

Московский государственный университет имени М. В. Ломоносова
Механико-математический факультет

В. М. Староверов

Лекции по курсу
«Работа на ЭВМ и программирование»
Часть 2

Учебное пособие



Издательство Московского Государственного Университета

2026

УДК 519.68+004.42(075.8)

ББК 32.97я73

C77

Рецензенты:

Заведующий кафедрой вычислительных технологий и моделирования
в геофизике и биоматематике физтех-школы
прикладной математики и информатики МФТИ,
чл.-корр.РАН, д.ф.-м.н., проф.
Василевский Юрий Викторович.

Доцент кафедры вычислительной математики
механико-математического факультета МГУ имени М. В. Ломоносова,
кандидат физико-математических наук
Валединский Владимир Дмитриевич.

Корректор:

старший научный сотрудник кафедры вычислительной математики
механико-математического факультета МГУ имени М. В. Ломоносова,
кандидат физико-математических наук
А. Е. Пентус

Староверов В. М.

C77 Лекции по курсу «Работа на ЭВМ и программирование». Часть 2 :
учебное пособие / В. М. Староверов. — Москва :
Издательство Московского Университета, 2026. —
311, [3] с. : ил.

ISBN 978-5-19-012325-8

Учебное пособие представляет собой конспекты второй половины лекций по курсу «Работа на ЭВМ и программирование», читаемых Староверовым В. М. в течение ряда лет на механико-математическом факультете МГУ им. М. В. Ломоносова. В курсе обсуждаются понятия *алгоритмов*, *структур данных*, их реализации на языках C/C++/Python, в частности в библиотеке шаблонов STL. Дается всесторонний обзор языка C++ на уровне разбора понятий *инкапсуляция* и *полиморфизм*. Формулируются и доказываются нижние и верхние оценки времени решения некоторых задач и времени работы различных алгоритмов.

Учебное пособие предназначено для студентов математических специальностей, специализирующихся в областях, связанных с прикладной математикой, а также в помощь преподавателям для проведения практических занятий.

УДК 519.68+004.42(075.8)

ББК 32.97я73

© Староверов В. М.

© Механико-математический факультет МГУ
имени М. В. Ломоносова, 2026

ISBN 978-5-19-012325-8 © Издательство Московского Университета, 2026

Содержание

1. Введение	7
2. C++. Простые и сложные классы. Правило трех. Исключения	11
2.1. Простые классы	13
2.2. Сложные классы. Правило трех	19
2.3. Два правила поведения языка C++ при работе со сложными классами	27
3. C++. Преобразование типов. Касты	31
4. C++. Сложные классы. Правило пяти. lvalue- и gvalue-ссылки	35
4.1. Правило пяти	35
4.2. lvalue- и gvalue-ссылки	39
5. Внутренние имена глобальных объектов	46
5.1. Внутренние имена переменных и функций	49
5.2. Шаблоны. Определение функций шаблона в отдельном сpp-файле	52
6. C/C++. Собственные процедуры отладки утечек памяти. Умные указатели. Функциональные объекты	58
6.1. Собственные процедуры отладки утечек памяти . .	58
6.2. Грамотная реализация realloc() на C++11. Placement new	71
6.3. Инструкция языка C++ static_assert	76
6.4. Emplace	78
6.5. Функциональные объекты	80
6.6. Умные указатели. unique_ptr. L1-список на основе умных указателей	84
6.7. Умные указатели. shared_ptr	91
7. Структуры данных. Вектор. STL	96
7.1. Правило нуля. Реализация вектора в стиле Python	97
7.2. Реализация вектора неограниченной длины	106
7.3. C++. Итераторы	112

7.4.	STL. Реализация вектора. Стандартные требования к итераторам STL	115
8.	STL. Структура данных <i>дек</i> и его адаптеры	123
8.1.	STL. Стек. Реализация стека	123
8.2.	STL. Очередь. Реализация очереди	125
8.3.	STL. Дек. Реализация дека	128
8.4.	C++. <code>initializer_list</code> . <code>range-based for</code>	129
8.5.	STL. Итераторы	132
9.	Структуры данных <i>списки</i> . Списки в STL	140
9.1.	L2-список с текущим положением	144
9.2.	L2-список с текущим элементом	145
9.3.	STL. Списки	150
9.4.	Циклический список с текущим положением	154
9.5.	L1-список с собственным отведением памяти	157
10.	C++. Исключения. Структурированные исключения Microsoft	163
11.	C++. Умные указатели. <code>weak_ptr</code> . L2-список на основе умных указателей	171
12.	Бинарные деревья	175
12.1.	Сбалансированные деревья (AVL-деревья)	200
12.2.	Операции с AVL-деревьями	202
12.3.	Красно-черные деревья	212
13.	Python. Аналоги массивов	225
13.1.	Python. Списки (<code>lists</code>). Срезы	225
13.2.	Python. Сортировки. Лямбда-функции	228
13.3.	Python. Кортежи (<code>tuples</code>)	229
13.4.	Python. Словари (<code>dictionaries</code> , <code>dict</code>)	230
13.5.	Python. Решение СЛУ. Матрица Гильберта. Классы	232
13.6.	Python. Неприятности. Изменение переменных. Параметры функций по умолчанию	241

13.7.	Лямбда-функции: Python <i>vs</i> C++ без захвата <i>vs</i> C++ с захватом	244
14.	В-деревья	254
14.1.	В-деревья	254
14.2.	Высота В-деревя	255
14.3.	Поиск элемента в В-дереве. Определение шаблона метода для класса, являющегося вложенным для некоторого класса. Алгоритм <code>lower_bound</code>	256
14.4.	Вставка элемента в В-дереве	259
14.5.	Удаление элемента из В-деревя	270
14.6.	В ⁺ -деревья	272
14.7.	Кратко обо всех основных контейнерах STL. Работа с пирамидой	275
15.	Графы	282
15.1.	Путь в графе с минимальным количеством шагов. Алгоритм волны	283
15.2.	Планарные графы. Формула Эйлера	287
15.3.	Поиск кратчайшего пути в графе. Алгоритм Дейкстры (Dijkstra's algorithm)	293
15.4.	Модифицированный алгоритм Дейкстры на основе STL	302
	Список использованных источников	310

1. Введение

Данное учебное пособие — это конспекты лекций второго полугодия годового курса «Работа на ЭВМ и программирование», читаемого в течение ряда лет на механико-математическом факультете МГУ им. М. В. Ломоносова. Это пособие является логичным продолжением пособия [11] с конспектами лекций предыдущего семестра.

Можно выделить следующие основные направления, обсуждаемые на лекциях:

- глубокий обзор языка C++ на уровне разбора понятий *инкапсуляция* и *полиморфизм*;
- разбор основных структур данных, используемых в программировании (в том числе их использование и реализация в STL);
- рассмотрение возможностей языка Python для работы с массивами и объектно-ориентированного программирования (как оно понимается в Python);
- введение в теорию графов на уровне разбора различных модификаций алгоритма Дейкстры.

Разбор понятия *наследование* не входит в данный курс — ему посвящен следующий семестр обучения. На данный момент это направление не рассматривается на лекциях и разбирается только на семинарских занятиях по окончании лекций. Все утверждения данного курса верны для структур/классов, в которых не используется наследование, далее это нигде упоминаться не будет.

Студент, тщательно изучивший данный курс, к концу семестра должен уметь пользоваться следующими понятиями языка C++ (версия C++11):

- 0) правила трех, пяти, нуля;
- 1) семантика перемещения и gvalue-ссылки;
- 2) лямбда-функции двух типов;
- 3) «умные» указатели (например, надо знать, как их использовать в списках и почему этого делать нельзя);
- 4) range-based циклы + initializer_list (нужно уметь делать свои классы доступными для range-based циклов);
- 5) default, delete;
- 6) auto, decltype;
- 7) nullptr;
- 8) static_assert.

Предполагается, что после окончания курса студент понимает эти термины и стоящие за ними принципы и механизмы, может объяснить их на теоретических собеседованиях и умеет использовать их в прикладном программировании.

Стоит отдельно сказать о компиляторах языков C/C++, которые использовались при подготовке данного пособия. На данный момент подавляющее количество программ на языках C/C++ предназначены для компиляции либо компилятором Microsoft, либо компилятором gcc. Данное пособие ориентируется именно на эти компиляторы.

Предполагается, что компилятор gcc по умолчанию используется с ключами

```
-W -Wall -Wfloat-equal -Wpointer-arith -Wwrite-strings -Wcast-align
-Wformat-security -Wmissing-format-attribute -Wformat=1
-Wno-long-long -Wcast-align -Winline -Werror -pedantic
-pedantic-errors -Wunused -Wuninitialized --param
inline-unit-growth=1000000 --param max-inline-insns-single=10000000
--param large-function-growth=10000000 -fPIC.
```

Данный набор ключей определяет параноидально жесткое поведение компилятора, проявляющееся в выдаче замечаний по мельчайшим поводам, и приводит к расцениванию этих замечаний как ошибок. Это приучает пользователей писать программы, которые при компиляции не дают замечаний. Подобный стиль программирования весьма полезен в начале обучения, так как очень часто именно замечания компилятора позволяют находить в программе реальные ошибки. Вообще, при написании программ имеет смысл с благодарностью принимать любую возможную помощь от компилятора и сопутствующих программ (мы будем разбирать такие программы в дальнейшем). В частности, весьма полезно добавлять к длинному приведенному выше списку ключей компилятора gcc ключ оптимизации `-O2`. Содержательно это приводит к тому, что часть выполнения программы переносится на уровень компиляции (на стадии компиляции выполняются некоторые простые действия, которые при обычной компиляции программы должны выполняться только во время работы программы). А это иногда позволяет отловить элементарные ошибки, которые могут совершать начинающие программисты (например, выявить неинициализированные переменные).

Предполагается, что читатель в процессе обучения будет программировать в OS Windows, используя файловый менеджер Far, компиляторы gcc и Microsoft (эти три программных продукта бесплатны для учебных целей). Ссылки на дистрибутивы файлового менеджера Far и компилятора gcc можно найти на сайте автора пособия <http://lectures.stargeo.ru>. В частности, на сайте есть ссылка на командные файлы для OS Linux и Windows (они называются соответственно `wgcc` и `wgcc.bat`), с помощью которых можно вызывать компилятор gcc с указанными выше жесткими ключами. Также на сайте можно найти аналогичные командные файлы `wg++` и `wg++.bat` для компилятора g++ программ на языке C++.

Программную среду для компилятора Microsoft можно установить, скачав соответствующий дистрибутив с сайта производителя. Имеет смысл приложить минимальные усилия для создания командного файла, с помощью которого можно будет вызывать компилятор Microsoft в командной строке. По умолчанию Microsoft не предоставляет такой возможности, но в папке компилятора Microsoft легко найти командный файл, который задает все необходимые переменные среды, необходимые для работы компилятора Microsoft. У автора для последней версии компилятора Microsoft полный путь к командному файлу, задающему необходимые переменные среды, такой: "C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Auxiliary\Build\vcvarsall.bat".

Остается только создать командный файл с именем `cc.bat`, расположенный в доступной для поиска исполняемых файлов папке, например, с таким содержанием:

```
1 call "C:\Program Files\Microsoft Visual Studio\2022\  
Community\VC\Auxiliary\Build\vcvarsall.bat" x86  
2 cl.exe /EHsc /Zi %1 %2 %3 %4 %5 %6
```

Теперь создать программу из исходного файла `q.cpp` можно из командной строки с помощью простой команды

```
1 cc q.cpp
```

Отметим, что в разных версиях Microsoft Visual Studio файл `vcvarsall.bat` может располагаться в разных папках. Например, в Microsoft Visual Studio 2010 командный файл `cc.bat` должен выглядеть следующим образом:

```
1 call "C:\Program Files (x86)\Microsoft Visual Studio 10.0\  
2 VC\vcvarsall.bat" x86  
3 cl.exe /EHsc /Zi %1 %2 %3 %4 %5 %6
```

Везде в дальнейшем для вызова соответствующих компиляторов мы будем использовать именно эти имена: `wgcc` (либо `gcc`, если соответствующий командный файл назван `gcc.bat`), `wg++` (либо просто `g++`) и `cc`.

Исторически сложилось так, что автор использует две версии компилятора Microsoft: старый компилятор 16-й версии, для которого полный путь к упомянутому командному файлу у автора выглядит как

```
C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\vcvarsall.bat,
```

и последний (на момент написания) компилятор 19-й версии с ранее указанным файлом, задающим переменные среды. Каждому из данных компиляторов соответствует своя версия среды программирования.

Пользователи могут установить среду компилятора Microsoft на другой диск, тогда пути к файлам скорректируются соответствующим образом. Надо иметь в виду, что смена версий компилятора Microsoft приводит к исправлению некоторых ошибок, появлению новых возможностей, общему замедлению среды программирования и появлению новых ошибок. Например, более ранняя из упомянутых сред программирования не поддерживает обновления кода программы на лету (технологии *Edit and Continue*) для 64-битных приложений, что является весьма существенным недостатком. Более поздняя версия среды программирования лишена этого недостатка, но она порой невообразимо медлительна. Дополнительная неприятность состоит в том, что код, написанный для старого компилятора, не компилируется новым компилятором. В результате приходится сохранять старый компилятор для работы со старыми большими проектами. Мораль, проистекающая из этих неприятностей, заключается в том, что следует учиться программировать на разных версиях языков C/C++, так как нельзя заранее предугадать, на каком компиляторе пользователю придется работать в дальнейшем.

Стоит напомнить, что данный курс лекций гармонично дополняется задачами, которые можно найти в [6].

2. C++. Простые и сложные классы. Правило трех. Исключения

Предполагается, что читатель знаком с материалом предыдущей части курса [11] и имеет начальное (*вульгарное*) представление о языке C++, т. е. знает общий синтаксис создания и использования структур и классов в языке C++, шаблонов, переопределения операторов (а это, в свою очередь, требует знания полного набора операторов языка C [4], поскольку язык C++ не позволяет вводить новых операторов и менять их приоритет), в частности операторов, используемых для ввода/вывода. Мы рассмотрим эти темы более подробно. В данной главе детально разбираются понятия *простого* и *сложного класса* и рассматриваются особенности работы с данными объектами. Безусловно, содержание этой темы (как и всего, что последует далее) покрывается классическим описанием языка C++ [19], но данная книга скорее интересна тем, кто уже знает язык C++ хорошо и хочет разбираться в его тонкостях.

Напомним, что функции класса принято называть *методами*, а данные внутри класса принято называть *элементами данных* или просто *элементами*.

Сразу надо заметить, что в языке C++ структуры и классы имеют всего два отличия:

- 1) все методы и данные в начале описания класса являются по умолчанию приватными, а внутри структуры — публичными;
- 2) хоть это и звучит тавтологично, структуры являются *структурами*, а классы — *классами*.

Мы обсудим эти отличия и далее уже не будем разделять понятия *структуры* и *класса*. Будем говорить только о *классах* всегда, когда это не будет приводить к недопониманию.

Под *приватностью* подразумевается невозможность обращения извне к элементам и методам класса, а под *публичностью* — возможность такого обращения. Надо иметь в виду, что понятие *вне класса* имеет контекстный характер, а не объектный, т. е. контекстно внутри описания класса можно обращаться к любым подобъектам объектов данного типа (данным и методам), а вне описания класса для объектов данного типа возможно прямое обращение только к публичным подобъектам. Например, корректны следующие обращения к приватному члену x класса T :

```

1 class T
2 {int x; // приватный элемент
3 // обращение к приватному члену класса:
4 void f() {T y; x=1; y.x=x;}
5 };

```

Следующее обращение к приватному члену класса вне описания класса некорректно:

```

1 void g(void)
2 {T z; z.x=0;}

```

Второе замечание имеет смысл в ситуациях, когда нужно указать компилятору, что данное имя является именем класса или структуры до собственно описания класса или структуры. При этом не требуется иметь информацию о содержании данного класса или структуры. Например, у нас есть два класса А и В, причем внутри класса А содержится указатель на класс В, а внутри класса В содержится указатель на класс А:

```

1 class A
2 {B *x;
3 };
4 class B
5 {A *x;
6 };

```

При любом порядке описания данных классов мы имеем указатель на класс, у которого в тексте программы нет описания до данной позиции, что пока является синтаксической ошибкой. Решить проблему можно, указав до приведенной выше конструкции, что А и В являются именами типов классов:

```

1 class A; class B;
2 class A
3 {B *x;
4 };
5 class B
6 {A *x;
7 };

```

При этом для структур А и В, конечно же, надо указывать, что данные имена являются именами структур:

```
1 struct A; struct B;
2 struct A
3 {B *x;
4 };
5 struct B
6 {A *x;
7 };
```

Существенным здесь является то, что пока нам не важно содержимое этих классов. Если же в классах содержатся методы, использующие данные указатели, то определения этих методов следует размещать уже после полного описания классов:

```
1 struct A; struct B;
2 struct A
3 {B *x;
4 void f();
5 };
6 struct B
7 {A *x;
8 void f();
9 };
10 void A::f() {x=nullptr;}
11 void B::f() {x=nullptr;};
```

Иногда (об этом речь пойдет позднее), когда требуется уточнить в программе, что данное имя А является именем типа, достаточно написать **typename** А, но в вышеприведенных примерах надо все же указывать явно, чем является А: классом или структурой.

2.1. Простые классы

Простыми классами называются классы, не подразумевающие отведения каких-либо ресурсов внутри своих объектов. Иными словами, простые классы содержат элементы только базовых типов.

Опишем работу с простым классом на примере структуры данных *вектор*, являющейся аналогом *массива* в языке С.

Для реализации простого класса требуется весьма ограниченный набор функций. Например, для работы с классом *вектор*

```

1 struct Vector
2 {int v[N];
3   ...
4 };

```

в простейшем случае достаточно определить только оператор *квадратные скобки*:

```

1 int &operator [] (size_t i){return v[i];}

```

Здесь в качестве N следует использовать либо целочисленную константу, либо соответствующее макроопределение.

Для созданной структуры желательно определить оператор <<, используемый для вывода содержимого структуры на экран:

```

1 ostream &operator<<(ostream&cout, const Vector &v)
2 {cout<<"("; for(int i=0;i<N;i++)
3   {cout<<v.v[i]<<(i<N-1?" ":"");} cout<<""); return cout;}

```

Полный код примера выглядит следующим образом:

```

1 #include<iostream>
2 #include<fstream>
3 #include<string>
4 #include<cstdlib>
5 #include<cstring>
6 using namespace std;
7 #ifndef N
8 #define N 10
9 #endif
10 struct Vector
11 {int v[N];
12   int &operator [] (size_t i){return v[i];}
13 };
14 ostream &operator<<(ostream&cout, const Vector &v)
15 {cout<<"("; for(int i=0;i<N;i++)
16   {cout<<v.v[i]<<(i<N-1?" ":"");} cout<<""); return cout;}
17 int main(void)

```

```

18 {Vector v; int m[N]={0,1,2,3,4,5,6,7,8,9};
19 for(int i=0;i<N;i++){v[i]=m[i];} cout<<"v"<<v<<endl;
20 return 0;
21 }

```

При обычной компиляции данного примера N задается макроопределением как 10. Однако если задать макроопределение N при компиляции:

```

1 g++ -c -D N=20 q.cpp

```

то N получит другое значение.

Пришло время разобраться с системой обработки ошибок в языке C++. Система обработки ошибок языка C, связанная с возвратом функциями кодов ошибок, здесь не годится, так как, например, при переопределении операторов для кода ошибки просто нет места (значение, возвращаемое функцией, переопределяющей соответствующий оператор, уже используется для других целей). Вместо этого в языке C++ используются *исключения*. Про исключения надо знать, что их можно *выбрасывать* и *ловить*. При выбрасывании исключения указывается переменная/значение определенного типа, содержащая информацию об ошибке. Далее происходит выход из всех функций без выполнения их кода, за исключением выполнения деструкторов (что тоже можно отменить соответствующими ключами компилятора), вплоть до момента ловли исключения с данным типом переменной. Логика языка запрещает использование исключений для каких-либо целей, отличных от обработки ошибочных ситуаций. Правила хорошего тона требуют задания внутри класса специального типа, который будет использоваться для исключений, связанных с работой данного класса. Например, в классе *вектор* можно создать тип (структуру) SErr, в которой будет храниться строка, описывающая произошедшую ошибку. При выходе за границы массива будем выбрасывать исключение данного типа с описанием ошибки:

```

1 struct SErr{string s; SErr(const char*s){this->s=s;}};
2 int &operator [] (size_t i)
3 {if(i>=N){throw SErr("vector: bad index");} return v[i];}

```

Для ловли исключения используется блок `try {...} catch(){...}` с указанием типа отлавливаемой переменной:

```

1  try {
2  for (int i=0; i<=N; i++)v[i]=m[i];
3  } catch (Vector::SErr err) {cout<<err.s<<endl;}

```

В конце данного цикла происходит выход за границу массива, что должно привести к выбросу соответствующего исключения.

Инструкция **catch** ловит исключение только указанного типа, далее (только в случае поимки исключения!) происходит выполнение блока, следующего за инструкцией **catch**, после чего продолжается выполнение программы обычным способом.

Далее следует упомянуть еще одно правило хорошего тона объектно-ориентированного программирования: все данные класса (элементы) обязаны быть приватными и все обращения к ним извне класса должны происходить при помощи соответствующих методов. Данный подход помогает легко менять внутреннюю структуру класса с помощью изменения собственно внутренней структуры и изменения соответствующих методов данного класса, не заботясь о способах обращения к элементам класса извне класса (поскольку прямых обращений к элементам класса извне класса просто нет). Таким образом, структуру вектора надо превратить в класс с приватными элементами. При этом станет невозможным прямое обращение к элементам класса и вышеприведенный вариант оператора вывода содержимого класса на экран станет неработоспособным. Его можно заменить на следующий:

```

1  ostream &operator<<(ostream&cout, const Vector &v)
2  {cout<<" "; for (int i=0; i<N; i++)
3      {cout<<v[i]<<(i<N-1?" ":""); cout<<" "; return cout;}

```

Здесь прямое обращение к элементу класса заменено на вызов оператора *квадратные скобки* (что сводится к вызову соответствующего метода класса). К сожалению, в результате мы получаем другую проблему: внутри данной функции нельзя использовать ранее определенный оператор *квадратные скобки*, поскольку он не гарантирует неизменяемости данного класса, а класс `Vector` в данной функции имеет спецификацию **const**. Проблема решается определением еще одного оператора *квадратные скобки*, дающего данные гарантии:

```

1 const int &operator [(size_t i)const { if (i>=N)
2   { throw SErr("vector: bad index"); } return v[i]; }

```

Здесь первая спецификация **const** гарантирует неизменяемость значения по возвращаемой ссылке (данный оператор нельзя использовать слева от знака присваивания), а вторая — неизменяемость данного класса при вызове данного оператора (=при вызове данной функции).

В качестве примера бинарной операции над векторами создадим метод, задающий бинарный оператор сложения:

```

1 Vector operator+(const Vector&b)const
2 { Vector r; for (int i=0; i<N; i++){ r[i]=v[i]+b.v[i]; } return r; }

```

Здесь при сложении двух векторов данный класс (***this**) выступает в качестве левого слагаемого в сложении, а аргумент функции выступает в качестве правого слагаемого.

Следует иметь в виду, что логика языка предполагает, что оператор сложения не изменяет своих аргументов. Данное требование является лишь логическим и не подкрепляется никакими синтаксическими требованиями. Однако, во избежание недоразумений, следует четко ему следовать. Отсюда вытекает необходимость создания локальной переменной, в которую следует поместить результат сложения. А поскольку данная переменная жива только внутри блока, мы получаем необходимость ее возврата из функции по значению. Вообще, следует в каждом возможном случае передавать «толстые» объекты внутрь функций и возвращать «толстые» значений из функций по ссылке (что сводится к фактической передаче лишь указателя на объект). Однако, если это невозможно (как в приведенном примере), приходится довольствоваться возвратом объекта по значению.

Окончательный пример определения и использования описываемого класса, оформленного в виде шаблона, выглядит следующим образом:

```

1 #include<iostream>
2 #include<fstream>
3 #include<string>
4 #include<cstdlib>
5 #include<cstring>

```

```

6  using namespace std;
7  //—
8  #ifndef N
9  #define N 10
10 #endif
11 template <class T>class Vector
12 {T v[N];
13 public:
14   struct SErr{string s; SErr(const char*s){this->s=s;}};
15   T &operator [] (size_t i){
16     if(i>=N){throw SErr("vector: bad index");} return v[i];}
17   const T &operator [] (size_t i) const
18     {if(i>=N){throw SErr("vector: bad index");} return v[i];}
19   Vector operator+(const Vector&b) const
20     {Vector r; for(int i=0;i<N;i++){r[i]=v[i]+b.v[i];} return r;}
21 };
22 template<class T>ostream &operator<<
23   (ostream&cout, const Vector<T> &v)
24   {cout<<" "; for(int i=0;i<N;i++)
25     {cout<<v[i]<<(i<N-1?" ":"");} cout<<" "; return
26     cout;}
27 //—
28 int main(void)
29 {Vector<int> v,w; int m[N]={0,1,2,3,4,5,6,7,8,9};
30   try{for(int i=0;i<=N;i++)v[i]=m[i];
31     } catch(Vector<int>::SErr err){cout<<err.s<<endl;}
32   cout<<"v="<<v<<endl;
33   w=v; cout<<"w="<<w<<endl;
34   w=v+w; cout<<"v+w="<<w<<endl;
35   return 0;
36 }

```

Здесь вектор реализован как шаблон, в котором тип `class T` является параметром; функция `main()` работает с вектором целых чисел, в шаблон передается `int`. Вместо класса описывается шаблон класса. Вместо функции, задающей оператор `<<` для вывода содержимого объекта на экран, создается шаблон соответствующей функции. Настоящий класс и функция вывода в реальности определяются только при создании объекта с заданным типовым параметром `Vector<int>`

и при вызове функции вывода соответственно. Безусловно, это является некоторой головной болью компилятора. Действительно, если класс `Vector<int>` создается в двух независимых исходных файлах одной программы, то получается, что тип объекта и функция с одинаковыми именами оказываются независимо заданными в двух местах программы. И сборщику программы остается с этим мириться (несмотря на то, что коварный программист может определить ту же функцию, задающую оператор `<<` для данного типа, по-разному в двух местах программы, что приведет к непредсказуемым последствиям). Не забудем также, что в цикле в функции `main()` происходит выход за границы массива.

Здесь также стоит отметить, что фрагмент кода между комментариями `//--` хорошо бы было поместить в отдельный `include`-файл, а уже этот файл вставить в данный код с помощью директивы `#include`. Этот `include`-файл можно будет вставлять и в другие исходные файлы программы, и это не приведет к неприятностям (в плане ошибок сборки, несмотря на явное множественное определение функций), но надо будет следить за тем, чтобы макроопределение `N` задавалось одинаково во всех файлах программы.

2.2. Сложные классы. Правило трех

Предельный минимализм, проповедуемый при работе с простыми классами, оказывается абсолютно неприменимым для сложных классов. Под *сложными классами* подразумеваются классы, в методах которых происходит явное отведение ресурсов. Безусловно, явное отведение ресурсов требует также их явного освобождения. Чаще всего под отведением/очисткой ресурсов подразумевается отведение/очистка памяти, но возможны и другие виды ресурсов: сокет, каналы/поток работы с файлами, семафоры и т. д. Основная причина, по которой невозможно работать со сложными классами так же, как с простыми, состоит в том, что при побайтовом присваивании или копировании объекта такого типа возникает две побайтовые копии объекта, владеющие одними и теми же ресурсами. Очистка ресурсов в одном объекте делает работу со второй копией объекта некорректной, причем вторая копия объекта в создавшейся ситуации может ничего не знать о своей нелегитимности, что приведет к некорректной работе программы. Отсюда вытекает основное

требование к сложным классам: должно быть обеспечено корректное присваивание друг другу и копирование объектов такого типа. Кроме того, сложные классы должны обеспечивать автоматическую очистку всех захваченных ресурсов при смерти объекта (реализуется с помощью деструктора). Но выполнение операций по очистке ресурсов требует корректного состояния всех переменных, идентифицирующих данные ресурсы, а для этого необходима правильная инициализация объекта (реализуется с помощью соответствующего конструктора/конструкторов).

Термины *присваивание* и *копирование* не являются синонимами. Под *присваиванием* подразумевается действие, осуществляемое при произведении операции вида $a=b$, при котором объект a уже существует перед произведением данной операции. Перед началом собственно копирования требуется очистка ресурсов в объекте a . *Копирование* происходит при передаче параметра в функцию по значению и при возврате значения из функции. Данное действие представляет собой способ создания новой переменной как копии другой переменной, при этом новая переменная создается буквально на голом месте и очищать никакие ресурсы предварительно здесь не требуется (банально потому, что все переменные объекта в начале копирования не инициализированы и с ними нельзя совершать никаких операций, кроме присваивания им корректных значений). В языке C++ копирование реализуется с помощью конструктора копирования, синтаксис которого будет описан ниже.

Все вышесказанное задает прожиточный минимум для нормального существования сложного класса:

- 1) конструктор по умолчанию (и/или другие необходимые конструкторы, создающие объект),
- 2) конструктор копирования,
- 3) деструктор,
- 4) оператор присваивания.

Правило трех — это требование наличия в классе трех последних пунктов этого списка, эти пункты однозначны. Понятно, что в правиле трех подразумевается также необходимость первого пункта, но однозначности здесь уже нет. Как правило, для создания объекта требуется конструктор по умолчанию (например, без него не создать массив объектов). Однако бывают специфические объекты, для ко-

торых конструктор по умолчанию просто запрещен (например, если создать приватный конструктор по умолчанию, то любая попытка создать объект конструктором без параметров приведет к синтаксической ошибке), и тогда объект должен создаваться конструктором с какими-то заданными параметрами. Подобная ситуация возникает в случае, когда существование объекта не имеет смысла без задания каких-то определенных значений. Например, число в кольце вычетов по модулю n не имеет смысла без задания этого самого n .

Разберемся с конструированием сложного класса на примере все того же вектора, но теперь размер вектора должен храниться в самом объекте и может меняться в процессе работы с вектором. Соответственно, данные вектора будут представлять собой указатель на массив объектов и размер этого массива:

```
1 T *v; size_t n;
```

Напомним, что мы работаем с шаблоном, задающим вектор объектов типа T .

Опишем технику программирования сложных классов с использованием трех базовых функций, которые далее будем называть `SetZero()`, `Clean()`, `CopyOnly()`. Практически все сложные классы можно задавать с использованием данной техники. Функция `SetZero()` должна приводить переменные класса в изначальное корректное состояние (как правило — обнулять их). Функция `Clean()` должна очищать захваченные классом ресурсы и после этого вызывать функцию `SetZero()`. Функция `CopyOnly()` должна копировать объект, передаваемый ей в качестве параметра, в данный объект, не заботясь о предварительной очистке ресурсов данного объекта. Если эти функции заданы, то для любого сложного класса конструктор копирования, деструктор и оператор присваивания задаются одинаковым образом:

```
1 Vector(const Vector&b) { CopyOnly(b); }
2 ~Vector() { Clean(); }
3 Vector &operator=(const Vector&b)
4 { if(&b!=this) { Clean(); CopyOnly(b); } return *this; }
```

Здесь конструкция `if(&b!=this)` предусмотрена на случай присваивания объекта самому себе (`a=a`), когда очистка объекта привела бы к невозможности дальнейшей работы с ним. Конструктор по умолчанию также задается универсальным образом:

```
1 Vector() {SetZero();}
```

Однако нам будет удобно объединить его с конструктором, задающим сразу вектор заданного размера:

```
1 Vector(size_t n=0) {if(n>0){this->n=n; v=new T[n];
2   memset(v,0,n*sizeof(*v));} else SetZero();}
```

Данный конструктор, вызванный без параметров, эквивалентен простой очистке объекта. Если же задать ненулевой параметр, то создается вектор заданного размера. Большим недостатком данной конструкции является то, что она неприменима для сложных типов *T*, поскольку далеко не всегда побайтовое обнуление объекта является его корректной инициализацией (для базовых типов это работать будет). Разберемся с этим чуть позже.

В нашем случае три вышеперечисленные функции задаются следующим образом:

```
1 void SetZero() {v=nullptr; n=0;}
2 void Clean() {delete [] v; SetZero();}
3 void CopyOnly(const Vector&b) {if(b.n){v=new T[n=b.n];
4   for(size_t i=0;i<n;i++)v[i]=b.v[i];} else SetZero();}
```

Разберемся с уже упомянутой проблемой инициализации, возникающей при конструировании шаблонов. Для нас было бы весьма удобным стандартно инициализировать все элементы создаваемого вектора (например, для вектора целых или вещественных чисел — нулевым значением). Если тип *T* представляет собой правильно созданный класс, то мы не будем иметь проблем, так как можем присвоить каждому элементу созданного класса объект *T()*. Под этой конструкцией подразумевается неименованный объект типа *T*, созданный с помощью конструктора по умолчанию (безусловно, мы рассчитываем, что у типа *T* такой конструктор есть). Отметим, что неименованные объекты весьма активно используются в языке *C++* с применением указанного синтаксиса. Однако чисто формально для базовых типов (еще их называют *POD-типами*, Plain Old Data) конструктора по умолчанию нет (можно считать, что он есть, но пустой). Для *POD*-типов конструкция *T()* приводит к созданию объекта соответствующего типа, инициализированного нулем (соответству-

ющего типа). Таким образом, создание локальной автоматической переменной `int x`; не приводит к ее инициализации, но переменную можно инициализировать либо явно конкретным значением, либо с помощью вышеописанной инициализации `int x=int()`; В рамках всего вышенаписанного было бы логичным инициализировать переменную с помощью вызова конструктора без параметров: `int x()`; однако согласно синтаксису языка C таким образом мы опишем **функцию**, возвращающую целое значение, и это совсем не то, чего мы хотим. Если углубляться в тему, то можно заметить, что переубедить компилятор можно с помощью конструкции `int x(0)`; которая уже не является описанием функции и которую можно использовать для определения целой переменной, инициализированной нулем. Однако данный подход не подойдет для использования внутри шаблона, когда вместо типа `int` используется параметр шаблона `T`.

Итак, конструктор для нашего вектора примет вид:

```
1 Vector(size_t n){SetZero(); if(n>0){this->n=n;
2   v=new T[n]; for(size_t i=0;i<n;i++)v[i]=T();}}
```

В конечном итоге мы получим следующий код для реализации и использования сложного класса шаблона вектора:

```
1 #include<iostream>
2 #include<fstream>
3 #include<string>
4 #include<cstdlib>
5 #include<cstring>
6 using namespace std;
7 #ifndef N
8 #define N 10
9 #endif
10 template<class T> class Vector
11 {T *v; size_t n;
12 public:
13 //—
14   Vector(){SetZero();}
15   Vector(size_t n){SetZero(); if(n>0)
16     {this->n=n; v=new T[n]; for(size_t i=0;i<n;i++)v[i]=T();}}
17   ~Vector(){Clean();}
18   Vector(const Vector&b){CopyOnly(b);}
```

```

19  Vector &operator=(const Vector&b)
20  { if(&b!=this){Clean(); CopyOnly(b);} return *this;}
21  //—
22  void SetZero(){v=nullptr; n=0;}
23  void Clean(){delete [] v; SetZero();}
24  void CopyOnly(const Vector &b){v=new T[n=b.n];
25    for(size_t i=0;i<n;i++)v[i]=b.v[i];}
26  //—
27  size_t size()const{return n;}
28  struct SErr{string s; SErr(const char *s){this->s=s;}};
29  T &operator[](size_t i){if(i>=n)
30    {throw SErr("vector: bad index");} return v[i];}
31  const T &operator[](size_t i)const{if(i>=n)
32    {throw SErr("vector: bad index");} return v[i];}
33  Vector operator+(const Vector&b)const{Vector r(n);
34    for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
35  };
36  template<class T> ostream &operator<<
37    (ostream&cout, const Vector<T> &v)
38  {cout<<" "; for(size_t i=0;i<v.size();i++)
39    {cout<<v[i]<<(i<v.size()-1?" ":"");} cout<<")"; return cout;}
40  int main(void)
41  {Vector<Vector<double>> v(N),w; int m[8]={0,1,2,3,4,5,6,7};
42    try{
43      for(int i=0;i<N;i++)
44        {v[i]=Vector<double>(8); for(int j=0;j<8;j++)v[i][j]=m[j];}
45    }catch(Vector<Vector<double>>::SErr err){cout<<err.s<<endl;}
46    cout<<"v="<<v<<endl;
47    w=v; cout<<"w="<<w<<endl;
48    w=v+w; cout<<"v+w="<<w<<endl;
49    return 0;
50  }

```

Здесь для проверки корректности написанного кода мы вместо вектора целых чисел создали вектор векторов вещественных чисел. Поскольку для вектора есть шаблон для оператора сложения, то и для нашего сложного вектора (вектора векторов) мы имеем оператор сложения, и для векторов — элементов этого (сложного) вектора опе-

ратор сложения задается тем же самым шаблоном оператора сложения.

Чтобы 44-я строка приняла более читаемый вид и для совместимости со стандартными реализациями векторов, в нашем векторе желательно определить функцию добавления элемента в конец (длина вектора при этом увеличивается на единицу). Такую функцию принято называть `push_back()`. Однако оказывается, что приведенная реализация не годится для этой цели. Если мы будем создавать вектор длины n , добавляя к нему все элементы по одному, то, как легко увидеть, суммарное время работы таких операций будет оцениваться как $\Theta(n^2)$. Для того чтобы время выполнения данных действий было $\Theta(n)$, модифицируем алгоритм способом, описанным в лекциях первого семестра [11]. Добавим в класс переменную `size_t nmax`, в которой будем держать размер реально созданного массива, хранящегося в памяти. В переменной `n` по-прежнему будем хранить длину вектора. При добавлении элементов к вектору `n` будет увеличиваться, пока не достигнет значения `n_max`; при достижении этого значения произойдет увеличение `n_max` вдвое с перераспределением памяти. Легко убедиться, что при этом время создания массива из n элементов будет равно $\Theta(n)$. Модифицированный код примет вид

```
1  #include<iostream>
2  #include<fstream>
3  #include<string>
4  #include<cstdlib>
5  #include<cstring>
6  using namespace std;
7  #ifndef N
8  #define N 10
9  #endif
10 template<class T> class Vector
11 {T *v; size_t n,nmax;
12 public:
13     //—
14     Vector(size_t n=0){SetZero(); if(n
15     {v=new T[nmax=this->n]; for(size_t i=0;i<n;i++)v[i]=T();}}
16     Vector(const Vector&b){CopyOnly(b);}
17     ~Vector(){Clean();}
18     Vector &operator=(const Vector&b)
```

```

19  { if(&b!=this){Clean(); CopyOnly(b);} return *this;}
20  //—
21  void SetZero(){v=nullptr; n=0; nmax=0;}
22  void Clean(){delete [] v; SetZero();}
23  void CopyOnly(const Vector&b){if(b.nmax)
24  {v=new T[nmax=b.nmax]; n=b.n;
25  for(size_t i=0;i<n;i++)v[i]=b.v[i];} else SetZero();}
26  //—
27  size_t size()const{return n;}
28  struct SErr{string s;SErr(const char
29      *s="error"){this->s=s;}};
30  T &operator[](size_t i)
31  {if(i>=n)throw SErr("bad index"); return v[i];}
32  const T &operator[](size_t i)const
33  {if(i>=n)throw SErr("bad index");return v[i];}
34  Vector operator+(const Vector&b)const{Vector r(n);
35  for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
36  void push_back(const T&x)
37  {if(n==nmax){T*w=new T[2*(nmax+1)]; size_t i=0;
38  for(;i<n;i++){w[i]=v[i];} for(;i<2*(nmax+1);i++)w[i]=T();
39  nmax=2*(nmax+1); delete [] v; v=w;} v[n++]=x;}
40  };
41  //—
42  template<class T>ostream &operator<<
43  (ostream &cout, const Vector<T> &v)
44  {cout<<" "; for(size_t i=0;i<v.size();i++)
45  {cout<<v[i]<<(i<v.size()-1?" ":"");} cout<<" ";return cout;}
46  //—
47  int main(void)
48  {Vector<Vector<double>> v(N),w; int m[8]={0,1,2,3,4,5,6,7};
49  try{
50  for(int i=0;i<N;i++)
51  {for(int j=0;j<8;j++)v[i].push_back(m[j]);}
52  }catch(Vector<Vector<double>>::SErr err){cout<<err.s<<endl;}
53  cout<<"v="<<v<<endl;
54  w=v; cout<<"w="<<w<<endl;
55  w=v+w; cout<<"v+w="<<w<<endl;
56  return 0;
57  }

```

Подводя общий итог, надо подчеркнуть, что для простых классов не требуется явная реализация основных конструкторов и операторов присваивания, в то время как для сложных классов эти реализации обязательны. Однако при программировании с использованием OpenMP ([2], [10], [20]) специфика компилятора Microsoft требует, тем не менее, явной реализации конструктора копирования даже для простого класса.

2.3. Два правила поведения языка C++ при работе со сложными классами

Можно говорить о двух принципиально различных правилах поведения программ на языке C++ при работе со сложными классами. Будем называть эти правила *стандартным поведением* и *нестандартным поведением* (данные понятия не встречаются в литературе). Варианты поведения легко объясняются на примере интерпретации языком выражения $a=b+c$. При *стандартном поведении* предполагается, что при возврате значения из функции (при возврате переменной по значению) компилятором должна создаваться временная переменная с помощью конструктора копирования от возвращаемой из функции переменной, т. е. при вызове функции, обслуживающей оператор сложения, должна с помощью конструктора копирования создаваться временная переменная, и именно эта переменная будет присваиваться переменной, стоящей слева от знака присваивания. После этого временная переменная получит право умереть. Вообще, при данном подходе все временные переменные, создаваемые при обработке выражения, имеют право умереть только после полной обработки выражения.

При *нестандартном поведении* временная переменная не создается. Вместо этого локальная переменная, возвращаемая из функции (в нашем примере из функции оператора сложения) напрямую используется в обработке всего выражения. И тогда именно эта локальная переменная будет присвоена переменной a при обработке выражения $a=b+c$. Естественно, компилятор позаботится о том, что все возвращаемые локальные переменные умрут только после полной обработки выражения. С одной стороны, этот подход позволяет существенно ускорить работу программы, но в сложных случаях он

может привести к ошибкам (все же использование локальных переменных вне тела функции — вещь потенциально небезопасная).

Выбор варианта поведения языка задается ключами компилятора.

Создадим пару функций для вывода служебной информации на экран:

```
1 void out(const char*s){cout<<s<<endl;}
2 template<class T> void out
3 (const char*s, const T&x, const char*s2){cout<<s<<x<<s2<<endl;}
```

Упростим в предыдущем примере главную функцию, тестирующую работу с вектором целых чисел:

```
1 int main(void)
2 {Vector<int> v; int m[5]={0,1,2,3,4};
3 for(int i=0;i<5;i++)v.push_back(m[i]);
4 v=v+v; cout<<"v+v="<<v<<endl;
5 return 0;}
```

Вставим соответствующий вывод в конструктор, деструктор и оператор присваивания:

```
1 Vector(size_t n=0){out("Vector(",n,")");SetZero();
2 if(n){v=new T[nmax=this->n=n];
3 for(size_t i=0;i<n;i++)v[i]=T();}}
4 Vector(const Vector&b)
5 {out("Vector(const Vector&b)"); CopyOnly(b);}
6 ~Vector(){out("~Vector()"); Clean();}
7 Vector &operator=(const Vector&b)
8 {out("Vector &operator=(const Vector&b)");
9 if(&b!=this){Clean(); CopyOnly(b); return *this;}}
```

а также в оператор сложения:

```
1 Vector operator+(const Vector&b) const
2 {out("Vector operator+(const Vector&b)");
3 if(n!=b.n){throw(SErr("n!=b.n"));} Vector r(n);
4 for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
```

Теперь мы можем узнать, как ведут себя компиляторы при работе с выражением `v=v+v;`,

При компиляции программы компилятором g++

```
1 g++ q.cpp
```

мы получаем нестандартное поведение языка C++, что выражается в следующем выводе на экран:

```
1 Vector (0)
2 Vector operator+(const Vector&b)
3 Vector (5)
4 Vector &operator=(const Vector&b)
5 ~Vector ()
6 v+v=(0 2 4 6 8)
7 ~Vector ()
```

При компиляции программы компилятором g++ с ключом:

```
1 g++ -fno-elide-constructors q.cpp
```

мы наблюдаем стандартное поведение языка с соответствующим выводом на экран:

```
1 Vector (0)
2 Vector operator+(const Vector&b)
3 Vector (5)
4 Vector (const Vector&b)
5 ~Vector ()
6 Vector &operator=(const Vector&b)
7 ~Vector ()
8 v+v=(0 2 4 6 8)
9 ~Vector ()
```

Аналогичное (стандартное) поведение наблюдается для компилятора C++ Microsoft. Компилятор Microsoft у автора везде далее вызывается командой cc:

```
1 cc q.cpp
```

Читателю предлагается самостоятельно разобраться, каким конкретным операциям и созданию каких переменных соответствуют строки полученного вывода на экран.

На учебных занятиях крайне рекомендуется использовать только стандартное поведение языка C++. Практика показывает, что

использование нестандартного поведения в некоторых случаях позволяет скрывать ошибки при написании программ (заведомо неправильно написанная программа может правильно работать), что довольно неприятно при обучении программированию.

3. C++. Преобразование типов. Касты

Простое явное преобразование типов в языке C (требуемый тип пишется в скобках перед преобразуемым выражением) заменено в языке C++ на различные типы преобразований, предназначенные для использования в различных специфических ситуациях. Преобразования типов в C++ называются *кастами* (*casts*). Соответственно, старое C-образное преобразование типов получило название *C-style-cast*. Его можно использовать в языке C++ (по крайней мере, пока), но это считается крайне плохим тоном с соответствующими выводами на собеседованиях при его использовании испытуемым. Если забыть про наследование, то можно говорить о трех видах кастов.

`static_cast` используется в случаях, когда программе требуется выполнить конкретный код для осуществления преобразования между различными типами данных (целыми, вещественными и т. д.), кроме случаев, относящихся к «константности» переменной. Например, если требуется преобразовать вещественную переменную к целой, это можно сделать следующим образом:

```
1 {int x; float y=1;
2 x=static_cast<int>(y); cout<<"1="<<x<<endl;}
```

`reinterpret_cast` применяется в случаях, когда требуется иная интерпретация данных, нежели предполагаемая в соответствии с используемыми типами, в частности, когда мы хотим использовать указатель на один тип как указатель на другой тип. Например, если мы хотим распечатать значения байтов целой переменной, то можно взять ее адрес, преобразовать его к указателю на `unsigned char` и использовать получившийся указатель как адрес начала массива переменных типа `unsigned char`:

```
1 {unsigned char *s; int x=1024;
2 s=reinterpret_cast<unsigned char*>(&x);
3 cout<<static_cast<int>(s[0])<<" "<<
4 static_cast<int>(s[1])<<" "<<static_cast<int>(s[2])<<" "<<
5 static_cast<int>(s[3])<<" "<<endl;
6 }
```

`const_cast` используется для изменения константности выражения (если так можно выразиться) и для изменения атрибута `volatile`.

В подавляющем большинстве случаев использование этого преобразования связано с неправильным проектированием программы и может приводить к ошибкам. Приведем корректный пример данного преобразования:

```
1 {int *p; const int x=1;
2   p=const_cast<int*>(&x); cout<<"1="<<*p<<endl;
3 }
```

В следующем примере использование данного преобразования синтаксически правильно, но не имеет никакого логического смысла и приведет к непредсказуемым последствиям:

```
1 void f(const int &x)
2 {const_cast<int&>(x)=0;}
3 int main(void)
4 {const int x=1; f(x); cout<<"x="<<x<<endl; return 0;}
```

Следует иметь в виду, что преобразование `reinterpret_cast` не может менять константность выражения и при необходимости приходится применять двойное преобразование:

```
1 {unsigned char *s; const int x=1;
2   s=reinterpret_cast<unsigned char*>(const_cast<int*>(&x));
3   cout<<static_cast<int>(s[0])<<" "<<
4     static_cast<int>(s[1])<<" "<<static_cast<int>(s[2])<<" "<<
5     static_cast<int>(s[3])<<" "<<endl;
6 }
```

Отдельного обсуждения требует атрибут `volatile`, который может применяться при определении/описании переменной. Чисто формально этот атрибут запрещает выполнять какую-либо оптимизацию работы с этой переменной. Приведем пример. Будем для определенности использовать 32-битный компилятор `g++` с оптимизацией первого уровня, т. е. компиляция программы будет происходить с помощью команды

```
1 g++ -O1 -m32 q.cpp
```

Рассмотрим следующий код:

```
1 #include<iostream>
2 using namespace std;
3 void f()
4 {int y=1; cout<<"f:"<<&y<<endl;}
5 int main(void)
6 {int x=1; cout<<&x<<endl;
7  while(x)
8  {f(); cout<<">"<<endl; exit(0);}
9  return 0;}
```

На компьютере автора на экран вывелось

```
1 0x102fe8c
2 f:0x102fe5c
3 >
```

что говорит о том, что разность адресов локальных переменных x и y равна 48. Здесь надо иметь в виду, что переменные x и y базируются на стеке и разность их адресов связана с механизмом вызова функций и передачи параметров в функции, т. е. для данного типа компиляции разность адресов можно считать постоянной.

Комментирование вызова функции `exit(0);` приведет к бесконечному циклу в главной функции. Зная смещение между адресами переменных x и y , мы можем попробовать изменить значение переменной x из функции `f()`:

```
1 #include<iostream>
2 using namespace std;
3 void f()
4 {int y=1; cout<<"f:"<<&y<<endl; (&y)[12]=0;}
5 int main(void)
6 {int x=1; cout<<&x<<endl;
7  while(x)
8  {
9    f(); cout<<">"<<endl; //exit(0);
10 }
11 return 0;
12 }
```

Однако цикл останется вечным, поскольку при оптимизации программы компилятор не видит возможности изменения значения переменной `x` в функции `main()` и просто выбрасывает проверку ее значения в цикле. Изменение значения переменной `y` также выбрасывается из программы, так как оно, с точки зрения компилятора, ни на что не влияет.

Однако мы можем запретить компилятору оптимизировать работу с переменными `x` и `y`:

```

1 #include<iostream>
2 using namespace std;
3 void f()
4 { volatile int y=1; cout<<"f:"<<const_cast<int*>(&y)<<endl;
5   (&y)[12]=0;
6 }
7 int main(void)
8 { volatile int x=1; cout<<const_cast<int*>(&x)<<endl;
9   while(x)
10    { f(); cout<<">"<<endl;}
11   return 0;
12 }
```

И тогда, в соответствии с нашей задумкой, цикл выполнится всего один раз (компилировали компилятором `gcc` без максимальной оптимизации с ключом `-m32`). В данной ситуации преобразование адресов переменных к типу `int*` нужно из-за особенностей компилятора: он отказывается выводить на экран адрес `volatile`-переменной (оператор вывода определен как пустой?), но согласен выводить адрес обычной переменной.

Для компилятора Microsoft, вызываемого без оптимизации:

```
1 cc q.cpp
```

разность адресов локальных переменных `x` и `y` оказывается равной 12. Тогда соответствующее изменение кода

```
1 (&y)[3]=0;
```

также приводит к изменению значение переменной `x`, и цикл прекращает свою работу.

4. C++. Сложные классы. Правило пяти. lvalue- и rvalue-ссылки

По большому счету, красивый и логичный язык C++ заканчивается вместе с завершением предыдущей главы. Ужасающая неэффективность языка, отраженная выше, приводит к необходимости создания многочисленных «костылей». К этим костылям можно относиться по-разному. Можно даже восхищаться их оригинальностью и красотой. Но сути это не меняет. Практика показывает, что, к сожалению, любое развитие объектно-ориентированного подхода рано или поздно приводит к существенным неприятностям (не будем говорить, что в тупик). В результате приходится добавлять в язык разнообразные новые конструкции, которые до поры до времени спасают ситуацию. Именно знание и понимание этих костылей определяет уровень программиста на языке C++. Существенная часть этой книги посвящена описанию данных черт языка.

4.1. Правило пяти

Как было показано ранее, стандартное поведение языка C++ при интерпретации выражения $a=b+c$ предполагает

- 1) создание локальной переменной, куда складывается сумма объектов (чего нельзя избежать),

а далее

- 2) создание временной переменной, куда с помощью отведения памяти помещается локальная переменная,

- 3) присваивание временной переменной a тоже реализуется с отведением памяти.

Использование отведения памяти на втором и третьем шагах, по логике, совершенно не является необходимым, так как на этих шагах отведенные ранее ресурсы (память) далее оказываются ненужными и можно было бы просто перенести эти ресурсы (т. е. просто присвоить соответствующие указатели) с предыдущего шага на следующий и обнулить ссылки на ресурсы на предыдущем шаге (обнулить указатели предыдущего шага). Для этой цели в языке C++ используются move-операции: move-конструкторы и move-присваивание. На этом этапе (в дальнейшем, при разборе lvalue- и rvalue-ссылок, эти операции будут описаны более формально) можно сформулировать ос-

нову данного подхода следующим образом: если компилятор увидит, что при присваивании $a=b$; или конструировании объекта с помощью конструктора копирования $T a(b)$; объект далее не нужен, то вместо обычного присваивания/копирования будет вызвана соответствующая move-операция. Определение соответствующих move-операций имеет обычно следующий синтаксис:

```

1 Vector ( Vector&&b) { MoveOnly ( b) ; }
2 Vector &operator=( Vector&&b)
3 { if (&b != this) { Clean () ; MoveOnly ( b) ; } return *this ; }

```

Здесь функция `MoveOnly()` осуществляет содержательную часть действия. В нашем случае эта функция выглядит следующим образом:

```

1 void MoveOnly ( Vector&b)
2 { if ( b.nmax ) { v=b.v ; n=b.n ; nmax=b.nmax ; b.SetZero () ; }
3   else SetZero () ; }

```

Заметим, что move-операции у нас определены способом, не зависящим от решаемой задачи. Специфической является только функция `MoveOnly()`.

Правило пяти при работе со сложным классом состоит в требовании определения необходимых методов из правила трех и двух описанных move-операций: move-конструктора и оператора move-присваивания.

Окончательный код выглядит следующим образом (сравните с кодом на с. 25–27):

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <cstdlib>
5 #include <cstring>
6 using namespace std;
7 //-----
8 void out ( const char*s ) { cout << s << endl ; }
9 template < class T > void out ( const char*s , const T&x ,
10                               const char*s2 ) { cout << s << x << s2 << endl ; }
11 //-----
12 template < class T > class Vector

```

```

13 {T *v; size_t n,nmax;
14 public:
15 //—
16 Vector(size_t n=0){out("Vector(",n,")");
17     if(n){v=new T[nmax=this->n=n];
18         for(size_t i=0;i<n;i++)v[i]=T();}
19     else SetZero();}
20 Vector(const Vector&b){out("Vector(const Vector&b)");
21                     CopyOnly(b);}
22 ~Vector(){out("~Vector()"); Clean();}
23 Vector &operator=(const Vector&b){
24     out("Vector &operator=(const Vector&b)");
25     if(&b!=this){Clean(); CopyOnly(b);} return *this;}
26 Vector(Vector&&b){out("Vector(Vector&&b)"); MoveOnly(b);}
27 Vector &operator=(Vector&&b){
28     out("Vector &operator=(Vector&&b)");
29     if(&b!=this){Clean(); MoveOnly(b);} return *this;}
30 void
31     MoveOnly(Vector&b){v=b.v;n=b.n;nmax=b.nmax;b.SetZero();}
32 //—
33 void SetZero(){v=nullptr;n=0;nmax=0;}
34 void Clean(){delete [] v; SetZero();}
35 void CopyOnly(const Vector&b){if(b.nmax
36 {v=new T[nmax=b.nmax]; n=b.n;
37     for(size_t i=0;i<nmax;i++)v[i]=b.v[i];} else SetZero();}
38 //—
39 size_t size()const{return n;}
40 struct SErr{string s;SErr(const
41     char*s="error"){this->s=s;}};
42 T &operator[](size_t i)
43     {if(i>=n)throw SErr("bad index"); return v[i];}
44 const T &operator[](size_t i)const
45     {if(i>=n)throw SErr("bad index"); return v[i];}
46 Vector operator+(const Vector&b)const{Vector r(n);
47     for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
48 void push_back(const T&x){if(n==nmax){T*w=new T[2*(nmax+1)];
49     size_t i=0; for(;i<n;i++){w[i]=v[i];}
50     for(;i<2*(nmax+1);i++)w[i]=T();
51     nmax=2*(nmax+1); delete [] v; v=w;} v[n++]=x;}

```

```

50 };
51 //—
52 template<class T>ostream &operator<<
53 (ostream &cout, const Vector<T> &v)
54 {cout<<" ("; for (size_t i=0;i<v.size();i++)
55   {cout<<v[i]<<(i<v.size()-1?" ":"");} cout<<"");return cout;}
56 //—
57 int main(void)
58 {Vector<int> v; int m[5]={0,1,2,3,4};
59   for (int i=0;i<5;i++)v.push_back(m[i]);
60   v=v+v; cout<<"v+v="<<v<<endl; return 0;
61 }

```

Стандартное поведение языка приведет к следующему выводу на экран:

```

1 Vector (0)
2 Vector (5)
3 Vector (Vector&&b)
4 ~Vector ()
5 Vector &operator=(Vector&&b)
6 ~Vector ()
7 v+v=(0 2 4 6 8)
8 ~Vector ()

```

Нестандартное поведение языка дает более короткую последовательность вызываемых функций:

```

1 Vector (0)
2 Vector (5)
3 Vector &operator=(Vector&&b)
4 ~Vector ()
5 v+v=(0 2 4 6 8)
6 ~Vector ()

```

В обоих случаях мы получаем лишь одно отведение памяти под переменную, используемую для хранения промежуточных данных, и именно эта отведенная память в конечном счете окажется в переменной, стоящей слева от знака присваивания. Теоретически при реализации операции $v=v+v$; на языке С мы смогли бы написать функцию, вообще не отводящую память (реализация данного действия

на C++ физически не может достичь данной оптимизации), но если бы мы строили реализацию выражения $w=v+v$; и в переменной w изначально не было бы отведенной памяти, то память под результат все равно пришлось бы отводить. Для последнего случая мы получили оптимальную реализацию данного действия. Накладные расходы, безусловно, есть, но они минимальны (время на их выполнение равно $O(1)$).

4.2. lvalue- и rvalue-ссылки

Усложним наш пример, заменив сложение двух переменных на сложение большего количества объектов:

```

1  int main(void)
2  { Vector<int> v; int m[5]={0,1,2,3,4};
3    for (int i=0; i<5; i++)v.push_back(m[i]);
4    v=v+v+v+v; cout<<"v+v+v+v="<<v<<endl;
5    return 0;
6  }
```

Для лучшего понимания ситуации добавим в оператор сложения вызов функции `out("+");` При стандартном поведении языка получим следующий вывод на экран:

```

1  Vector (0)
2  +
3  Vector (5)
4  Vector (Vector&&b)
5  ~Vector ()
6  +
7  Vector (5)
8  Vector (Vector&&b)
9  ~Vector ()
10 +
11 Vector (5)
12 Vector (Vector&&b)
13 ~Vector ()
14 Vector &operator=(Vector&&b)
15 ~Vector ()
16 ~Vector ()
```

```

17 ~Vector ()
18 v+v+v+v=(0 4 8 12 16)
19 ~Vector ()

```

При нестандартном поведении языка вывод будет иметь следующий вид:

```

1 Vector (0)
2 +
3 Vector (5)
4 +
5 Vector (5)
6 +
7 Vector (5)
8 Vector &operator=(Vector&&b)
9 ~Vector ()
10 ~Vector ()
11 ~Vector ()
12 v+v+v+v=(0 4 8 12 16)
13 ~Vector ()

```

Как мы видим, в обоих случаях создаются три временные переменные, в которых хранятся промежуточные результаты трех операций сложения. Если бы мы реализовывали действие $a=b+c+d+e$; на языке C, то смогли бы обойтись всего одной временной переменной, сведя данное действие к следующей последовательности операций:

```

1 tmp=b+c; tmp+=d; tmp+=e; a=tmp;

```

Для подобных мероприятий в языке C++ существует механизм *rvalue*- и *lvalue*-ссылок. Понятия *rvalue* и *lvalue* присутствовали уже в языке C, но там они были тривиальными: под *lvalue* (*left value*) подразумевалось то, что может стоять слева от знака присваивания, а под *rvalue* (*right value*) — то, что может стоять только справа от знака присваивания. В языке C++ эти понятия несколько изменились. Чисто формально под *rvalue* здесь подразумеваются либо литералы (например, 5 или 5.), либо временные переменные, создаваемые компилятором (про них мы много говорили ранее), либо именованные объекты (объекты вида T()). Все остальное считается *lvalue*. Например, в определенном выше операторе сложения аргумент является *lvalue* (хотя он и имеет модификатор **const**) и сам объект класса

также является lvalue. Но при обработке выражения $a=b+c+d+e$; результат $b+c$ помещается в gvalue (во временную переменную), что дает компилятору возможность отличить эту переменную от видимых нам переменных, а это, в свою очередь, дает нам возможность написать другой оператор сложения для случая, когда левый операнд сложения является gvalue. Напомним, что операция $b+c$ имеет дело с двумя lvalue.

Чисто формально (синтаксически) один знак `&` говорит о том, что данная ссылка (в нашем случае — аргумент функции) является lvalue-ссылкой, а два знака `&` свидетельствуют о том, что данная ссылка является gvalue-ссылкой. В случае, когда левый операнд является gvalue-ссылкой, а правый — lvalue-ссылкой, используется следующий синтаксис для определения оператора сложения:

```

1 Vector &&operator+(const Vector&b)&&{out("&&+");
2   for(size_t i=0;i<n;i++){v[i]=v[i]+b.v[i];}
3   return static_cast<Vector&&>(*this);}
```

Содержимое данного метода фактически повторяет содержимое оператора $+=$, но здесь определяется именно оператор $+$ с особой сигнатурой: в качестве первого операнда (неявного, `*this`) принимается gvalue-ссылка (`&&`). Для уменьшения количества копирований и выделений памяти вместо создания нового объекта для результата используется (изменяется и возвращается как gvalue-ссылка) `*this`.

Отметим, что (согласно логике использования данного оператора) gvalue-ссылка используется как раз для того, чтобы значение по данной ссылке можно было бы изменять (что вступает в противоречие с тем, что gvalue как раз изменять нельзя!). Чисто формально gvalue-ссылкой является фактический аргумент функции, но как только мы переходим к формальному параметру функции, получаемое функцией значение уже изменять можно, и данная переменная превращается в lvalue (хотя параметр и описан как gvalue-ссылка; шизофреничность последнего абзаца подтверждает осмысленность использованного выше понятие «костыли»). Понятно, что компилятору приходится проявлять изрядную смекалку, чтобы разрешить это противоречие. Например, если в качестве параметра выступает gvalue-ссылка на целую переменную, то в функцию можно передавать число 5, но внутри функции, согласно вышеприведенной логике, это значение можно изменять. Таким образом, компилятор должен

озаботиться тем, чтобы в функцию в реальности передавалась обычная ячейка памяти с помещенным туда целочисленным литералом 5. Напомним, что, вообще говоря, если в программе в некотором контексте встречается литерал (например, 5), то это не значит, что он будет храниться в какой-то конкретной ячейке памяти. Например, выражение `x=5;`, скорее всего, преобразуется в команду ассемблера `mov`, в которой первый операнд является адресом переменной, куда надо поместить результат, а второй — именно данной константой.

Последний оператор сложения будет возвращать rvalue-ссылку на `*this` (для ее дальнейшего использования), но поскольку `*this` уже успело превратиться в lvalue, то в операторе `return` необходим `static_cast` для преобразования lvalue в rvalue.

Заметим, что теперь надо подправить старый оператор сложения, чтобы уточнить, что в нем левый аргумент сложения — lvalue-ссылка (ранее это было не принципиально):

```
1 Vector operator+(const Vector&b) const&{out("+"); Vector r(n);
2   for (size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
```

В результате в случае стандартного поведения языка C++ получим следующий вывод на экран:

```
1 Vector (0)
2 +
3 Vector (5)
4 Vector (Vector&&b)
5 ~Vector ()
6 &&+
7 &&+
8 Vector &operator=(Vector&&b)
9 ~Vector ()
10 v+v+v+v=(0 4 8 12 16)
11 ~~Vector ()
```

В случае нестандартного поведения языка C++ вывод на экран будет иметь вид:

```
1 Vector (0)
2 +
3 Vector (5)
```

```

4  &&+
5  &&+
6  Vector &&operator=(Vector&&b)
7  ~Vector ()
8  v+v+v+v=(0 4 8 12 16)
9  ~Vector ()

```

Таким образом, мы получили всего одно отведение памяти в первом операторе сложения, что оптимально даже для языка C.

Для простоты будем далее рассматривать только стандартное поведение языка C++ в более сложном случае сложения четырех слагаемых:

```

1  v=(v+v)+(v+v);

```

Здесь мы имеем попытку сложения двух rvalue, что требует соответствующего оператора сложения. А в случае

```

1  v=v+(v+v+v);

```

нам необходимо складывать lvalue и rvalue. Таким образом, нам надо дописать еще два оператора сложения: первый — для сложения rvalue и rvalue, второй — для сложения lvalue и rvalue.

```

1  Vector &&operator+(Vector&&b)&&{out("&&+&&");
2  for(size_t i=0;i<n;i++){v[i]=v[i]+b.v[i];}
3  return static_cast<Vector&&>(*this);}
4  Vector &&operator+(Vector&&b) const&&{out("&&");
5  for(size_t i=0;i<n;i++){b.v[i]=v[i]+b.v[i];}
6  return static_cast<Vector&&>(b);}

```

Для случая интерпретации выражения `v=(v+v)+(v+v)`; при стандартном поведении языка C++ мы получим следующий вывод на экран:

```

1  Vector (0)
2  +
3  Vector (5)
4  Vector (Vector&&b)
5  ~Vector ()
6  +
7  Vector (5)

```

```

8 Vector (Vector&&b)
9 ~Vector ()
10 &&+&&
11 Vector &operator=(Vector&&b)
12 ~Vector ()
13 ~Vector ()
14 v+v+v+v=(0 4 8 12 16)
15 ~Vector ()

```

Легко увидеть, что отведение памяти под две временные переменные здесь неизбежно. Можно показать, что даже на языке С нельзя выполнить эту операцию, не создавая двух дополнительных объектов.

Весьма полезной является возможность использования ссылок на временные объекты, возвращаемые функциями:

```

1 {Vector<double> v; int m[]={0,1,2,3,4,5,6,7,8,9};
2 for (int i=0;i<10;i++)v.push_back(m[i]);
3 Vector<int> &&w=(v+v);
4 cout<<"v="<<v<<endl<<"v+v="<<w<<endl;
5 }

```

Здесь переменная *w* является rvalue-ссылкой на временную переменную, которая создана для результата, возвращаемого оператором сложения. Наличие такой ссылки продлевает время жизни переменной, на которую она ссылается, поэтому данной переменной можно пользоваться после выполнения оператора сложения. Однако использовать подобный трюк надо с осторожностью (скорее, с пониманием того, что мы делаем).

Рассмотрим простой пример (визуально почти не отличающийся от предыдущего):

```

1 {Vector<double> v; int m[]={0,1,2,3,4,5,6,7,8,9};
2 for (int i=0;i<10;i++)v.push_back(m[i]);
3 Vector<double> &&w=v+v+v;
4 cout<<"v+v+v="<<w<<endl;
5 }

```

Результат может нас удивить. Мы не увидим на экране суммы векторов. Подправим немного деструктор, чтобы он выводил на экран также текущее количество элементов в векторе:

```
1 ~Vector() {out("~Vector(",n,")"); Clean();}
```

Тогда отладочный вывод в случае стандартного поведения языка C++ будет иметь вид:

```
1 Vector(0)
2 +
3 Vector(10)
4 &&+
5 ~Vector(10)
6 v+v+v=( )
7 ~Vector(10)
```

С легкой грустью мы замечаем, что деструктор для временной переменной оказывается вызванным сразу после выполнения оператора сложения. Тогда на экран выводится значение формально уже не существующей переменной (фактически ее место в стеке никто испортить не успел, поэтому мы получили вывод, соответствующий состоянию данной переменной сразу после ее гибели). Это происходит и в случае использования компилятора gcc, и в случае компилятора Microsoft. Однако если разобраться, то компилятор винить не в чем. В нашем случае происходит присваивание переменной `w` значения, возвращаемого вторым оператором сложения. А это значение, в свою очередь, является `rvalue`-ссылкой на временную переменную, возвращаемую первым оператором сложения. Компилятор не может разобраться с тем, ссылку на какую именно переменную возвращает оператор сложения (быть может, когда-то в будущем он это и сможет, но не сейчас), поэтому время жизни временной переменной не продлевается, и мы получаем то, что получаем.

5. Внутренние имена глобальных объектов

Как мы уже давно знаем, процесс создания исполняемого файла из файла с исходной программой (например, на языке C/C++) состоит из двух основных этапов: компиляции и сборки. В реальности каждый из этих этапов может состоять из подэтапов (например, при компиляции с языка C сначала программу обрабатывает препроцессор, а потом уже запускается собственно компилятор), но сейчас нам это не важно. Программа, осуществляющая процесс компиляции, называется *компилятором*. Компилятор из каждого файла с исходной программой создает *объектный файл*. Объектные файлы собираются вместе для создания *исполняемого файла* с помощью программы, называемой *линковщиком* или *сборщиком*.

В реальности весь процесс создания программы может идти немного по другому пути. Например, Microsoft активно продвигает технологию *Edit and Continue*, позволяющую перекомпилировать и пересобирать проект в процессе его выполнения: приостанавливаем программу, меняем исходный код, среда разработки перекомпилирует и пересобирает программу, продолжаем исполнение, и все это на лету. Или, например, все тот же компилятор Microsoft позволяет осуществлять общую оптимизацию кода на этапе сборки, т. е. существенная часть процесса компиляции практически осуществляется не на этапе компиляции, а на этапе сборки. Это, безусловно, существенно замедляет процесс сборки проекта (проект, создаваемый автором текста, размером примерно в пару миллионов строк на языке C++ собирается на топовом компьютере 4–5 минут), но общая оптимизация кода оправдывает эти затраты. Однако мы не будем останавливаться на таких отклонениях от стандартного процесса создания исполняемого файла. Разберемся с тем, что делают компилятор и сборщик в классическом случае.

В простейшем случае компилятор может почти полностью создать исполняемый код и поместить его в объектный файл. В каждый момент исполнения программы компилятор знает, как вычислить адреса всех локальных переменных по значению указателя на вершину стека, хранящемуся в соответствующем регистре (регистрах). Таким образом, зная адреса всех локальных переменных, компилятор может выписать весь исполняемый код (в машинных командах), который с ними оперирует.

Рассмотрим простейший пример:

```
1  int a,b,c;
2  b=1;
3  c=2;
4  a=b+c;
```

При компиляции без оптимизации программы с данным кодом компилятор Microsoft создает следующий ассемблерный код:

```
1  _c$ = -12           ; size = 4
2  _b$ = -8           ; size = 4
3  _a$ = -4           ; size = 4
4  ; 5      : {int a,b,c;
5      mov ebp, esp
6      sub esp, 12
7  ; 6      : b=1;
8      mov DWORD PTR _b$[ebp], 1
9  ; 7      : c=2;
10     mov DWORD PTR _c$[ebp], 2
11 ; 8      : a=b+c;
12     mov eax, DWORD PTR _b$[ebp]
13     add eax, DWORD PTR _c$[ebp]
14     mov DWORD PTR _a$[ebp], eax
```

Здесь точка с запятой является символом комментария. Первые три строки задают константы, которые используются в дальнейшем ассемблером в соответствующих операциях (они задают смещения от вершины стека к адресам трех целых переменных), т. е. этим строкам не соответствует никаких машинных операций в коде. Остальные команды задают соответствующие машинные команды в исполняемом файле.

Команда `mov` копирует второй аргумент в первый, команда `sub` вычитает из первого аргумента второй, команда `add` — прибавляет.

Регистр `esp` указывает на вершину стека. Команда `mov ebp, esp` сохраняет в регистр `ebp` значение регистра `esp`. Вычитание 12 из значения регистра `esp` фактически является отведением памяти в стеке под три целые четырехбайтовые переменные (на случай, если внутри текущего блока появятся еще блоки, а в них потребуется отводить память под свои локальные переменные).

Восьмая команда копирует четырехбайтовую (DWORD значит *double word*, а под *словом* имеется в виду пара байт) единичку по адресу, взятому из регистра `ebx` минус 8 (константа перед `[ebx]` задает требуемое смещение). Аналогично двойка копируется по адресу, взятому из регистра `ebx` минус 12.

Далее двумя командами в регистре `eax` образуется сумма значений созданных переменных, после чего значение данного регистра записывается по адресу, взятому из регистра `ebx` минус 4, что и будет значением переменной `a`.

Из данного примера следует, что весь требуемый в данном случае машинный код будет полностью записан в объектный файл.

Легко понять, что единственное, с чем компилятор не сможет разобраться на этапе создания объектного файла — это подстановка адресов статических объектов (глобальных или локальных в файле). Под *статическим объектом* подразумевается статическая переменная или функция. Если бы какая-то из используемых в примере переменных была статической, то компилятор бы просто не знал, откуда брать ее адрес. Все, что ему остается — это вместо адреса статической переменной подставить в объектный файл (не важно, каким образом) ее внутреннее имя (имя, которое используется внутри объектного файла). При этом у компилятора должны быть четкие правила создания внутренних имен глобальных объектов. Сборщик при обработке всех объектных файлов создаст таблицу имен глобальных объектов, проверит, что каждый глобальный объект создается ровно один раз и далее заменит в машинном коде все одинаковые имена глобальных объектов на один и тот же выделенный им адрес.

Из всего написанного выше вытекает идея реализации полиморфизма при работе с глобальными (статическими) переменными, функциями, шаблонами функций, переменными, созданными при помощи шаблонов классов. Не вдаваясь в тонкости, можно сказать, что внутренние имена объектов, допускающих одинаковые видимые (внешние) имена, должны состоять из внешних имен с дополнительными символами, конкретизирующими имя данного объекта. Например, для имен функций к внешнему имени функции должны добавляться символы, соответствующие типам параметров функции, для функции, являющейся членом класса, к внутреннему имени функции также должно добавляться имя класса и так далее. Далее разбе-

рем подробнее алгоритмы создания внутренних имен для отдельных типов объектов.

5.1. Внутренние имена переменных и функций

Разберем простой пример, включающий в себя глобальную переменную и функцию:

```
1 #include<stdio.h>
2 struct SMyStruct{int My__Value;}My__Struct;
3 void My__Fun()
4 {printf("%d\n",My__Struct.My__Value);}
5 int main(void)
6 {My__Fun(); return 0;}
```

Данный пример можно компилировать и как программу на языке C, и как программу на языке C++. Имена объектов в данной программе начинаются на `My__`, чтобы их можно было легко искать в большом списке глобальных объектов, который у нас появится позднее.

Сначала поместим пример в файл `q.c` и скомпилируем его компилятором `gcc`:

```
1 gcc q.c
```

К получившемуся исполняемому файлу `a.exe` применим команду `nm`, выводящую в стандартный поток вывода (по умолчанию на экран) список имен глобальных объектов. Стандартный поток вывода перенаправим в файл `nm.txt`, чтобы было проще искать нужные нам имена глобальных объектов:

```
1 nm a.exe >nm.txt
```

Поискав в получившемся тексте слово `My__`, мы обнаружим, что есть всего два таких глобальных объекта:

```
1 0000000000401520 T My__Fun
2 0000000000407030 B My__Struct
```

Так мы узнали, что внутренние имена глобальных объектов (переменных и функций) для компилятора `gcc` на языке C совпадают с исходными именами объектов.

С помощью ключа /FAS

```
1 cc /FAS q.c
```

можно попросить компилятор Microsoft параллельно компиляции создать ассемблерный файл, в котором также будут присутствовать имена глобальных объектов. В файле `q.asm` найдутся следующие указания на имена глобальных объектов:

```
1 COMM    _My__Struct :DWORD
2 PUBLIC  _My__Fun
```

Значит, компилятор Microsoft для языка С создает внутренние имена глобальных объектов, приписывая к исходным именам спереди символ подчеркивания. Мы не получили ничего неожиданного. При таких правилах создания внутренних имен полиморфизм невозможен.

Тот же самый текст поместим в файл `q.cpp`. Тогда после компиляции/сборки программы компилятором `gsc` в списке глобальных объектов исполняемого файла мы найдем имена

```
1 0000000000401520 T _Z7My__Funv
2 0000000000407030 B My__Struct
```

Значит, внутренние имена статических переменных образуются тем же способом, что и в языке С (понятие *поллиморфизма* для обычных переменных в языке С++ не имеет смысла, но это понятие осмысленно в том плане, что возможны одинаковые имена элементов в различных классах), а вот к внутреннему имени функции дописываются буквы, указывающие на тип параметров функции.

В ассемблерном файле компилятора Microsoft после компиляции того же файла `q.cpp` обнаружатся имена

```
1 PUBLIC ?My__Struct@@3USMyStruct@@A ; My__Struct
2 PUBLIC ?My__Fun@@YAXXZ ; My__Fun
```

Здесь все оказывается интереснее: к имени глобальной переменной приписывается имя типа(!). Значит, можно ожидать, что компилятор Microsoft дает недокументированную возможность создавать в разных файлах разные глобальные переменные с одинаковыми именами, но разными типами (смысла в этом, с точки зрения языка С++,

нет, но это может служить причиной весьма неожиданных ошибок в программе). Далее мы покажем, что при использовании компилятора Microsoft можно иметь разные функции с одинаковыми именами и типами параметров, но различными типами возвращаемых значений (о такой возможности надо знать скорее для понимания возможных ошибок; язык C++ сам по себе такого не допускает, и если компилятор обнаружит такую экзотику в каком-то исходном файле, то предсказуемо выдаст ошибку).

Рассмотрим пример. Программа состоит из двух исходных файлов. Первый файл q.cpp:

```

1 #include<stdio>
2 int MyVar=0;
3 void fun1(void);
4 int fun2(void){ printf("%d\n",MyVar); return 0;}
5 int main(void)
6 {fun1();
7  fun2();
8  return 0;
9 }
```

Второй файл q2.cpp:

```

1 #include<stdio>
2 float MyVar=1;
3 void fun1(void){ printf("%g\n",MyVar);}
```

После компиляции первого файла компилятором Microsoft `cc -c q.cpp /FAS` найдем в файле `q.asm` имена глобальных объектов:

```

1 PUBLIC ?fun2@@YAHXZ ; fun2
2 PUBLIC ?MyVar@@@3HA ; MyVar
```

Аналогично найдем имена глобальных объектов в файле `q2.asm`:

```

1 PUBLIC ?fun1@@YAXXZ ; fun1
2 PUBLIC ?MyVar@@@3MA ; MyVar
```

Надо обратить внимание на то, что дополнительные символы во внутренних именах функций `fun1` и `fun2` различны, что (как упоминалось ранее) говорит о том, что признак типа переменной, возвраща-

емой функцией, в данном случае также включается во внутреннее имя объекта.

Программа успешно скомпилируется, соберется и отработает с выводом на экран

```
1
2
```

```
1
2
```

Компилятор `g++` запрещает такие вольности. Тип возвращаемого из функции значения и тип глобальной переменной не добавляются к внутренним именам соответствующих объектов, поэтому при попытке собрать данную программу линковщик `g++` выдаст ошибки из-за повторяющихся имен глобальных объектов и для переменной, и для функции.

5.2. Шаблоны. Определение функций шаблона в отдельном `сpp`-файле

Разберемся с правилами образования внутренних имен функций и объектов, задаваемых шаблонами. Надо понимать, что сам по себе шаблон функции не определяет никакой функции. Собственно функция определяется компилятором в момент ее использования. Рассмотрим простой пример:

```
1 #include<iostream>
2 using namespace std;
3 template<class T> T Min(const T&a ,const T&b){return a<b?a:b;}
4 template<class T> T Max(const T&a ,const T&b){return a>b?a:b;}
5 int main(void)
6 {cout<<Min(1,2)<<" "<<Max(1,2)<<endl; //это эквивалентно
7   cout<<Min<int>(1,2)<<" "<<Max<int>(1,2)<<endl;
8   return 0;
9 }
```

В момент использования шаблонов функций `Min` и `Max` компилятором в объектном файле, получаемом из данного исходного файла, создаются (определяются) соответствующие функции, во внутреннее имя которых добавляются символы, задающие типы параметров функции. В данном примере определяются функции с целочисленными параметрами, возвращающие целое значение. Если аналогичный

код появится в другом исходном файле программы, то и там при компиляции будут определены соответствующие аналогичные функции. При сборке линковщик получит множественное определение одного и того же имени глобального объекта, но делать нечего, ему придется смириться с этим. Он будет надеяться, что программист определил шаблон в едином include-файле, поэтому определения данных функций будут везде аналогичны. А на деле в качестве определения функции он будет использовать первое встретившееся определение функции с данным именем.

Рассмотрим функцию — член класса на простом примере:

```

1  template<class T,int N>struct STest
2  {
3    T vTest [N];
4    void funTest () {vTest [0]=T ();}
5  };
6  STest<int ,1024> xTest ;
7  int main(void)
8  {xTest . funTest ();
9    return 0;
10 }
```

Здесь параметрами шаблона являются тип переменной и целочисленное значение.

Для компилятора gcc мы получим следующую информацию о внутреннем имени функции:

```
1  _ZN5STestIiLi1024EE7funTestEv
```

Имя типа структуры (STest) и текстовое представление целой переменной 1024 из параметра шаблона включаются во внутреннее имя глобального объекта (функции) funTest. Также во внутреннее имя функции включаются буквы, соответствующие типу T, передаваемому в шаблон при определении переменной xTest. В данном случае это буква i после буквы I (соответствует типу **int**). Например, если бы мы параметризовали шаблон типом **float**, то внутреннее имя функции было бы следующим:

```
1  _ZN5STestIfLi1024EE7funTestEv
```

Внутреннее имя переменной `xTest` для компилятора `gcc` совпадает с внешним.

Таким образом, для различных параметров шаблона мы получим различные функции с различными внутренними именами.

Для компилятора `Microsoft` все немного более запутанно. Внутренние имена интересующих нас переменной и функции имеют вид:

```

1 PUBLIC ?xTest@@@3U?$STest@H$0EAA@@@A          ; xTest
2 PUBLIC ?funTest@?$STest@H$0EAA@@@QAEXXZ      ;
   STest<int ,1024 >::funTest

```

Здесь число `N` представлено во внутреннем имени в некоторой системе счисления, которую чисто визуально сложно понять.

Рассмотрим другой простой пример файла с программой (пусть его имя будет `q.cpp`):

```

1 #include<iostream>
2 using namespace std;
3 #include"q"
4 int main(void)
5 {S<int ,10> y; y.fun ();
6  return 0;
7 }

```

include-файл с именем `q` имеет вид:

```

1 #pragma once
2 template<class T,int N> struct S
3 {
4  T x;
5  void fun () {this->x=N; cout<<"x="<<x<<endl;}
6 };

```

Программа из одного файла `q.cpp` будет замечательно компилироваться, запускаться и выдавать на экран число `10`.

Поставим своей целью вынести определение функции из описания класса. Данное желание вполне разумно в случае, когда определение функции занимает много места и при просмотре include-файла мешает сразу увидеть описания всех переменных и методов класса.

Самым простым способом решения данной проблемы является вынесение шаблона функции из шаблона структуры в том же самом include-файле:

```
1 #pragma once
2 template<class T,int N> struct S
3 {T x;void fun();};
4 template<class T,int N> void S<T,N>::fun()
5 {this->x=N; cout<<"x="<<x<<endl;}
```

Здесь мы не определили никакой функции (только шаблон!), но в файле q.cpp имеется вызов данной функции для конкретного экземпляра класса x, что заставляет компилятор определить функцию fun с параметрами шаблона int и 10. Данный подход будет успешно работать.

Следующим шагом будет попытка полностью вынести определение функции fun() из include-файла. Так стоит поступать, когда в include-файле содержится описание многих классов и лишний текст усложняет обзор возможностей этих классов. Вынесение определения шаблона функции в отдельный cpp-файл возможно, но, к сожалению, приводит к определенным неудобствам. Действительно, мы легко можем вынести определение шаблона функции класса в cpp-файл (пусть это будет q2.cpp) ровно в том виде, как оно написано в include-файле. Однако в этом случае при вызове соответствующей функции x.fun(); компилятор не будет иметь информации о теле функции и никакой функции с соответствующим внутренним именем определено не будет. В файле q2.cpp также не появится никакого определения функции, так как шаблон сам по себе не иницирует никаких определений функций. Тогда нам следует после определения шаблона указать компилятору, что необходимо определить функцию, соответствующую шаблону с требуемыми параметрами шаблона. Файл q2.cpp в этом случае должен иметь вид:

```
1 #include<iostream>
2 using namespace std;
3 #include"q"
4 template<class T,int N> void S<T,N>::fun()
5 {this->x=N; cout<<"x="<<x<<endl;}
```

```
6 template void S<int,10>::fun();
```

Последнюю инструкцию придется повторять для всех вариантов параметров шаблона, которые встречаются в программе, что, безусловно, крайне неудобно.

Если синтаксис последней инструкции сложно запомнить, то в качестве альтернативы можно рассмотреть простое создание технической переменной с типом заданного шаблона с требуемыми параметрами шаблона и вызовом требуемого метода для этой переменной. Это приведет к определению в программе требуемой функции, которую можно будет использовать в других файлах программы. В этом случае файл q2.cpp может иметь вид:

```

1 #include<iostream>
2 using namespace std;
3 #include"q"
4 template<class T,int N> void S<T,N>::fun()
5 {this->x=N; cout<<"x"<<x<<endl;}
6 int Def_S_Fun(void)//создаем переменную tmp для вызова
   функции:
7 {S<int,10> tmp; tmp.fun(); return 0;}
8 static int i_tmp=Def_S_Fun();

```

Следует признать, что данное решение является весьма некрасивым и в некоторых случаях неприемлемым (из-за побочных эффектов при вызове функции).

Безусловно, определение функций шаблона в cpp-файлах может приводить к ошибкам. В частности, если определять функции с одинаковыми внутренними именами и параметрами в разных файлах по-разному, то мы получим различное поведение программы в зависимости от порядка указания объектных файлов при сборке программы. Например, если в последнюю программу добавить файл q3.cpp вида

```

1 #include<iostream>
2 using namespace std;
3 #include"q"
4 template<class T,int N> void S<T,N>::fun() {this->x=N+1;
   cout<<"x"<<x<<endl;}
5 template void S<int,10>::fun();

```

то следующие два варианта компиляции и сборки приведут при запуске программы к различной выдаче на экран:

```
1 g++ q.cpp q2.cpp q3.cpp  
2 g++ q.cpp q3.cpp q2.cpp
```

Наиболее непротиворечивым видится решение, при котором определения всех шаблонов методов классов, определенных через шаблоны, выносятся в отдельный `include`-файл, который включается в файл с определением шаблона класса. Недостатком этого подхода является существенное увеличение размера объектных файлов и времени компиляции из-за того, что определения всех функций, задаваемых шаблоном, могут появиться во всех объектных файлах программы (если эти функции там вызываются).

6. C/C++. Собственные процедуры отладки утечек памяти. Умные указатели. Функциональные объекты

Данная глава посвящена работе с памятью в языке C++. Сначала разберемся с переопределением пользователем процедур отведения памяти, что может быть полезным при отладке приложений, когда недоступны специализированные средства отладки. Далее рассмотрим, что может предложить язык C++ в качестве компенсации отсутствия в нем аналога функции `realloc()`. Наконец, разберемся с умными указателями, которые призваны решить проблемы управления памятью в языке C++. Для понимания механизма работы умных указателей необходимо предварительно ознакомиться с понятием *функциональных объектов*.

6.1. Собственные процедуры отладки утечек памяти

В [11] мы уже обсуждали различные механизмы, позволяющие выявлять некорректные отведения и утечки памяти в языках C/C++. В ряде случаев эти механизмы оказываются недоступными. Например, при использовании компилятора `gcc` в ОС Windows не удается работать с отладчиком `gdb` (см. [13]) и программой отладки `valgrind`. И в этом случае мы вспоминаем известный тезис о том, в чьих руках находится спасение утопающих. Оказывается, довольно несложно написать свои процедуры для работы с памятью, заменяющие стандартные. Собственноручно написанные функции будут способны отследить утечки памяти и другие ошибки, связанные с отведением/очисткой памяти. В частности, в данные процедуры можно встроить проверку выхода за границы массивов: при отведении памяти можно автоматически выделять несколько дополнительных байтов после отведенного куска памяти, эти байты заполняются определенными значениями, и при очистке памяти программа проверяет, что эти байты не поменялись). Компилятор Microsoft дает стандартную возможность таких проверок в среде программирования Microsoft Visual Studio. Безусловно, приведенная далее реализация не претендует на эффективность. Быстрые алгоритмы отведения/очистки памяти требуют отдельного обсуждения. Более того, их реализация невозможна без доступа к механизмам, обеспечивающим

работу с виртуальной памятью. Но это уже тема другой беседы. Любый желающий сможет усовершенствовать предлагаемый вариант работы с памятью самостоятельно. Написанных далее функций будет вполне достаточно, чтобы, например, отслеживать корректность работы с памятью во всех программах, которые пишутся в рамках данного курса.

Разберем далее только случай работы с языком C++.

Начнем с создания include-файла `malloc.h`, содержимое которого желательно (не обязательно!) включать в заголовок исходных файлов отлаживаемой программы. Начало файла будет иметь следующий вид:

```
1 #pragma once
2 #include<cstdio>
3 #include<cstring>
4 #ifndef LOG
5 #define LOG 0
6 #else
7 #undef LOG
8 #define LOG 1
9 #endif
10 #ifndef OUT
11 #define OUT
12 inline void out(const char*s){ if (LOG) printf("%s ",s);}
13 template<class T>
14 void out(const char*s, const T&x, const char*s2)
15 { if (LOG) printf("%s%d%s ",s,static_cast<int>(x),s2);}
16 template<class T>
17 void out(const char*s, const T&x, const T&y, const char*s2)
18 { if (LOG) printf("%s%d,%d%s
19     ",s,static_cast<int>(x),static_cast<int>(y),s2);}
19 #endif
```

Далее будут идти описания функций, которые мы определим в файле `malloc.cpp`. В конце `malloc.h` потребуется записать еще пару строк:

```
1 #define new new(__FILE__,__LINE__)
2 #define malloc(n) malloc(n, __FILE__, __LINE__)
```

Объясним смысл этих строк позднее.

В заголовке файла обеспечивается правильное задание макроопределения LOG. Предполагается, что исходные файлы программы могут компилироваться либо обычным образом, либо с дополнительным ключом `-D LOG`. Первые строки `include`-файла приводят к тому, что если макроопределение при компиляции не задано, то задается макроопределение

```
1 #define LOG 0
```

Если же макроопределение при компиляции задано (ключ `-D LOG`), то в результате задается макроопределение

```
1 #define LOG 1
```

В случае если LOG равно 0, все функции отладочного вывода не будут делать ничего. Если LOG равно 1, то все функции отладочного вывода будут выдавать надлежащую отладочную информацию о вызове функций отведения/очистки памяти. Сами функции выдачи отладочной информации `out()` довольно просты, и легко понять, что они делают. В конце работы программы в любом случае будет произведена проверка отсутствия утечек памяти.

Мы будем определять функции или шаблоны функций `out(...)` при реализации различных структур данных и в дальнейшем. Во избежание их множественного определения здесь и далее используется макроопределение `OUT`.

Надо понимать, что работа процедуры проверки утечек памяти сильно зависит от системы. Приведенные далее процедуры будут корректно работать для текущих компиляторов Microsoft (последней версии) и `gcc` только при отсутствии оптимизации программы (т. е. если при компиляции не используются ключи `-O*`).

Для использования написанных процедур будет достаточно скомпилировать файл `malloc.cpp` и линковать получившийся объектный файл вместе с объектными файлами собственной программы.

Перейдем к файлу с исходным кодом `malloc.cpp`. Его заголовок будет иметь вид:

```
1 #include<cstdio>
2 #include<cstdlib>
3 #include<cstring>
4 using namespace std;
```

```
5 #include "malloc.h"
6 #undef malloc
7 #undef new
8 //_____
9 #define N 1000
10 #define L 1000000
11 static char buf[N][L];
12 static char used[N];
13 static size_t lbuf[N];
14 char sf[N][128];
15 int il[N];
16 #ifdef __GNUC__
17 static int i0=4;
18 #else
19 static int i0=0;
20 #endif
21 static size_t First=static_cast<size_t>(-1);
22 //_____
```

Здесь следует обратить внимание на задание константы `i0`, которой мы воспользуемся позже. Она задается равной 4 для компилятора `gcc` и равной 0 иначе. Макроопределение `__GNUC__` по умолчанию определено для компилятора `gcc`, поэтому конструкция препроцессора

```
1 #ifdef __GNUC__
2 ...
3 #else
4 ...
5 #endif
```

всегда служит для отделения инструкций, предназначенных компилятору `gcc`, от инструкций, предназначенных другим компиляторам.

Сразу после включения файла `malloc.h` мы разопределяем макроопределения `malloc` и `new`, поскольку именно эти операторы мы будем переопределять в `malloc.cpp`.

Далее мы задаем макроопределения: `N` — максимально возможное количество кусков отводимой памяти стандартными средствами отведения памяти, которые мы переопределим, и `L` — максимально возможная длина одного отводимого куска памяти. Заданные выше

значения захватывают 1 ГБ оперативной памяти. При желании эти значения можно изменить.

Массив `buf` представляет собой массив кусков памяти, которые будут выдаваться при вызове функций выделения памяти.

В массиве `used` будет отмечаться, что соответствующий кусок памяти в массиве кусков памяти `buf` выделен пользователю.

В массиве `lbuf` будут храниться длины фактически выделенных кусков памяти.

В массиве строк `sf` будут храниться имена файлов, в которых произведено выделение соответствующего куска памяти.

В массиве целых переменных `il` будут храниться номера строк файлов, в которых произведено выделение соответствующего куска памяти.

Здесь стоит сказать несколько слов о понятии *модели кода*, используемой программой. В приведенном заголовке файла создан массив размером в 1 ГБ ($N * L$). Это не должно приводить к каким-либо проблемам. Более того, мы без проблем можем создать даже три однокигабайтных массива. Но при увеличении размера исходного массива втрое будет невозможно скомпилировать программу, несмотря на то, что мы будем использовать 64-битный компилятор и размер оперативной памяти позволяет вместить данный массив. Это говорит о том, что используемые нами компиляторы/архитектуры/операционные системы имеют внутренние ограничения на размер одного массива. Например, в ОС Unix на привычной нам архитектуре эти ограничения задаются используемой в программе *моделью кода*. Для 64-битного компилятора при создании программы по умолчанию используется модель кода `small`, что предполагает, что размер одного массива в программе не может превышать 2 ГБ. Модель кода можно заменить на `medium` или `large`, указанный предел при этом увеличится. Для компилятора `gcc` задать модель кода можно с помощью ключа `-mcmodel=medium` или `-mcmodel=large`. Данная тема является весьма обширной, но мы ограничимся только упоминанием указанных моментов.

Итак, если пользователю выделен кусок памяти `buf[i]`, то `used[i]=1` и выделение произошло в строке `il[i]` файла `sf[i]`. В случае если файл `malloc.h` не включен в исходный файл программы пользователя, переопределение функций отведения памяти будет происходить,

но информация о номерах строк и именах файлов, где произошло отведение памяти, запоминаться не будет. При этом проверка утечек памяти все равно должна работать.

Свяжем все куски неотведенной памяти в список. Это будет сделано в следующей функции:

```
1 static void InitMemory ()
2 { First=0; size_t i=0;
3   for (i=0; i<N; i++)*(static_cast<size_t*>(buf [ i ] ))=i+1;}
```

Здесь в начале каждого куска памяти будет храниться номер следующего куска свободной памяти. Заметим, что содержимое неотведенных кусков памяти больше никем не используется, поэтому его можно использовать для списка свободных блоков. Если же кусок памяти используется основной программой, то для наших процедур работы со списком свободного места его содержимое уже несущественно. Номер первого куска свободной памяти будет храниться в переменной `First`. Данная функция будет вызываться при первом вызове любой функции отведения памяти. Признаком того, что произошел первый вызов функции отведения памяти будет выполнение условия `First==static_cast<size_t>(-1)`. Связывание кусков свободной памяти в список позволит нам при отведении памяти быстро получать номер куска неотведенной памяти и при освобождении памяти быстро занести освобождаемую память обратно в список кусков неотведенной памяти.

Стоит отметить, что итоговая реализация всех функций отведения/очистки памяти позволяет выполнять их за постоянное время (не зависящее от количества отведенных блоков памяти). Безусловно, данный способ отведения памяти принципиально не позволяет отводить большое количество кусков памяти. Ситуация отчасти исправляется заведением нескольких массивов блоков разной длины для отведения кусков памяти различного размера (именно так реализуется отведение небольших кусков памяти в стандартных функциях отведения памяти). По большому счету это также не является окончательным решением при требовании эффективного отведения большого количества кусков памяти различного размера. Но это уже тема отдельного довольно длинного обсуждения.

Надо также понимать, что в стандартных библиотеках языков C/C++ может присутствовать явный вызов исходных функций отве-

дения памяти (наши переопределенные функции не будут задействованы). Такие неприятности у нас будут происходить, например, в программе, скомпилированной компилятором Microsoft, при выводе на экран при помощи оператора <<. Именно поэтому мы используем для этих целей стандартный вывод языка C — при использовании функций семейства printf отведения памяти не происходит. Компилятор gcc не доставляет подобных неудобств в данном случае, но это не значит, что другие функции из стандартной библиотеки языков C/C++ будут столь же любезны.

Приступим к переопределению функций и операторов отведения и очистки памяти. Данное переопределение не приведет к множественному определению для соответствующих функций и операторов, но, безусловно, следует следить за тем, чтобы подобное переопределение действовало для всей программы. Например, это условие может нарушаться при оптимизации компиляции.

Переопределенная функция malloc() будет иметь следующий вид:

```

1 void *malloc(size_t n)
2 { if(n>L) return nullptr; //инициализация при первом вызове:
3   if(First==static_cast<size_t>(-1))
4     { atexit(CheckMem); InitMemory(); }
5   size_t i=First; if(First==N) return nullptr;
6   First=*static_cast<size_t*>(buf[First]); //извлекаем кусок
      из списка
7   out("malloc(#", i, n, ")\n");
8   used[i]=1; lbuf[i]=n; return buf[i];
9 }

```

Здесь при первом вызове функции отведения памяти с помощью функции atexit() происходит назначение функции CheckMem() на выполнение при завершении работы программы. Также вызывается ранее определенная функция InitMemory(), связывающая куски свободной памяти в список. В переменной i=First лежит номер первого свободного куска памяти. Значение этой переменной возвращается из функции. Переменной First присваивается номер следующего в списке куска свободной памяти: First=*static_cast<size_t*>(buf[First]);. Отмечается, что кусок памяти buf[i] занят, и запоминается длина занятого куска: lbuf[i]=n;. Вызывается функция, извещающая о вызове функции malloc() в случае, если LOG равно 1.

Содержательно отведение памяти полностью объясняется подробным описанием действий переопределенной функции `malloc()`. Но, кроме нее, память еще может отводиться функциями `realloc()`, `calloc()` и `strdup()`. Определим их, используя написанную функцию `malloc()`:

```

1 void *realloc(void *p, size_t n)
2 {if(n>L)return nullptr; out("realloc(",n,")");
3   if(p==NULL)return malloc(n);
4   //ищем номер куска
5   size_t i=(static_cast<char*>(p)-static_cast<char*>(buf))/L;
6   if(i>=N){out("bad realloc\n");return nullptr;}
7   lbuf[i]=n;
8   return p;
9 }
10 //—
11 void *calloc(size_t n, size_t sz)
12 {out("calloc(",n,sz,")");return memset(malloc(n*sz),0,n*sz);}
13 //—
14 char *strdup(const char*s)
15 {out("strdup");return
    strcpy(static_cast<char*>(malloc(strlen(s)+1)),s);}

```

Здесь функция `realloc()` изрядно «халтурит»: она не меняет указатель на отведенный кусок памяти, но корректирует размер отведенной памяти в рамках старого куска.

Реализация функции `free()` также очень проста:

```

1 void free(void* p)
2 {if(p==nullptr)return; size_t
    i=(static_cast<char*>(p)-static_cast<char*>(buf))/L;
3   if(i>=N){out("bad free\n");return;} //вносим кусок в список:
4   used[i]=0; *static_cast<size_t*>(buf[i])=First; First=i;
5   if(static_cast<int>(i)>=i0)out("free(#",i,lbuf[i],")\n");
6 }

```

Содержательно в функции вычисляется номер `i` отведенного куска памяти, отмечается, что кусок памяти свободен: `used[i]=0`, и кусок памяти помещается обратно в список кусков свободной памяти: `*static_cast<size_t*>(buf[i])=First; First=i`;

Следует обратить внимание на то, что вывод сообщения об очистке памяти должен происходить только при `static_cast<int>(i)>=i0` (константа `i0` задается в заголовке исходного файла `malloc.cpp`). Оказывается, компилятор `gcc` не очищает память, отведенную под внутренние нужды четырьмя первыми операциями отведения памяти, и мы принимаем во внимание эту шалость компилятора. Здесь число 4, безусловно, зависит от компилятора. Для рассмотренных автором компиляторов `gcc` и `Microsoft` (последней версии) значение данной переменной определяется корректно. Но гарантий, что для любой реализации компилятора это число останется тем же, нет.

Для простоты мы не переопределяли экзотические функции отведения `_alloca()`, `_malloca()` и т. п.

Осталось определить функцию проверки утечек памяти, которая назначена на выполнение при завершении работы программы:

```

1 void CheckMem()
2 {int good=1;
3  for(int i=i0;i<N;i++)if(used[i])
4  {good=0;
5    if(sf[i][0]!='\0') //если есть информация об имени файла:
6      printf("memory leak:%d size=%d: '%s' line#%d\n",
7            i,static_cast<int>(lbuf[i]),sf[i],il[i]);
8    else //если нет информации об имени файла:
9      printf("memory leak:%d
10             size=%d\n",i,static_cast<int>(lbuf[i]));
11 }
12 if(good)printf("No memory leaks\n");
13 }
```

При наличии утечек памяти функция распечатывает размеры кусков неочищенной памяти, а также номера строк файлов и имена файлов, где была отведена память, если такая информация имеется. При отсутствии такой информации выдаются только размеры кусков неочищенной памяти. Настало время разобраться, откуда берется информация о месте отведения памяти.

В стандартах компиляторов `gcc` и `Microsoft` предусматривается наличие макроопределений препроцессора `__LINE__` и `__FILE__`. При обнаружении таких макроопределений вместо них подставляются соответственно номер текущей строки текущего файла и строка с име-

нем текущего файла. Эти макроопределения удобно использовать в отладочных целях. У нас в конце include-файла `malloc.h` записано макроопределение

```
1 #define malloc(n) malloc(n, __FILE__, __LINE__)
```

В результате в каждом исходном файле программы, в который включен данный include-файл, вызов функции `malloc()` с одним параметром грубо заменится на вызов функции с тем же именем, но уже с двумя дополнительными параметрами: именем текущего файла и номером текущей строки. Осталось добавить в файл `malloc.cpp` определение соответствующей функции:

```
1 void *malloc(size_t n, const char*sf, int il)
2 {if(n>L)return nullptr;
3  if(First==static_cast<size_t>(-1))
4    {atexit(CheckMem);InitMemory();}
5  size_t i=First; if(First==N)return nullptr;
6  First==static_cast<size_t*>(buf[First]);
7  if(static_cast<int>(i)>=i0)out("malloc(#",i,n,")\n");
8  used[i]=1; lbuf[i]=n;
9  ::il[i]=il; strncpy(::sf[i],sf,128); ::sf[i][127]='\0';
10 return buf[i];
11 }
```

Данная функция почти полностью повторяет предыдущую функцию `malloc()`, но в ней еще запоминаются в соответствующие переменные имя текущего файла и номер текущей строки в файле. Аналогичные определения с добавлением двух дополнительных параметров нужны для функций `realloc()`, `calloc()` и `strdup`, но мы для краткости их пропустим. В файле `malloc.cpp` сразу после включения файла `malloc.h` мы разопределили макроопределение `malloc`. Аналогично надо разопределить определения `realloc()`, `calloc()` и `strdup`, если они присутствуют.

Мы задали все основные функции, необходимые для отладки программ на языке C (надо иметь в виду, что некоторые конструкции из языка C++, которые мы использовали, не будут работать при компиляции файла компилятором языка C Microsoft). Осталось переопределить операторы `new` и `delete` для языка C++. Функции, соответствующие данным операторам, должны заниматься только

отведением памяти, поэтому в них передается только размер отводимой памяти. Вызов конструкторов находится полностью в ведении компилятора, и мы об этом заботиться не должны.

Надо иметь в виду, что оператор **new** может использоваться как для создания объектов, так и для создания массивов объектов. Функции, соответствующие данным действиям, различны. Более того, для каждого из этих случаев следует определить сразу две функции, соответствующие оператору **delete**. Одна из них получает только указатель на очищаемую память, а вторая, кроме этого, получает вторым параметром размер одного элемента в отведенном под массив элементов куске памяти. Для нас реализации этих функций ничем отличаться не будут.

Функции для отведения/очистки памяти под отдельные объекты и под массивы объектов будут иметь простой вид:

```

1 void *operator new(size_t n)
2 {out("new(",n,")"); return malloc(n);}
3 void operator delete(void*p)
4 {out("delete"); free(p);}
5 void operator delete(void*p,size_t sz)
6 {out("delete(",sz,")"); free(p);}
7 //_____
8 void *operator new[](size_t n)
9 {out("new [](",n,")"); return malloc(n);}
10 void operator delete [] (void*p)
11 {out("delete []"); free(p);}
12 void operator delete [] (void*p,size_t sz)
13 {out("delete [](",sz,")"); free(p);}

```

Операторы **new** и **new[]** могут иметь дополнительные параметры, в которые можно передавать имя текущего файла и номер строки текущего файла:

```

1 void *operator new(size_t n,const char*sf,int il)
2 {out("new(",n,")"); return malloc(n,sf,il);}
3 void *operator new[](size_t n,const char*sf,int il)
4 {out("new [](",n,")"); return malloc(n,sf,il);}

```

Чтобы можно было использовать данные операторы, в конце `include`-файла `malloc.h` с помощью макроопределения переопределим вызов оператора `new`:

```
1 #define new new(__FILE__, __LINE__)
```

Соответственно, сразу после вставки файла `malloc.h` в файл `malloc.cpp` надо не забыть разопределить данное макроопределение:

```
1 #undef new
```

Осталось напомнить, что объектный файл, получаемый из файла `malloc.cpp`, должен быть прилинкован к тестируемой программе. Весьма полезным оказывается вставка файла `malloc.h` во все исходные файлы программы (только в этом случае будет запоминаться информация об именах файлов и номерах строк, в которых отводилась память).

Упомянутые файлы с небольшими модификациями выложены на сайте <http://lectures.stargeo.ru>. Файл `malloc.c` содержится в архиве <http://lectures.stargeo.ru/malloc.zip> вместе с тестом на функции отведения памяти. Файл `malloc.cpp` вместе с тестом на языке C++ содержится в архиве <http://lectures.stargeo.ru/mallocCpp.zip>. Суть модификаций заключается в том, что вместо одного массива `char buf[N][L]` рассматриваются пять массивов с кусками памяти разной длины:

```
1 #define NMAX 300000000
2 #define L0 128
3 #define L1 512
4 #define L2 8192
5 #define L3 131072
6 #define L4 2097152
7 char p0 [NMAX/L0][L0];
8 char p1 [NMAX/L1][L1];
9 char p2 [NMAX/L2][L2];
10 char p3 [NMAX/L3][L3];
11 char p4 [NMAX/L4][L4];
```

При отведении памяти выбирается массив, содержащий кусок памяти минимально возможного размера, в котором помещался бы требуемый кусок. При этом для каждого размера `L0–L4` придется

создавать свой набор переменных, используемых в описанной выше программе.

Простейший тест позволяет отследить утечки памяти (или отсутствие утечек, если убрать комментирование строк кода):

```

1 #include<iostream>
2 using namespace std;
3 #include"malloc.h"
4 int main(void)
5 {{ printf("begin\n");
6   {int *m=(int*) malloc(5*sizeof(int));
7     //free(m);//"забыли" очистить память
8     m=m; m=nullptr;
9   }
10  {int *m=new int;
11    //delete m;//"забыли" очистить память
12    m=m; m=nullptr;
13  }
14  {int *m=new int[10];
15    //delete m;//"забыли" очистить память
16    m=m; m=nullptr;
17  }
18 }
19 CheckMem();
20 return 0;
21 }

```

Будем предполагать, что данный код размещен в файле q.cpp.

В результате совместной компиляции и сборки файла теста q.cpp и файла malloc.cpp и запуска получившейся программы мы получим следующий вывод на экран с сообщениями об утечках памяти:

```

1 memory leak:4 size=20: 'q.cpp' line#6
2 memory leak:5 size=4: 'q.cpp' line#10
3 memory leak:6 size=40: 'q.cpp' line#14

```

К сожалению, изменение вышеописанного кода для его совместимости с программой на языке С не сводится к простому выбрасыванию всего, что относится к операторам **new** и **delete**. Проблема заключается в отсутствии полиморфизма в языке С. Так, не при-

ведет к проблемам замена стандартных функций отведения/очистки памяти на свои функции с тем же количеством параметров. Но определение тех же функций с дополнительными параметрами `const char*sf, int il` (для имени текущего файла и номера текущей строки в файле) приведет к синтаксическим ошибкам при использовании компилятора с языка C. Поэтому придется по-разному определять эти функции для случая компиляции с ключом `-D LOG` и без этого ключа. В конечном итоге в `include`-файле для программ на языке C после вышеприведенного заголовка будет написано следующее:

```
1 void *malloc(size_t n);
2 void *realloc(void *p, size_t n);
3 void free(void* p);
4 void *calloc(size_t n, size_t sz);
5 char *strdup(const char*s);
6 void *mallocXXX(size_t n, const char*sf, int il);
7 void *reallocXXX(void *p, size_t n, const char*sf, int il);
8 void *callocXXX(size_t n, size_t sz, const char*sf, int il);
9 char *strdupXXX(const char*s, const char*sf, int il);
10 //—
11 #if LOG==1
12 #define malloc(n) mallocXXX(n, __FILE__, __LINE__)
13 #define realloc(s, n) reallocXXX(s, n, __FILE__, __LINE__)
14 #define calloc(n1, n2) callocXXX(n1, n2, __FILE__, __LINE__)
15 #define strdup(s) strdupXXX(s, __FILE__, __LINE__)
16 #endif
```

Здесь имеется в виду, что и в файле `malloc.c` для случаев `LOG==0` (соответствует отсутствию ключа `-D LOG` при компиляции) и `LOG==1` должны определяться функции работы с памятью с различными именами.

6.2. Грамотная реализация `realloc()` на C++11. Placement new

Настало время обратить внимание на отсутствие в языке C++ адекватной замены функции `realloc()` из языка C. Имеется в виду, что, например, вместо функции `malloc()` в языке C++ используется оператор `new[]`, обеспечивающий не только отведение памяти под массив объектов, но и вызов соответствующих конструкторов для

элементов созданного массива (впрочем, здесь особого разнообразия не допускается, так как для элементов массива могут вызываться только конструкторы без параметров).

Было бы логичным наличие в языке C++ оператора `renew`, который обеспечивал бы создание нового массива объектов и перенос в него объектов из старого массива с корректным вызовом необходимых конструкторов/деструкторов. К сожалению, подобная постановка данной задачи не выглядит четкой и аналога описанного оператора в языке C++ не существует. Пока все, что мы можем — это создать новый массив требуемого размера (при этом будут вызываться конструкторы по умолчанию для каждого элемента массива), скопировать требуемую часть старого массива в новый массив (в лучшем случае с помощью `move`-присваивания для каждого переносимого элемента массива) и очистить старый массив, автоматически вызывая деструкторы для всех элементов старого массива.

Получить возможность объединения трех описанных действий (конструктора по умолчанию, конструктора копирования, деструктора) в языке C++ можно с помощью операции *placement new*. Использование данной возможности стало фактически стандартом при работе с STL, поэтому надо понимать, что это такое и когда данную операцию можно использовать. В частности, с *placement new* непосредственно связана функция *emplace*, постоянно используемая при конструировании объектов.

Для примера рассмотрим предельно урезанную структуру вектора:

```

1 #include<iostream>
2 using namespace std;
3 template<class T> struct CVector
4 {T *v; size_t n,nmax;
5   CVector(size_t m=0){v=nullptr; nmax=n=m; if(m){v=new T[m];}}
6   ~CVector(){delete [] v;v=nullptr; nmax=n=0;}
7   T &operator [(size_t i){if(i>=n)throw -1; return v[i];}
8   void push_back(T x)
9   {
10    if(n>=nmax)
11      {T *w=new T[(nmax+1)*2]; //переводим память
12        for(size_t i=0;i<nmax;i++){w[i]=v[i];}
13        nmax=(n+1)*2; delete [] v; v=w;

```

```

14 }
15 v[n++]=x;
16 }//добавляем элемент (память переотведена)
17 };

```

Для данного вектора возможны только рождение, смерть, обращение по индексу через оператор [] и добавление элемента в конец. Для наших целей этого будет достаточно, так как здесь присутствуют операции отведения, очистки и переотведения памяти. Отметим также, что данный шаблон структуры вполне допускает работу с полноценным сложным классом T, параметризующим шаблон.

Простейший тест на данный класс может выглядеть следующим образом:

```

1 int main(void)
2 {int q[]={0,1,2,3,4,5,6,7,8,9};CVector<int> v;
3 for(size_t i=0;i<sizeof(q)/sizeof(*q);i++)v.push_back(q[i]);
4 for(size_t i=0;i<v.n;i++){cout<<v.v[i]<<" ";} cout<<endl;
5 return 0;
6 }

```

В этом примере оказывается весьма удобным применить *placement new*. Идея данного подхода заключается в том, что мы отводим память обычными средствами (не предполагающими вызовов конструкторов), а необходимые конструкторы и деструкторы вызываем вручную. Это может избавить нас от вызовов лишних функций. Вариация *placement new* отличается от обычного оператора **new** тем, что не предполагается отведения памяти: в качестве параметра в круглых скобках после ключевого слова **new** передается адрес объекта, для которого должен быть вызван требуемый конструктор. Создадим три шаблона функций для работы с памятью: функцию отведения памяти под массив Alloc, функцию очистки отведенной памяти Free и функцию переотведения памяти Realloc.

Шаблон функции отведения памяти примет вид:

```

1 template<class T> T* Alloc(size_t n,T*w=nullptr)
2 {if(n==0)return nullptr;//отводим память простым способом:
3 T*v=reinterpret_cast<T*>(new char[n*sizeof(T)]);
4 if(w)//вызов placement new для конструктора копирования:
5 for(size_t i=0;i<n;i++)new(v+i) T(w[i]);

```

```

6  else //вызов placement new для конструктора по умолчанию:
7      for (size_t i=0;i<n;i++)new(v+i) T();
8  return v;}

```

Данную функцию можно вызывать либо только для отведения памяти под n элементов массива (при этом функция возвращает `nullptr`, если требуется создать массив из нуля элементов) типа T с вызовом конструкторов по умолчанию:

```

1  T*m=Alloc<T>(n);

```

либо для отведения памяти с инициализацией элементов массива элементами другого массива m :

```

1  T*m=Alloc<T>(n,m);

```

Отметим, что во втором случае указание типа объекта массива не обязательно, так как компилятор сможет взять его из типа элемента массива m .

Функция очистки памяти примет вид:

```

1  template<class T> void Free(size_t n,T*v)
2  { //вызов деструктора в явном виде:
3      for (size_t i=0;i<n;i++)v[i].~T();
4      delete [] reinterpret_cast<char*>(v);
5  }

```

Обычная смерть массива объектов сопровождается автоматическим вызовом деструкторов для каждого элемента массива и очисткой памяти, отведенной под массив объектов. Здесь же явно вызывается деструктор для каждого элемента массива без автоматической очистки памяти, отведенной под массив. Чистить память, отведенную под массив объектов, придется вручную.

Наконец, функция переотведения памяти от массива, состоящего из n_0 элементов, до массива, состоящего из n_1 элементов, будет выглядеть следующим образом:

```

1  template<class T> T*Realloc(size_t n0,size_t n1,T*w)
2  {T*v=w; static_assert(sizeof(char)==1,"sizeof(char)!=1");
3      if (n1>n0)
4      {v=reinterpret_cast<T*>(new char [n1*sizeof(T)]);

```

```

5   for (size_t i=0;i<n0;i++)//placement new для move-констр-ра:
6     new(v+i) T(static_cast<T&&>(w[i]));
7   for (size_t i=n0;i<n1;i++)new(v+i) T();//констр-р по умолч.
8   Free<T>(n0,w);//=> delete || reinterpret_cast<char*>(w);
9   }
10  else if(n1<n0)
11  { //деструктор вызывается явным образом в следующем виде:
12    for (size_t i=n1;i<n0;i++)v[i].~T();
13  }
14  return v;
15  }

```

Отметим, что элементы в новом массиве создавались здесь с помощью вызова move-конструктора, поэтому если он полностью подчистил содержимое переносимых объектов, то вызов функции Free может быть заменен простой очисткой памяти `delete|| reinterpret_cast<char*>(v)`. Теперь для каждого элемента нового массива вместо вызова трех функций (конструктора по умолчанию, move-конструктора, деструктора для элемента старого массива) нужен вызов всего лишь одной функции — move-конструктора. Также мы не забыли вызвать конструкторы по умолчанию для вновь созданных элементов массива. Объявлению `static_assert` посвящен следующий раздел.

После введения данных функций реализация нашего вектора примет следующий довольно элегантный вид:

```

1  template<class T> struct CVector
2  {T *v=nullptr; size_t n=0,nmax=0;
3    CVector(size_t m=0){v=nullptr; v=Alloc(nmax=n=m,v);}
4    ~CVector(){Free(n,v);v=nullptr;nmax=n=0;}
5    T &operator [(size_t i){if(i>=n)throw -1; return v[i];}
6    void push_back(T x)
7    {if(n>=nmax)//перепроводим память, если необходимо:
8      {v=Realloc(nmax,(nmax+1)*2,v); nmax=(nmax+1)*2;}
9      v[n++]=x;//добавляем элемент (память перепроведена)
10   }
11  };

```

6.3. Инструкция языка C++ `static_assert`

Разберемся с понятием `static_assert`. Для начала вспомним, что в языке C существует весьма полезная конструкция `assert`, позволяющая проверять условия, невыполнение которых должно приводить к останову программы:

```
1 #include<cstdio>
2 #include<cstdlib>
3 #include<cassert>
4 int main(void)
5 {int *m=(int*) malloc(4000LL*1000*1000);
6  assert(m!=NULL); printf("Ok\n");
7  //...
8  return 0;}
```

В этом примере указатель `m` должен быть ненулевым, иначе оператор `assert` прервет выполнение программы (стоит отметить, что далеко не всегда при невозможности отведения памяти функция `malloc()` возвращает нулевой указатель, хотя и должна это делать). Активное использование подобных конструкций существенно облегчает отладку программы, указывая программисту, что программа ведет себя не так, как ему хотелось бы. Однако надо понимать, что использование данного оператора в библиотечных функциях вместо помощи может приносить проблемы для большой программы: большая программа не имеет права падать в ситуации, когда мелкой библиотеке что-то сильно не понравилось. В этом случае в языке C предпочтительнее использовать механизм выдачи соответствующих кодов ошибок для сообщения внешнему миру, что что-то пошло не так.

Даже если мы действительно хотим прерывать программу при обнаружении неприятностей, недостатком описанного подхода является то, что проблема может быть обнаружена только при выполнении программы. Но существуют ситуации, когда проблему можно обнаружить уже при компиляции программы (например, при исследовании каких-то константных выражений, значение которых известно уже на этапе компиляции). Для подобных целей в языке C++ и введен оператор `static_assert`. Если целесообразность использования оператора `assert` в ряде случаев бывает спорной, то оператор `static_assert` должен употребляться по возможности всегда. Его использование

является признаком очень хорошего тона. Использование данной конструкции всегда помогает программисту при написании больших программ.

В приведенном выше примере определения шаблона функции `Realloc()` с помощью данного оператора проверяется, что размер переменной `char` равен одному байту. Иначе весь написанный код становится некорректным. И эта проверка происходит уже на этапе компиляции. Поддерживают данный оператор C++ многочисленные конструкции, позволяющие понять, выполняются ли различные условия при компиляции программы. Например, следующим способом можно проверить, можно ли присваивать переменной типа `T&` значения типа `const T&`:

```
1 static_assert(is_assignable<T&,const T&>::value);
```

На самом деле конструкции, подобные `is_assignable`, являются шаблонами классов. Обращаться к проверяемому выражению можно либо через статический элемент данного класса `value`, либо через оператор `()`.

Приведенная конструкция полезна в шаблонах. Она позволяет сразу же при компиляции кода выявить ошибку программиста, если тот попытается создать шаблон, параметризовав его типом, которому ничего нельзя присвоить (например, массивом). Аналогично можно проверить, есть ли для данного типа конструктор, позволяющий передать в него параметр определенного типа:

```
1 #include<iostream>
2 #include<type_traits>
3 using namespace std;
4 struct S1{S1(){} int x=0;};
5 struct S2{S2(int y=0){x=y;} int x=0;};
6 int main(void)
7 {
8     cout<<is_constructible<S1,int>()<<endl;
9     cout<<is_constructible<S2,int>()<<endl;
10 }
```

В результате выполнения функции на экране появятся нолик и единичка. Аналогично предыдущему случаю данные выражения следует применять в операторе `static_assert`. Например, конструкция

```

1 struct S1{S1(){} int x=0;};
2 static_assert(is_constructible<S1,int>(),"S1(int) not def.");

```

выдаст ошибку компилятора с соответствующим сообщением, напомнив программисту, что он забыл создать конструктор объекта, инициализирующего структуру целой переменной.

6.4. **Emplace**

Мы уже говорили, что понятие *placement new* в языке C++ неразрывно связано с понятием *emplace*. Функции добавления или помещения объекта в контейнер, в имени которых фигурирует «*emplace*», вместо того чтобы добавить/поместить в контейнер передаваемый через параметры функции объект, получают на входе параметры конструктора нового объекта, а затем вызывают *placement new* с этими параметрами для соответствующего места в памяти контейнера.

Рассмотрим пример, использующий приведенную выше структуру `CVector` (с. 72). Создадим класс, являющийся оберткой над обычным целым числом:

```

1 struct Int
2 {int x; Int(int x=0){this->x=x; }
3 operator int()const{return x;}
4 };

```

Теперь вызов метода `v.push_back(1);` для полноценного класса `CVector<Int>` и сложного класса `Int` (разумеется, если мы сделаем сложным класс `Int` и если мы полноценно определим класс `CVector`) превратится в вызов пяти(!) функций: сначала вызовется конструктор для объекта `Int` с целочисленным параметром, потом будет вызван конструктор копирования для передачи созданного объекта в формальный параметр функции, далее будет вызван оператор присваивания формального параметра элементу массива, потом будут вызваны деструкторы для двух созданных объектов типа `Int`. Можно оптимизировать работу, используя *move*-операции, но от вызова пяти функций избавиться не удастся.

Альтернативой данному беспределу является создание метода `emplace_back()`, служащего той же цели, что и исходный метод `push_back()`, но получающего на входе вместо вставляемого объ-

екта параметры конструктора данного объекта и применяющего *placement new* для создания нового объекта по указанному адресу. Полный код шаблона класса `Vector` (отличающегося от шаблона класса `CVector` наличием двух параметров шаблона) будет выглядеть следующим образом:

```

1  template<class T, class I> struct Vector
2  {T *v; size_t n, nmax;
3    Vector(size_t m=0){v=nullptr; nmax=n=m; if(m){v=new T[m];}}
4    ~Vector(){delete [] v;v=nullptr;nmax=n=0;}
5    T &operator [] (size_t i){if(i>=n)throw -1; return v[i];}
6    void realloc(){if(n>=nmax)
7      {T *w=new T[(nmax+1)*2];//переводим память, если
8        требуется
9        for(size_t i=0;i<nmax;i++){w[i]=v[i];}
10       nmax=(n+1)*2; delete [] v; v=w;
11     }
12   void push_back(const T &x)
13   {realloc();//если требуется, переводим память
14     v[n++]=x;//добавляем элемент (память переведена)
15   }
16   void emplace_back(const I &x)
17   {realloc();//если требуется, переводим память
18     new(v+(n++)) T(x);//конструируем элемент по адресу v+n
19   }
20 };

```

Теперь более правильное заполнение вектора будет выглядеть следующим образом:

```

1  int main(void)
2  {int q[]={0,1,2,3,4,5,6,7,8,9}; Vector<Int, int> v;
3    for(size_t i=0;i<sizeof(q)/sizeof(*q);i++)
4      v.emplace_back(q[i]);
5    for(size_t i=0;i<v.n;i++){cout<<v.v[i]<<" ";} cout<<endl;
6    return 0;
7  }

```

6.5. Функциональные объекты

Для начала вспомним подход языка C к ситуации, когда надо выполнить некоторое глобальное действие (которое логично оформить в виде функции), локальную часть которого необходимо задавать особо для каждого конкретного случая. В языке C это локальное действие обычно оформляется в виде функции, указатель на которую передается в общую функцию. Стандартный пример данной ситуации — задача сортировки. Есть стандартные способы сортировки, и необходима функция сравнения сортируемых элементов, указатель на которую передается в функцию сортировки. В языке C существует стандартная функция быстрой сортировки `qsort()` (в современном C она имеет весьма эффективную реализацию [11]). Приведем пример ее использования для сортировки массива целых чисел по возрастанию:

```
1 #include<iostream>
2 using namespace std;
3 int cmp(const void*p1,const void*p2)
4 {
5     if(*(int*)p1<*(int*)p2)return -1;
6     if(*(int*)p1>*(int*)p2)return 1;
7     return 0;
8 }
9 int main(void)
10 {int m[]={5,2,6,1,4,2,8,3,1,2},
11     n=static_cast<int>(sizeof(m)/sizeof(*m));
12     qsort(m,n,sizeof(*m),cmp);
13     for(int i=0;i<n;i++){printf("%d ",m[i]);}printf("\n");
14     return 0;
15 }
```

Несмотря на эффективность данного подхода, у него есть два недостатка: во-первых, налицо накладные расходы на вызов функции сравнения, и при простой функции сравнения они сопоставимы по времени выполнения с самой операцией сравнения. Во-вторых, вместе с указателем на функцию невозможно передать в функцию сортировки никакой дополнительной информации. Например, если мы захотим сортировать целые числа в кольце вычетов по модулю N , то передать это N в функцию сравнением мы сможем только через гло-

бальную переменную, а это является признаком весьма дурного тона и неприменимо в ряде ситуаций (в частности, в случае, когда сортировки одновременно выполняются в параллельных нитях). Решением данной проблемы (с параллельным устранением перечисленных недостатков) в языке C++ является использование *функциональных объектов (функторов)*.

Напишем собственный шаблон простейшей функции сортировки (сортировки пузырьком):

```

1 #include<iostream>
2 #include<algorithm>
3 #include<cstdlib>
4 using namespace std;
5 template<class T, class U>void sort (T*m, int n, U lt)
6 { for (int i=0; i<n; i++) for (int j=1; j<n; j++)
7   if (!lt (m[j-1], m[j])) swap (m[j-1], m[j]);
8 }
```

На вход данной функции передается массив объектов типа T и некоторая на первый взгляд странная вещь lt типа U. В функции используется стандартный алгоритм swap(), который, кстати, является стандартным шаблоном (описанным в include-файле algorithm). От нашего объекта lt мы хотим только одного: иметь возможность приписывать справа от него круглые скобки и передавать в них два сравниваемых объекта типа T. lt должен возвращать *истину*, если первый объект меньше второго (в требуемом смысле), и ложь в противном случае. Например, в качестве lt можно передавать указатель на функцию (как и в языке C). Данную функцию можно также определить с помощью шаблона:

```

1 template<class T>bool cmp(const T&a, const T&b){return a<b;}
```

Тогда функцию сортировки можно выполнять следующим образом:

```

1 {int m[]={5,2,6,1,4,2,8,3,1,2},
2   n=static_cast<int>(sizeof(m)/sizeof(*m));
3   bool (*Cmp)(const int&a, const int&b)=cmp<int>;
4   sort (m, n, Cmp);
5   for (int i=0; i<n; i++){cout<<m[i]<<" ";} cout<<endl;
6 }
```

либо более коротко (но менее понятно):

```

1 {int m[] = {5, 2, 6, 1, 4, 2, 8, 3, 1, 2},
2   n = static_cast<int>(sizeof(m) / sizeof(*m));
3   sort(m, n, cmp<int>);
4   for (int i = 0; i < n; i++) {cout << m[i] << " ";} cout << endl;
5 }
```

По сути, описанный подход ничем не отличается от подхода языка С. Единственным бонусом является универсальность написанных шаблонов для решения задач сортировки для массивов всех типов, допускающих оператор сравнения <.

Теперь настало время вспомнить, что функция не является единственной сущностью в языке С++, к которой можно приписывать круглые скобки. Имеются объекты с определенным оператором (), которые также можно использовать для этой цели. Наконец мы дошли до понятия *функционального объекта* — объекта созданного пользователем типа, в котором определен оператор *круглые скобки*. В нашем случае в эти скобки надо передавать сравниваемые величины, а возвращать этот оператор должен результат сравнения:

```

1 template<class U>struct slt
2 {bool operator () (const U&a, const U&b) {return a < b;}};
```

Здесь объект данного типа не содержит вообще никаких данных, но при необходимости он может содержать переменные с требуемой информацией (например, ту самую N для сравнения чисел в кольце вычетов по модулю N). Более того, поскольку `operator()` определен как метод класса с определением внутри класса, то он по умолчанию считается *inline-функцией*, т. е. при формальном вызове данной функции тело функции должно непосредственно подставляться в вызываемую функцию. Безусловно, слово *должно* здесь существенно. Например, старые компиляторы умели это делать далеко не всегда, но в данном случае даже они сделали бы это. Теперь мы можем использовать все тот же шаблон функции сортировки, передавая ему в качестве способа сравнения функциональный объект вместо указателя на функцию сортировки:

```

1 {int m[] = {5, 2, 6, 1, 4, 2, 8, 3, 1, 2},
2   n = static_cast<int>(sizeof(m) / sizeof(*m));
```

```

3  sort(m,n,slt<int>());
4  for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;

```

Здесь `slt<int>()` — неименованный объект типа `slt<int>`. Для него определен требуемый оператор *круглые скобки*, т. е. на языке C++ корректна запись вида

```

1  if(slt<int>(1,2))cout<<"1<2\n"; else cout<<"!(1<2)\n";

```

В случае использования нашего шаблона для функции сортировки вместо имени `lt` подставляется `slt<int>()`.

Вообще говоря, не было необходимости определять функциональный объект самостоятельно, потому что в библиотеке алгоритмов, описанной в стандартном include-файле `algorithm`, уже присутствует стандартный функциональный объект `less`, который можно использовать в тех же целях:

```

1  {int m[]={5,2,6,1,4,2,8,3,1,2},
2  n=static_cast<int>(sizeof(m)/sizeof(*m));
3  sort(m,n,less<int>());
4  for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
5  }

```

Более того, в шаблоне метода сортировки можно было сразу задать функцию сравнения по умолчанию для сортировки массива по возрастанию:

```

1  template<class T,class U>void sort(T*m,int n,U lt=less<T>())
2  {for(int i=0;i<n;i++)for(int j=1;j<n;j++)
3    if(!lt(m[j-1],m[j]))swap(m[j-1],m[j]);
4  }

```

Наш код можно записать так, что будет казаться, что осуществляется передача указателя на функцию:

```

1  {int m[]={5,2,6,1,4,2,8,3,1,2},
2  n=static_cast<int>(sizeof(m)/sizeof(*m));
3  slt<int> lt;
4  sort(m,n,lt);
5  for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
6  }

```

Легко понять, что смысл и реализация данного кода принципиально отличаются от кода на с. 81, где действительно передается указатель на функцию.

Забегая на полгода занятий вперед, скажем, что в языке C++ подход с использованием функциональных объектов не единственный для решения подобных задач. Более мощным механизмом, который здесь можно применить, является использование техники *наследования*. Этот подход элегантнее и, безусловно, приветствуется на собеседованиях, но, по сути, не имеет никаких достоинств по сравнению с описанным выше подходом.

6.6. Умные указатели. `unique_ptr`.

L1-список на основе умных указателей

Для простоты изложения (и проверки работоспособности подхода) далее мы будем использовать описанную выше реализацию шаблона `Vector`, помещенную в `include`-файл `Vector` (напомним, что это реализация шаблона вектора с одним параметром). Для вывода требуемой диагностической информации о вызове конструкторов и деструкторов написанные далее `src`-файлы надо компилировать с ключом `-D LOG`, добавляя при сборке написанный ранее файл `malloc.cpp`, например:

```
1 g++ -D LOG q.cpp malloc.cpp
```

Разберем следующий тривиальный пример:

```
1 #include<iostream>
2 using namespace std;
3 #include"Vector"
4 {CVector<int> *v=new CVector<int>;
5 //...
6 delete v;//надо помнить!!!
7 }
```

В данном примере видна основная проблема языка C (пусть никого не вводит в заблуждение использование здесь семантики языка C++): при использовании указателей на вручную созданные ресурсы нужно не забыть эти ресурсы освободить. При соблюдении блочной логики программы это не сложно: можно, как в показанном при-

мере, отвести память (захватить ресурс), потом сразу ее очистить (освободить ресурс), а потом уже вписывать все действия с данным ресурсом между двумя написанными конструкциями. Однако алгоритмы далеко не всегда подчиняются таким правилам. Например, при написании алгоритмов работы со списками отведение/очистка памяти имеют намного более сложную логику. По большому счету, именно возможность подобного использования указателей является основной претензией к языку C. Эта проблема наследуется языком C++. Ограничимся, однако, пока только ситуацией, когда захватываемый ресурс необходимо использовать только в рамках текущего блока. Отметим, что ресурс надо освобождать как при обычном выходе из блока, так и при выполнении операций **return** или **break**, если данный блок является телом цикла или оператора **switch**. Для устранения данной проблемы в языке C++ применяются умные указатели вида `unique_ptr`. К сожалению, в русском языке не прижилось устойчивых терминов для именованной данной сущности, и нам придется называть такие объекты просто `unique_ptr` (не забывая, что на самом деле `unique_ptr` — это не тип класса, а шаблон для типа класса).

`unique_ptr` — весьма простая конструкция. Этот умный указатель является классом, в котором хранится обычный указатель. Конструктор класса задает значение этого обычного указателя. Деструктор класса применяет к этому указателю оператор **delete**. Таким образом, при выходе из блока, в котором определена локальная автоматическая переменная типа `unique_ptr`, происходит уничтожение объекта, которым *владеет* данный умный указатель. Для того чтобы не дать возможности двум объектам типа `unique_ptr` владеть одним ресурсом (указателем на отведенную память), для объектов данного типа запрещен оператор присваивания и конструктор копирования. Безусловно, для находчивых двоечников это не помеха, и они вполне могут напрямую создать пару умных указателей, владеющих одним обычным указателем. К сожалению, язык C++ не может воспрепятствовать этому. Таким образом, использование умных указателей не является панацеей, никто не гарантирует отсутствия ошибок при работе с ними.

Обычный синтаксис создания и использования `unique_ptr` полностью соответствует приведенному описанию:

```

1 {Vector<int> *v=new Vector<int>;
2   unique_ptr<Vector<int>> up(v);
3   //...
4 }
```

Для `unique_ptr` доступны операторы `*` и `->`, эмулирующие работу с обычным указателем. Для извлечения из умного указателя обычного указателя используется метод `get()`:

```

1 {unique_ptr<Vector<int>> up(new Vector<int>);
2   Vector<int> *v=up.get();
3   //...
4 }
```

Здесь мы получили обычный указатель `v`, указывающий на объект, которым владеет умный указатель `up`.

Как было сказано выше, для объектов типа `unique_ptr` запрещены операторы присваивания и копирования. Однако для них разрешены `move`-операции (`move`-присваивание и `move`-копирование), и при их помощи реализован метод данного класса `swap`, позволяющий обменять местами содержимое двух умных указателей. Для тех же целей можно использовать алгоритм `swap` с двумя параметрами. Итак, корректна следующая конструкция:

```

1 {unique_ptr<Vector<int>> up(new Vector<int>(1)),
2   up2(new Vector<int>(2));
3   //up=up2;// ERROR!!!
4   up.swap(up2);
5 }
```

Правила хорошего тона не рекомендуют создавать `unique_ptr` таким способом. Следует использовать шаблон функции `make_unique()`, создающей умный указатель и требуемый объект с помощью одной операции отведения памяти, а не двух (при описанных выше операциях память отводится отдельно под сам умный указатель и отдельно под объект, которым будет владеть данный умный указатель). Синтаксис такого действия следующий:

```

1 {unique_ptr<Vector<int>> up(make_unique<Vector<int>>());
2 }
```

Здесь имеет смысл задаться нетривиальным вопросом: а что же возвращает шаблон функции `make_unique()`? Данная конструкция не может возвращать объект типа `unique_ptr`, поскольку для данных умных указателей операции копирования/присваивания запрещены. Ответ лежит на поверхности: `make_unique()` возвращает `rvalue`-ссылку на умный указатель. Тогда конструирование объекта `up` превращается в вызов `move`-конструктора, который разрешен для `unique_ptr`.

Указатели `unique_ptr` могут использоваться и для более сложных целей. В первой части лекций мы рассматривали реализацию однонаправленного списка с фиктивным элементом. Шаблон для такого класса весьма прост:

```

1 #pragma once
2 template<class T> struct SList1
3 {struct SNode
4   {T v; SNode *next; SNode(){v=T{}; next=nullptr;}
5   SNode(const T&v){this->v=v; next=nullptr;}};
6   SNode b,*cur; int n;
7   SList1(){cur=&b;n=0;}
8   ~SList1(){clear();}
9   void clear(){GoToBegin(); while(DelNext());}
10  int size(){return n;}
11  bool empty(){return n==0;}
12  void GoToBegin(){cur=&b;}
13  bool GoToNext(){if(cur->next){cur=cur->next; return true;}
14    return false;}
15  bool Get(T&v){if(cur==&b){return false;}
16    v=cur->v;return true;}
17  void insert(const T&v){SNode *nd=new SNode(v);
18    nd->next=cur->next;cur->next=nd; this->n++;}
19  bool find(const T&v){T x; for(GoToBegin();GoToNext();)
20    {Get(x);if(x==v)return true;} return false;}
21  bool DelNext()
22  {if(cur->next==nullptr)return false; SNode *nd=cur->next;
23    cur->next=cur->next->next;delete nd;this->n--;return true;}
24 };

```

Приведем простейший пример использования такого списка строк типа `string`:

```
1 #include<iostream>
2 #include<fstream>
3 #include<cstdlib>
4 #include<cstring>
5 #include<string>
6 using namespace std;
7 #include "List"
8 int main(void)
9 {{ SList1<string> l; string s;
10  for (int i=0;i<10;i++)l.insert("0");
11  l.clear();
12  cout<<(l.empty()?"empty":"not empty")<<endl;
13  }cout<<"done";
14 return 0;
15 }
```

Здесь `string` — стандартный класс языка C++, описанный в include-файле `string`. Строкам этого типа можно присваивать строки в понимании языка C, но преобразования типа по умолчанию к строке в понимании языка C (т. е. к указателю `char*` на массив символов, terminated нулем) не существует. Вместо этого у класса `string` имеется метод `c_str()`, возвращающий указатель на строку в понимании языка C.

Также заметим, что здесь использован стандартный подход: заключение всего жизненного цикла объекта `l` в блок с последующим (после закрытия блока) выводом отладочного сообщения. Такой подход дает возможность убедиться в корректности работы указанного объекта.

Изменим созданный выше шаблон класса `SList1` так, чтобы в каждом элементе списка хранился не обычный указатель на следующий элемент, а умный указатель с аналогичным содержимым, т. е. вместо `SNode *next` будем использовать `unique_ptr<SNode> next`. Тип указателя `cur` менять не будем. Данный подход (при внесении соответствующих небольших изменений в код) избавит нас от необходимости следить за утечками памяти. Действительно, при смерти всего объекта списка сначала умрет элемент списка `b`, а при его смерти умрет умный указатель `next` внутри `b`. Смерть данного умного указателя повлечет за собой смерть объекта, на который он указывает, т. е. первого

реального элемента списка. Далее последовательно при смерти каждого элемента списка будет умирать умный указатель внутри него, что приведет к смерти следующего элемента списка, и т. д. Таким образом, теперь не надо определять деструктор для всего списка. Вот как выглядит код с `unique_ptr`:

```
1 #pragma once
2 #include<memory>
3 using namespace std;
4 template<class T> struct SList1
5 {struct SNode
6   {T v; unique_ptr<SNode> next; SNode()=default;
7     SNode(const T&v){this->v=v;} ~SNode(){cout<<"~"<<endl;}};
8   SNode b,*cur; int n;
9   SList1() {cur=&b;n=0;}
10  ~SList1()=default;
11  void clear() {b.next.reset();n=0;}
12  int size(){return n;}
13  bool empty(){return n==0;}
14  void GoToBegin() {cur=&b;}
15  bool GoToNext() {if(cur->next.get())
16    {cur=cur->next.get(); return true;} return false;}
17  bool Get(T&v) {if(cur==&b){return false;}
18                v=cur->v; return true;}
19  void insert(const T&v){unique_ptr<SNode>
20    nd=make_unique<SNode>(v);
21    nd->next.swap(cur->next); cur->next.swap(nd); this->n++;}
22  bool find(const T&v){T x; for(GoToBegin();GoToNext();)
23    {Get(x); if(x==v)return true;} return false;}
24  bool DelNext()
25    {if(cur->next==nullptr)return false; unique_ptr<SNode> nd;
26     nd.swap(cur->next); cur->next.swap(nd->next); this->n--;
27     return true;}
28  };
```

Отметим, что функция `clear()` теперь реализована с применением метода `reset()` умного указателя внутри элемента списка `b`. Этот метод очищает память, обычный указатель на которую хранится в данном `unique_ptr`, и замещает этот обычный указатель значением из параметра метода. Если параметра нет, то он по умолчанию считается

равным `nullptr`. Также надо было несколько изменить логику методов `insert()` и `DelNext()`. Один из вариантов реализации данных методов приведен выше. Для обеспечения работы с умными указателями подключен заголовочный файл `memory`.

Приведенный метод реализации однонаправленного списка является классическим и обязательным для воспроизведения на собеседованиях. Аналогичная реализация однонаправленного списка приведена на сайте Microsoft. Можно, например, добавить в реализованный выше список строк 10000 строк "0", и все будет великолепно работать. Почему-то почти никто не упоминает, что данная реализация не является работоспособной. Например, попытка выполнения

```

1 #include<iostream>
2 #include<cstring>
3 #include<string>
4 using namespace std;
5 #include "List"
6 int main(void)
7 {{SList1<string> l; string s;
8   for (int i=0;i<100000;i++)l.insert("0");
9   }cout<<"done";
10 return 0;
11 }
```

приводит к падению программы. Текст `done` не будет выведен на экран ни при выполнении программы, скомпилированной обычным способом компилятором `gcc`, ни при выполнении программы, скомпилированной компилятором Microsoft.

Выяснить причину неприятностей позволяет четкое понимание процесса очистки памяти, отведенной в списке, при вызове деструктора. Действительно, деструктор каждого элемента списка рекурсивно вызывается из деструктора предыдущего элемента списка. А рекурсия глубиной в 100000 шагов необратимо вызывает переполнение стека, что приводит к падению программы. Ситуация лечится возвращением старого деструктора данного класса и функции `clear()`:

```

1 ~SList1 () { clear ();}
2 void clear () {GoToBegin (); while(DelNext ());}
```

В этом случае никакой рекурсии уже не будет. Но все изменения при переходе от обычного указателя к умному тогда вообще теряют какой-либо смысл.

Следует обратить внимание на важную тонкость: `unique_ptr` владеет объектом, созданным с помощью операции `new`, и уничтожить его надо оператором `delete`. Но в языках C/C++ указатели могут указывать как на одиночный объект, так и на массив объектов. Создание/очистка массивов в языке C++ производятся другими операторами: `new[]/delete[]`. Попытка освобождения оператором `delete` памяти из-под массива, созданного с помощью оператора `new[]`, для массива объектов сложных классов всегда приводит к падению программы (для массивов переменных элементарных типов это происходит не всегда). Таким образом, если объект `unique_ptr` владеет указателем на массив, требуется передача в объект информации об особом способе очистки памяти. Начиная с первоначальных версий реализации `unique_ptr` для этого требовалась передача в объект `unique_ptr` функционального объекта, выполняющего данное действие. Функциональный объект требуется параметризовать типом, на который указывает хранящийся в объекте указатель. Сам указатель должен передаваться в функциональный объект через оператор *круглые скобки*. В нашем случае для хранения в умном указателе указателя на массив требуется определить шаблон структуры/класса следующего вида:

```
1  template<class T>struct Deleter
2  { void operator () (T*p) { delete [] p; } };
```

Объект данного типа следует передавать в `unique_ptr`, хранящий указатель на массив объектов типа `Vector<int>` например, следующим образом:

```
1  unique_ptr<Vector<int>, Deleter<Vector<int>>>
2  p(new Vector<int> [2], Deleter<Vector<int>>());
```

6.7. Умные указатели. `shared_ptr`

`shared_ptr` является существенно более сложной реализацией умного указателя по сравнению с `unique_ptr`. Обычным указателем на объект могут владеть несколько `shared_ptr`. Каждый `shared_ptr` хранит

указатель на объект и счетчик ссылок. При корректном появлении нового *shared_ptr*, владеющего указателем на тот же объект, счетчик увеличивается на единицу. При смерти *shared_ptr* счетчик уменьшается на единицу. Пока (позднее мы вернемся к этому вопросу) будем говорить, что при обнулении счетчика происходит вызов функции, уничтожающей подвластный объект. Легко понять, что в реальности для корректной реализации *shared_ptr* должен хранить указатель на общую структуру, в которой как раз и содержится счетчик ссылок. При этом указатель на подвластный объект должен храниться в самом умном указателе. Тогда все *shared_ptr*, владеющие указателем на один и тот же объект, будут иметь доступ к одной и той же контрольной структуре. Безусловно, все эти умные указатели должны создаваться путем присваивания или копирования от одного умного указателя. Если мы ошибочно создадим два независимых умных указателя, владеющих одним и тем же обычным указателем, то ничего хорошего из этого не получится (это верно и для *unique_ptr*, и для *shared_ptr*). Итак, умные указатели очень хорошо помогают обеспечить очистку памяти из-под созданных объектов, но все же, как уже говорилось ранее, не являются в этом вопросе панацеей. Их неправильное использование не может не привести к ошибкам при работе программы.

Приведем простой пример, иллюстрирующий потенциальные проблемы при работе с обычными указателями:

```
1 #include<iostream>
2 #include "Vector"
3 using namespace std;
4 Vector<int> *fun ()
5 { Vector<int> *p=new Vector<int >(1000);
6 return p;
7 }
8 int main(void)
9 { Vector<int> *v=fun ();
10 //...
11 delete v;v=nullptr;
12 return 0;
13 }
```

Внутри функции `main()` нет явного отведения памяти под вектор (отведение происходит внутри функции `fun()`), и, глядя только на функцию `main()`, невозможно понять, что делать с указателем `v`. Например, если `v` в реальности является указателем на статический вектор, то очищать память из-под него не надо, а если на динамический, то надо. Информацию о том, как надо разрушать созданный объект, желательно закладывать в программу при создании объекта, когда видно, как объект создан. Для упрощения жизни программиста в этом случае удобно использовать умный указатель `shared_ptr`, который выполняет только что озвученное пожелание. Будем везде вместо обычного указателя использовать умный указатель:

```
1 #include<iostream>
2 #include<memory>
3 #include "Vector"
4 using namespace std;
5 shared_ptr<Vector<int>> fun ()
6 {shared_ptr<Vector<int>> p=make_shared<Vector<int>>(1000);
7   return p;
8 }
9 int main(void)
10 {shared_ptr<Vector<int>> v=fun ();
11   Vector<int> *w=v.get ();
12   //...
13   return 0;
14 }
```

Теперь пользователь не должен заботиться о происхождении вектора, с которым идет работа в функции `main()`. Пока жива хотя бы одна копия исходного умного указателя, созданного в функции `fun()`, вектор из этой функции будет жив. Когда умрет последний умный указатель, владеющий этим объектом, то и сам вектор умрет, т. е. для указателя, хранящегося в `make_shared<>()` будет вызван оператор `delete`. Опять же не надо забывать, что обычный указатель, которым будет владеть умный указатель, должен быть создан либо оператором `new`, либо с помощью шаблона функции `make_shared<>()`. В данном контексте недопустимо использование обычного указателя на статическую или автоматическую переменную.

Как и для `unique_ptr<>()`, в случае если память отводится под массив объектов, можно использовать функциональный объект, который указывает, как разрушать объект, на который указывает указатель. Синтаксис использования данного функционального объекта для случая `shared_ptr<>()` немного отличается от аналогичного синтаксиса для `unique_ptr<>()`. Для `shared_ptr<>()` при задании соответствующего типа не надо указывать тип функционального объекта, а достаточно просто передать функциональный объект во второй параметр умного указателя:

```

1 #include<iostream>
2 #include<memory>
3 #include"Vector"
4 using namespace std;
5 template<class T> struct Del
6 {void operator ()(T*p){delete [] p;}};
7 shared_ptr<Vector<int>> fun()
8 {shared_ptr<Vector<int>>
9   p(new Vector<int>[5],Del<Vector<int>>());
10  return p;
11 }
12 int main(void)
13 {shared_ptr<Vector<int>> v=fun();
14  Vector<int> *w=v.get();
15  //...
16  return 0;
17 }
```

В новых версиях C++ доступен отдельный вид умного указателя `shared_ptr<>()`, для которого параметр шаблона имеет вид `T[]`, где `T` — тип некоторого класса. Для объекта данного типа не определены операторы `*` и `->`, но определен оператор *квадратные скобки*, позволяющий обращаться к элементам соответствующего массива:

```

1 #include<iostream>
2 #include<memory>
3 #include"Vector"
4 using namespace std;
5 shared_ptr<Vector<int>[]> fun()
6 {shared_ptr<Vector<int>[]> p(new Vector<int>[5]);return p;}
```

```
7  int main(void)  
8  {shared_ptr<Vector<int>> v=fun ();  
9    Vector<int> *w=v.get (); v[1].push_back(1);v[1].push_back(2);  
10   cout<<v[1]<<endl;return 0;  
11 }
```

7. Структуры данных. Вектор. STL

В [11] уже было введено понятие *структуры данных* как некое обобщение понятия множества (либо семейства) элементов некоторого вида с предопределенным набором функций (*предписаний*) для работы с этими элементами. Под определением структуры данных обычно как раз и имеется в виду выписывание набора предписаний, доступных для структуры данных. Описание стандартного набора структур данных можно найти в классических книгах [7], [8]. При реализации структуры данных (а она может быть различной!) принято каждому предписанию ставить в соответствие функцию, реализующую это предписание. Существует четыре базовых предписания, общих для всех структур данных:

1. Создать.
2. Уничтожить.
3. Очистить.
4. Пуста ли?

Для структуры данных *вектор* первое предписание логично модифицируется в

1. Создать вектор заданной длины.

Для библиотеки STL, о которой мы будем говорить позже, стало стандартом включать во все реализации структур данных (контейнеров) предписание

5. Получить количество элементов множества.

Кроме того, среди предписаний структуры данных желательно иметь некоторые стандартные способы перебора элементов данного множества, но об этом речь пойдет позже.

Остальные предписания специфичны для каждой конкретной структуры данных. По большому счету, для структуры данных *вектор* обязательно только одно дополнительное предписание:

6. Задать/получить значение элемента вектора (множества) с заданным порядковым номером.

Все остальные предписания зависят от конкретной реализации структуры данных *вектор*. Среди них могут быть:

7. Добавить элемент в конец вектора.
8. Получить значение последнего элемента вектора.
9. Вставить элемент на заданную позицию вектора.
10. Уничтожить элемент на заданной позиции вектора.

Далее мы разберем пару нетривиальных реализаций описанной структуры данных на языке C++.

7.1. Правило нуля. Реализация вектора в стиле Python

Правило нуля является в каком-то смысле вершиной возможностей реализации объектов в языке C++. Оно предельно просто: если элементы класса являются либо базовыми переменными (целыми или вещественными), либо полноценными классами (удовлетворяющими правилу трех, пяти или нуля), либо простыми классами, то при реализации данного объекта в соответствующем классе не обязательно определять конструкторы, деструкторы и операторы присваивания.

Действительно, в этом случае при создании объекта автоматически вызовутся все необходимые конструкторы для элементов класса (для базовых переменных и простых классов этого не произойдет, но это и не важно с точки зрения обеспечения нормального процесса жизнедеятельности объектов данного типа: они нормально родятся и умрут), при смерти объекта автоматически вызовутся все необходимые деструкторы для элементов класса, а при различных видах присваивания/копирования объектов класса автоматически вызовутся соответствующие операции присваивания/копирования для всех элементов класса (присваивание/копирование классов происходит поэлементно). Здесь важно, чтобы все элементы класса умели выполнять все перечисленные операции. В обычной жизни желательно создавать классы исключительно на основе правила нуля, но хорошо понимать правило пяти и уметь создавать следующие ему классы, так как в основе всего лежат именно они. Например, скоро мы будем разбирать стандартную реализацию вектора в языке C++. Эта реализация покрывает существенную часть наших потребностей при работе с векторами. Однако для стандартных векторов, например, нет операции сложения (весьма нужной в работе), поэтому в ряде случаев приходится писать свою реализацию этой структуры данных (после разбора понятия *наследование* это уже не будет нужно, но пока мы притворяемся, что ничего об этом не знаем). Нам пригодится реализация, которую мы разбираем уже в течение длительного времени.

Вернемся к созданной ранее реализации шаблона вектора, размещенной в include-файле Vector:

```

1 #pragma once
2 #include<iostream>
3 #include<fstream>
4 #include<string>
5 #include<cstdlib>
6 #include<cstring>
7 using namespace std;
8 //—
9 #ifndef LOG
10 #define LOG 0
11 #else
12 #undef LOG
13 #define LOG 1
14 #endif
15 #ifndef OUT
16 #define OUT
17 inline void out(const char*s){ if (LOG) cout<<s<<endl;}
18 template<class T>void out(const char*s, const T&x,
19     const char*s2){ if (LOG) cout<<s<<x<<s2<<endl;}
20 #endif
21 //—
22 template<class T> class Vector
23 {T *v; size_t n, nmax;
24 public:
25     //—
26     Vector(size_t n=0){out("Vector(",n,")"); if(n)
27     {v=new T[nmax=this->n=n]; for(size_t i=0;i<n;i++)v[i]=T();}
28     else SetZero();}
29     Vector(const Vector&b){out("Vector(const Vector&b)");
30         CopyOnly(b);}
31     ~Vector(){out("~Vector(",n,")"); Clean();}
32     Vector &operator=(const Vector&b){
33         out("Vector &operator=(const Vector&b)");
34         if(&b!=this){Clean(); CopyOnly(b);} return *this;}
35     Vector(Vector&&b){out("Vector(Vector&&b)"); MoveOnly(b);}
36     Vector &operator=(Vector&&b){out("operator=(Vector&&b)");

```

```

37         if(&b!=this) { Clean (); MoveOnly (b); } return *this; }
38     //—
39     void SetZero () { v=nullptr; n=0; nmax=0; }
40     void Clean () { delete [] v; SetZero (); }
41     void CopyOnly ( const Vector& b) { if (b.nmax)
42         { v=new T[nmax=b.nmax]; n=b.n;
43         for (size_t i=0; i<nmax; i++) v[i]=b.v[i]; } else SetZero (); }
44     void
45         MoveOnly ( Vector&b) { v=b.v; n=b.n; nmax=b.nmax; b.SetZero (); }
46     //—
47     struct SErr { string s; SErr ( const char*s="err" ) { this->s=s; }; };
48     T &operator [] ( size_t i)
49     { if (i>=n) throw SErr ("bad index"); return v[i]; }
50     const T &operator [] ( size_t i) const
51     { if (i>=n) throw SErr ("bad index"); return v[i]; }
52     void push_back ( const T&x) { if (n==nmax) { size_t nmax2=2*nmax+1;
53     T*w=new T[nmax2]; size_t i=0; for (; i<n; i++) { w[i]=v[i]; }
54     for (; i<nmax2; i++) { w[i]=T(); } delete [] v; v=w; nmax=nmax2; }
55     v[n++]=x; }
56     Vector operator + ( const Vector&b) const & {
57         out ("Vector operator+(const Vector&b)&");
58         if (n!=b.n) { throw SErr ("n!=b.n"); } Vector r (*this);
59         for (size_t i=0; i<n; i++) { r[i]=r[i]+b[i]; } return r; }
60     Vector &&operator + ( const Vector&b) && {
61         out ("Vector operator+(const Vector&b)&&");
62         if (n!=b.n) { throw SErr ("n!=b.n"); }
63         for (size_t i=0; i<n; i++) { v[i]=v[i]+b[i]; }
64         return static_cast<Vector&&> (*this); }
65     Vector &&operator + ( Vector&&b) const & {
66         out ("Vector operator+(Vector&&b)&");
67         if (n!=b.n) { throw SErr ("n!=b.n"); }
68         for (size_t i=0; i<n; i++) { b.v[i]=v[i]+b[i]; }
69         return static_cast<Vector&&> (b); }
70     Vector &&operator + ( Vector&&b) && {
71         out ("Vector operator+(Vector&&b)&&");
72         if (n!=b.n) { throw SErr ("n!=b.n"); }
73         for (size_t i=0; i<n; i++) { v[i]=v[i]+b[i]; }
74         return static_cast<Vector&&> (*this); }
75     size_t size () const { return n; }

```

```

75 T &back(){return v[n-1];}
76 template<class U> friend ostream &operator<<
77 (ostream &cout, const Vector<U> &v);
78 };
79 //—
80 template<class V>
81 ostream &operator<<(ostream &cout, const Vector<V> &v)
82 {cout<<" ("; for(size_t i=0;i<v.n;i++){cout<<v[i]<<" ";}
      cout<<")"; return cout;}
83 //—

```

Простейший тест для данной реализации имеет вид:

```

1 #include<iostream>
2 #include<fstream>
3 #include<cstdlib>
4 #include<cstring>
5 using namespace std;
6 #include "Vector"
7 int main(void)
8 {Vector<double> v; int m[]={0,1,2,3,4,5,6,7,8,9};
9 for(int i=0;i<10;i++)v.push_back(m[i]);
10 Vector<double> w=v+v; cout<<"v+v="<<w<<endl;
11 return 0;
12 }

```

Попробуем изменить данную реализацию так, чтобы вместо обычного указателя в ней хранился умный указатель `shared_ptr` на массив данных типа `T`. Безусловно, можно подправить всю реализацию вектора для совместимости с новым типом данных. Однако мы поступим по-другому. Создадим вектор, аналогичный по поведению объектам (в частности, спискам) языка Python, т. е. копирование/присваивание нашего вектора будет создавать объект, ссылающийся на тот же самый массив данных. А для создания полноценной копии вектора будем использовать новый метод `copy()`. При этом мы хотим получить реализацию вектора на основе правила нуля, поэтому будем стремиться отбросить все конструкторы и операторы присваивания нового класса `Vector` (кроме, возможно, одного, создающего вектор заданной длины).

Первое, что приходит в голову — это замена указателя, хранящегося в классе `Vector`, умным указателем:

```
1 shared_ptr<T[]> v; size_t n=0,nmax=0;
```

Здесь используется инициализация элементов структуры, доступная начиная с 11 версии C++.

К сожалению, данный подход является неправильным. Умные указатели данного типа можно присваивать друг другу, и все они будут указывать на одну область данных, но если после присваивания друг другу изменить указатель, хранящийся внутри одного из таких `shared_ptr`, то указатели внутри других `shared_ptr` останутся неизменными. Чтобы изменение указателя внутри одного `shared_ptr` приводило к изменению указателей внутри других `shared_ptr`, необходимо хранить внутри `shared_ptr` не указатель на данные, а указатель на указатель на данные. Тогда если несколько `shared_ptr` хранят указатели на один и тот же указатель на данные, то изменение одного указателя на данные затронет все упомянутые `shared_ptr`.

Предложенный подход, однако, также не решает проблему. Мы добились того, что изменение одного `shared_ptr` затрагивает другие, равные ему `shared_ptr`. Но теперь гибель всех `shared_ptr`, указывающих на один и тот же объект, не приводит к автоматической гибели данных, на которые указывает указатель, хранящийся внутри указателя, хранящегося внутри `shared_ptr`. Следовательно, внутри `shared_ptr` надо хранить не обычный указатель, а умный указатель. Более того, достаточно хранить умный указатель вида `unique_ptr`. Таким образом, в классе `Vector` будут храниться следующие данные:

```
1 shared_ptr<unique_ptr<T[]>> v=make_shared<unique_ptr<T[]>>();
2 size_t n=0,nmax=0;
```

Данная инициализация делает необязательной наличие конструктора, так как после указанных действий создается полноценный умный указатель `shared_ptr`, указывающий на созданный умный указатель `unique_ptr`, хранящий внутри себя указатель, равный `nullptr`. Тем не менее полезно создать конструктор, создающий вектор заданной длины:

```
1 Vector(size_t n=0){out("Vector(",n,")");
2 if(n){(*v).reset(new T[nmax=this->n]);
```

```
3 for (size_t i=0;i<n;i++)(*v)[i]=T();}
```

Отметим, что использование здесь метода `reset()` предполагает наличие уже корректно созданного внутри `v` умного указателя.

Функции, обслуживающие конструкторы и операторы присваивания, достаточно заменить на одну новую функцию очистки вектора:

```
1 void clear() {v.reset(); n=nmax=0;}
```

Функция добавления элемента в список будет иметь вид:

```
1 void push_back(const T&x) {if (n==nmax) {size_t nmax2=2*nmax+1;
2 T*w=new T[nmax2]; size_t i=0; for (;i<n;i++){w[i]=(*v)[i];}
3 for (;i<nmax2;i++){w[i]=T();} (*v).reset(w); nmax=nmax2;}
4 (*v)[n++]=x;}
```

Наконец, копия объекта будет создаваться функцией `copy()`:

```
1 Vector copy() const {Vector r(n);
2 for (size_t i=0;i<n;i++){r[i]=(*v)[i];} return r;}
```

В функции сложения придется создавать полную копию текущего объекта с помощью вызова конструктора `Vector r(copy());`:

```
1 Vector operator+(const Vector&b) const&{
2 out("Vector operator+(const Vector&b)&");
3 if (n!=b.n){throw SErr("n!=b.n");} Vector r(copy());
4 for (size_t i=0;i<n;i++){r[i]=(*v)[i]+b[i];} return r;}
```

Но можно обойтись и без созданного конструктора класса `Vector`:

```
1 Vector operator+(const Vector&b) const&{
2 out("Vector operator+(const Vector&b)&");
3 if (n!=b.n){throw SErr("n!=b.n");} Vector r;
4 for (size_t i=0;i<n;i++){r.push_back((*v)[i]+b[i]);}
5 return r;}
```

Для получившегося шаблона класса `Vector` (с необходимыми заменами `v` на `(*v)`) написанный выше тест будет замечательно работать. Однако полученная реализация вектора на самом деле будет неработоспособной. Проблема заключается в том, что, хотя присваивание одного вектора другому создает корректную ссылку на один и тот

же массив данных, элементы `n` и `nmax` этих двух векторов будут ссылаться на различные переменные. Поэтому если мы в один из этих двух векторов добавим элемент, то значение `n` (и, возможно, `nmax`) изменится только у этого вектора. Для разрешения этой ситуации следует целые переменные `n` и `nmax` также заменить на умные указатели `shared_ptr` на целые переменные. Безусловно, обращаться к значениям этих переменных придется через оператор `*`: соответственно, `*n` и `*nmax`. Тогда данные вектора будут описываться следующим образом:

```
1 shared_ptr<unique_ptr<T[]>>v=make_shared<unique_ptr<T[]>>();
2 shared_ptr<size_t> n=make_shared<size_t>(0),
3     nmax=make_shared<size_t>(0);
```

Теперь присваивание одного вектора другому будет создавать два объекта, указывающих полностью на одни и те же данные. После упомянутых выше изменений мы получим следующий код шаблона вектора:

```
1 #pragma once
2 #include<iostream>
3 #include<fstream>
4 #include<string>
5 #include<memory>
6 #include<cstdlib>
7 #include<cstring>
8 using namespace std;
9 //—
10 #ifndef LOG
11 #define LOG 0
12 #else
13 #undef LOG
14 #define LOG 1
15 #endif
16 #ifndef OUT
17 #define OUT
18 inline void out(const char*s){ if (LOG) cout<<s<<endl;}
19 template<class T>void out(const char*s, const T&x,
20     const char*s2){ if (LOG) cout<<s<<x<<s2<<endl;}
21 #endif
```

```

22 //—
23 template<class T> class Vector
24 {
25     shared_ptr<unique_ptr<T[]>>v=make_shared<unique_ptr<T[]>>();
26     shared_ptr<size_t>
27         n=make_shared<size_t>(0),nmax=make_shared<size_t>(0);
28 public:
29     //—
30     Vector(size_t n=0){out("Vector(",n,")");
31         if(n){(*v).reset(new T[*nmax=*(this->n)=n]);
32         for(size_t i=0;i<n;i+)(*v)[i]=T();}
33     Vector copy()const{Vector r(*n);
34         for(size_t i=0;i<*n;i++){r[i]=(*v)[i];} return r;}
35     //—
36     struct SErr{string s; SErr(const char*s="err"){this->s=s;}};
37     T &operator[(size_t i){if(i>=*n)throw SErr("bad index");
38         return (*v)[i];}
39     const T &operator[(size_t i)const{return (*v)[i];}
40     void push_back(const T&x){if(*n==*nmax){/*Begin if (...)*/
41         size_t nmax2=2*(nmax)+1; T*w=new T[nmax2]; size_t i=0;
42         for(;i<*n;i++){w[i]=(*v)[i];} for(;i<nmax2;i++){w[i]=T();}
43         (*v).reset(w);*nmax=nmax2;}/*End if (...)*/ (*v)[(*n)++]=x;}
44     Vector operator+(const Vector&b)const&{
45         out("Vector operator+(const Vector&b)&");
46         if(*n!=*(b.n)){throw SErr("Different sizes");} Vector r;
47         for(size_t i=0;i<*n;i++){r.push_back((*v)[i]+b[i]);}
48         return r;}
49     Vector &&operator+(const Vector&b)&&{
50         out("Vector operator+(const Vector&b)&&");
51         if(*n!=*(b.n)){throw SErr("Different sizes");}
52         for(size_t i=0;i<*n;i++){(*v)[i]=(*v)[i]+b[i];}
53         return static_cast<Vector&&>(*this);}
54     Vector &&operator+(Vector&&b)const&{
55         out("Vector operator+(Vector&&b)&");
56         if(*n!=*(b.n)){throw SErr("Different sizes");}
57         for(size_t i=0;i<*n;i++){(*b.v)[i]=(*v)[i]+b[i];}
58         return static_cast<Vector&&>(b);}
59     Vector &&operator+(Vector&&b)&&{
60         out("Vector operator+(Vector&&b)&&");

```

```

61     if (*n != *(b.n)) {throw SErr("Different sizes");}
62     for (size_t i=0; i<*n; i++){(*v)[i]=(*v)[i]+b[i];}
63     return static_cast<Vector&&>(*this);}
64 size_t size() const {return *n;}
65 T &back() {return (*v)[(*n)-1];}
66 template<class U> friend ostream &operator<<
67 (ostream &cout, const Vector<U> &v);
68 };
69 //—
70 template<class V>
71 ostream &operator<<(ostream &cout, const Vector<V> &v)
72 {cout<<" "; for (size_t i=0; i<*(v.n); i++){cout<<v[i]<<" ";}
73 cout<<" "; return cout;}
74 //—

```

Здесь надо обратить внимание на комментарий */*End if(...)*/*. Подобные комментарии, указывающие на то, какой блок закрывает данная фигурная скобка, являются признаком хорошего тона. Они существенно улучшают читаемость кода, в котором присутствуют большие блоки.

Следующий тест показывает возможности созданного вектора передаваться по значению в функцию, возвращаться по значению из функции и копироваться так, как если бы все это происходило по ссылке. Отметим, все указанные операции выполняются быстро, так как реальный размер объекта `Vector` небольшой (например, для 64-битного `gcc` по умолчанию размер вектора равен 48 байтам независимо от декларируемого размера вектора, и, соответственно, он вдвое меньше для 32-битного компилятора). Для созданного вектора передача по ссылке, по сути, не нужна.

```

1 #include<iostream>
2 using namespace std;
3 #include "Vector"
4 Vector<double> fun()
5 {Vector<double> v; int m[]={0,1,2,3,4,5,6,7,8,9};
6 for (int i=0; i<10; i++)v.push_back(m[i]);
7 return v;
8 }
9 void fun2(Vector<double> v)

```

```

10 {v.push_back(25);}
11 int main(void)
12 {Vector<double> v=fun(),q; cout<<"v="<<v<<endl;
13  Vector<double> w=v+v;      cout<<"v+v="<<w<<endl;
14  v=w;                      cout<<"v="<<v<<endl;
15  q=v;
16  fun2(q);                  cout<<"q="<<q<<endl;   cout<<"v="<<v<<endl;
17  return 0;
18 }

```

Следует иметь в виду, что при максимальном уровне замечаний существующему на данный момент компилятору `gcc` не удастся скомпилировать приведенный вариант программы. Ему не удастся корректно обработать все `inline`-функции, что заставляет его выдать замечание, а это при максимальном уровне замечаний интерпретируется как ошибка. Компиляция данного кода без дополнительных ключей проходит успешно. Последняя версия компилятора Microsoft также успешно справляется с написанной программой.

7.2. Реализация вектора неограниченной длины

Вернемся к нашей стандартной (на основе правила пяти) реализации вектора (с. 36). Для такой реализации существует стандартная задача языка C++: *требуется реализовать вектор неограниченной (формально) длины*. Вместо реализации метода `push_back()` или задания размера вектора в конструкторе мы хотим просто обращаться к элементу вектора с произвольным индексом. При этом если обращение к элементу вектора происходит справа от знака присваивания (или в выражении, передаваемом в качестве параметра функции), то гарантируется сохранение размера вектора (отсутствие переотведения памяти). Иными словами, если мы пытаемся **получить значение** элемента вектора, расположенного в рамках отведенной памяти, то используется соответствующий элемент вектора. Если при этом индекс выходит за рамки отведенной памяти, то возвращается значение некоторой технической переменной, значение которой, условно говоря, обнулено. Если же мы **присваиваем значение** некоторому элементу вектора, то при необходимости память должна переотводиться. При присваивании значения элементу вектора по индексу, расположенному в рамках отведенной памяти, должно осуществ-

ляться обычное присваивание. Если же индекс выходит за границы отведенной памяти, то перед присваиванием должно происходить автоматическое переотведение памяти под массив объектов вектора. В сухом остатке мы хотим получить возможность прямого выполнения различных действий при получении значения элемента вектора и при присваивании элементу вектора некоторого значения.

Практика показывает, что манипуляции с модификатором `const` в соответствующем операторе *квадратные скобки* здесь, к сожалению, не помогают (можно было бы рассчитывать, что компилятор будет использовать `const T & operator[]` справа от знака присваивания, а `T & operator[]` — слева, но такого не происходит). К решению данной задачи приводит создание нового типа данных, содержащего ссылку на используемый вектор (шаблон вектора) элементов типа `T` и текущий индекс вектора. Оператор *квадратные скобки* теперь будет возвращать не ссылку на элемент данных, а объект нашего нового типа. При этом оператор *квадратные скобки* будет помещать в создаваемый объект ссылку на текущий вектор и индекс, получаемый оператором в качестве параметра. Для нового типа определим функции для двух операторов: преобразования нового типа к типу `T` и присваивания новому типу элемента типа `T`. Первый оператор будет использоваться, если оператор *квадратные скобки* для нашего вектора применяется справа от знака присваивания, а второй — слева от знака присваивания. При этом остается надеяться на мудрость компилятора, понимающего, какие преобразования типов надо делать в соответствующих случаях (далее мы увидим, что компилятор может применять в подобных случаях редкостную особенность).

Итак, создадим внутри шаблона класса `Vector` новый тип

```
1 struct SV
2 {
3     Vector *v; size_t i;
4     SV(Vector *v, size_t i){this->v=v;this->i=i;}
5     operator T()const{if(i>=v->n)return Vector::TMP;
6         return v->v[i];}
7     T &operator=(const T&x)
8     {if(i>=v->n)v->resize(i+1); v->v[i]=x;return v->v[i];}
9 };
```

Здесь `TMP` — статическая переменная типа `T`, размещенная в классе `Vector`, а `resize()` — новый метод класса `Vector`, позволяющий изменять его размер. В пятой и седьмой строчках кода определяются оператор преобразования типа и оператор присваивания.

К сожалению, одного такого класса оказывается недостаточно, так как в коде реализации вектора, кроме обычного вектора, присутствует вектор с модификатором `const`, а такой вектор не может быть передан в созданную структуру. Придется создать аналогичную структуру, обслуживающую `const Vector` (естественно, везде речь идет о соответствующих шаблонах):

```

1 struct SVC
2 {
3     const Vector *v; size_t i;
4     SVC(const Vector *v, size_t i){this->v=v; this->i=i;}
5     operator T() const {if (i>=v->n) return Vector::TMP;
6                     return v->v[i];}
7 };

```

Оператор присваивания для объектов типа `SVC` не нужен по понятным причинам.

Осталось заменить операторы *квадратные скобки* соответствующим образом:

```

1 SV operator [] (size_t i){return SV(this, i);}
2 SVC operator [] (size_t i) const {return SVC(this, i);}

```

и мы получим работоспособный код.

Заметим, что при создании статического члена класса мы не определяем его, а просто описываем. Фактически, объявляя член класса статическим (принято говорить, что мы создаем член класса, общий для всех экземпляров данного класса), мы просто описываем глобальную переменную, область видимости которой ограничена данным классом. Но эту переменную надо еще где-то определить. Если бы у нас был обычный класс (не шаблон) `Vector` с целой статической переменной внутри него

```

1 static int TMP;

```

то для этой переменной в одном из `src`-файлов (ровно в одном!) должно было бы присутствовать ее определение в виде

```
1 int Vector::TMP;
```

Если бы это определение присутствовало более чем в одном файле, то при сборке программы мы получили бы ошибку `multiply definition` — множественное определение переменной. Именно поэтому данное определение нельзя включать в `include`-файл: если этот файл будет включен в несколько исходных файлов программы, то мы сразу же получим множественное определение переменной. Однако в случае шаблонов требование к единственности определения глобального объекта уже не является необходимым. Ранее мы наблюдали это на примере определения функций. То же самое относится и к определению переменных. Для определения статической переменной шаблона достаточно определить эту переменную внутри `include`-файла вне класса в виде

```
1 template<class T> T Vector<T>::TMP;
```

Безусловно, в этом случае мы получим множественное определение переменной, но в случае шаблона это не является ошибкой.

В конечном счете мы получим следующее содержание `include`-файла с определением шаблона вектора бесконечной длины:

```
1 #pragma once
2 #include<iostream>
3 #include<string>
4 using namespace std;
5 //—
6 template<class T> class Vector
7 {T *v; size_t n,nmax; static T TMP;
8 public:
9     struct SV
10    {Vector *v; size_t i;
11     SV(Vector *v, size_t i){this->v=v; this->i=i;}
12     operator T() const
13     {if(i>=v->n) return Vector::TMP; return v->v[i];}
14     T &operator=(const T&x)
15     {if(i>=v->n) v->resize(i+1); v->v[i]=x; return v->v[i];}
16 };
17 struct SVC
```

```

18 {const Vector *v; size_t i;
19   SVC(const Vector *v, size_t i){this->v=v; this->i=i;}
20   operator T() const
21     {if(i>v->n)return Vector::TMP; return v->v[i];}
22 };
23 //—
24 Vector(size_t n=0){if(n){v=new T[nmax=this->n=n];
25   for(size_t i=0;i<n;i++)v[i]=T();} else SetZero();}
26 Vector(const Vector&b){CopyOnly(b);}
27 ~Vector(){Clean();}
28 Vector &operator=(const Vector&b)
29   {if(&b!=this){Clean(); CopyOnly(b);} return *this;}
30 Vector(Vector&&b){MoveOnly(b);}
31 Vector &operator=(Vector&&b)
32   {if(&b!=this){Clean(); MoveOnly(b);} return *this;}
33 //—
34 void SetZero(){v=nullptr; n=0; nmax=0;}
35 void Clean(){delete [] v; SetZero();}
36 void CopyOnly(const Vector& b)
37   {if(b.nmax){v=new T[nmax=b.nmax]; n=b.n;
38   for(size_t i=0;i<nmax;i++)v[i]=b.v[i];} else SetZero();}
39 void MoveOnly(Vector&b)
40   {v=b.v; n=b.n; nmax=b.nmax; b.SetZero();}
41 //—
42 struct SErr{string s; SErr(const char*s="err"){this->s=s;}};
43 SV operator [] (size_t i){return SV(this, i);}
44 SVC operator [] (size_t i) const{return SVC(this, i);}
45 void push_back(const T&x){(*this)[n] = x;}
46 void resize(size_t m){if(m>nmax){size_t nmax2=2*m+1;
47   T*w=new T[nmax2]; size_t i=0; for(; i<n; i++){w[i]=v[i];}
48   for(; i<nmax2; i++){w[i]=T();} delete [] v; v=w; nmax=nmax2; n=m;}
49 Vector operator+(const Vector&b) const&{
50   if(n!=b.n){throw SErr("n!=b.n");} Vector r(*this);
51   for(size_t i=0; i<n; i++){r[i]=r[i]+b[i];} return r;}
52 Vector &&operator+(const Vector&b)&&{
53   if(n!=b.n){throw SErr("n!=b.n");}
54   for(size_t i=0; i<n; i++){v[i]=v[i]+b[i];}
55   return static_cast<Vector&&>(*this);}
56 Vector &&operator+(Vector&&b) const&{

```

```

57     if (n!=b.n){throw SErr("n!=b.n");}
58     for (size_t i=0;i<n;i++){b.v[i]=v[i]+b[i];}
59     return static_cast<Vector&&>(b);}
60 Vector &&operator+(Vector&&b)&&{
61     if (n!=b.n){throw SErr("n!=b.n");}
62     for (size_t i=0;i<n;i++){v[i]=v[i]+b[i];}
63     return static_cast<Vector&&>>(*this);}
64 size_t size()const{return n;}
65 T &back(){return v[n-1];}
66 template<class U> friend ostream &operator<<
67 (ostream &cout ,const Vector<U> &v);
68 };
69 //—
70 template<class V>
71 ostream &operator<<(ostream &cout ,const Vector<V> &v)
72 {cout<<" "; for (size_t i=0;i<v.n;i++)
73 {cout<<v[i]<<(i<v.size()-1?" ":"");}cout<<" ";return cout;}
74 //—
75 template<class T> T Vector<T>::TMP;

```

В качестве теста для данной реализации вектора можно использовать следующий код:

```

1  #include<iostream>
2  using namespace std;
3  #include"Vector"
4  //—————
5  int main(void)
6  {Vector<double> v;  int m[]={0,1,2,3,4,5,6,7,8,9};
7   for (int i=0;i<10;i++)v[i]=(float)m[i];
8   cout<<"v="<<v<<endl;
9   for (int i=0;i<10;i++){float x=v[i];cout<<x<<" ";}cout<<endl;
10  v=v+v;  cout<<"v+v="<<v<<endl;
11  for (size_t i=0;i<v.size();i++)v[i]=v[i]+v[i];
12  cout<<"v+v+v="<<v<<endl;
13  return 0;
14  }

```

Здесь следует отметить отменную сообразительность компилятора при выполнении совершенно нетривиальных процедур $v[i]=\text{(float)}m[i]$ и $v[i]=v[i]+v[i]$;

7.3. C++. Итераторы

Как уже говорилось в начале главы, для каждой структуры данных желательно иметь некоторый стандартизированный механизм перебора элементов множества. Этот механизм не должен опираться на внутренние возможности конкретного множества. Например, для однонаправленных списков существует понятие *текущего элемента*. Перемещая позицию текущего элемента по списку, можно перебирать элементы списка. У этой возможности есть существенный недостаток: если мы захотим перебрать все пары элементов списка, то наличие всего одного текущего элемента не позволит нам это сделать.

В языке C++ для стандартного перебора элементов реализации множества вводится понятие *итератора*. Изначально был некоторый набор договоренностей о том, какие операции (например, разыменовывание, инкрементация, сравнение) необходимы для объекта, осуществляющего перебор. Перегрузка операторов позволила выработать унифицированный синтаксис для объектов-итераторов. С развитием языка C++ появились новые синтаксические конструкции, опирающиеся на итераторы. Об этом речь пойдет позднее.

В качестве примера синтаксиса использования обычных итераторов рассмотрим простой цикл для перебора элементов множества. Для нашей реализации вектора должна быть предусмотрена следующая конструкция перебора элементов вектора (в приведенном примере — с выводом элементов вектора на экран):

```

1 Vector<int> v;
2 //...
3 for (Vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
4 {cout<<*it<<" ";}cout<<endl;
```

В стандартной библиотеке шаблонов (Standard Templates Library — STL) языка C++ данный вид итераторов допустим для всех контейнеров, для которых понятие итератора вообще имеет смысл.

Принято говорить о четырех видах итераторов: обычном (перебирающем элементы в естественном порядке), обратном, обычном для const-объектов, обратном для const-объектов. Соответствующие итераторы должны иметь имена типов: `iterator`, `reverse_iterator`, `const_iterator`, `const_reverse_iterator`.

Из нашего примера работы с обычным итератором видно, что для него должны быть определены как минимум унарный оператор `*`, операторы `++`, `!=`, а внутри обслуживаемого класса должны быть определены функции `begin()` и `end()`, возвращающие соответствующие итераторы. Для трех остальных типов итераторов соответствующие функции должны иметь имена `rbegin()` и `rend()`, `cbegin()` и `chend()`, `crbegin()` и `crend()`. Еще раз подчеркнем, что слово *должны* здесь используется в прямом смысле. Далее мы будем разбирать конструкции языка C++, в которых участвуют данные операторы и функции, поэтому синтаксис определения итераторов должен жестко соблюдаться.

Когда все, о чем говорилось выше, задано, остается выполнить весьма небольшую работу по созданию кода, обслуживающего обычный итератор для класса `Vector` (код должен располагаться внутри публичной части класса `Vector`):

```
1 struct iterator{ Vector *v; size_t i;
2   iterator(Vector*v=nullptr, size_t i=0){this->v=v;this->i=i;}
3   T &operator*( ) {return v->v[i];}
4   bool operator!=(const iterator &b) const{return i!=b.i;}
5   iterator operator++(){i++; return *this;} //префиксный
6   iterator operator++(int tmp) //постфиксный оператор ++
7     {tmp=tmp; iterator x=*this; i++; return x;}
8 };
9 iterator begin(){return iterator(this,0);}
10 iterator end(){return iterator(this,n);}
```

Здесь имеет смысл обратить внимание на определение двух операторов инкрементирования `++`: префиксного и постфиксного. Сразу же становится понятно, почему в языке C++ принято использовать именно префиксный оператор инкрементирования (несмотря на привычку из языка C к постфиксной форме вызова данного оператора). Действительно, постфиксный оператор возвращает значение инкрементируемой переменной, которое было до собственно выполнения

операции инкрементирования. Поэтому для возвращаемого значения приходится создавать временную переменную, сохраняющую исходное значение данного итератора. В префиксном операторе можно просто возвращать `*this`. Эту тонкость всегда надо иметь в виду для грамотного использования описываемой операции.

Код, обеспечивающий работу трех оставшихся итераторов, имеет следующий вид:

```

1  struct const_iterator{ const Vector *v; size_t i;
2  const_iterator(const Vector*v=nullptr, size_t i=0)
3  {this->v=v; this->i=i;}
4  const T &operator*(){return v->v[i];}
5  bool operator!=(const const_iterator &b)const{return
6  i!=b.i;}
7  const_iterator operator++(){i++; return *this;}
8  const_iterator operator++(int tmp)
9  {tmp=tmp; const_iterator x=*this; i++; return x;}
10 };
11 const_iterator cbegin()const{return const_iterator(this,0);}
12 const_iterator cend()const{return const_iterator(this,n);}
13 //—
14 struct reverse_iterator{ Vector *v; size_t i;
15 reverse_iterator(Vector*v=nullptr, size_t i=0)
16 {this->v=v; this->i=i;}
17 T &operator*(){return v->v[i];}
18 bool operator!=(const reverse_iterator &b)const
19 {return i!=b.i;}
20 reverse_iterator operator++(){i--; return *this;}
21 reverse_iterator operator++(int tmp)
22 {tmp=tmp; reverse_iterator x=*this; i--; return x;}
23 };
24 reverse_iterator rbegin(){return reverse_iterator(this,n-1);}
25 reverse_iterator rend(){return reverse_iterator(this,-1);}
26 //—
27 struct const_reverse_iterator{ const Vector *v; size_t i;
28 const_reverse_iterator(const Vector*v=nullptr, size_t i=0)
29 {this->v=v; this->i=i;}
30 const T &operator*(){return v->v[i];}
31 bool operator!=(const const_reverse_iterator &b)const

```

```

31 {return i!=b.i;}
32 const_reverse_iterator operator++(){i--; return *this;}
33 const_reverse_iterator operator++(int tmp)
34 {tmp=tmp; const_reverse_iterator x=*this; i--; return x;}
35 };
36 const_reverse_iterator crbegin() const
37 {return const_reverse_iterator(this, n-1);}
38 const_reverse_iterator crend() const
39 {return const_reverse_iterator(this, -1);}

```

Например, оператор << для класса Vector теперь можно переписать следующим образом:

```

1  template<class V>
2  ostream &operator<<(ostream &cout, const Vector<V> &v)
3  {cout<<" "; for (Vector<double>::const_reverse_iterator
4  it=v.crbegin(); it!=v.crend(); ++it){cout<<*it<<" ";}
5  cout<<" "; return cout;}

```

Здесь для разнообразия элементы вектора выводятся в обратном порядке. **const**-версия итератора необходима, так как обслуживаемый вектор имеет модификатор **const**, что запрещает изменение его содержимого.

7.4. STL. Реализация вектора.

Стандартные требования к итераторам STL

Принято говорить, что структура данных *вектор* реализована в STL как шаблон `vector`. Однако на самом деле в STL существуют две реализации вектора с одинаковым набором предписаний, касающихся собственно вектора. Первая реализация уже названа, а вторая — шаблон `deque`. Несмотря на то, что шаблон `deque` предназначен для реализации структуры данных *дек*, речь о которой пойдет позднее, в `deque` перенесены предписания для работы с вектором. И наоборот: некоторые предписания из структуры данных *дек* перенесены в шаблон `vector`. Таким образом, для работы с вектором (аналогом массива) можно пользоваться обоими шаблонами совершенно одинаковым образом, не забывая, что реализации в них совершенно разные. А это, в свою очередь, требует понимания, какую реализацию в каких случаях следует использовать.

Для работы с данными контейнерами и для некоторых дополнительных операций нам потребуются следующие include-файлы:

```

1 #include<vector>
2 #include<deque>
3 #include<algorithm>
4 #include<functional>

```

Существует два основных способа обращения к элементам вектора: через оператор `[]` (как к элементам обычного массива) и через метод `at()`. Во втором случае происходит обязательная проверка выхода за границы массива с выбросом соответствующего исключения в соответствующем случае. В следующем примере произойдет выброс исключения:

```

1 vector<int> v;
2 try
3 {
4   for (int i=0;i<20;i++){ printf("%d\n",i);v.push_back(i);}
5   for (int i=0;i<=20;i++){ printf("%d\n",v.at(i));}
6 }catch (...) {cout<<"Out of scope\n";}

```

По умолчанию оператор `[]` не производит проверку выхода за границу массива. Для шаблона `vector` время работы такой операции не отличается от обращения к элементу обычного массива (т. е. использование шаблона `vector` может проигрывать по времени работы обычному массиву только в момент создания с отведением памяти: обычный массив как автоматическая переменная базируется на стеке, а `vector` в любом случае захватывает память из кучи). Однако такое поведение можно изменить. Для автоматической проверки выхода за границы массива при вызове оператора `[]` при использовании компилятора `g++` требуется задать макроопределение `_GLIBCXX_DEBUG`, причем это макроопределение должно быть задано до включения include-файлов, т. е. либо с помощью ключа компилятора `-D _GLIBCXX_DEBUG`, либо с помощью макроопределения в первой строке файла с программой:

```

1 #define _GLIBCXX_DEBUG

```

Здесь стоит отметить, что при выходе за границы массива в этом случае поведение программы может оказаться отличным от ожидае-

мого. На данный момент проверка выхода за границы вектора, несомненно, происходит, но вместо желаемого выброса исключения возможно простое прерывание работы программы с выводом сообщений о проблемах внутри реализации шаблона вектора, что не указывает на проблемное место в коде. Возможно, эта явная ошибка реализации STL под gcc будет в дальнейшем исправлена.

Для компилятора Microsoft требование к проверке выхода за границу вектора при использовании оператора `||` сводится к установке макроопределения `_DEBUG` в начале файла с программой (опять же либо с помощью ключа компиляции `-D _DEBUG`, либо с помощью прямого задания макроопределения). При этом собирать исполняемый файл следует, например, с ключом `-MTd`, что приводит к использованию отладочной версии стандартной библиотеки C++. Например, программу из одного файла можно скомпилировать и собрать с помощью команды

```
1 g++ -D _DEBUG q.cpp /MTd
```

Безусловно, проще все это сделать в среде разработки Microsoft Visual Studio, где все необходимые ключи устанавливаются автоматически в конфигурации проекта Debug.

Возвращаясь к теме итераторов, следует заметить, что в STL они активно используются как для перебора элементов множества, так и для указания позиции в множестве. Например, алгоритм `sort()` требует в качестве параметров итераторы на начало и на пустой конец сортируемого множества (*пустым концом* принято называть элемент, расположенный после последнего элемента массива). Сортировка вектора с распечаткой результата может быть произведена с помощью следующих команд:

```
1 sort(v.begin(), v.end());  
2 for(int i=0; i<20; i++){ printf("%d ", v[i]); } printf("\n");
```

Сортировку по убыванию можно произвести следующим способом:

```
1 sort(v.begin(), v.end(), greater<int>());  
2 for(int i=0; i<20; i++){ printf("%d ", v[i]); } printf("\n");
```

Здесь `greater` — соответствующий стандартный функциональный объект.

То же самое более понятно:

```

1 greater<int> gt;
2 sort(v.begin(), v.end(), gt);
3 for(int i=0; i<20; i++){ printf("%d ", v[i]); } printf("\n");

```

Постараемся сделать так, чтобы последняя тройка примеров кода корректно работала одновременно для шаблонов `vector`, `deque`, `Vector` (последний определен нами). Легко проверить, что данный код корректно компилируется и выполняется для шаблонов `vector` и `deque`. Осталось разобраться с вектором, который мы сами определили. Для него придется доопределить довольно много методов для уже созданных итераторов, но проблем возникнуть не должно: сообщения об ошибках, выдаваемые компилятором, подскажут, какие методы надо доопределить. Менее тривиально требование STL доопределить внутри итераторов несколько типов, необходимых для корректного обслуживания итераторов нашего класса `Vector`. А именно, внутри итераторов надо доопределить следующие типы:

```

1 typedef output_iterator_tag iterator_category;
2 typedef T value_type;
3 typedef ptrdiff_t difference_type;
4 typedef T* pointer;
5 typedef T& reference;

```

Данные определения являются стандартными для создаваемых пользователями итераторов. Они предоставляют метаинформацию об итераторах, чтобы шаблоны STL могли с ними работать. Легко понять смысл данных определений. Например, с помощью `iterator::value_type` шаблоны STL, параметризуемые типом итераторов класса `Vector`, получают доступ к типу объектов, которые содержатся в классе `Vector`. Подчеркнем, что часто шаблоны STL параметризуются даже не классом обслуживаемого объекта (к примеру, `Vector`), а типом итератора для данного объекта (`Vector::iterator`). Например, шаблон `sort` для класса `Vector` можно вызывать следующим образом:

```

1 sort<Vector<int>::iterator>(v.begin(), v.end());

```

При этом в силу особенностей семантики языка эти шаблоны будут иметь сложности доступа к типу объекта, содержащемуся в классе

Vector. Действительно, если тип итератора является параметром шаблона, то откуда можно получить тип объекта, которым параметризуется класс, содержащий данный итератор? Остается только вариант задания стандартного имени для данного базового типа T посредством typedef.

Также (увы) придется доопределить следующие методы класса Vector::iterator:

```

1 size_t operator-(const iterator&b) const {return i-b.i;}
2 iterator operator+(int b) const {return iterator(v, i+b);}
3 iterator operator-(int b) const {return iterator(v, i-b);}
4 bool operator==(const iterator&b) const {return
   v==b.v&&i==b.i;}
5 bool operator<(const iterator&b) const {return i<b.i;}
6 bool operator>(const iterator&b) const {return i>b.i;}
7 iterator operator--(){i--; return *this;}
8 iterator operator--(int tmp){tmp=tmp; iterator r=*this; i--;
   return r;}

```

В конечном счете определение обычного итератора внутри класса Vector примет весьма сложный вид:

```

1 struct iterator {Vector *v; size_t i;
2   iterator(Vector *v=nullptr, size_t
   i=0){this->v=v; this->i=i;}
3   bool operator!=(const iterator&b) const {return i!=b.i;}
4   iterator operator++(){i++; return *this;}
5   iterator operator++(int tmp){tmp=tmp; iterator r=*this; i++;
   return r;}
6   T &operator*(){return v->v[i];}
7   size_t operator-(const iterator&b) const {return i-b.i;}
8   iterator operator+(int b) const {return iterator(v, i+b);}
9   iterator operator-(int b) const {return iterator(v, i-b);}
10  bool operator==(const iterator&b) const {return
   v==b.v&&i==b.i;}
11  bool operator<(const iterator&b) const {return i<b.i;}
12  bool operator>(const iterator&b) const {return i>b.i;}
13  iterator operator--(){i--; return *this;}
14  iterator operator--(int tmp){tmp=tmp; iterator r=*this; i--;
   return r;}

```

```

15  typedef output_iterator_tag iterator_category;
16  typedef T value_type;
17  typedef ptrdiff_t difference_type;
18  typedef T* pointer;
19  typedef T& reference;
20  };

```

Аналогично придется исправить три оставшихся итератора.

Перейдем к особенностям реализации вектора для шаблонов `vector` и `deque`. Рассмотрим простой код:

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  #include "malloc.h"
5  int main(void)
6  {vector<int> v;
7   for (size_t i=0;i<100;i++)v.push_back(i);
8   return 0;
9  }

```

Скомпилируем и соберем нашу программу с соответствующим ключом:

```

1  g++ -D LOG q.cpp malloc.cpp

```

Файл `malloc.cpp` с отладочной версией функций/операторов отведения памяти мы обсуждали в разделе 6.1..

Мы получим следующий вывод на экран:

```

1  new(4) malloc(#4,4)
2  new(8) malloc(#5,8)
3  delete(4) free(#4,4)
4  new(16) malloc(#4,16)
5  delete(8) free(#5,8)
6  new(32) malloc(#5,32)
7  delete(16) free(#4,16)
8  new(64) malloc(#4,64)
9  delete(32) free(#5,32)
10 new(128) malloc(#5,128)
11 ...

```

Отсюда мы сразу получаем, что вектор в STL компилятора `gcc` устроен по тому же принципу, что и созданный нами шаблон `Vector`: при добавлении элементов к концу вектора каждый раз при заполнении отведенного куска памяти происходит переотведение памяти с увеличением размера отведенной памяти вдвое (для компилятора `Microsoft` размер отведенного куска памяти увеличивается в полтора раза). Каждое такое переотведение памяти требует переноса старых объектов со старого куска памяти на новый кусок (например, с помощью `move-присваивания`; подробно особенности этого переноса будут обсуждаться позднее). Если вектор обслуживает достаточно сложные объекты, то это может привести к довольно затратным операциям.

Если тип вектора `vector<int> v` заменить на `deque<int> v` с добавлением `include-файла deque`, то у аналогичной программы будет другой вывод на экран:

```
1 new(64) malloc(#4,64)
2 new(512) malloc(#5,512)
3 new(512) malloc(#6,512)
4 new(512) malloc(#7,512)
5 new(512) malloc(#8,512)
6 new(512) malloc(#9,512)
7 new(144) malloc(#10,144)
8 delete(64) free(#4,64)
9 new(512) malloc(#4,512)
10 new(512) malloc(#11,512)
11 new(512) malloc(#12,512)
12 new(512) malloc(#13,512)
13 new(512) malloc(#14,512)
14 new(512) malloc(#15,512)
15 new(512) malloc(#16,512)
16 new(304) malloc(#17,304)
17 delete(144) free(#10,144)
18 new(512) malloc(#10,512)
19 new(512) malloc(#18,512)
20 ...
```

Мы видим, что устройства шаблона `deque` совершенно другое. Отведение памяти размером 64, 144 и т. д. байт соответствует отведению памяти под массив указателей (при необходимости каждый

раз его размер увеличивается вдвое + некоторый кусок памяти под служебные нужды). Каждый такой указатель указывает на кусок памяти размером 512 байт (отведение этих кусков памяти также отображается на экране). Собственно данные будут заполнять именно эти куски по 512 байт (если размер объекта большой, то куски будут иметь другой размер). Таким образом, переотведение памяти (под массив указателей) никак не сказывается на месторасположении самих объектов. Их адрес остается неизменным в течение всей жизни данной версии реализации вектора. Минусом данной реализации является более медленное обращение к элементам вектора, так как здесь используется двойная ссылка на элемент: сначала на блок размером 15 байт, а только потом на элемент в блоке.

8. STL. Структура данных *дек* и его адаптеры

Существует некоторое различие между классическими структурами данных *стек*, *дек*, *очередь* и их аналогами в STL, набор предписаний в определениях этих структур данных отличается от методов шаблонов соответствующих классов. Однако при создании контейнеров STL была использована некоторая внутренняя логика, которая позволяет понять, почему все было сделано именно так, а не иначе. Эта логика включала в себя требование соответствия понятий STL официально принятым определениям указанных структур данных, а также соображения об удобстве и универсальности использования соответствующих контейнеров. Рассмотрим более подробно указанные контейнеры.

8.1. STL. Стек. Реализация стека

Стеком называется структура данных, организованная по принципу LIFO — last in, first out, т. е. элемент, попавший в множество последним, должен первым его покинуть. При практическом использовании часто налагается ограничение на длину стека, т. е. требуется, чтобы количество элементов не превосходило некоторого целого N.

Создание исполнителя *стек* предполагает реализацию следующих предписаний:

1. Создать.
2. Уничтожить.
3. Очистить.
4. Пуст ли?
5. Добавить элемент на вершину стека.
6. Взять/извлечь элемента с вершины стека.

Этот набор операций, собственно, и определяет структуру данных *стек*.

Простейшая реализация стека на базе массива фиксированного размера выглядит следующим образом:

```
1  template<class T,int m=10> class StackFixed
2  {T v[m]; size_t n=0; static T tmp;
3  public:
4  void clear () {n=0;} //Очистить
```

```

5  size_t size() {return n;}
6  bool empty() {return n==0;} //Пуст ли?
7  bool filled() {return n>=m;} //!!! Заполнен ли?
8  void push(const T&x) //Добавление элемента
9  { if (!filled()) v[n++] = x; /* else что-то можно сделать */ }
10 void pop() //Извлечение элемента
11 { if (!empty()) n--; /* else ... */ }
12 T&top() //Взятие элемента с вершины стека
13 { if (!empty()) {return v[n-1];} /* else что-то... */ return tmp; }
14 };
15 template<class T, int m> T StackFixed<T, m>::tmp;

```

Поскольку данная реализация основана на массиве фиксированной длины (задается через параметры шаблона), то к стандартным предписаниям добавляется предписание, проверяющее, заполнен ли массив элементов стека (filled()). В случае пустого стека метод top() возвращает ссылку на статическую переменную tmp.

Легко реализовать стек на основе уже созданного шаблона Vector:

```

1  #pragma once
2  #include "Vector"
3  template<class T> class Stack
4  { Vector<T> v;
5  public:
6  void clear() {v.clear();}
7  size_t size() {return v.size();}
8  bool empty() {return v.size()==0;}
9  void push(const T&x) {v.push_back(x);}
10 void pop() { if (!empty()) {v.resize(v.size()-1);} /* else ... */ }
11 T&top() {return v.back();}
12 };

```

Для использования шаблона stack из STL необходимо включить в файл с программой на языке C++ соответствующий include-файл:

```

1  #include<stack>

```

Использование данного шаблона ничем не отличается от созданного нами шаблона Stack:

```
1 {stack<int> v;  
2   for (int i=0;i<10;i++){v.push(i);}  
3   while (!v.empty()){cout<<v.top()<<" ";v.pop();}  
4 }
```

Если при сборке программы, реализующей цикл

```
1 stack<int> v;  
2 for (int i=0;i<10000;i++){v.push(i);}
```

подключить также файл `malloc.cpp` (`malloc.o`), то мы сразу увидим, что стек реализован так же, как и описанный выше `deque`. Поэтому говорят, что `stack` является *адаптером* шаблона `deque`.

8.2. STL. Очередь. Реализация очереди

Очередью называется структура данных, организованная по принципу FIFO — first in, first out, т. е. элемент, попавший в множество первым, должен первым его покинуть.

Создание исполнителя *очередь* предполагает наличие следующих предписаний:

1. Создать.
2. Уничтожить.
3. Очистить.
4. Пуста ли?
5. Добавление элемента в конец очереди.
6. Взятие/извлечение элемента из начала очереди.

Этот набор операций, собственно, и определяет структуру данных *очередь*.

В STL структуре данных *очередь* соответствует контейнер `queue`, организованный тоже как адаптер контейнера `deque`. Основные методы добавления элемента, удаления элемента, взятия элемента из начала очереди прогнозируемо имеют имена `push`, `pop`, `front`. Также доступен метод `back` для получения значения элемента из конца очереди:

```
1 #include<iostream>  
2 #include<queue>  
3 using namespace std;
```

```

4  int main(void)
5  {queue<int> v;
6   for(int i=0;i<10;i++)v.push(i);
7   cout<<v.front()<<" "<<v.back()<<endl;
8   while(!v.empty()){cout<<v.front()<<" "; v.pop();}cout<<endl;
9   return 0;
10 }
```

Заметим, что методы `front` и `back` контейнера `queue` делегируют вызовы соответствующим методам базового контейнера `deque`.

Стандартной самостоятельной реализацией структуры данных *очередь* является так называемая *циклическая очередь* на основе массива. Мы можем реализовать класс `Queue` как шаблон, параметризованный типом переменных, хранящихся в очереди, и количеством элементов в массиве, в котором будет храниться очередь:

```

1  template<class T,int n=10> class Queue
2  {T v[n]; int fr=n-1,bk=n-1;
3   int next(int&i){if(i==0)i=n-1;else i--; return i;}
4  public:
5   struct err{string s; err(const char*ss){s=ss;}};
6   void push(const T&x){if(!filled()){v[bk]=x;next(bk);}
7   /*else throw err("push:filled");//можно выбросить искл.*/}
8   T &front(){if(empty())throw err("front:empty");
9   else return v[fr];}
10  void pop(){if(empty())throw err("pop:empty");else next(fr);}
11  bool empty(){return fr==bk;}
12  bool filled(){return (fr+1)%n==bk;}
13  };
```

В данной реализации мы создаем отдельный приватный метод `next(int&x)`, который заменяет значение индекса в массиве на его следующее значение (вычисляемое циклически), причём движение осуществляется от больших индексов к меньшим. Индекс `fr` указывает на начало очереди, а `bk` — на элемент, предшествующий концу очереди. Поэтому новый элемент мы заносим по индексу `bk`, а извлекаем элемент по индексу `fr` (в обоих случаях индекс затем меняется на `next`). Извлечение элемента производится двумя инструкциями: `front()` (получение значения) и `pop()` (уничтожение первого элемента

очереди). Отметим, что при этой реализации один элемент в массиве останется неиспользованным, поскольку иначе мы не смогли бы отличить пустую очередь от заполненной. Легко увидеть, что если этот недостаток окажется существенным, то легко ввести в класс целую переменную `n`, содержащую количество элементов в очереди. С использованием этой переменной легко запишутся методы `empty()` и `filled()`. Однако тогда придется вставлять инкрементацию/декрементацию переменной `n` в функции `push()/pop()`.

Тест на использование данного класса можно написать в следующем виде:

```
1 #include<iostream>
2 #include"Queue"
3 #include<cstring>
4 using namespace std;
5 int main(void)
6 {
7     {Queue<int> v;
8         for (int i=0;i<10;i++)v.push(i);
9         cout<<v.front()<<" "<<v.back()<<endl;
10        while (!v.empty()) {cout<<v.front()<<" ";v.pop();} cout<<endl;
11    }
12    return 0;
13 }
```

Этот же тест без изменений будет работать для реализации очереди в STL:

```
1 #include<iostream>
2 #include<queue>
3 using namespace std;
4 int main(void)
5 {
6     {queue<int> v;
7         for (int i=0;i<10;i++)v.push(i);
8         cout<<v.front()<<" "<<v.back()<<endl;
9         while (!v.empty()) {cout<<v.front()<<" ";v.pop();} cout<<endl;
10    }
11    return 0;
12 }
```

Легко увидеть, что в тесте для нашей реализации очереди один заносимый элемент не поместится в очереди, а в очереди STL ограничений на длину нет. Вообще, именно реализация очереди на базе дека STL позволяет для сложных объектов, хранящихся в контейнере, относительно безболезненно расширять массив, ассоциированный с очередью. Это связано с тем, что при добавлении элементов в дек адреса объектов контейнера не меняются.

8.3. STL. Дек. Реализация дека

Деком (от англ. *deque* — double-ended queue, двусторонняя очередь) называется абстрактная структура данных, представляющая собой последовательность элементов, где можно добавлять и удалять элементы как с начала, так и с конца последовательности. По сути, дек обладает набором предписаний как стека (элементы можно извлекать в порядке добавления), так и очереди (элементы можно извлекать в порядке, обратном порядку добавления).

Наиболее простой является реализация дека опять же на основе массива фиксированной длины. И здесь опять напрашивается реализация на основе циклического дека (по аналогии с циклической очередью). Данная реализация довольно проста, но (поэтому) у нее есть существенное преимущество перед реализацией дека STL, заключающееся в ее эффективности. Работа с деком на основе массива фиксированной длины оказывается существенно более быстрой по сравнению с работой с деком STL, поэтому приводимая далее реализация оказывается вполне конкурентоспособной в случае повышенных требований к быстродействию программы.

```

1  template<class T, int n=10> class Deque
2  {T v[n]; int fr=n-1, bk=n-1;
3  public:
4      struct err{string s; err(const char*ss){s=ss;}};
5      void push_back(const T&x){//занести элемент в хвост:
6          if(!filled())v[bk]=x;next(bk);}else//можем игнорировать:
7              if(0)throw err("push_back:filled");}
8      void push_front(const T&x){//занести элемент в голову:
9          if(!filled()){prev(fr);v[fr]=x;}else//можем игнорировать:
10             if(0)throw err("push_front:filled");}
11     T &front(){if(empty())throw err("front:empty");else

```

```

12         return v[fr];} //начало дека
13 T &back() {if(empty())throw err("back:empty"); else
14         return v[(bk+1)%n];} //хвост дека
15 void pop_front() {if(empty())throw err("pop_front:empty");
16         else next(fr);} //уничтожить голову дека
17 void pop_back() {if(empty())throw err("pop_back:empty");
18         else prev(bk);} //уничтожить хвост дека
19 bool empty() {return fr==bk;} //пуст?
20 bool filled() {return (fr+1)%n==bk;} //заполнен?
21 int next(int&x) {if(x==0)x=n-1;else x--;return x;} //к след-му
22 int prev(int&x) {x=(x+1)%n; return x;} //-> к предыдущему
23 };

```

Здесь условные операторы `if(0)` указывают на то, что их наличие/отсутствие зависит от целей нашей реализации: контейнер при невозможности выполнения соответствующих операций может либо игнорировать данную неприятность, либо (если убрать операторы `if(0)`) будет выбрасывать исключение.

Следующий тест для дека STL эмулирует работу с очередью:

```

1 #include<iostream>
2 #include<deque>
3 using namespace std;
4 int main(void)
5 {deque<int> v; for(int i=0;i<10;i++)v.push_back(i);
6  cout<<v.front()<<" "<<v.back()<<endl;
7  while(!v.empty())
8  {cout<<v.front()<<" ";v.pop_front();} cout<<endl;
9  return 0;
10 }

```

Простая замена в тесте слова `deque` на `Deque` и строки `#include<deque>` на строку `#include"Deque"` создаст аналогичный тест для нашей реализации циклического дека.

8.4. C++. `initializer_list`. `range-based for`

В языке C существует довольно удобный способ инициализации массива путем перечисления значений через запятую в фигурных скобках:

```
1 int m[]={1,2,3,4,5,6};
```

Здесь размер массива определяется количеством элементов в фигурных скобках. Если задать размер массива явно, то начальные элементы будут инициализированы заданными значениями, остальные — нулями. Было бы весьма удобным использовать подобную инициализацию для подходящих контейнеров. В языке C++ 11-й версии такая возможность появилась. Для этого используется `initializer_list`.

Несмотря на то, что конструкция `initializer_list` используется как шаблон, для которого определен стандартный итератор, данная сущность является внутренней операцией языка. Физически объект этого типа обычно реализуется как пара, состоящая из указателя на отведенный компилятором кусок памяти с данными и количества элементов по данному адресу. Передавать такой объект в функцию следует по значению. Из-за небольшого размера описанной пары процедура передачи данных оказывается весьма эффективной.

Рассмотрим простой пример заполнения и распечатки созданного нами ранее класса `Vector` и стандартного контейнера STL `vector`:

```
1 #include<iostream>
2 #include<vector>
3 #include<cstdlib>
4 #include<cstring>
5 using namespace std;
6 #include "Vector"
7 //—————
8 int main(void)
9 {
10 { int m[]={1,2,3,4,5,6}; Vector<int> v;
11 for (size_t i=0;i<sizeof(m)/sizeof(*m);i++)v.push_back(m[ i ] );
12   for (int x:v){cout<<x<<" ";}cout<<endl;
13 }
14 { int m[]={1,2,3,4,5,6}; vector<int> v;
15 for (size_t i=0;i<sizeof(m)/sizeof(*m);i++)v.push_back(m[ i ] );
16   for (int x:v){cout<<x<<" ";}cout<<endl;
17 }
18 return 0;
19 }
```

Для распечатки значений векторов здесь использован *range-based for*, появившийся в C++11. Данный вид оператора **for** можно применять для обычных массивов и для объектов, для которых определен стандартный итератор `iterator`. А поскольку мы определили для шаблона `Vector` все возможные итераторы, то и *range-based for* автоматически стал доступным для нашего вектора. Собственно, с этого момента простые договоренности о виде итератора в стиле STL становятся необходимой потребностью языка C++, поскольку стандартные итераторы начинают использоваться в конструкциях языка.

Следует отличать конструкцию `for(int x:v)` или `for(auto x:v)` от вариантов цикла `for(int &x:v)` или `for(auto &x:v)`. В первом случае цикл идет по копиям значений подобъектов объекта `v`, а втором — по самим подобъектам, т. е. во втором случае, в отличие от первого, можно изменять значения подобъектов, например путем присваивания:

```
1 for (int &x:v)x=0;
```

Добавим конструктор класса `Vector`, в который значения передаются через `initializer_list` :

```
1 Vector(initializer_list<T> l):Vector()  
2 {for(auto it=l.begin();it!=l.end();++it)push_back(*it);}
```

Тогда инициализация нашего вектора примет весьма простую форму, аналогичную возможной инициализации контейнера STL `vector`:

```
1 {vector<int> v={1,2,3,4,5,6};  
2 for(auto &x:v){cout<<x<<" ";}cout<<endl;  
3 }  
4 {Vector<int> v={1,2,3,4,5,6};  
5 for(auto &x:v){cout<<x<<" ";}cout<<endl;  
6 }
```

Особенностью *range-based for* является то, что данный цикл преобразуется к форме с участием итератора, причем необходимо используются методы базового класса с именами `begin()` и `end()`. Это верно даже в том случае, когда обслуживаемый объект имеет спецификатор `const`. В результате код

```

1 const Vector<int> v; //может прийти в виде параметра функции
2 for (const auto &x:v) {cout<<x<<" ";} cout<<endl;

```

скомпилируется с ошибками, так как `Vector::iterator` нельзя использовать для константного объекта. Ситуацию легко исправить, добавив пару методов к содержимому шаблона класса `Vector`:

```

1 const_iterator begin() const{return const_iterator(this,0);}
2 const_iterator end() const{return const_iterator(this,n);}

```

Поскольку имеет место полиморфизм, компилятор позволит это сделать и использует данные функции в разворачивании *range-based for* для константного объекта. Тогда последний пример будет интерпретирован следующим образом:

```

1 const Vector<int> v; //может прийти в виде параметра функции
2 for (Vector<int>::const_iterator i=v.begin(); i!=v.end(); ++i)
3 {cout<<*i<<" ";} cout<<endl;

```

8.5. STL. Итераторы

Опишем коротко основные виды итераторов, используемые в STL. Мы уже говорили об итераторах, но пришло время упорядочить понимание данной сущности. В этом разделе в примерах встретятся контейнеры и алгоритмы, которые мы будем рассматривать позднее. Так получилось потому, что STL весьма сложно разбирать последовательно по шагам. Между различными областями библиотеки шаблонов — контейнерами, алгоритмами, функциональными объектами, итераторами, потоками — много взаимозависимостей. Поэтому в идеале надо разбирать эту библиотеку, идя параллельно по всем ее компонентам, что практически нереально в стандартном процессе обучения. Чтобы лучше разобраться в STL, весьма полезно прочитать [15] — классическую книгу в данной области.

Далее в данном разделе под *итераторами* мы будем понимать *STL-итераторы*.

Для работы со специфическими итераторами необходимо включить в исходный код include-файл `iterator`.

Максимально упрощенный подход

В максимально упрощенном виде итераторы являются обобщением/заменой указателей, используемых вне STL, поэтому работа с итератором часто напоминает работу с обычным указателем. В реальности итератор является классом, в котором (скорее всего) содержится только указатель на объект, хранящийся в контейнере, с переопределенными операциями, необходимыми для работы с данным объектом. Некоторым типам итераторов приходится хранить еще некоторые дополнительные данные.

Итераторы не используются для работы со стеком, очередью и приоритетной очередью в силу специфики этих структур данных (в классическом понимании).

Существует несколько разновидностей итераторов, но пока не будем заострять на этом внимания. Для каждого контейнера существует свой тип итератора, который можно использовать для работы с данным контейнером: `vector::iterator`, `list::iterator`, `deque::iterator` и т. д. Они имеют почти одинаковый набор функций, что унифицирует работу с итераторами.

Следует помнить, что итераторы (как указатели на объекты в контейнере) в общем случае можно использовать после получения их значения, только пока контейнер не изменялся. После изменения контейнера полученный ранее итератор становится неопределенным. Например, такая ситуация складывается с итераторами контейнера `set` (поскольку данный контейнер реализуется с помощью красно-черного дерева), но, например, для контейнеров `deque` и `list` в случае, если элемент, содержащийся в контейнере, жив, то итератор, указывающий на него, останется рабочим, несмотря на изменения в множестве. Такое поведение непосредственно вытекает из особенностей реализации контейнеров, которые мы подробно разбираем в нашем курсе.

В большинстве контейнеров STL вводится некоторая упорядоченность элементов. Упорядоченность задается либо местом элемента в последовательности данных (как в массиве), либо порядком, задаваемым оператором `<`, который должен быть определен для элементов хранимого в контейнере множества. В таких контейнерах для элементов корректны понятия *первый*, *последний*, *следующий*, *предыдущий*.

В большинстве контейнеров STL (в классе контейнеров) содержатся следующие функции, возвращающие итераторы, указывающие на соответствующие элементы текущего контейнера (читай: псевдоуказатели на объекты, хранящиеся в текущем контейнере):

`iterator begin()` — возвращает итератор, указывающий на первый элемент в контейнере;

`iterator end()` — возвращает итератор, указывающий на элемент, следующий после последнего элемента в контейнере;

`reverse_iterator rbegin()` — возвращает итератор, указывающий на первый элемент в контейнере для обратного перебора элементов, т. е. на последний элемент в контейнере;

`reverse_iterator rend()` — возвращает итератор, указывающий на элемент, следующий после последнего элемента в контейнере при обратном переборе элементов, т. е. на элемент, стоящий перед первым элементом в контейнере;

`const_iterator cbegin()` — возвращает итератор, указывающий на первый элемент в константном контейнере;

`const_iterator cend()` — возвращает итератор, указывающий на элемент, следующий после последнего элемента в константном контейнере;

`const_reverse_iterator crbegin()` — возвращает итератор, указывающий на первый элемент в константном контейнере при обратном переборе элементов;

`const_reverse_iterator crend()` — возвращает итератор, указывающий на элемент, следующий после последнего элемента в константном контейнере при обратном переборе элементов.

К итераторам применима префиксная операция `++`, возвращающая итератор, указывающий на следующий элемент в контейнере (постфиксная операция `++` тоже может существовать, но это нечто более сложное).

К итераторам применима префиксная операция `*`, возвращающая значение элемента данных, содержащегося в элементе контейнера, на который указывает итератор.

Приведем пример перебора элементов контейнер `set`.

```
1 set<int> v; set<int>::iterator it;
2 v.insert(2);v.insert(1);v.insert(3);
3 v.insert(4);v.insert(4);
```

```

4 for ( it=v.begin () ; it!=v.end (); ++it )
5   cout<<*it<<" ";

```

Итераторы ввода

Все, что было сказано в предыдущем разделе, относится к *итераторам ввода*. Под *вводом* имеется в виду ввод данных из контейнера в окружающую программу (например, ввод данных из контейнера требуется при выводе содержимого контейнера на экран). Полный список операций для итераторов различных типов можно найти, например, в [12]. Итераторы ввода поддерживают перемещение вперед по элементам контейнера и дают возможность получить значение элемента в контейнере без права его изменения. Итераторы ввода можно присваивать друг другу и сравнивать на равенство/неравенство. Пример:

```

1  istream_iterator<int> itIn , itEnd=istream_iterator<int>();
   int m[10] , n;
2  ostream_iterator<int> itOut ( cout , " " );
3  itIn=cin ;
4  for ( n=0; itIn!=itEnd; ++itIn , n++) m[n]=*itIn ;
5  for_each ( m, m+n , print ) ; cout<<endl ;

```

Стандартный алгоритм STL `for_each` применяет функцию `print` для каждого значения, получаемого с помощью итератора ввода от `m` до (не включая) `m+n`. Фактически данный алгоритм STL заменяет оператор цикла. Здесь и далее функция `print()` должна быть определена в программе следующим образом:

```

1  void print ( int v ) { cout<<v<<" " ; }

```

Итераторы вывода

Итераторы вывода не дают возможности для сравнения итераторов, но для них можно использовать оператор `*` слева от знака присваивания.

```

1  vector<int> v ; v.resize ( 10 ) ; size_t n ;
2  istream_iterator<int> it ; //итератор ввода для привязки к cin

```

```

3 ostream_iterator<int> itOut(cout, " "); // итератор вывода
4 for (it=cin, n=0; it!=istream_iterator<int>(); ++it, n++) v[n]=*it;
5 for (vector<int>::iterator it=v.begin(); it!=v.end(); ++it)
6     *itOut=*it;
7 cout<<endl;

```

Здесь итератор ввода `it` привязывается к потоку ввода `cin` и с его помощью происходит передача элементов типа `int`, получаемых из стандартного потока ввода (с клавиатуры), элементам массива `v[n]`. Итератор вывода `itOut` привязывает к потоку вывода `cout`. После этого присваивание элементу `*itOut` значения целой переменной приводит к выводу этого значения на экран с последующим символом-разделителем пробелом.

Последовательные итераторы

Последовательные итераторы объединяют возможности итераторов ввода и вывода.

Двунаправленные итераторы

Двунаправленные итераторы поддерживают все свойства последовательных итераторов плюс операцию `--`. Например, итераторы ассоциативных контейнеров являются двунаправленными:

```

1 multimap<int, double> v; multimap<int, double>::iterator it;
2 v.insert(pair<int, int>(1,1)); v.insert(pair<int, int>(2,2));
3 v.insert(pair<int, int>(3,40)); v.insert(pair<int, int>(4,30));
4 v.insert(pair<int, int>(4,40));
5 it=v.find(4); cout<<"[4]="<<it->second<<endl;
6 --it; cout<<"[3]="<<it->second<<endl;

```

Итераторы произвольного доступа

К свойствам двунаправленного итератора добавляются возможности прибавления/вычитания целого числа и оператор `[]`. Например, итератор вектора (`vector`) является итератором произвольного доступа.

```
1 vector<int> v; vector<int>::iterator it; v.resize(10);
2 for(it=v.begin(); it<v.end(); it+=2)*it=(it-v.begin());
3 for_each(v.begin(), v.end(), print);
```

Итераторы вставки

Все перечисленные итераторы позволяют перебирать и модифицировать элементы контейнеров, но для списков (о которых речь пойдет в следующей главе) требуется дополнительная операция вставки, поэтому пришлось создать итератор вставки. Достаточно рассмотреть итераторы вставки в начало объекта, в хвост объекта и на определенную позицию объекта. Начнем с того, что полегче.

`inserter` — функция, возвращающая итератор вставки на определенную позицию.

`back_inserter` — функция, возвращающая итератор вставки в хвост контейнера.

`front_inserter` — функция, возвращающая итератор вставки в голову контейнера.

Пример использования:

```
1 #include<iostream>
2 #include<algorithm>
3 #include<list>
4 #include<iterator>
5 using namespace std;
6 int main(void)
7 {int x[]={1,2,3,4,5}; size_t n=sizeof(x)/sizeof(x[0]);
8   list<int> l; list<int>::iterator it;
9   copy(x,x+n, front_inserter<list<int>>(l)); //вставка в начало
10  copy(x,x+n, back_inserter<list<int>>(l)); //вставка в конец
11  it=l.begin(); ++it; copy(x,x+n, inserter<list<int>>(l, it));
12    //вставляем со второй позиции (после первого элемента)
13  copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
14  cout<<endl; return 0;
15 }
```

Здесь стандартный алгоритм STL `copy` все элементы от итератора ввода из первого аргумента до итератора ввода из второго аргумента копирует в итератор из третьего аргумента.

Поясним, что происходит в предыдущем примере. В нем в трех случаях `copy` используются не итераторы, а функции, возвращающие итераторы, причем возвращаемый из них итератор передается в алгоритм по значению. Внутри алгоритмов `copy` происходят:

присваивание значения разадресованному итератору вставки в начало контейнера, приводящее к вставке этого значения в начало контейнера;

присваивание значения разадресованному итератору вставки в конец контейнера, приводящее к вставке этого значения в конец контейнера;

присваивание значения разадресованному итератору вставки на определенную позицию контейнера, приводящее к вставке этого значения в данную позицию и перемещению итератора на позицию после вставленного элемента.

Таким образом, первый алгоритм `copy` создаст список 5 4 3 2 1, поскольку каждая вставка будет происходить в начало списка, второй — дополнит его до списка 5 4 3 2 1 1 2 3 4 5, а третий последовательно вставит элементы после первого элемента списка (каждая вставка будет перемещать итератор вставки на позицию после вставленного элемента). Мы получим следующий вывод на экран:

```
1 | 5 1 2 3 4 5 4 3 2 1 1 2 3 4 5
```

В следующем примере итераторы используются напрямую (это выглядит несколько сложнее, но сути не меняет):

```
1 | int x[]={1,2,3,4,5}; list<int> l;
2 | list<int>::iterator it;
3 | front_insert_iterator<list<int>> itFront(l);
4 | copy(x,x+sizeof(x)/sizeof(x[0]), itFront);
5 | back_insert_iterator<list<int>> itBack(l);
6 | copy(x,x+sizeof(x)/sizeof(x[0]), itBack); //первый вывод:
7 | copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
8 | cout<<endl;
9 | insert_iterator<list<int>> itAtPlace(l,++l.begin());
10| copy(x,x+sizeof(x)/sizeof(x[0]), itAtPlace); //второй вывод:
```

```
11 copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));  
12 cout<<endl;
```

В результате мы получим следующий вывод на экран:

```
1 5 4 3 2 1 1 2 3 4 5  
2 5 1 2 3 4 5 4 3 2 1 1 2 3 4 5
```

Надо помнить, что итераторы вставки используются только для вставки. Эта фраза подразумевает, что, например, в последнем примере операция `++itAtPlace` не изменит позиции итератора (хотя синтаксически не приведет к ошибке!).

Не удастся использовать итераторы вставки для вектора и ассоциативных контейнеров (хотя, функция `insert` для вектора существует), но их можно использовать для дека.

9. Структуры данных *списки*. Списки в STL

Под *списком* понимается структура данных, которая строится на понятиях *текущего элемента* или *текущего положения между двумя элементами*. Под *элементом* здесь подразумевается некоторая сущность, внутри которой хранится один элемент множества, с которым работает наша структура данных (напомним, что каждая структура данных работает с некоторым множеством элементов определенного типа). Иногда элементы списка называют *вершинами списка*. Все элементы в списке имеют некоторую упорядоченность, и в каждый момент времени возможно получение информации и модификация/добавление/удаление элементов только в непосредственной близости от текущего элемента/положения. За одну операцию возможно перемещение текущего элемента только либо в следующее положение (к следующему элементу или к следующему положению), либо к предыдущему элементу/положению. Если в последовательности элементов списка имеется начало или конец, то за одну операцию возможно перемещение текущего элемента соответственно в начало или конец списка.

Как видно из общего описания *списка*, можно определять различные структуры данных, удовлетворяющие различным вариантам описания. Так, сразу же стоит отдельно рассматривать списки с *текущим элементом* и с *текущим положением* (между двумя соседними элементами списка).

Обычно рассматривают следующие виды списков.

Однонаправленные (L1-списки): возможно перемещение только к следующему элементу/положению; задаются только начало списка и текущий элемент/положение.

Двунаправленные (L2-списки): возможно перемещение к следующему и предыдущему элементу/положению; задаются начало и конец списка.

Циклические (C-списки): возможно перемещение текущего элемента/положения либо в одну, либо в две стороны; список завязан в кольцо, у которого нет ни начала, ни конца.

Визуально список проще всего представлять себе как бусы с ручкой, держащей одну бусину, или промежуток между соседними бусинами. Каждая бусина здесь представляет собой одну вершину списка. Бусы могут быть замкнуты в кольцо, а могут представлять со-

бой последовательность, у которой есть начало и конец. Во втором случае есть либо одна рука, держащая начало бус, либо есть еще одна рука, держащая конец бус. Какие-либо действия с бусами можно производить только первой рукой: ею можно перебирать бусины, изменять бусины, которые она держит (в том числе удалять их и добавлять новые около текущего положения), перемещать первую руку ко второй или третьей руке, если они есть.

Стандартной (далеко не простейшей!) реальной реализацией списка является обычный текстовый файл при работе с ним в текстовом редакторе. Фактически в этом случае файл представляется пользователю двунаправленным списком строк. Среди строк есть текущая, ее можно редактировать. По списку строк можно перемещаться с помощью соответствующих клавиш (по большому счету, только к предыдущей или к следующей строке). Строки около текущей можно редактировать, добавлять, удалять. Более сложные операции с текстовым файлом представляются как комбинации трех основных указанных действий.

Отметим, что до сих пор мы, в принципе, ничего не говорили о конкретных реализациях списка, оставаясь на уровне простого перебора возможных операций со списком (разговор про бусы не в счет). Как правило, обсуждая списки, имеют в виду *ссылочную реализацию* списка. Для ссылочной реализации принято в каждой вершине списка хранить один элемент данных исходного множества и одну или две ссылки, на следующую или еще на предыдущую вершину списка. В случае списка с текущим элементом для всего списка хранится ссылка на текущую вершину списка. Для случая списка с текущим положением приходится хранить две ссылки на соседние вершины списка. В зависимости от вида списка (L1, L2, C) для всего списка также хранятся либо две ссылки, на начало и конец списка, либо только ссылка на первый элемент списка. Никакой дополнительной информации о списке не хранится.

Важно понимать, что ссылочная реализация списка является только одной из возможных. И тут на сцену выходит *бесссылочная реализация* списка, которая стандартно представляет собой два стека, вершины которых направлены навстречу друг другу. Вершины стеков представляют собой элементы вокруг текущего положения списка (как вариант, вершину первого стека можно считать теку-

щим элементом). Перемещение по списку к следующему/предыдущему положению осуществляется перебрасыванием вершины одного стека в другой. Возможна модификация вершин стеков.

Отметим, что возможна весьма простая реализация стека на обычном компьютере: обычный файл с возможностью дописывания данных в конец, модификацией конца файла, обрезания куса данных в конце файла. Таким образом может быть реализован текстовый редактор, который не требует загрузки всего редактируемого файла в оперативную память. В данной реализации потребуются работа с двумя рабочими файлами: в одном будут лежать строки начала редактируемого файла, а в другом — строки конца файла в обратном порядке. Перемещение курсора по строкам редактируемого файла будет реализовано как перебрасывание строки из конца одного рабочего файла в конец другого. Отметим, что здесь за рамки разговора выходит сам способ реализации файловой системы в целом, но это уже тема другой беседы. Безусловно, обычные операции работы с файлами не предусматривают изменения длин строк внутри рабочих файлов, но нам это и не требуется, так как модификация рабочих файлов будет происходить только в их концах. Можно предъявить претензию к неустойчивости всей системы в целом: при падении программы редактирования файлов все данные пропадут, но способ загрузки всего файла в оперативную память (которым, как правило, пользуются все редакторы) ничем в этом плане не лучше. На данном принципе был основан старый добрый редактор *Micromir*, или просто *MIM*, который очень любило старшее поколение сотрудников мехмата. Увы, из-за проблем совместимости он не дожил до наших дней, хотя и сейчас его легко скачать и запускать в эмуляторе старого MS-DOS. Некоторых очень полезных особенностей данного редактора критически не хватает во всех ныне существующих текстовых редакторах.

Выделим несколько конкретных разновидностей структур данных *списки*, для которых существуют удобные реализации и приведем набор предписаний для каждой такой структуры данных. Под 1, 2, 3, 4, 5 имеется в виду набор стандартных предписаний для любой структуры данных.

1. Создать.
2. Удалить.
3. Очистить.
4. Пуст ли?
5. Количество элементов.

Последнее предписание можно считать стандартным с некоторой натяжкой, но оно вошло в стандарт всех контейнеров STL, о чем пойдёт речь позднее. Итак, приведем набор предписаний для основных используемых разновидностей списков.

L1-список с текущим элементом

- 1, 2, 3, 4, 5.
6. Встать в начало.
7. Перейти к следующему элементу.
8. Уничтожить следующий элемент.
9. Получить значение текущего элемента.
10. Добавить элемент после текущего элемента.

L2-список с текущим положением

- 1, 2, 3, 4, 5.
6. Встать в начало (перед первым элементом).
7. Встать в конец (после последнего элемента).
8. Перейти к следующему положению.
9. Перейти к предыдущему положению.
10. Уничтожить следующий элемент.
11. Уничтожить предыдущий элемент.
12. Вставить элемент в текущее положение.
13. Получить значение следующего элемента.
14. Получить значение предыдущего элемента.

L2-список с текущим элементом

- 1, 2, 3, 4, 5.
6. Встать в начало.
7. Встать в конец.
8. Перейти к следующему элементу.
9. Перейти к предыдущему элементу.
10. Уничтожить текущий элемент.
11. Вставить элемент после текущего элемента.
12. Вставить элемент перед текущим элементом.
13. Получить значение текущего элемента.

Циклический список с текущим элементом (C-L1)

1, 2, 3, 4, 5.

6. Перейти к следующему элементу.
7. Уничтожить следующий элемент.
8. Вставить элемент после текущего элемента.
9. Получить значение текущего элемента.

Циклический список с текущим элементом (C-L2)

1, 2, 3, 4, 5.

6. Перейти к следующему элементу.
7. Перейти к предыдущему элементу.
8. Уничтожить текущий элемент.
9. Вставить элемент после текущего элемента.
10. Вставить элемент перед текущим элементом.
11. Получить значение текущего элемента.

Безусловно, в каждой реализации каждому предписанию должна соответствовать эффективная функция, выполняющая предписание.

Реализацию L1-списка с текущим элементом (в виде списка с фиктивным элементом) мы рассматривали в предыдущей книге [11], поэтому не будем на ней останавливаться.

9.1. L2-список с текущим положением

Для L2-списка с текущим положением существует стандартная бессмысленная реализация в виде двух стеков, о которой мы говорили ранее. Воспользуемся стандартными контейнерами `stack` из STL. Поскольку в качестве данных внутри шаблона класса списка будут выступать только полноценные классы `stack<T> b,e`, мы получим класс, организованный по правилу нуля. Для него не нужны никакие конструкторы, деструкторы, операторы присваивания. Кроме методов, соответствующих предписаниям класса, добавим для удобства пару новых методов, позволяющих определять, есть ли элементы до и после текущего положения (точнее, проверяющих отсутствие таких элементов): `empty_begin()`, `empty_end()` .

`include`-файл, содержащий определение шаблона класса, примет весьма простой вид:

```
1 #pragma once
2 #include<stack>
```

```
3 using namespace std;
4 template<class T> class L2S
5 {stack<T> b,e;
6 public:
7     bool empty() {return b.empty()&&e.empty();}
8     bool empty_begin() {return b.empty();}
9     bool empty_end() {return e.empty();}
10    void clear() {b.clear();e.clear();}
11    void insert_before(const T&x){b.push(x);}
12    void insert_after(const T&x){e.push(x);}
13    void GoToBegin(){while(GoToPrev());}
14    void GoToEnd(){while(GoToNext());}
15    bool GetBefore(T&v)
16    {if(empty_begin())return false;else{v=b.top();return
17     true;}}
18    bool GetAfter(T&v)
19    {if(empty_end())return false;else{v=e.top();return true;}}
20    bool DelBefore()
21    {if(empty_begin())return false;else{b.pop();return true;}}
22    bool DelAfter()
23    {if(empty_end())return false;else{e.pop();return true;}}
24    bool GoToNext()
25    {if(empty_end())return false;
26     else{b.push(e.top());e.pop();return true;}}
27    bool GoToPrev()
28    {if(empty_begin())return false;
29     else{e.push(b.top());b.pop();return true;}}
};
```

Соответствие между методами класса и предписаниями легко определить по названиям методов.

9.2. L2-список с текущим элементом

Для L2-списка с текущим элементом существует опять же стандартная ссылочная реализация. На языках C/C++ она предполагает создание двух новых типов данных: для элемента списка (будем называть соответствующий тип *Node*) и для самого списка (тип *List*). На языке C++ структуру элемента списка удобно спрятать в структу-

ру всего списка. Более того, описание структуры Node надо сделать приватным, чтобы никто из внешнего для данного класса мира не увидел бы внутренней кухни реализации всей структуры данных. Из этого следует, что, когда мы будем говорить о добавлении *элемента в список*, под *элементом* будем иметь в виду не вершину списка, а элемент множества, хранящегося в используемой структуре данных.

В списке будут храниться три указателя на вершины списка Node *b,*e,*cur, соответствующие первому, последнему, текущему элементам списка.

В основе реализации будет лежать простая договоренность: для пустого списка все три указателя внутри списка должны быть нулевыми. Для непустого списка все три указателя должны быть ненулевыми. Например, это предполагает, что для непустого списка всегда определен текущий элемент списка.

Для работы с исключениями создадим внутри класса структуру типа err, внутри которой будем хранить строку с информацией об ошибке. В случае когда возвращение кода ошибки из метода невозможно (пока это насущно только для метода получения значения текущего элемента списка Get()), именно структуру этого типа будем выбрасывать через исключение. Договоримся, что там, где это возможно, вместо кода ошибки будем из соответствующих функций возвращать **true** в случае успеха функции и **false** в случае невозможности успешного выполнения функции. Безусловно, подобное смешение стилей работы с ошибками является признаком дурного тона, но в реальной жизни оно может оказаться весьма удобным (в силу вынужденного, надо признать, неудобства стиля C++ работы с ошибками).

Имеет смысл немного изменить имена методов ранее определенного класса списка, и тогда реализация списка будет иметь следующий вид:

```

1 #include<string>
2 using namespace std;
3 template<class T> class List
4 {struct Node{Node *next,*prev; T v;
5     Node(const T&x){v=x;next=prev=nullptr;}};
6 public:
7     struct err{string s; err(const char*ss){s=ss;}};

```

```

8 Node *b,*e,*cur;
9 List(){b=e=cur=nullptr;}
10 ~List(){while(Del());}
11 bool empty(){return cur==nullptr;}
12 bool empty_begin(){return cur==nullptr || cur->prev==nullptr;}
13 bool empty_end(){return cur==nullptr || cur->next==nullptr;}
14 void insert_before(const T&x){Node *n=new Node(x);
15     if(empty())cur=b=e=n;
16     else {n->prev=cur->prev;n->next=cur;
17         if(n->prev)n->prev->next=n; else b=n;
18         n->next->prev=n; cur=n;}}
19 void insert_after(const T&x){Node *n=new Node(x);
20     if(empty())cur=b=e=n;
21     else {n->prev=cur;n->next=cur->next;
22         n->prev->next=n;
23         if(n->next)n->next->prev=n; else e=n; cur=n;}}
24 void GoToBegin(){cur=b;}
25 void GoToEnd(){cur=e;}
26 T &Get(){if(empty())throw err("Get:empty");
27         else {return cur->v;}}
28 bool Del(){if(empty())return false; else {Node*t=cur;
29     if(t->prev){t->prev->next=t->next; cur=t->prev;}
30     else b=cur=t->next;
31     if(t->next)t->next->prev=t->prev; else e=t->prev;
32     delete t; return true;}}
33 bool GoToNext(){if(empty_end())return false;
34     else {cur=cur->next; return true;}}
35 bool GoToPrev(){if(empty_begin())return false;
36     else {cur=cur->prev; return true;}}
37 };

```

Здесь мы сохранили, несколько изменив, методы `empty_begin()` и `empty_end()`, полезные в данной реализации. По сравнению с предыдущей реализацией списка здесь пришлось добавить очистку памяти. Для упрощения реализации класс не доведен до согласованности с правилом трех.

Безусловно, единственными реально нетривиальными методами данной реализации являются функции вставки и удаления элементов. Здесь надо иметь в виду элементарное требование аккуратности

при работе с языком С (даже не С++): если используется значение, взятое по адресу некоторого указателя, то предварительно надо обязательно проверить, что данный указатель ненулевой. Надо обратить внимание на то, что при вставке элементов в список новый элемент становится текущим элементом списка (такое поведение позволяет легко реализовать итератор вставки, с которым мы знакомы по работе с STL-итераторами). При удалении элемента списка необходимо переместить указатель на текущий элемент на вершину списка, соседнюю с удаляемой. Если же ее нет (список пуст), то указатель `cur` должен стать нулевым. И при вставке, и при удалении элемента надо учесть, что могут измениться указатели на первый и последний элементы списка.

У данной реализации есть одно неочевидное на первый взгляд неудобство: нет возможности эстетично написать цикл перебора элементов списка. Простейший пример заполнения элементов списка и их распечатки может иметь следующий вид:

```

1 List<int> l; int m[]={1,2,3,4,5,6};
2 for(int x:m)l.insert_before(x);
3 try{
4 if(!l.empty()){l.GoToBegin();
5 do{cout<<l.Get()<<" ";}while(l.GoToNext());} cout<<endl;
6 }catch(List<int>::err v){cout<<"error:"<<v.s<<endl;}
```

Поскольку элементы вставляются до текущего, порядок чисел в списке будет обратным по сравнению с исходным массивом.

Постараемся свести созданный нами список к реализации, совместимой с набором методов для стандартного списка STL. Мы должны принять во внимание различие подходов к понятию *списка* стандартной *структуры данных* «список» и *контейнера* «список» в STL. Разница заключается в том, что в структуре данных *список* перебор множества осуществляется внутренними ресурсами списка (что согласуется, например, с бессмысленной реализацией списка в виде двух стеков), в то время как в списках STL перебор элементов реализуется исключительно с помощью итераторов (надо иметь в виду, что использование итераторов весьма удобно, но далеко не всегда его можно спроектировать на физическую реализацию списка: в том же приведенном выше примере с реализацией текстового редактора есть только одно текущее положение и нескольких итераторов вве-

сти нельзя). Вообще, в списках STL текущего элемента как такового нет. Вместо этого используются итераторы, что ломает наше представление о списках, так как можно использовать параллельно сразу несколько итераторов для работы с одним списком. Безусловно, это ничему не противоречит, так как списки в STL — всего лишь конкретная реализация общей структуры данных *список*. А это, в свою очередь, позволяет использовать абсолютно любой набор дополнительных предписаний, лишь бы они имели достаточно эффективную реализацию.

Мы уже работали с итераторами в стиле STL ранее, поэтому для нас не составит проблем добавить в шаблон списка структуру обычного итератора и функции `begin()` и `end()` для работы с данными типа *итератор*:

```

1 struct iterator{List *l=nullptr; Node*n=nullptr;
2   iterator(List*ll=nullptr,Node*nn=nullptr){l=ll;n=nn;}
3   bool operator!=(const iterator&b)const{return n!=b.n;}
4   iterator operator++(){if(n)n=n->next; return *this;}
5   iterator operator++(int t)
6     {t=t; iterator it=*this; if(n)n=n->next; return it;}
7   T&operator*(){if(l&&l->cur==nullptr)throw err("*:empty");
8     if(n==nullptr)throw err("*:is not initialized");
9     else return n->v;}
10 };
11   iterator begin(){return iterator(this,b);}
12   iterator end(){return iterator(this,nullptr);}

```

Это позволит нам более изящно записать предыдущий пример бегой проверки работоспособности написанной реализации списка:

```

1 List<int> l; int m[]={1,2,3,4,5,6};
2   for(int x:m)l.insert_before(x);
3   for(List<int>::iterator it=l.begin();it!=l.end();++it)
4     {cout<<*it<<" "; cout<<endl;

```

Здесь ничто не мешает нам заменить тип `List<int>::iterator` на `auto`, что сократит код, но несколько усложнит его анализ в дальнейшем (когда мы банально забудем, что возвращает `l.begin()`). Более того, поскольку мы определили стандартный итератор для нашего списка,

нам доступен *range-based for*. Тогда наш пример можно упростить еще:

```

1 List<int> l; int m[]={1,2,3,4,5,6};
2   for (int x:m) l.insert_before(x);
3   for (int x:l) {cout<<x<<" ";} cout<<endl;

```

Наконец, добавив инициализацию списка с помощью конструктора с использованием `initializer_list`

```

1 List(initializer_list<T> l):List()
2 {for (auto &x:l) insert_after(x);}

```

мы получим еще более короткий код теста:

```

1 List<int> l={1,2,3,4,5,6};
2   for (int x:l) {cout<<x<<" ";} cout<<endl;

```

9.3. STL. Списки

Для созданного нами шаблона списка `List` было бы весьма полезным задать несколько дополнительных методов и определений:

```

1   using value_type=T;
2   iterator insert(List::iterator it, const T&x)
3   {cur=it.n; insert_before(x); return iterator(this, cur);}
4   iterator push_front(const T&x)
5   {cur=b; insert_before(x); return iterator(this, cur);}
6   iterator push_back(const T&x)
7   {cur=e; insert_after(x); return iterator(this, cur);}

```

Конструкция `using` в данном контексте является заменой понятного нам по языку C определения типа

```

1 typedef int value_type;

```

Данное определение понадобится в дальнейшем для итераторов вставки. Можно вспомнить, что оно входило в список требований к реализации итераторов, которые были сформулированы в разделе *STL. Реализация вектора. Стандартные требования к итераторам STL*. Однако здесь будет достаточно только этого определения.

Созданные методы являются аналогами одноименных методов контейнера STL `list`. Теперь мы можем вызывать эти методы и для нашего шаблона `List` и для контейнера STL `list`, например:

```
1 #include<iostream>
2 #include<list>
3 using namespace std;
4 #include "List"
5 int main(void)
6 {
7     {List<int> l; int m[]={1,2,3,4,5,6};
8     for(int x:m)l.push_back(x);
9     for(int x:l){cout<<x<<" ";} cout<<endl;
10 }
11 {list<int> l; int m[]={1,2,3,4,5,6};
12 for(int x:m)l.push_back(x);
13 for(int x:l){cout<<x<<" ";} cout<<endl;
14 }
15 return 0;
16 }
```

Отметим, что метод `push_back()` доступен также для контейнеров `vector` и `deque`. Поэтому контейнер `list` в этом примере можно безболезненно (с точки зрения синтаксиса) заменить на контейнер `vector` или `deque`:

```
1 #include<iostream>
2 #include<vector>
3 #include<deque>
4 using namespace std;
5 int main(void)
6 {
7     {vector<int> l; int m[]={1,2,3,4,5,6};
8     for(int x:m)l.push_back(x);
9     for(int x:l){cout<<x<<" ";} cout<<endl;
10 }
11 {deque<int> l; int m[]={1,2,3,4,5,6};
12 for(int x:m)l.push_back(x);
13 for(int x:l){cout<<x<<" ";} cout<<endl;
14 }
```

```

15 return 0;
16 }

```

Метод `insert` также доступен для контейнеров `vector` и `deque`. Но метод `push_front` доступен только для контейнера `deque` (здесь имеет смысл вспомнить разницу в реализации этих контейнеров в STL).

Мы уже проходили итераторы вставки STL, осталось разобраться, как сделать их доступными для созданного нами типа. Для стандартного итератора вставки STL `insert_iterator` необходим метод `insert()`. Поскольку мы реализовали этот метод с правильной сигнатурой, `insert_iterator` можно применять и для стандартного контейнера STL `list`, и для нашего шаблона `List`:

```

1 #include<iostream>
2 #include<list>
3 #include<iterator>
4 #include<cstring>
5 using namespace std;
6 #include "List"
7 int main(void)
8 {
9     { list<int> l; int m[]={1,2,3,4,5};
10     for (auto x:m)
11         { insert_iterator<list<int>> it(l,l.end()); *it=x;}
12     for (auto x:l){cout<<x<<" "; cout<<endl;
13     }
14     { List<int> l; int m[]={1,2,3,4,5};
15     for (auto x:m)
16         { insert_iterator<List<int>> it(l,l.begin()); *it=x;}
17     for (auto x:l){cout<<x<<" "; cout<<endl;
18     }
19     return 0;
20 }

```

Определение `value_type` необходимо для того же итератора вставки, чтобы добраться до типа данных, которым параметризуется контейнер, тип которого передается в качестве шаблона списка (итератор видит тип контейнера как параметр шаблона, но не видит тип, которым параметризуется контейнер).

Поскольку можно использовать итератор вставки, в той же функции `main()` можно использовать шаблон функции `inserter`, возвращающий итератор вставки:

```

1  {List<int> l; int m[]={1,2,3,4,5};
2  for(auto x:m)*inserter<List<int>>(l,l.begin())=x;
3  for(auto x:l){cout<<x<<" ";} cout<<endl;
4  }
5  {list<int> l; int m[]={1,2,3,4,5};
6  for(auto x:m)*inserter<list<int>>(l,l.begin())=x;
7  for(auto x:l){cout<<x<<" ";} cout<<endl;
8  }
9  return 0;
10 }
```

Описание шаблона функции `inserter` и итераторов вставки было дано ранее.

Наконец, метод `push_front` позволяет использовать итератор вставки `front_insert_iterator` и шаблон функции `front_inserter`, а метод `push_back` позволяет использовать итератор вставки `back_insert_iterator` и шаблон функции `back_inserter`:

```

1  {list<int> l; int m[]={1,2,3,4,5};
2  for(auto x:m){front_insert_iterator<list<int>>it(l);*it=x;}
3  for(auto x:l){cout<<x<<" ";} cout<<endl;
4  }
5  {List<int> l; int m[]={1,2,3,4,5};
6  for(auto x:m){front_insert_iterator<List<int>>it(l);*it=x;}
7  for(auto x:l){cout<<x<<" ";} cout<<endl;
8  }
9  {list<int> l; int m[]={1,2,3,4,5};
10 for(auto x:m){back_insert_iterator<list<int>>it(l); *it=x;}
11 for(auto x:l){cout<<x<<" ";} cout<<endl;
12 }
13 {List<int> l; int m[]={1,2,3,4,5};
14 for(auto x:m){back_insert_iterator<List<int>>it(l); *it=x;}
15 for(auto x:l){cout<<x<<" ";} cout<<endl;
16 }
17 {list<int> l; int m[]={1,2,3,4,5};
18 for(auto x:m){*front_inserter<list<int>>(l)=x;}
```

```

19  for(auto x:1){cout<<x<<" ";} cout<<endl;
20  }
21  {List<int> l; int m[]={1,2,3,4,5};
22  for(auto x:m){*front_inserter<List<int>>(l)=x;}
23  for(auto x:1){cout<<x<<" ";} cout<<endl;
24  }
25  {list<int> l; int m[]={1,2,3,4,5};
26  for(auto x:m){*back_inserter<list<int>>(l)=x;}
27  for(auto x:1){cout<<x<<" ";} cout<<endl;
28  }
29  {List<int> l; int m[]={1,2,3,4,5};
30  for(auto x:m){*back_inserter<List<int>>(l)=x;}
31  for(auto x:1){cout<<x<<" ";} cout<<endl;
32  }

```

9.4. Циклический список с текущим положением

Рассмотрим реализацию L1-циклического списка с текущим положением. Она оказывается довольно изящной.

Содержательно в данной реализации есть всего две нетривиальные операции: вставка элемента после текущего и уничтожение элемента после текущего. К нашему удовольствию, эти операции имеют весьма простую реализацию:

```

1  #pragma once
2  #include<string>
3  template<class T> class List1
4  {struct Node{Node *next;T v;
5   Node(const T&x,Node *n){v=x;next=(n?n:this);}};
6   Node *cur=nullptr; size_t sz=0;
7  public:
8   struct err{string s;err(const char*ss){s=ss;}};
9   ~List1(){clear();}
10  bool empty(){return cur==nullptr;}
11  void clear(){while(DelNext());}
12  size_t size(){return sz;}
13  void InsertAfter(const T&x)
14  {if(empty())cur=new Node(x,nullptr); else
15   {cur->next=new Node(x,cur->next);} sz++;}

```

```

16  bool DelNext() {if(empty()) {return false;} sz--;
17      if(cur==cur->next){delete cur; cur=nullptr;} else
18      {Node *n=cur->next; cur->next=cur->next->next; delete n;}
19      return true;}
20  T &Get() {if(empty())throw err("Get:empty");return cur->v;}
21  bool GoToNext() {if(empty())return false; cur=cur->next;
22      return true;}
23  };

```

Здесь мы добавили элемент класса *sz*, в котором будет храниться текущее количество элементов в списке, и соответствующий метод *size()*. Безусловно, это надо было делать и для предыдущих реализаций списков (по аналогии с STL, где метод *size()* присутствует во всех контейнерах). Отсутствие данных объектов в предыдущих реализациях можно объяснить только нашей ленью. Однако для данного класса хранить количество элементов списка станет насущной необходимостью, речь об этом пойдет дальше.

Обратим внимание на реализацию конструктора

```

1  Node(const T&x, Node *n) ...

```

здесь *n* должен указывать на следующий элемент списка, но если *n* имеет нулевое значение, то следующим элементом списка делается текущий элемент (**this**). Было бы весьма удобно описать конструктор сразу в виде

```

1  Node(const T&x, Node *n=this) ...

```

но правила конструирования объектов и обработки значений по умолчанию не позволяют нам это сделать.

Простой тест для такого списка имеет вид:

```

1  #include<iostream>
2  using namespace std;
3  #include "List1"
4  int main(void)
5  {List1<int> l; l.InsertAfter(1);l.InsertAfter(2);
6    cout<<l.Get()<<endl; l.GoToNext();
7    cout<<l.Get()<<endl; l.GoToNext();
8    return 0;
9  }

```

Мы не можем реализовать итератор для нашего класса по аналогии с предыдущей реализацией итератора, поскольку в данной реализации списка банально нет начала. Запоминание элемента, с которого мы начнем перебор элементов списка, тоже нас не спасет. Действительно, в этом случае метод `begin()` должен возвращать итератор, в котором будет храниться ссылка на текущий элемент списка. Но тогда и метод `end()` должен возвращать итератор, в котором хранится все тот же элемент списка (здесь мы даже не обращаем внимания на то, что метод `begin()` вызывается в цикле всего один раз (и позволяет запомнить ссылку на начало перебора элементов цикла), а метод `end()` вызывается много раз и при этом должен хранить ссылку на один и тот же последний перебираемый элемент списка, что создает проблемы при реализации). Получается, что условие завершения цикла итератора будет выполняться сразу на первом же элементе цикла, что не позволит произвести цикл по всем элементам списка. Однако в данной реализации мы храним количество элементов в списке `sz`, поэтому повторим цикл с помощью итератора ровно `sz` раз. Итератор и методы `begin()` и `end()` в этом случае будут иметь вид:

```

1  struct iterator
2  {Node *cur; size_t i,n;
3   iterator(List1*1, size_t j){cur=l->cur;i=j;n=l->size();}
4   bool operator!=(const iterator&b)const{return i!=b.i;}
5   iterator operator++(){i++;if(cur)cur=cur->next;return
   *this;}
6   iterator operator++(int t)
7   {t=t;iterator it(*this);i++;if(cur)cur=cur->next;return it;}
8   T &operator*(){return cur->v;}
9  };
10 iterator begin(){return iterator(this,0);}
11 iterator end(){return iterator(this,sz);}

```

Фактически, стандартный цикл итератора превращается в обычный цикл

```

1  for (size_t i=0;i<n;i++)...

```

Теперь нам доступен перебор элементов списка с помощью стандартного итератора, а следовательно, и с помощью *range-based for*:

```

1  #include<iostream>
2  using namespace std;
3  #include "List1"
4  int main(void)
5  { List1<int> l; int m[]={1,2,3,4,5};
6    for(auto x:m)l.InsertAfter(x);
7    for(auto it=l.begin();it!=l.end();++it)
8      {cout<<*it<<" ";} cout<<endl;
9    for(auto x:l){cout<<x<<" ";} cout<<endl;
10   return 0;
11  }

```

Добавление в тело шаблона списка пары описанных ранее инструкций

```

1  using value_type=T;
2  iterator insert(List1::iterator it, const T&x)
3  {cur=it.cur; InsertAfter(x); return iterator(this,0);}

```

позволит использовать для нашего списка также и итератор вставки:

```

1  #include<iostream>
2  #include<iterator>
3  using namespace std;
4  #include "List1"
5  int main(void)
6  { List1<int> l; int m[]={1,2,3,4,5};
7    for(auto x:m)*inserter<List1<int>>(l,l.begin())=x;
8    for(auto x:l){cout<<x<<" ";} cout<<endl;
9    return 0;
10  }

```

Безусловно, слово `begin` особого содержательного смысла здесь не несет и служит лишь для совместимости с STL.

9.5. L1-список с собственным отведением памяти

Вернемся еще раз к ссылочной реализации однонаправленного списка с фиктивным элементом:

```

1  template<class T> class SList1
2  {struct SNode
3    {T v; SNode *next; SNode(){next=nullptr;}
4    SNode(const T&v){this->v=v; next=nullptr;}};
5  SNode b,*cur; int n;//b - фиктивный эл-т, cur - текущий эл-т
6  SList1 () {cur=&b;n=0;}
7  ~SList1 () {clear ();}
8  void clear () {GoToBegin (); while(DelNext ());}
9  int size () {return n;}
10 bool empty () {return n==0;}
11 void GoToBegin () {cur=&b;} //cur всегда != nullptr
12 bool GoToNext () {if(cur->next){cur=cur->next; return true;}
13     return false;}
14 bool Get(T&v){if(cur==&b){return false;} v=cur->v;
15     return true;}
16 void insert (const T&v){SNode *n=new SNode(v);
17     n->next=cur->next;cur->next=n; this->n++;}
18 bool find (const T&v){T x; for(GoToBegin ();GoToNext ();)
19     {Get(x); if(x==v)return true;} return false;}
20 bool DelNext ()
21 {if(cur->next==nullptr)return false; SNode *n=cur->next;
22     cur->next=cur->next->next; delete n; this->n--; return
23     true;}
};

```

Пора обратить внимание на ужасную расточительность любой обычной ссылочной реализации списка как в плане расхода оперативной памяти (для каждого элемента множества приходится выделять память под отдельную структуру, в которой еще хранится указатель; более того, каждое отведение памяти в языках C/C++ обычно приводит к реальному захватыванию оперативной памяти большего, чем требуется, размера), так и времени добавления/удаления элементов (операции с памятью достаточно медленны). Из-за указанных проблем при профессиональной работе со списками (а также с деревьями, речь о которых пойдет позднее) желательно использовать собственные процедуры отведения/очистки памяти, которые хотя бы частично нивелируют указанные недостатки.

Стандартный подход к решению данной проблемы заключается в создании массива вершин списка (элементов типа `SList1::SNode`), элементы которого изначально связываются в структуру данных *список* (это легко сделать естественным путем: каждый элемент ссылается на следующий, причем в каждой вершине списка уже присутствует указатель, который можно использовать для реализации данного списка). Данный список элементов типа `SList1::SNode`, лежащих в массиве, объявляется *списком свободного места*. Теперь операция отведения памяти под вершину списка заменяется операцией извлечения вершины списка из списка свободного места, после чего вершину можно использовать в нашем основном списке `SList1`. Операцию освобождения памяти, отведенной под вершину списка, следует заменить операцией возвращения данной вершины в список свободного места. Таким образом, в нашем списке `SList1` будет фактически присутствовать два списка: основной, с которым идет работа, и отдельный список свободного места. Перебрасывания элементов между двумя списками можно осуществлять довольно быстро, причем память будет отводиться только под сами элементы списка, никакой дополнительной служебной памяти отводиться не будет.

В данной реализации проблема возникнет в тот момент, когда список свободного места окажется исчерпанным. Тогда необходимо переотвести память под весь массив элементов и связать дополнительные элементы нового массива в новый список свободного места. Но при переотведении памяти все указатели, используемые в основном списке, станут недействительными. Их несложно скорректировать (зная, что адреса всех элементов в массиве свободного места изменились на одну и ту же константу), но мы поступим по-другому. Поскольку теперь в нашем списке используются только элементы из массива свободного места, мы можем заменить указатели на вершины списка на их индексы в массиве свободного места. Тогда при переотведении памяти индексы останутся действительными и никакой коррекции для них не потребуется.

Теперь заголовок класса списка вместе с описанием вершины списка приобретет следующий вид:

```
1  template<class T> class SList1
2  {struct SNode{ T v; size_t next; SNode(){next=0;}}
3    SNode(const T&v){this->v=v; next=0;}};
```

```

4 size_t cur; int n; //cur=индекс тек. эл-та в списке св. места
5 SNode *f; size_t nf=100; //массив для списка свободного места

```

Конструктор списка, определение которого следует за заголовком, усложнится. В нем придется создавать массив вершин и связывать его в список свободного места с фиктивным элементом `f[0]`. Заметим, что теперь мы не можем, как ранее, завести обычный элемент класса `SList1` в качестве фиктивного элемента рабочего списка, так как мы постулировали, что все элементы рабочего списка будут поступать из списка свободного места. Поэтому выделим элемент массива `f[1]` как фиктивный элемент рабочего списка.

```

6 SList1 () { cur=1;n=0;f=new SNode[nf];
7           for (size_t i=0;i<nf;i++){ f[i].next=i+1;}
8           f[0].next=2;f[1].next=0;f[nf-1].next=0;}

```

Здесь мы, фактически, создаем однонаправленный список свободного места с фиктивным элементом. В качестве фиктивного элемента используется первый элемент массива `f[0]`. Последний элемент массива ссылается на элемент с индексом 0. Тогда признаком исчерпания списка свободного места будет выполнение условия `f[0].next==0`. Как уже говорилось, элемент массива `f[1]` выступает фиктивным элементом рабочего списка, что отражается в том, что в начальный момент номер текущего элемента равен 1 (номеру фиктивного элемента).

Нам придется создать новые функции, заменяющие операторы создания и уничтожения элемента списка. Вместо использования оператора `new SNode(v)` будем вызывать функцию `NewSNode(v)`, а вместо очистки памяти оператором `delete n` будем вызывать функцию `Delete(n)`. Реализация этих функций довольно проста и сводится к обычной работе с однонаправленным списком. Единственная проблема, как уже говорилось, возникает при исчерпании списка свободного места. Тогда придется переотводить память и связывать новые элементы списка в новый список:

```

9 size_t NewSNode(const T&x)
10 {size_t i=f[0].next; //инд. св. места
11  if (i==0){SNode *ff=new SNode[2*nf]; //если массив исчерпан
12   for (size_t i=0;i<nf;i++)ff[i]=f[i];
13   delete [] f; f=ff; //создаем новый массив и переносим старый
14   for (size_t i=nf;i<2*nf;i++)f[i].next=i+1; //новый список

```

```

15   f[0].next=i; f[2*nf-1].next=0; i=nf; nf*=2;
16       }//извлекаем вершину из списка св.места:
17   f[0].next=f[i].next; f[i].next=0; f[i].v=x;
18   return i;
19   }

```

Функция Delete(n) определена ниже.

Оставшаяся часть класса почти полностью повторяет код класса SList1 с обычным отведением памяти (с учетом замены nullptr на 0 и операторов отведения/очистки памяти на новые функции отведения/очистки памяти):

```

19   ~SList1() {clear();}
20   void clear() {GoToBegin(); while(DelNext());}
21   int size() {return n;}
22   bool empty() {return n==0;}
23   void GoToBegin() {cur=1;}
24   bool GoToNext() {if(f[cur].next)
25   {cur=f[cur].next; return true;} return false;}
26   bool Get(T&v)
27   {if(cur==1){return false;} v=f[cur].v; return true;}
28   void insert(const T&v)
29   {size_t n=NewSNode(v);
30   f[n].next=f[cur].next; f[cur].next=n; this->n++;
31   }
32   bool find(const T&v){T x; for(GoToBegin();GoToNext();)
33   {Get(x);if(x==v)return true;} return false;}
34   bool DelNext()
35   {if(f[cur].next==0)return false; size_t n=f[cur].next;
36   f[cur].next=f[f[cur].next].next; Delete(n); this->n--;
37   return true;
38   }
39   size_t NewSNode(const T&x)
40   {size_t i=f[0].next;
41   f[0].next=f[i].next; f[i].next=0; f[i].v=x; return i;
42   }
43   void Delete(size_t i){f[i].next=f[0].next; f[0].next=i;}
44   };

```

Единственный момент, который может раздражать в данной реализации — это отсутствие реальной очистки памяти при уменьшении размера рабочего списка. Можно либо игнорировать данную неприятность (в случае, когда нас интересует только объем захватываемой памяти в худшем случае), либо добавить функцию «компактификации» массива со списком свободной памяти, когда n станет сильно меньше nf .

10. C++. Исключения. Структурированные исключения Microsoft

Как уже обсуждалось в [11], простейший вид цикла выполнения команды состоит из трех этапов:

- 1) считывание команды по адресу из счетчика команд в процессор;
- 2) увеличение счетчика команд на размер команды с ее аргументами (каждая команда со своими аргументами занимает сколько-то байт);
- 3) выполнение команды.

На самом деле в существующих ЭВМ этот цикл более сложный (см. [14]). К указанным пунктам надо, как минимум, добавить проверку наличия запроса на прерывание и, при наличии запроса, вызов процедуры обслуживания прерывания. Под *прерыванием* понимается приостановка обычного выполнения программы (в любой момент, когда разрешено выполнение прерываний) и выполнение некоторой специально назначенной процедуры. В процессе выполнения такой процедуры прерывания обычно запрещаются. Прерывания бывают *аппаратными* и *программными*. Аппаратные прерывания происходят в результате каких-либо аппаратных событий (например, нажатия на клавишу на клавиатуре). Способ выдачи запроса на прерывания при этом очень сильно зависит от архитектуры ЭВМ. Программные прерывания назначаются при запросе на прерывание непосредственно из программы каким-то специфическим программным вызовом. Мы не будем вдаваться в подробности этих действий в связи с их сильной аппаратной и системной зависимостью, но добавим к указанным трем пунктам четвертый:

- 4) проверка запросов на прерывание и выполнение прерываний.

Выполнение прерываний заложено в самой основе любой современной ЭВМ. Они позволяют программе не сосредотачиваться на обслуживании каких-то редко происходящих событий, а просто заниматься своими текущими делами или вообще уходить в сон и не тратить при этом процессорное время. И только при возникновении прерываний будут вызываться процедуры их обслуживания. Классическим примером использования исключений являются процедуры асинхронного ввода/вывода. Мы не будем приводить конкретного синтаксиса, поскольку он сильно зависит от версии языка и опера-

ционной системы. Суть асинхронного ввода/вывода состоит в том, что после вызова операции практически сразу происходит выход из нее, а собственно ввод/вывод происходит параллельно с дальнейшим выполнением программы. Когда операция ввода/вывода будет завершена, программа получит уведомление об этом через механизм прерываний.

Можно сказать, что исключения в языках C, C++, Python являются высокоуровневым аналогом прерываний.

Компания Microsoft в своих компиляторах C/C++ ввела отдельное понятие *структурированных исключений Microsoft* (SEH, Structured Exception Handling). SEH имеет исключительно системно-зависимую реализацию и используются только в OS Windows. Особый синтаксис SEH (по сравнению с обычными исключениями C++) подчеркивает принципиальное отличие данной сущности от стандартных исключений C++. Отметим, что SEH можно использовать и в языке C, и в языке C++. Самым главным отличием SEH от стандартных исключений C/C++ является возможность программного выбора дальнейшего поведения после выбрасывания структурированного исключения (=прерывания) и ловли его в инструкции ловли исключения (аналогичной инструкции стандартного исключения C++). После ловли исключения программа может выбрать один из трех вариантов дальнейших действий:

1) выполнить процедуру обработки исключения по аналогии с обработкой стандартного исключения C++;

2) пробросить исключение дальше (в надежде, что его поймает кто-то еще);

3) попробовать исправить причину выброса исключения и вернуться в точку, в которой произошло исключение (в надежде, что теперь все будет хорошо).

При этом структурированные исключения Microsoft могут ловить аппаратные исключения, чего не умеют стандартные исключения C++, например в компиляторе gcc. Зато в OS UNIX (Linux) существует понятие *сигналов* и в языке C есть функция `signal()`, с помощью которой возможно назначение функций обработки для определенных сигналов. Например, после вызова функции

```
1 signal(SIGFPE, handler);
```

можно надеяться, что при целочисленном делении на ноль (посылке системой сигнала SIGFPE) будет происходить автоматический вызов функции `handler()`. Безусловно, конкретная реакция программы на деление на ноль может отличаться в различных системах на различных архитектурах ЭВМ.

До конца этой главы для компиляции примеров будем использовать только компилятор Microsoft.

Рассмотрим элементарную программу подсчета гармонического среднего последовательности целых чисел, читаемых из файла `t.txt`:

```

1 #include<stdio.h>
2 double av=0; int n=0,x;
3 int main(void)
4 {FILE *f; f=fopen("t.txt","r");
5  while(fscanf(f,"%d",&x)==1)if(x){av+=1./x; n++;}
6  av=1/(av/n); printf("av=%lg\n",av);
7  fclose(f);
8  return 0;
9 }
```

Содержательный код данной программы (если не брать в расчет способ получения последовательности чисел) довольно короткий:

```

1 if(x){av+=1./x; n++;}
```

Разумно желание исключить проверку деления на 0 из данного кода.

Компилятор Microsoft в режиме работы по умолчанию вполне разрешает делить на 0 вещественные числа. Это ничему не противоречит, так как в представлении вещественных чисел с плавающей точкой предусмотрено значение $\pm\infty$ (см. [11]). Можно попробовать заменить данное деление на целочисленное деление, например, таким способом:

```

1 if(x){av+=1000000000/x; n++;}
```

Результат такого деления будет иметь относительную погрешность $1.e-6$ для чисел, по модулю не превосходящих 1000, что хуже относительной погрешности представления чисел `double`, но лучше относительной погрешности представления чисел `float`. Тогда результат легко получить по формуле

```
1 av=1000000000/(((double)av)/n);
```

Избавимся от проверки деления на 0 в содержательной части программы, т. е. сведем ее к виду

```
1 {av+=1000000000/x; n++;}
```

При делении на 0 будет происходить прерывание, которое можно ловить с помощью структурированных исключений Microsoft. SEH обслуживаются инструкциями `__try` и `__except`. Инструкция `__except` имеет параметр, в котором можно записать выражение или вызвать функцию, и блок обработки исключения по аналогии с исключением C++. Нам придется добавить пару `include`-файлов, и конечный код будет выглядеть следующим образом:

```
1 #include<stdio.h>
2 #include<windows.h>
3 #include<except.h>
4 double av=0; int n=0,x, K=1000000000;
5 //—
6 int ErrFun(int code)
7 {
8     if(code==EXCEPTION_INT_DIVIDE_BY_ZERO)
9     {
10         x=K; av-=1; n--;
11         return -1;
12     }
13     return 1;
14 }
15 //—
16 int main(void)
17 {FILE *f; f=fopen("t.txt","r");
18     __try{
19         while(fscanf(f,"%d",&x)==1){av+=K/x; n++;}
20     }__except(ErrFun(GetExceptionCode())){printf("exception\n");}
21     av=K/(av*1./n); printf("av=%lg\n",av);
22     fclose(f);
23     return 0;
24 }
```

Здесь инструкция `__except` предполагает в своем параметре вызов функции, возвращающей интуитивно понятное значение (конечно, вызов функции можно заменить требуемым выражением или просто константой):

–1 = вернуться после выполнения функции в точку исключения, не выполняя тело оператора `__except`;

0 = пробросить исключение дальше, не выполняя тело оператора `__except`;

1 = выполнить тело оператора `__except` и продолжить выполнение программы (полный аналог исключения C++).

Внутри выражения, вычисляемого в параметре `__except` (и только здесь!), можно использовать оператор `GetExceptionCode()`. Заметим, что данный оператор имеет вид функции, но в реальности ею не является в силу ограничения на место вызова. Данный оператор возвращает код последнего выброшенного прерывания. При целочисленном делении на 0 значение данного кода равно `EXCEPTION_INT_DIVIDE_BY_ZERO` (понятно, что данная константа определяется макроопределением, заданным в глубинах добавленных `include`-файлов).

При возникновении деления на 0 мы заменяем значение переменной `x` (именно для этого мы сделали переменную глобальной) на некоторое ненулевое значение и возвращаемся к точке программы, в которой произошло деление на 0. Нам придется выполнить операцию, нивелирующую изменение переменной `av`, которое произойдет на этом шаге:

```
1 x=K; av -=1; n--;
```

Для компилятора Microsoft можно определить характер взаимодействия обычных исключений и механизма `SEH` (по большей части связанного с блоками `catch(...)`). Инструкция `catch(...)` может обслуживать все исключения, а может перехватывать только обычные и игнорировать структурированные. По умолчанию инструкция `catch(...)` ловит все исключения, включая структурированные. Это эквивалентно компиляции файлов программы с ключом `/EHa` (здесь `a` — от слова *all*):

```
1 cc -EHa q.cpp
```

Тогда написанную программу можно заменить на следующий код:

```

1 #include<stdio.h>
2 #include<windows.h>
3 #include<except.h>
4 double av=0; int n=0,x, K=1000000000;
5 int main(void)
6 {FILE *f; f=fopen("t.txt","r"); l0 ;
7 try{
8 while(fscanf(f,"%d",&x)==1){av+=K/x; n++;}
9 }catch(...){printf("exception\n"); goto l0;}
10 av=K/(av*1./n); printf("av=%lg\n",av);
11 fclose(f);
12 return 0;
13 }
```

Здесь блок обслуживания исключения будет вызываться (в отличие от предыдущего случая), и на экране при делении на 0 будет появляться текст `exception`.

Если обслуживание структурированных исключений не требуется, то компиляцию файлов программы надо проводить с ключом `/EHs` (здесь `s` — от слова *standard*):

```
1 cc -EHs q.cpp
```

В этом случае последний приведенный вариант программы корректно работать не будет.

Стоит добавить несколько слов об инструкциях `__leave` и `__finally`, которые можно использовать при работе со структурированными исключениями. Первая из инструкций является аналогом операции `continue`. При ее выполнении происходит переход в конец блока `__try`. Инструкцию `__finally` можно написать вместо инструкции `__except()` (но не вместе с инструкцией!). Блок инструкции `__finally` будет гарантированно выполнен независимо от того, произошло исключение или нет.

В следующем примере вычисляется среднее гармоническое элементов последовательности, записанных в файле. При появлении чисел по модулю больше 1000 (слишком большая относительная ошибка) процесс вычисления прерывается (что отлавливается весьма неэф-

фективно на фоне нашей параноидальной борьбы за оптимизацию). Как и ранее, в случае деления на ноль исключение ловится и происходит исправление ситуации. Деление на нулевое n также отлавливается с помощью обработки структурированного исключения.

```
1 #include<stdio.h>
2 #include<windows.h>
3 #include<excpt.h>
4 int av=0; int n=0,x, K=10000000;
5 //—
6 int ErrFun(int code)
7 {
8     if (code==EXCEPTION_INT_DIVIDE_BY_ZERO)
9         {x=K; av--=1; n--; return -1;}
10    return 1;
11 }
12 //—
13 int main(void)
14 {FILE *f; f=fopen("t.txt","r");
15   __try{__try{__try{
16     while (fscanf(f,"%d",&x)==1)
17       {if (x>1000||x<-1000) __leave; av+=K/x; n++;}
18     }__except(ErrFun(GetExceptionCode())){printf("exception\n");}
19     }__finally{printf("av=%lg\n",K*1./(av/n));}
20     }__except(1){printf("Последовательность пуста");}
21   fclose(f);
22   return 0;
23 }
```

Надо обратить внимание на то, что активное использование структурированных исключений позволяет сделать любую программу практически непотопляемой, несмотря на постулат, говорящий о том, что в теле любой большой программы присутствуют фатальные ошибки. Тела всех более или менее крупных функций программы можно погрузить в блок `try{...}catch(...)`, разрешить использование SEH, после чего любая попытка программы упасть будет ловиться соответствующей инструкцией `catch(...)`. При обработке исключительной ситуации можно постараться сохранить текущее состояние программы и корректно выйти из нее.

К сожалению, использование структурированных исключений для исправления возникшей неприятности и возврата к точке выброса исключения (а это является потенциально самым интересным при рассмотрении SEH) не получило распространения. Связано это с тем, что оптимизатор программы обычно настолько исправляет исходный код, что программисту оказывается совершенно непонятным, что он, собственно, должен исправлять.

11. C++. Умные указатели. `weak_ptr`.

L2-список на основе умных указателей

Мы уже разбирали классическую реализацию двунаправленного списка с текущим элементом. Попробуем реализовать ту же самую структуру данных на основе умных указателей. Мы рассчитываем в качестве бонуса получить отсутствие необходимости очистки памяти. С другой стороны, как и в случае реализации L1-списков на основе умных указателей, мы имеем все шансы на возможное переполнение стека при автоматической очистке памяти всего списка (если элементов списка слишком много). Затем мы создадим полноценный итератор и введем все необходимые конструкции для обеспечения возможности работы стандартного итератора вставки, поставляемого STL.

Первое, что приходит в голову — это просто заменить обычные указатели в структуре вершины списка на умные указатели. Тогда заголовок класса всего списка примет вид:

```
1 struct Node{shared_ptr<Node> prev, next; T v;
2   Node(const T&x){v=x;}};
3   shared_ptr<Node> b, e, cur;
```

К сожалению, подобная реализация не обеспечивает автоматической очистки памяти при уничтожении всего объекта списка. Действительно, пусть список состоит всего из двух элементов. После гибели умных указателей `b`, `e`, `cur` первый элемент списка будет содержать умный указатель на второй элемент, а второй — на первый. Таким образом, счетчики ссылок на эти элементы никогда не достигнут нуля, т. е. не будет создано условия для их автоматической очистки. Стандартным способом решения проблемы циклических ссылок является использование так называемых *слабых умных указателей* — `weak_ptr`. Рождение/смерть слабых умных указателей не изменяет количества ссылок на объект, которым управляет `shared_ptr`. Чтобы использовать объект, на который указывает `weak_ptr`, нужно преобразовать `weak_ptr` в `shared_ptr` с помощью метода `lock()`.

Код заголовка класса списка мы реализуем теперь следующим способом:

```

1 struct Node{weak_ptr<Node> prev; shared_ptr<Node> next; T v;
2   Node(const T&x){v=x;}
3 };
4   shared_ptr<Node> b,e,cur;

```

Теперь после смерти указателей `b`, `e`, `cur` счетчик ссылок на первый элемент списка будет равен нулю (на него ссылается только один слабый указатель, не влияющий на счетчик ссылок), что приведет к гибели этого элемента. После этого счетчик ссылок на второй элемент обнулится, что погубит и его. Таким образом, будут вызваны деструкторы для всех элементов списка.

Теперь, если вывести за скобки все определения, связанные с итераторами, и отложить определение некоторых методов, начало `include`-файла с шаблоном класса списка примет вид:

```

1 #pragma once
2 #include<string>
3 #include<memory>
4 template<class T> class List
5 {struct Node{weak_ptr<Node> prev; shared_ptr<Node> next; T v;
6   Node(const T&x){v=x;}};
7   shared_ptr<Node> b,e,cur;
8 public:
9   struct err{string s; err(const char*ss){s=ss;}};
10  //—
11  List(){b=e=cur=nullptr;}
12  ~List(){clear();}
13  void clear(){GoToBegin(); while(Del());}
14  bool empty(){return cur==nullptr;}
15  void GoToBegin(){cur=b;}
16  void GoToEnd(){cur=e;}
17  bool GoToNext()
18  {if(empty_end())return false; else
19   {cur=cur->next; return true;}}
20  bool GoToPrev()
21  {if(empty_begin())return false; else
22   {cur=cur->prev; return true;}}
23  bool Del();
24  void insert_before(const T&x);

```

```

25 void insert_after(const T&x);
26 T &Get() {if(empty())throw err("Empty"); else return cur->v;}
27 bool empty_end(){return cur==nullptr || cur->next==nullptr;}
28 bool empty_begin(){return cur==nullptr || cur->prev==nullptr;}
29 };

```

Метод для уничтожения текущего элемента можно задать внутри include-файла следующим образом:

```

1 template<class T>
2 bool List<T>::Del(){if(empty())return false; else
3   {shared_ptr<Node>n=cur;
4     if(n->prev.lock())
5       {n->prev.lock()->next=n->next; cur=n->prev.lock();} else
6       {cur=b=n->next;}
7     if(n->next){n->next->prev=n->prev; cur=n->next;} else
8     {cur=e=n->prev.lock();}
9     return true;}}

```

В данном методе мы не заботимся об уничтожении старого текущего элемента, так как он автоматически будет уничтожен вместе со смертью умного указателя *n*.

Определение методов вставки будут иметь вид:

```

1 template<class T> void List<T>::insert_before(const T&x){
2   if(empty())b=e=cur=make_shared<Node>(x);else
3   {shared_ptr<Node>n=make_shared<Node>(x); n->next=cur;
4     n->prev=cur->prev; cur->prev=n;
5     if(n->prev.lock())n->prev.lock()->next=n; else b=n;
6     cur=n;}}
7 template<class T> void List<T>::insert_after(const T&x) {
8   if(empty())b=e=cur=make_shared<Node>(x);else
9   {shared_ptr<Node>n=make_shared<Node>(x); n->prev=cur;
10    n->next=cur->next; cur->next=n;
11    if(n->next)n->next->prev=n; else e=n;
12    cur=n;}}

```

Реализация станет полноценной, если добавить в шаблон списка определения, связанные с итераторами: сам итератор (хотя бы один простой итератор) и методы *begin()*, *end()*, *insert()*, *push_back()*, *push_front()*:

```
1  struct iterator{List *l; shared_ptr<Node>n;
2      iterator(List*l,shared_ptr<Node>n){this->l=l;this->n=n;}
3      bool operator!=(const iterator&b)const{return n!=b.n;}
4      iterator operator++(){if(n)n=n->next; return *this;}
5      iterator operator++(int t){t=t; iterator it=this;
6          if(n)n=n->next; return it;}
7      T&operator*((){if(l->cur==nullptr)throw err("*:empty");
8          else return n->v;}
9  };
10     iterator begin(){return iterator(this,b);}
11     iterator end(){return iterator(this,nullptr);}
12 //—
13     using value_type=T;
14     iterator insert(List::iterator it, const T&x)
15     {cur=it.n; insert_before(x); return iterator(this,cur);}
16     iterator push_front(const T&x)
17     {cur=b; insert_before(x); return iterator(this,cur);}
18     iterator push_back(const T&x)
19     {cur=e; insert_after(x); return iterator(this,cur);}
```

Приведенный выше тест работоспособности итераторов вставки будет пригоден и для данной реализации списка.

12. Бинарные деревья

Дадим несколько определений, которые позволят нам комфортно работать с *бинарными деревьями*.

Графом называется пара $G = (V, E)$, где V — множество элементов произвольной природы, E — семейство пар элементов из V . Множество V в дальнейшем будем называть *множеством вершин* графа или *множеством элементов*, а семейство E — *семейством ребер* графа G . Здесь стоит подчеркнуть, что ребра графа образуют именно семейство, так как в нем допустимы одинаковые пары вершин графа.

Ориентированным графом называется граф, в котором порядок элементов в ребрах существенен.

Везде далее мы будем рассматривать только конечные графы и данное уточнение будем опускать.

Ветвью графа $G = (V, E)$ называется последовательность $\{a_1, \dots, a_n\}$, где $a_i \in V$ ($1 \leq i \leq n$), $(a_i, a_{i+1}) \in E$ ($1 \leq i < n$).

Длиной ветви $\{a_1, \dots, a_n\}$ называется число n .

Будем говорить, что в графе есть *путь* от вершины a до вершины b , если найдется ветвь графа $\{a, \dots, b\}$.

Связным графом называется граф, в котором для любых $a, b \in V$ существует путь от a до b .

Будем называть путь $\{a_0, \dots, a_k\}$ ($a_i \in V$) *циклом*, если $a_0 = a_k$.

Петлей называется ребро графа, в котором вершины совпадают: $(a, a) \in E$, где $a \in V$.

Ориентированным деревом называется конечный связный ориентированный граф без циклов и петель.

Везде в данной главе, говоря про *деревья*, мы будем иметь в виду именно *ориентированные деревья*.

Для ориентированного дерева ребро (a, b) называется *входящим* в вершину b и *исходящим* из вершины a . Ориентация, введенная в граф, порождает «родственные» отношения вершин в графе, т. е. можно говорить, что для ребра (a, b) вершина a является *родителем* вершины b , а вершина b является *дочерней* вершиной для вершины a . Вершина b является *потомком* вершины a , если существует последовательность вершин дерева v_0, \dots, v_k , в которой $v_0 = a, v_k = b$ и для всех $0 \leq i < k$ (v_i, v_{i+1}) является ребром дерева.

Для ориентированного дерева существует ровно одна вершина, для которой нет входящих ребер. Будем называть такую вершину *корнем дерева*. Для остальных вершин количество входящих ребер равно 1.

Бинарным деревом называется ориентированное дерево, в котором для каждой вершины количество исходящих ребер не превосходит двух. Исходящие из вершины ребра принято называть *левым* (*left*) и, если есть, *правым* (*right*). В стиле языка С для дочерних вершин вершины дерева v будем использовать обозначения $v \rightarrow \text{left}$ и $v \rightarrow \text{right}$.

Будем называть *листьями дерева* вершины с не более чем одним исходящим ребром. Отметим, что данное определение не стандартно, но оно будет нам удобно, и мы везде будем им пользоваться.

Высотой дерева называется максимальная длина ветвей дерева.

Будем говорить, что для некоторой вершины дерева v дерево $T(v)$, состоящее из вершин — потомков v , является *поддеревом* с корневой вершиной v .

Далее будем предполагать, что вершины графа допускают операции сравнения, удовлетворяющие аксиомам из раздела *Сортировки* в [11].

Введем несколько вольные обозначения для графа T и вершины v : будем писать $T < v$, если для всех вершин x графа T выполнено $x < v$. Аналогично определим $v < T$, $T_1 < T_2$ и т. д.

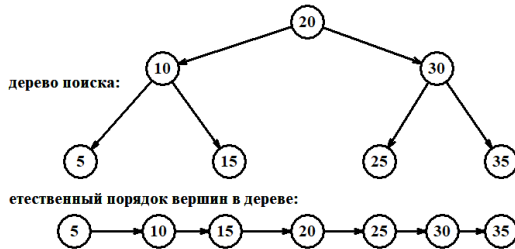
Бинарное дерево называется *деревом поиска*, если для каждой вершины $v \in V$ выполнено

$$T(v \rightarrow \text{left}) < v < T(v \rightarrow \text{right}).$$

Наряду с этим можно ввести понятие *мультидерева поиска*, для которого для каждой вершины $v \in V$ должно быть выполнено

$$T(v \rightarrow \text{left}) \leq v < T(v \rightarrow \text{right}).$$

Определение дерева поиска задает естественную упорядоченность вершин, поэтому при изображении такого дерева на рисунке имеет смысл располагать вершины дерева так, чтобы проекции вершин на горизонтальную прямую представляли бы собой упорядоченную последовательность элементов:



Далее будем определять структуру данных *дерево поиска*, для которой введем следующие предписания (см. с 142).

- 1, 2, 3, 4, 5.
6. Вставить элемент.
7. Удалить элемент.
8. Найти элемент.
9. Найти минимальный элемент.
10. Найти максимальный элемент.
11. Найти следующий элемент.
- 12.a. Объединить деревья T_1 и T_2 , такие что $T_1 < T_2$.
- 12.b. Объединить с помощью стыковочного элемента v деревья T_1 и T_2 , такие что $T_1 < v < T_2$.
13. Разбить с помощью стыковочного элемента v дерево T на деревья T_1 и T_2 , такие что $T_1 < v < T_2$ и объединение элементов деревьев T_1 и T_2 и элемента v дает множество элементов дерева T .
14. Поиск элемента по индексу.

Как всегда, первые пять предписаний стандартны: создать дерево, уничтожить дерево, очистить дерево, проверить, пусто ли дерево, получить количество элементов в дереве.

Далее будем совмещать формальные описания алгоритмов для работы с деревом поиска с описанием всего необходимого для их реализации на языке C++. Мы будем использовать имена и опираться на описания методов контейнера `set` из библиотеки STL, поскольку он реализован на основе бинарного дерева поиска. Такой подход позволит читателю не только разобраться в соответствующих алгоритмах, но и освоить интерфейс класса и в дальнейшем эффективно использовать стандартный контейнер `set` в своих проектах.

Стоит упомянуть, что контейнер `set` используется для работы с множеством, элементы которого допускают упорядочивание (для чего модификации бинарного дерева поиска очень хорошо подходят). В C++23 для работы с множествами предлагаются так называемые *плоские* контейнеры. Например, для работы со стандартным множеством предложен контейнер `flat_set`. Обладая тем же набором предписаний, что и контейнер `set`, плоское множество реализуется на основе обычного массива упорядоченных элементов. Такая реализация (далее в лекциях описание этой реализации сведется к *идеально сбалансированному дереву*) дает ощутимо более быстрый поиск элемента и экономию оперативной памяти по сравнению с бинарным деревом поиска. Однако вставка и удаление элемента оказываются принципиально более медленными.

Существует две принципиально различные ссылочные реализации вершины бинарного дерева: со ссылкой на родителя и без таковой. При отсутствии ссылки на родителя возникнут сложности с реализацией некоторых предписаний дерева. Например, алгоритм перебора вершин дерева можно будет реализовать с помощью рекурсивной функции, но это сразу наложит ограничение на количество элементов в дереве в связи с ограниченностью стека, либо придется хранить массив с ветвью дерева от корня до текущей вершины, что потребует отведения памяти в куче. Далее мы будем разбирать только первый вариант, когда для каждой вершины есть три ссылки (указателя): на левого и правого потомков вершины и на родительскую вершину. Также в классе вершины будет храниться статический указатель на вершину, необходимый в случаях неуспешного вызова метода класса дерева, возвращающего ссылку на указатель на вершину дерева. В подобных случаях выбрасывать исключение не всегда удобно из-за существенных накладных расходов при его обслуживании.

На самом деле ничто не мешает нам при невозможности возвращения из функции ссылки на реальный указатель возвращать ссылку на указатель, размещенный по нулевому адресу. Проверка «нереальности» данной ссылки будет иметь вполне предсказуемый вид:

```

1 int *&fun() {int **x=nullptr; return *x;} //возвращаем ссылку
2 int main(void) //проверяем, что адрес ссылки нулевой:
3 {if(&fun()==nullptr) cout<<"null reference"<<endl;return 0;}

```

Однако мы не будем пользоваться таким подходом, исходя из почти подсознательного ощущения его неправильности ☹.

В конечном итоге начало описания шаблона класса дерева и вершины дерева будет иметь вид:

```
1 template<class T> class Tree
2 {struct Node{Node *par,*left,*right; T v; static Node*tmp;
3   Node(const T&x,Node*p=nullptr);
4   bool IsLeft() const;
5   bool IsRight() const;
6   Node *&Par();
7 };
```

Необходимо помнить, что в реальности статический член класса является одной обычной статической переменной, которую необходимо определить (внутри класса содержится лишь описание данной переменной). Везде далее мы предполагаем, что шаблон класса дерева определяется в include-файле. Если бы мы определяли не шаблон класса, а непосредственно класс, то определение статической переменной можно было бы помещать только в исходном сpp-файле программы, поскольку переменную может быть определена только один раз. Определение переменной внутри include-файла приводило бы к ее множественному определению (*multiply definition*) при включении include-файла в несколько исходных файлов программы. Однако для шаблонов множественное определение шаблонов функций и переменных допускается (безусловно, это приведет к ситуации *multiply definition*, но сборщик закроет на это глаза, перевалив всю возможную ответственность на пользователя). Поэтому внутри include-файла отдельно от шаблона класса дерева можно поместить шаблон определения переменной tmp:

```
1 template<class T> typename Tree<T>::Node*
2   Tree<T>::Node::tmp=nullptr;
```

Внутри определения шаблона дерева зададим три основных метода. Методы IsLeft() и IsRight() будут проверять, является ли данная вершина соответственно левым и правым потомком своего родителя. При отсутствии родителя данные методы должны возвращать **false**. Метод Par() должен возвращать ссылку на указатель на данную вершину дерева, содержащийся в родителе данной вершины,

т. е. для вершины дерева v данный метод должен возвращать либо $v \rightarrow \text{par} \rightarrow \text{left}$, либо $v \rightarrow \text{par} \rightarrow \text{right}$. В случае когда родителя у вершины нет, нам как раз пригодится переменная `Tree<T>::Node::tmp`, которую в этом случае и будет возвращать метод `Par()`.

Многие методы класса дерева (например, функция поиска элемента) должны возвращать в том или ином виде ссылку на вершину дерева. Это противоречит принципу сокрытия внутренней кухни класса от глаз пользователя. Поэтому внутри класса дерева надо сразу же определить итератор, позволяющий перебирать элементы дерева или же просто служащий ссылкой на некоторый элемент дерева. Будем оформлять этот итератор в стиле STL (соответствующая структура задается внутри класса дерева):

```

1 struct iterator{Tree *t; Node*n;
2   iterator(Tree*t=nullptr,Node*n=nullptr){this->t=t;this->n=n;}
3   bool operator!=(const iterator&b)const{return n!=b.n;}
4   iterator operator++(); //определим позднее
5   T&operator*(){return n->v;}
6   operator bool()const{return n!=nullptr;}
7 };
8 iterator begin(){return Min(root);}
9 iterator end(){return iterator(this,nullptr);}

```

Метод итератора `operator++()` должен будет возвращать следующий по порядку элемент дерева (упорядоченность элементов дерева будет задавать операцией `<`, которая должна быть определена для элементов дерева, т. е. для базового типа шаблона `T`), а метод `Min(v)` — минимальный элемент поддерева с корнем v . Соответственно, метод `Min(root)` будет возвращать минимальный элемент всего дерева.

Мы определили или описали все необходимое для того, чтобы элементы класса дерева t можно было бы перебирать стандартным циклом для итератора STL, например:

```

1 for (auto it=t.begin(); it!=t.end(); ++it) {cout <<*it <<endl;}

```

Не забываем, что во всех случаях, когда это возможно, при описании методов следует указывать спецификацию `const`, чтобы данный метод можно было использовать как для экземпляра обычного класса дерева, так и для константного класса дерева.

Внутри класса дерева, кроме обычного итератора, нам надо создать как минимум `const`-итератор для перебора элементов `const`-класса дерева:

```

1  struct const_iterator{const Tree *t; const Node*n;
2     const_iterator(const Tree*t=nullptr, const Node*n=nullptr)
3         {this->t=t; this->n=n;}
4  bool operator!=(const const_iterator&b)const{return n!=b.n;}
5     const_iterator operator++();
6     const T&operator*(){return n->v;}
7     operator bool()const{return n!=nullptr;}
8 };
9  const_iterator cbegin()const{return Min(root);}
10 const_iterator cend()const
11     {return const_iterator(this, nullptr);}
12 const_iterator begin()const{return Min(root);}
13 const_iterator end()const//для range-based for const вектора
14     {return const_iterator(this, nullptr);}

```

Последние два метода пригодятся нам для обслуживания `range-based for` в случае перебора элементов вектора со спецификацией `const`. Отметим, что для константного контейнера `begin()` и `end()` возвращают константные итераторы, но `cbegin()` и `cend()` полезны для явного выражения намерений и для написания шаблонов.

Безусловно, для полноценной жизни необходимы также `reverse`-итератор и `const-reverse`-итератор, но мы обойдемся без них, поскольку наши цели сейчас другие.

Минимальный элемент дерева ищется с помощью прохождения по левой ветви дерева от корня до вершины, у которой не будет левого потомка. Понятно, что минимальная вершина дерева является листом (у нее либо вообще нет потомков, либо есть только правый потомок). Будет логично, если метод класса дерева `Min()` будет возвращать итератор, указывающий на соответствующую вершину (метод задается вне класса дерева; описание методов внутри класса мы явно пропишем позднее).

```

1  template<class T> typename Tree<T>::iterator
2      Tree<T>::Min(typename Tree<T>::Node *r) //для внутр.
        целей
3  { if(r==nullptr) return iterator(this);
4      while(r->left)r=r->left;
5      return iterator(this,r);
6  }
7  template<class T> typename Tree<T>::iterator
8      Tree<T>::Min() {return Min(root);} //для внешнего мира
9  //—
10 template<class T> typename Tree<T>::const_iterator
11     Tree<T>::Min(const typename Tree<T>::Node *r) const
12 { if(r==nullptr) return const_iterator(this); //для внутр. целей
13     while(r->left)r=r->left;
14 return const_iterator(this,r);
15 }
16 template<class T> typename Tree<T>::const_iterator
17     Tree<T>::Min() const {return Min(root);} //для внешнего мира

```

Для каждого итератора метод Min() определен в двух видах: для внутреннего пользования (с указанием вершины поддерева, в котором ищется минимум) и для внешнего использования (без параметров, минимум ищется по всему дереву).

Следует обратить внимание на необходимость использования ключевого слова **typename**. Вместо него можно напрямую использовать слово **struct**, что является необходимым в языке C при определении переменной типа данного класса. Например, в языке C если мы определили структуру **struct S {...}**;, то при объявлении переменной x такого типа мы обязаны использовать ключевое слово **struct: struct S x**;. В языке C++ использование слова **struct** (**class**) при объявлении переменной не является обязательным. Однако в шаблонах синтаксический разбор изначального шаблона происходит в весьма мягкой форме (т. е. изначально допустимо весьма широкое толкование слов, используемых в шаблоне), поэтому компилятору придется подсказывать: является ли выражение Tree<T>::Node именем типа или именем переменной. На самом деле компилятор может догадываться (только догадываться!), что данное выражение является именно именем типа. Поэтому при отсутствии ключевого слова

typename в выдаваемых замечаниях он обычно четко предлагает добавить данную спецификацию (т. е. если при беглом просмотре многословного текста, описывающего ошибки компиляции, обнаруживается слово **typename**, то надо внимательно искать место, куда его надо вставить). Увы, в некоторых особо сложных случаях компилятору не хватает сообразительности даже догадываться об этом, поэтому выдаваемые ошибки могут быть весьма непонятными.

Наличие итератора для `const`-дерева и методов `begin()` и `end()`, возвращающих `const_iterator`, позволяет нам переопределить метод `<<` для вывода дерева `t` на экран обычным способом: `cout<<t<<endl;`. Шаблон соответствующей функции может иметь, например, следующий вид:

```

1  template<class T> ostream
2  &operator<<(ostream &cout, const Tree<T> &t)
3  {cout<<"{"; for(const auto &x:t){cout<<x<<" ";}
4  cout<<"}"; return cout;}

```

Здесь предполагается, что для базового типа `T` шаблона класса дерева определен соответствующий оператор `<<`.

В классе дерева будем хранить указатель на корневую вершину `root` и целую переменную с количеством элементов в текущем дереве. Начало класса дерева тогда будет иметь вид:

```

1  template<class T> class Tree
2  {struct Node{Node *par,*left,*right; T v; static Node*tmp;
3  Node(const
4  T&x,Node*p=nullptr){v=x;par=p;left=right=nullptr;}
5  bool IsLeft()const{return par&&par->left==this;}
6  bool IsRight()const{return par&&par->right==this;}
7  Node *&Par()
8  {
9  if(IsLeft())return par->left;
10 else if(IsRight())return par->right;
11 return tmp;
12 }*root=nullptr; size_t n=0;

```

Здесь мы добавили определения вышеописанных методов класса. Текущая версия языка `C++` позволяет задавать инициализацию эле-

ментов класса непосредственно в их описании (чего нельзя было делать в изначальном C++), хотя это можно было бы сделать и в конструкторе класса.

Зададим три основных конструктора класса дерева:

```

1 Tree() {root=nullptr;n=0;}
2 Tree(initializer_list<T> l):Tree(){for(auto &x:l)insert(x);}
3 template<class U> Tree(const U&l):Tree()
4                                     {for(auto &x:l)insert(x);}

```

Конструктор по умолчанию будет просто дублировать обнуление элементов класса `root` и `n`.

Второй конструктор задаст возможность инициализации дерева начальным набором значений в стандартной форме вида

```

1 Tree<int> t={1,2,3,4,5};

```

Как и ранее, данная возможность обеспечивается конструкцией языка C++ `initializer_list`. Напомним, что использование переменной данного типа подразумевает наличия итератора для данного типа (еще раз подчеркнем, что `initializer_list` не является обычным шаблоном; это конструкция собственно языка C++).

Третий конструктор имеет весьма интересный вид. Фактически он является шаблоном (внутри шаблона класса дерева!), позволяющим инициализировать экземпляр класса дерева объектом любого типа, имеющим стандартный конструктор (который автоматически обеспечивает возможность использования `range-based for`). Например:

```

1 std::vector<int> v={1,2,3,4,5}; Tree<double> t=v;

```

В соответствии с набором предписаний дерева поиска класс дерева должен иметь методы добавления и удаления элементов:

```

1 pair<iterator, bool> insert(const T&x);
2 bool erase(const T&x){auto it=find(x);
3   if(it){erase(it); return true;} else return false;}
4 void erase(iterator it);

```

Метод `erase()` удаления элемента типа `T` из дерева сначала вызывает функцию поиска элемента (которую еще предстоит описать и определить) и в случае ее успешности вызывает метод удаления элемента по

итератору, указывающему на найденный элемент. Метод возвращает признак успешности выполнения процедуры (успешности поиска удаляемого элемента).

При определении метода вставки элемента в дерево мы сталкиваемся с необходимостью возвращения данным методом сразу двух сущностей: во-первых, признака успешности выполнения данной операции (если вставляемый метод уже есть в дереве, то вставка должна быть признана неуспешной), во-вторых, ссылки (итератора) на вставленную вершину (если вставка прошла успешно). Данный подход повторяет семантику, используемую соответствующим методом класса `set` библиотеки `STL`, но противоречит тому, что функция может вернуть только одно значение. Однако в практике очень часто встречаются ситуации, когда в программе необходимо оперировать парой переменных определенных типов. Во многих языках программирования в таком случае используется сущность `pair`. В `STL` для этих целей используется соответствующий шаблон. Несомненно, недостатком языка является то, что при использовании шаблона `pair` практически всегда в угловых скобках после слова `pair` надо указывать типы переменных, содержащихся в данном шаблоне (во многих случаях компилятор достаточно сообразителен, чтобы понять, какими типами задается шаблон без непосредственного их указания, но для случая `pair` это часто не срабатывает).

Итак, метод вставки значения в дерево `insert` возвращает пару переменных. Первая — итератор, указывающий на вставленный или присутствующий в дереве элемент с данным значением, вторая имеет тип `bool` и отвечает за успешность операции вставки. Обращение к первой и второй переменной шаблона `pair` происходит через элементы шаблона класса `first` и `second`. Здесь надо иметь в виду, что под элементом с заданным значением v подразумевается элемент дерева x , для которого не верны соотношения $v < x$ и $x < v$. Мы уже упоминали, что для типа T , параметризующего шаблон дерева, должен быть задан оператор `<`, для которого выполняются аксиомы, которые мы задали в разделе *Сортировки* в [11].

Последний кусок кода можно подправить, используя возможность задания переменной внутри оператора `if` (по аналогии задания переменной внутри оператора `for`), которая появилась в 17-й версии языка `C++`. Получившийся код будет иметь вид:

```

1 pair<iterator , bool> insert (const T&x);
2 bool erase (const T&x) { if (auto it=find(x); it)
3   { erase(it); return true; } else return false; }
4 void erase (iterator it);

```

На данный момент компилятору gcc надо указывать при помощи соответствующего ключа, что при компиляции следует использовать 17-ю версию языка:

```

1 g++ -std=c++17 q.cpp

```

Метод `void erase(iterator it);` будет определен позднее.

Вставка элемента в дерево производится путем поиска в цикле листа, после которого можно вставить новый элемент, а потом собственно создания и вставки новой вершины вместо отсутствующего потомка найденного листа. Если вставляемый элемент уже есть в дереве, то функция вставки должна вернуть пару, состоящую из итератора, указывающего на найденную вершину, и логической переменной `false`. При этом оператор `new Node(x,r)` создает экземпляр класса с новой вершиной, внутри которой хранится элемент `x` и сразу задается указатель на родителя `r`. В случае успешности вставки вершины надо не забыть увеличить на единицу количество вершин дерева: `this->n++`. В результате код метода вставки будет иметь вид:

```

1 template<class T> pair<struct Tree<T>::iterator , bool>
2   Tree<T>::insert (const T&x)
3 { if (root==nullptr) { root=new Node(x); this->n++;
4   return pair<iterator , bool>(iterator (this , root) , true); }
5 for (Node *r=root;;)
6   { //новые вершины вставляются только в листья
7     if (x<r->v)
8       { if (r->left) r=r->left; else { r->left=new
9         Node(x, r); this->n++;
10        return pair<iterator , bool>(iterator (this , r->left) , true); } }
11    else if (r->v<x)
12      { r->right=new Node(x, r); this->n++;
13      return
14        pair<iterator , bool>(iterator (this , r->right) , true); }

```

```

14 }
15 else
16     return pair<iterator , bool>(iterator (this , r) , false);
17 }
18 }

```

Удаление вершины дерева более сложно.

Если у вершины нет потомков, то данная вершина n просто удаляется, а указатель от родителя на нее $n \rightarrow \text{Par}()$ обнуляется.

Если у вершины n есть только один потомок, то указатель на данную вершину от родителя перебрасывается на существующего потомка вершины n , а указатель на родителя от потомка n перебрасывается на родителя вершины n .

Наконец, если у вершины n есть оба потомка, то ищем минимальный элемент m в дереве с корнем в вершине $n \rightarrow \text{right}$. Значение из вершины m перебрасываем в вершину n . Далее удаляем вершину m этой же процедурой. Мы это можем сделать, так как вершина m является листом дерева, а удаление листа мы разобрали ранее в двух предыдущих пунктах алгоритма.

Осталось не забыть уменьшить на единицу количество элементов в дереве $\text{this} \rightarrow n$ (если происходит реальное удаление вершины дерева). Также надо каждый раз отдельно рассматривать случай отсутствия родителя у удаляемой вершины (т. е. случай, когда данная вершина является корнем).

В результате получится следующий код:

```

1  template<class T> bool Tree<T>::erase(iterator it)
2  {if(!it) return false; Node *n=it.n;
3   if(n->left==nullptr&& n->right==nullptr) //нет потомков
4   {this->n--; if(n->par)n->Par()=nullptr; else root=nullptr;}
5   else if(n->left==nullptr) //есть только правый потомок:
6       n->right
7   {this->n--; if(n->par)n->Par()=n->right; else {root=n->right;
8   n->right->par=n->par;}
9   else if(n->right==nullptr) //есть только левый потомок:
10      n->left
11  {this->n--; if(n->par)n->Par()=n->left; else {root=n->left;
12  n->left->par=n->par;}
13  else //есть и левый и правый потомок

```

```

12 { iterator m=Min(n->right); *it=std::move(*m); erase(m);
13     return true;}
14 delete n; return true;
15 }

```

Поиск элемента в дереве происходит естественным путем, начиная с корня дерева. Если искомое значение меньше значения корня, то далее аналогично рассматриваем левое поддерево от корня. Если значение корня дерева меньше искомого значения, то далее аналогично рассматриваем правое поддерево от корня. Если не верно ни то, ни то, то мы нашли вершину с заданным значением. Если после очередного шага дочерней вершины нет, то искомого значения в дереве нет.

Поиск следующего по порядку элемента дерева будет реализован в рамках оператора ++ для итератора дерева. Алгоритм поиска следующего элемента за вершиной n состоит в следующем: если у вершины есть правый потомок, то следующим элементом будет минимальный элемент в правом поддереве вершины n. Если правого потомка нет, то следует идти вверх по иерархии дерева, пока текущая вершина не станет левым потомком своего родителя. Легко увидеть, что в конце такого цикла родитель текущей вершины будет следующим по порядку элементом дерева.

Приведем окончательный код дерева поиска.

```

1  template<class T> class Tree
2  {struct Node{Node *par,*left,*right; T v; static Node*tmp;
3   Node(const T&x,
4     Node*p=nullptr){v=x; par=p; left=right=nullptr;}
5   bool IsLeft() const{return par&&par->left==this;}
6   bool IsRight() const{return par&&par->right==this;}
7   Node *&Par(){if(IsLeft())return par->left;
8     else if(IsRight())return par->right;return tmp;}
9   }*root; size_t n;
10  iterator Min(Node*r);
11  const_iterator Min(const Node*r) const;
12  public:
13  struct iterator{Tree *t; Node*n;
14  iterator(Tree*t=nullptr, Node*n=nullptr){this->t=t; this->n=n;}

```

```

14  bool operator!=(const iterator&b) const{return n!=b.n;}
15  iterator operator++();
16  T&operator*(){return n->v;}
17  operator bool()const{return n!=nullptr;}
18  };
19  iterator begin(){return Min(root);}
20  iterator end(){return iterator(this, nullptr);}
21  //—
22  struct const_iterator{const Tree *t; const Node*n;
23    const_iterator(const Tree*t=nullptr, const Node*n=nullptr)
24    {this->t=t; this->n=n;}
25  bool operator!=(const const_iterator&b) const{return
26    n!=b.n;}
27  const_iterator operator++();
28  const T&operator*(){return n->v;}
29  operator bool()const{return n!=nullptr;}
30  };
31  const_iterator cbegin()const{return Min(root);}
32  const_iterator kend()const
33  {return const_iterator(this, nullptr);}
34  const_iterator begin()const{return Min(root);}
35  const_iterator end()const//для range-based for const вектора
36  {return const_iterator(this, nullptr);}
37  //—
38  bool empty(){return root==nullptr;}
39  size_t size(){return n;}
40  //—
41  Tree(){root=nullptr;n=0;}
42  Tree(initializer_list<T> l):Tree(){for(auto &x:l)insert(x);}
43  template<class U> Tree(const U&l):Tree()
44  {for(auto &x:l)insert(x);}
45  ~Tree(){while(!empty())erase(iterator(this, root));}
46  //—
47  iterator find(const T&x);
48  bool erase(const T&x);
49  bool erase(iterator it);
50  pair<iterator, bool> insert(const T&x);
51  iterator Min(){return Min(root);}
52  const_iterator Min()const{return Min(root);}

```

```

52 };
53 //=====
54 template<class T> typename Tree<T>::Node*
55   Tree<T>::Node::tmp=nullptr;
56
57 template<class T> typename Tree<T>::iterator
58   Tree<T>::Min(typename Tree<T>::Node *r)
59 { if(r==nullptr) return iterator(this);
60   while(r->left)r=r->left;
61   return iterator(this,r);
62 }
63 //-----
64 template<class T> typename Tree<T>::const_iterator
65   Tree<T>::Min(const typename Tree<T>::Node *r) const
66 { if(r==nullptr) return const_iterator(this);
67   while(r->left)r=r->left;
68   return const_iterator(this,r);
69 }
70 //-----
71 template<class T> typename Tree<T>::iterator
72   Tree<T>::iterator::operator++()
73 { if(!n) return iterator(t);
74   if(n->right)
75   { *this=t->Min(n->right); return *this;}
76   else
77   { while(n->IsRight())n=n->par;
78     n=n->par; return *this;
79   }
80 }
81 //-----
82 template<class T> typename Tree<T>::const_iterator
83   Tree<T>::const_iterator::operator++()
84 { if(!n) return const_iterator(t);
85   if(n->right)
86   { *this=t->Min(n->right); return *this;}
87   else
88   { while(n->IsRight())n=n->par;
89     n=n->par; return *this;
90   }

```

```

91 }
92 //-----
93 template<class T> pair<struct Tree<T>::iterator, bool>
94     Tree<T>::insert(const T&x)
95 {if(root==nullptr){root=new Node(x); this->n++;
96     return pair<iterator, bool>(iterator(this, root), true);}
97     for(Node *r=root;;)
98     {//новые вершины вставляются только в ~листья
99     if(x<r->v)
100    {if(r->left)r=r->left; else {r->left=new
101        Node(x, r); this->n++;
102        return pair<iterator, bool>(iterator(this, r->left), true);}}
103    else if(r->v<x)
104    {if(r->right)r=r->right; else
105    {r->right=new Node(x, r); this->n++;
106    return
107        pair<iterator, bool>(iterator(this, r->right), true);}}
108    }
109    else
110    return pair<iterator, bool>(iterator(this, r), false);
111 }
112 //-----
113 template<class T> typename Tree<T>::iterator
114     Tree<T>::find(const T&x)
115 {if(empty())return iterator(this);
116     for(Node*n=root;n;)
117     if(x<n->v)n=n->left; else if(n->v<x)n=n->right;
118     else return iterator(this, n);
119     return iterator(this);
120 }
121 //-----
122 template<class T> ostream
123     &operator<<(ostream&cout, const Tree<T> &t)
124 {cout<<"{"; for(auto it=t.cbegin(); it!=t.cend(); ++it)
125     {cout<<*it<<" ";} cout<<"}"; return cout;}
126
127 template<class T> bool Tree<T>::erase(const T&x)
128 {auto it=find(x); return erase(it);}

```

```

128 //-----
129 template<class T> bool Tree<T>::erase(iterator it)
130 { if (!it) return false; Node *n=it.n;
131   if (n->left==nullptr&& n->right==nullptr)
132   { this->n--; if (n->par)n->Par()==nullptr; else root=nullptr;}
133   else if (n->left==nullptr) //n->right!=nullptr
134   { this->n--; if (n->par)n->Par()==n->right; else {root=n->right;}
135     n->right->par=n->par;}
136   else if (n->right==nullptr) //n->left!=nullptr
137   { this->n--; if (n->par)n->Par()==n->left; else {root=n->left;}
138     n->left->par=n->par;}
139   else
140   {iterator m=Min(n->right); *it=std::move(*m); erase(m);
141     return true;}
142   delete n; return true;
143 }

```

Поиск элемента по индексу реализуется при незначительном дополнении вышеописанного класса дерева. Данная функция потребует хранения в каждой вершине дерева количества nl элементов в левом поддереве данной вершины. Теперь структура вершины дерева будет иметь следующее описание:

```

1 struct Node{Node *par,*left,*right; T v;
2   static Node*tmp; size_t nl=0;
3   Node(const T&x,
4     Node*p=nullptr){v=x; par=p; left=right=nullptr;}
5   bool IsLeft() const{return par&&par->left==this;}
6   bool IsRight() const{return par&&par->right==this;}
7   Node *&Par() {if (IsLeft()) return par->left; else
8     if (IsRight()) return par->right; return tmp;}
9   }*root; size_t n;

```

Вставка элемента в дерево потребует всего лишь одной дополнительной операции $r->nl++$ в случае, если новая вершины будет вставлена в левое поддерево вершины r . Проблема заключается в том, что в начале поиска вершины, после которой должна происходить вставка, мы еще не знаем: есть ли вставляемое значение в дереве (если есть, то в дереве ничего измениться не должно) или его нет (тогда

должна произойти реальная вставка). Поэтому в начале функции нам придется вызвать функцию поиска в дереве вставляемого элемента. Если вставляемый элемент найдется, то нужно сразу выйти из функции вставки. Таким образом, алгоритм вставки станет двухпроходным.

Код метода вставки элемента в дерево примет следующий вид:

```

1  template<class T> pair<struct Tree<T>::iterator , bool>
2      Tree<T>::insert (const T&x)
3  { if (auto it=find(x); it) return pair<iterator , bool>(it , false);
4  if (root==nullptr){ root=new Node(x); this->n++;
5      return pair<iterator , bool>(iterator (this , root) , true);}
6  for (Node *r=root;;)
7  {
8      if (x<r->v)
9          {r->n1++;/* !!! */ if (r->left) r=r->left; else
10             {r->left=new Node(x,r); this->n++;
11                 return pair<iterator , bool>(iterator (this , r->left) , true);}
12         }
13     else if (r->v<x)
14         {if (r->right) r=r->right; else {r->right=new Node(x,r);
15             this->n++;
16             return
17                 pair<iterator , bool>(iterator (this , r->right) , true);}
18         }
19     }
20 }

```

Здесь мы снова использовали определение переменной внутри оператора **if**: **if**(auto it=find(x);it).

Метод удаления элемента из дерева оказывается более сложным. После того как мы найдем вершину дерева, которую реально надо удалять (напомним, что такая вершина обязана быть листом), нужно будет пройти по ветке от найденного элемента до корня дерева, корректируя значение n1 в тех вершинах, в которые мы пришли из левого поддерева. Метод, обеспечивающий корректировку значений n1, будет иметь вид:

```

1 template<class T> void Tree<T>::Dec(Node *n)
2 {this->n--;
   for (; n->par; n=n->par) if (n->IsLeft ()) n->par->nl--;}
```

Тогда метод уничтожения вершины дерева примет вид:

```

1 template<class T> bool Tree<T>::erase(iterator it)
2 {if (!it)return false; Node *n=it.n;
3   if (n->left==nullptr&& n->right==nullptr)
4     {Dec(n); if (n->par)n->Par()==nullptr;else root=nullptr;}
5   else if (n->left==nullptr) //n->right!=nullptr
6     {Dec(n); if (n->par)n->Par()==n->right;else {root=n->right;}
7       n->right->par=n->par;}
8   else if (n->right==nullptr) //n->left!=nullptr
9     {Dec(n); if (n->par)n->Par()==n->left;else {root=n->left;}
10      n->left->par=n->par;}
11  else
12    {iterator m=Min(n->right); *it=static_cast<T&&>(*m);
13      erase(m); return true;}
14  delete n; return true;
15 }
```

В класс дерева потребуется добавить описания метода Dec() и оператора [], который мы обсудим далее:

```

1 T&operator [] (size_t i);
2 void Dec(Node *n);
```

Поиск элемента дерева по порядковому номеру i происходит с помощью следующего алгоритма. Зададим текущую вершину дерева $n=root$.

Если $n->nl==i$, то легко увидеть, что данная вершина является искомой.

Если $i < n->nl$, то искомая вершина находится в левом поддереве вершины n , и мы переходим к левому поддереву: $n=n->left$.

Если $i > n->nl$, то искомая вершина находится в правом поддереве вершины n , и мы переходим к правому поддереву: $n=n->right$. При этом следует скорректировать искомый индекс: $i -= n->nl + 1$, поскольку мы теперь игнорируем левое поддерево исходной вершины n и саму исходную вершину n .

Код определения оператора `||` будет иметь вид:

```

1 template<class T> T& Tree<T>::operator |( size_t i)
2 { if (i>=n) throw -1;
3   for (Node*n=root ;;)
4     if (n->n1==i) return n->v;
5     else if (i<n->n1) n=n->left;
6     else { i=n->n1+1; n=n->right;}
7 }

```

Слияние двух деревьев

Для двух деревьев поиска T_1 и T_2 , таких что все элементы в T_1 меньше или равны всех элементов в T_2 , требуется слить элементы деревьев в одно дерево поиска T .

Алгоритм решения задачи сводится к следующему. Выбираем из дерева T_2 наименьший элемент (самый левый), исключаем его из дерева T_2 и делаем его корнем нового дерева T . Его левым потомком будет корень дерева T_1 , а правым — корень дерева T_2 . Дерево T будет деревом поиска.

Далее нам встретится немного другая задача: пусть задан некоторый элемент v и два дерева T_1 и T_2 , такие что все элементы в T_1 меньше v , а все элементы из T_2 — больше. Требуется слить все указанные данные в одно дерево поиска T . Элемент v в этой ситуации называется *стыковочным*. Задача в данной формулировке тривиальна.

Разбиение дерева по разбивающему элементу

Для данной вершины дерева v требуется разбить дерево поиска T на два дерева поиска T_0 и T_1 , таких что все элементы в T_0 меньше v , все элементы в T_1 больше v и множество элементов дерева T есть объединение множеств элементов деревьев T_0 , T_1 и элемента v .

Для наглядности рассуждений расположим геометрически дерево поиска таким образом, чтобы его вершины были упорядочены по оси ОХ.

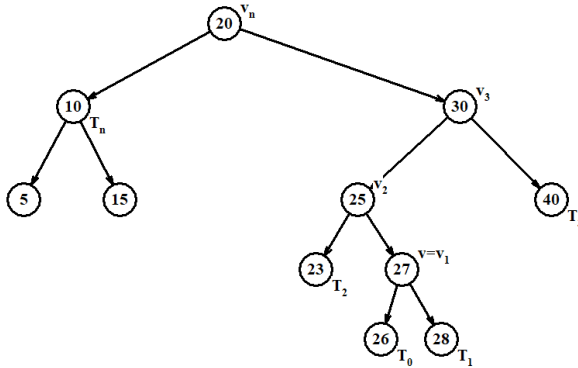
Проведем на графике из вершины v вертикальную линию. Все элементы в дереве слева от этой линии меньше v , а справа — больше v . Вершина v имеет два поддерева, из нее выходящих, в одном из которых все элементы меньше v , а в другом — больше. Назовем эти деревья T_0 и T_1 соответственно.

Занумеруем вершины ветви дерева от v до корня дерева r . Они образуют ветвь $V' = \{v_1 = v, v_2, v_3, \dots, v_n = r\}$: Поддерево, выходящее из вершины, являющейся потомком $v_i \in V'$, назовем T_i (см. рисунок).

Легко доказать, что верны следующие факты для любого $1 \leq i \leq n$:
 либо для любого $j < i$ выполнено $T_i < v_i < T_j$, $T_i < v_i < v_j$ (v_j принадлежит правому поддереву v_i),

либо для любого $j < i$ выполнено $T_j < v_i < T_i$, $v_j < v_i < T_i$ (v_j принадлежит левому поддереву v_i).

Легко увидеть также, что каждая вершина дерева T принадлежит либо некоторому T_i , либо V' .



Пример разбиваемого дерева

Конструирование двух требуемых поддеревьев будем производить следующим способом. Рассмотрим сначала поддеревья T_0 и T_1 . Будем последовательно добавлять в них данные, чтобы получить искомые деревья.

Будем последовательно перебирать вершины v_i для i от 2 до n . На каждом шаге если v_{i-1} является правым потомком v_i , то $v_i < v_{i-1}$, $T_i < v_i < T_0$, и мы сливаем деревья T_i и T_0 с помощью стыковочного элемента v_i , помещая результат слияния в T_0 . Иначе v_{i-1} является левым потомком v_i , тогда $v_{i-1} < v_i$, $T_1 < v_i < T_i$, и мы сливаем деревья T_1 и T_i с помощью стыковочного элемента v_i , помещая результат слияния в T_1 .

В конечном итоге деревья T_0 и T_1 будут искомым разбиением исходного дерева T с помощью разбивающей вершины v .

Как результат проделанных выкладок, сформулируем несколько теорем.

Теорема 1. *Операции добавления, удаления, поиска, поиска элемента по индексу, поиска минимального и максимального элементов реализуются за время $O(h(T))$, где $h(T)$ — высота дерева T .*

Теорема 2. *Пусть для деревьев T_1 и T_2 и элемента v выполняются соотношения $T_1 < v < T_2$. Тогда объединение деревьев T_1 и T_2 с помощью стыковочного элемента v можно выполнить за время $O(1)$.*

Теорема 3. *Пусть для деревьев T_1 и T_2 выполнено соотношение $T_1 < T_2$. Тогда объединение деревьев T_1 и T_2 можно выполнить за время $O(\min(h(T_1), h(T_2)))$.*

Далее речь пойдет только про деревья поиска.

Введем несколько определений для бинарных деревьев поиска. На самом деле в них никак не будет использоваться то, что бинарное дерево является именно деревом поиска, но в дальнейшем все построения будут опираться на это. Поэтому мы все же будем требовать во всех определениях, чтобы деревья были деревьями поиска.

Сбалансированным деревом (AVL-деревом) называется такое бинарное дерево поиска, что для каждой его вершины высоты левого и правого поддеревьев отличаются не более чем на 1: для каждой вершины $v \in V$

$$|h(T(v \rightarrow \text{left})) - h(T(v \rightarrow \text{right}))| \leq 1.$$

Балансом вершины v называется разность высот ее левого и правого поддеревьев, т. е. $h(T(v \rightarrow \text{left})) - h(T(v \rightarrow \text{right}))$.

Идеально сбалансированным деревом называется такое бинарное дерево поиска, что длины всех его ветвей от корня до листа отличаются не более чем на 1.

Напомним, что *листом* мы называем вершину, у которой менее двух потомков.

В литературе часто встречается другое определение идеально сбалансированного дерева. Чтобы не вносить путаницы, мы будем называть такие деревья *идеально сбалансированными'*.

Идеально сбалансированным' деревом называется такое бинарное дерево поиска, что для каждой его вершины количества элементов

в левом и правом поддереве отличаются не более чем на 1, т. е. для каждой вершины $v \in V$ справедливо

$$|\#(T(v \rightarrow \text{left})) - \#(T(v \rightarrow \text{right}))| \leq 1.$$

Здесь и далее будем обозначать через $\#T$ количество вершин (элементов) в дереве T , а через $\#V$ для множества V — количество элементов в множестве V .



Примеры различных видов деревьев

Для идеально сбалансированного дерева T высоты h все слои дерева, кроме, быть может, нижнего слоя, заполнены, поэтому для него выполняется соотношение

$$2^{h-1} \leq \#(T) < 2^h.$$

Прологарифмировав последнее соотношение, получим следующее утверждение.

Утверждение 1. Для идеально сбалансированного дерева T высоты h , состоящего из N вершин, выполнено соотношение

$$\log_2(N) < h \leq \log_2(N) + 1.$$

Докажем, что между введенными определениями деревьев существует взаимосвязь.

Теорема 4. Если дерево идеально сбалансированное', то оно идеально сбалансированное.

Доказательство. Докажем теорему по индукции по высоте дерева.

Для дерева высоты 1 теорема верна.

Пусть теорема верна для деревьев высоты не больше h . Докажем, что тогда теорема верна для дерева высоты $h + 1$.

По предположению индукции для поддеревьев $T(\text{root} \rightarrow \text{left})$ и $T(\text{root} \rightarrow \text{right})$ теорема верна, следовательно, они идеально сбалансированные.

Поскольку для поддеревьев $T(\text{root} \rightarrow \text{left})$ и $T(\text{root} \rightarrow \text{right})$ количества элементов отличаются не более чем на 1 и эти деревья идеально сбалансированные, по утверждению 1 либо их высоты равны, либо в одном поддереве нижний уровень заполнен полностью, а во втором (более длинном) на нижнем уровне присутствует ровно один элемент. Следовательно, длины всех ветвей от корня до листа в поддеревьях отличаются не более чем на 1. Отсюда сразу следует, что все дерево идеально сбалансированное. \square

На самом деле легко увидеть, что идеально сбалансированное' дерево можно рассматривать как обычный упорядоченный массив элементов, где корень дерева — медиана данного множества, вершины второго уровня — медианы левой и правой частей массива относительно медианы и т. д. Мы сразу получаем, что для построения идеально сбалансированного' дерева достаточно упорядочить массив элементов, после чего мы автоматически получаем индексы вершин соответствующего идеально сбалансированного' дерева. Отсюда вытекает утверждение о времени, необходимом для построения идеально сбалансированного' дерева, а следовательно, по доказанной теореме 4, и для идеально сбалансированного дерева.

Утверждение 2. *Идеально сбалансированное' (а значит, и идеально сбалансированное) дерево можно построить алгоритмами на основе сравнений за время $O(N \cdot \log(N))$, где N — количество вершин в дереве.*

Поскольку длины всех ветвей от корня до листа дерева, проходящих через одну и ту же вершину v , отличаются на константу (на длину ветви от корня до вершины v минус 1) от длин ветвей от v до листа, то для идеально сбалансированного дерева мы сразу получаем определение, эквивалентное исходному, что отражено в следующем утверждении.

Утверждение 3. *Длины всех ветвей от корня до листа отличаются не более чем на 1 тогда и только тогда, когда для любой*

вершины $v \in V$ длины всех ветвей от v до листа отличаются не более чем на 1.

Тогда, поскольку для идеально сбалансированного' дерева для всех вершин v длины всех ветвей от v до листа отличаются не более чем на 1, высоты левого и правого поддеревьев вершины v тоже отличаются не более чем на 1. Отсюда сразу вытекает следующая теорема.

Теорема 5. *Если дерево идеально сбалансированное, то оно сбалансированное.*

Итак, мы получили, что из идеальной сбалансированности' дерева следует его идеальная сбалансированность, а из идеальной сбалансированности следует сбалансированность.

12.1. Сбалансированные деревья (AVL-деревья)

Для сбалансированного дерева верна оценка высоты дерева через количество вершин в нем, аналогичная по порядку оценке для идеально сбалансированных деревьев.

Теорема 6. *Для сбалансированного дерева высоты h , состоящего из N вершин, верна оценка*

$$h = \Theta(\log_2 N).$$

Доказательство. Пусть t_n — минимальное количество элементов в сбалансированном дереве высоты n . Тогда верна рекурсивная формула

$$t_n = t_{n-1} + t_{n-2} + 1,$$

т. е. для сбалансированного дерева высоты n с минимальным количеством вершин одно из поддеревьев, дочерних корневому элементу, должно быть сбалансированным деревом высоты $n - 1$ с минимальным количеством вершин, а другое — сбалансированным деревом высоты $n - 2$ с минимальным количеством вершин.

Уравнение $t_n - t_{n-1} - t_{n-2} = 1$ можно рассматривать как бесконечную неоднородную систему линейных уравнений, для которой верно, что ее общее решение представляется как общее решение соответствующей однородной системы плюс частное решение неоднородной системы. Легко увидеть, что у системы $t_n - t_{n-1} - t_{n-2} = 1$ есть простое частное решение $t_n = -1$.

Найдем общее решение однородной системы линейных уравнений $t_n - t_{n-1} - t_{n-2} = 0$. У этой задачи существует стандартное общее решение (см. [1]), которое будет разбираться на старших курсах. Однако оно нам не понадобится. Разберемся с решением конкретного уравнения. Легко увидеть, что если задать t_1 и t_2 , то все остальные t_i однозначно восстанавливаются по рекуррентной формуле. Следовательно, нам достаточно найти два линейно независимых решения t_i для $i = 1, 2$ (а следовательно, и для всех $i > 2$), и тогда задачу можно считать решенной.

Будем искать решение задачи $t_n - t_{n-1} - t_{n-2} = 0$ в виде $t_n = \lambda^n$. Тогда получаем соотношение

$$\lambda^n - \lambda^{n-1} - \lambda^{n-2} = 0.$$

Вынося λ^{n-2} за скобки, получаем

$$\lambda^2 - \lambda - 1 = 0.$$

Данное уравнение в поставленной задаче называется *характеристическим*. У него есть два решения (в этом есть суть нашего везения!):

$$\lambda_1^n = \frac{1 + \sqrt{5}}{2} \quad \text{и} \quad \lambda_2^n = \frac{1 - \sqrt{5}}{2}.$$

Таким образом, система $t_n - t_{n-1} - t_{n-2} = 1$ имеет общее решение вида

$$t_n = C_1 \cdot \left(\frac{1 + \sqrt{5}}{2} \right)^n + C_2 \cdot \left(\frac{1 - \sqrt{5}}{2} \right)^n - 1.$$

Поскольку

$$\left| \frac{1 + \sqrt{5}}{2} \right| > 1, \quad \text{а} \quad \left| \frac{1 - \sqrt{5}}{2} \right| < 1,$$

мы получаем, что

$$t_n = C_1 \cdot \left(\frac{1 + \sqrt{5}}{2} \right)^n (1 + o(1)).$$

После логарифмирования последнего равенства мы сразу получаем требуемое соотношение:

$$\log_2 C_2 + \log_2 t_n \geq n \log_2 \left(\frac{\sqrt{5} + 1}{2} \right),$$

$$n \leq \frac{\log_2 t_n + \log_2 C_2}{\log_2((\sqrt{5} + 1)/2)}$$

для некоторого $C_2 > 0$ или, в исходных обозначениях,

$$h \leq \frac{\log_2 N}{\log_2((\sqrt{5} + 1)/2)} + \log_2 C_3 \leq 1,45 \log_2 N + C$$

для некоторой константы C . □

Если сравнить последнее соотношение с аналогичной оценкой высоты идеально сбалансированного дерева, то мы получим, что высота сбалансированного дерева не более чем в 1,45 раза больше высоты идеально сбалансированного дерева, состоящего из того же количества элементов. А поскольку все рассмотренные алгоритмы работы с деревьями поиска напрямую зависят от высоты дерева, мы, чисто формально, выяснили, во сколько раз все алгоритмы изменения сбалансированного дерева будут медленнее соответствующих алгоритмов для идеально сбалансированного дерева. Разумеется, после изменения дерево должно оставаться в том же классе, в котором оно было исходно. Мы не умеем быстро возвращать дерево в состояние идеальной сбалансированности, но AVL-дерево после всех рассмотренных выше операций можно достаточно быстро привести опять к состоянию сбалансированности. Как это делается, мы рассмотрим далее.

12.2. Операции с AVL-деревьями

Оказывается, что для сбалансированных деревьев все описанные выше операции можно модифицировать так, что будет сохраняться сбалансированность дерева, при этом время их выполнения не будет превышать $O(\log_2 N)$ операций. Везде далее, говоря о сбалансированных деревьях, будем подразумевать, что мы имеем дело со сбалансированными деревьями поиска.

Как обычно, правым и левым поддеревьями некоторой вершины дерева v называются поддерева с корнями $v \rightarrow \text{left}$ и $v \rightarrow \text{right}$ соответственно. Баланс вершины дерева — разность высот левого и правого поддеревьев этой вершины.

Будем везде для простоты считать, что у каждой вершины есть ровно два дочерних поддерева (левое и правое), которые могут быть пустыми.

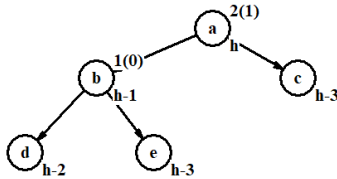
Поиск элемента в дереве

Поиск в сбалансированных деревьях не отличается от поиска в стандартных деревьях поиска.

Добавление элемента в дерево

После добавления нового элемента обычным способом в лист дерева (новые элементы добавляются только в листья) пойдем вверх от добавленной вершины (балансы вершин могут нарушиться только в данной ветви, и их изменение не может быть больше единицы) и найдем первую вершину в ветви (a), для которой баланс стал по модулю больше одного, т. е. его модуль стал равен двум. Для всех вершин, лежащих ниже a , баланс по модулю не превосходит 1.

Пусть для определенности элемент добавляется в левое поддерево вершины a . Справа сверху от вершины будем писать ее баланс после добавления новой вершины, а рядом в круглых скобках — баланс до добавления. Высоту соответствующего данной вершине поддерева будем писать справа снизу от вершины:



Будем обозначать баланс вершины с помощью квадратных скобок: $[a]$ — баланс вершины a . То, что дерево после добавления вершины разбалансировалось, означает, что до добавления вершины $[a] = 1$, а после добавления $[a] = 2$. Возможны три варианта баланса вершины b после изменения дерева: 1, 0, -1 (поскольку ниже вершины a нет разбалансированных вершин). Рассмотрим два, как окажется, принципиально различных случая: 1) $[b] = 1$ или $[b] = 0$, 2) $[b] = -1$. На рисунке варианты $[b] = 1$ или $[b] = 0$ печатаются через косую черту.

1. После добавления вершины получаем $[b] = 1$ или $[b] = 0$ ($[b] = 1$ соответствует удлинению левого поддерева b). Произведем следующую трансформацию дерева:



Такую перестановку вершин (с соответствующими поддеревьями) будем называть правым поворотом (в соответствии с перемещением вершин $d-b-a$). Возможно, данную трансформацию проще (чисто визуально) представлять себе как утягивание вверх вершины b и стряхивание оставшихся вершин вниз.

Будем обозначать высоту дерева с корнем в некоторой вершине x через h_x , баланс этой вершины — $[x]$. Пусть $h_a = h$ (см. рисунок). Тогда для случая $[b] = 1$ получаем:

$$h_a = h,$$

$$h_b = h - 1 \text{ (так как } h_b > h_c),$$

$$h_c = h - 3 \text{ (= } h_b - 2, \text{ так как } h_b > h_c \text{ и } [a] = 2),$$

$$h_d = h - 2 \text{ (так как } h_d > h_e),$$

$$h_e = h - 3 \text{ (= } h_d - 1, \text{ так как } h_d > h_e \text{ и } [b] = 1).$$

После изменения дерева высоты поддеревьев $T(d)$, $T(e)$, $T(c)$ не изменяются и легко рассчитать высоты поддеревьев с вершинами в b и a : $h_a = h - 2$, $h_b = h - 1$. Значит, после правого поворота

$$[a] = [b] = 0,$$

$$[c], [d], [e] \text{ не изменились.}$$

Таким образом, данное дерево сбалансировалось. При этом, если изменение дерева произошло в результате добавления вершины, его высота не изменилась. Действительно, перед добавлением вершины $h_a = h - 1$, что совпадает с высотой дерева $T(b)$ после добавления вершины и правого поворота. Таким образом, в случае $[b] = 1$ процесс балансировки дерева завершен.

Для случая $[b] = 0$ изображенное дерево тоже остается сбалансированным:

$$h_a = h,$$

$$h_b = h - 1 \text{ (так как } h_b > h_c),$$

$$h_c = h - 3 \text{ (= } h_b - 2, \text{ так как } h_b > h_c \text{ и } [a] = 2),$$

$h_d = h - 2$ (так как $h_d = h_e$),

$h_e = h - 2$.

Значит, после правого поворота

$[a] = 1, [b] = -1,$

$[c], [d], [e]$ не изменились.

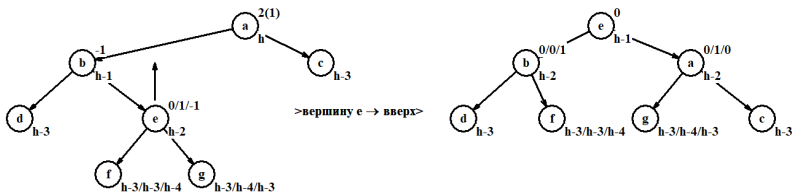
Однако высота всего изображенного дерева изменяется (была до добавления $h_a = h - 1$, стала $h_b = h$).

Итак, если перед изменением дерева $[b] = 1$, то процесс балансировки завершен.

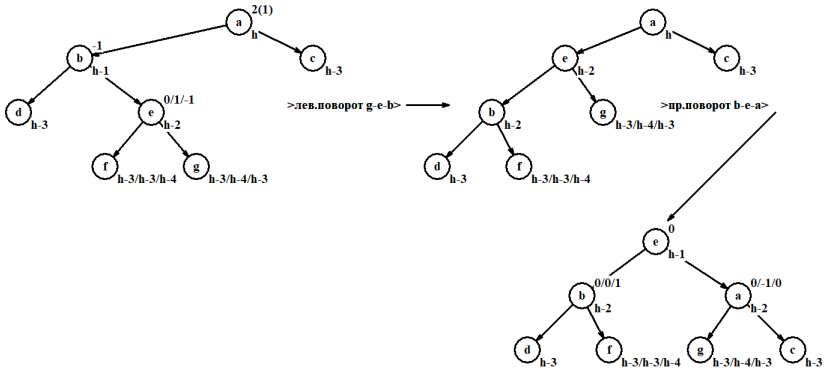
Осталось заметить, что случай $[b] = 0$ не может реализоваться при добавлении вершины к дереву. Действительно, если после добавления вершины выполняется соотношение $[b] = 0$, то это значит, что либо высоты поддеревьев вершины b вообще не изменились, либо новая вершина добавлена к более короткому из этих поддеревьев (при добавлении вершины баланс изменяется не более чем на единицу). Но тогда высота поддерева $T(b)$ в любом случае не изменилась и никакой разбалансировки всего поддерева $T(a)$ произойти не могло.

В дальнейшем мы покажем, что случай $[b] = 0$ может реализоваться, например, при удалении вершины из дерева (если сократится более высокое поддерево вершины b). Поэтому данный случай все равно надо было разбирать.

2. После добавления вершины получаем $[b] = -1$ (удлинение правого поддерева b). В этом случае поднимем вверх вершину e и в соответствии с порядком вершин перебросим ссылки между родителями и потомками в дереве:



Описанная перестановка может быть проведена за два вращения: левое $g-e-b$ и правое $b-e-a$:



Возможны три варианта баланса вершины e : $[e] = 0$, или $[e] = 1$, или $[e] = -1$. Пусть $h_a = h$. Тогда в соответствии с возможными вариантами $[e]$ имеем:

$$h_a = h,$$

$$h_b = h - 1,$$

$$h_c = h - 3,$$

$$h_d = h - 3,$$

$$h_e = h - 2,$$

$$h_f = h - 3, \text{ или } h_f = h - 3, \text{ или } h_f = h - 4,$$

$$h_g = h - 3, \text{ или } h_g = h - 4, \text{ или } h_g = h - 3.$$

Значит, после трансформации дерева

$$[e] = 0,$$

$$[b] = 0, \text{ или } [b] = 0, \text{ или } [b] = 1,$$

$$[a] = 0, \text{ или } [a] = -1, \text{ или } [a] = 0,$$

$$[d], [f], [g], [c] \text{ не изменились.}$$

Таким образом, изображенное дерево сбалансировалось. При этом, если изменение дерева произошло в результате добавления вершины, его высота не изменилась. Действительно, перед добавлением вершины выполнялось $h_a = h - 1$ и после добавления высота не изменилась ($h_e = h - 1$). Итак, в случае добавления вершины при $[b] = -1$ процесс балансировки дерева завершен.

Таким образом, процедура вставки вершины в сбалансированное дерево поиска сводится к нахождению листа v , после которого следует вставить новую вершину, и проверке условия сбалансированно-

сти всех вершин на ветке от вершины v до корня. Если в процессе проверки встретится разбалансированная вершина, то с помощью одного или двух вращений все дерево можно привести к сбалансированному.

Итак, мы доказали следующую теорему.

Теорема 7. *В сбалансированное дерево поиска, состоящее из N вершин, можно добавить одну вершину за время $O(\log_2 N)$. При этом для балансировки дерева потребуется не более двух поворотов.*

Отметим, что, хотя балансировок требуется не более двух, весь процесс балансировки все же требует времени $O(\log_2 N)$, так как требуется еще найти, в какой вершине следует производить балансировку.

Удаление элемента из дерева

Удаление вершины из дерева поиска описано на с. 187. Нам остается только сбалансировать возможно разбалансированное дерево. Таким образом, процедура удаления вершины v из сбалансированного дерева поиска сводится к следующему:

- найти вершину v , которую следует реально удалить (в любом случае это будет лист),
- удалить вершину из дерева поиска (возможно, с предварительным замещением значения в некоторой другой вершине v' значением из данной вершины v),
- проверить для каждой вершины ветви дерева от v' до корня условие балансировки, и если оно нарушилось, то операциями вращения произвести балансировку соответствующего поддерева.

Вернемся к уже рассмотренному алгоритму исправления баланса вершины (алгоритм останется неизменным). Баланс корневой вершины исходного поддерева a (см. предыдущие рисунки) после удаления вершины дерева, равный двум, свидетельствует о том, что произошло уменьшение высоты именно правого поддерева вершины a , а левое (более высокое) поддерево не изменилось. При этом при удалении вершины высота всего поддерева не изменилась. Поэтому баланс вершины b (левого потомка вершины a в исходном несбалансированном дереве) может быть любым из трех рассмотренных: -1 ,

0, 1. Следовательно, после балансировки высота всего поддерева может как сократиться на единицу (при $[b] = 1$ или $[b] = -1$), так и остаться неизменной (если $[b] = 0$). Если высота поддерева в результате балансировки осталась прежней, то на этом процесс балансировки дерева завершается. Если же высота поддерева уменьшилась, то придется идти дальше вверх по ветви дерева в поисках очередной проблемной вершины. Таким образом, в худшем случае нам придется делать столько балансировок, сколько вершин находится в ветви.

Алгоритм удаления вершины из сбалансированного дерева позволяет сформулировать следующую теорему.

Теорема 8. *Из сбалансированного дерева поиска, состоящего из N вершин, можно удалить одну вершину за время $O(\log_2 N)$.*

Поиск минимального и максимального элемента в дереве

Поиск минимального и максимального элементов в сбалансированном дереве не отличается от соответствующего поиска для стандартных деревьев поиска.

Поиск следующего/предыдущего элемента в дереве

Поиск следующего и предыдущего элементов в сбалансированном дереве не отличается от соответствующего поиска для стандартных деревьев поиска.

Слияние двух деревьев

Два сбалансированных дерева поиска T_1 и T_2 , такие что все элементы в T_1 меньше всех элементов в T_2 , требуется слить в одно дерево поиска T .

Выбираем из дерева T_2 наименьший элемент v (самый левый) и удаляем его из дерева T_2 . Элемент v стыковочный, для него верно $T_1 < v < T_2$. Возможна ситуация, когда стыковочный элемент присутствует уже в постановке задачи.

Для определенности будем предполагать, что высота дерева T_1 больше или равна высоте дерева T_2 .

Рассмотрим правую ветвь дерева T_1 : $\{v_1, \dots, v_K\}$ (нумерация от корня к листу). Поскольку дерево сбалансированное, $h(v_i) - h(v_{i+1}) \leq 2$. Тогда на этой ветви найдется вершина v_l , такая что

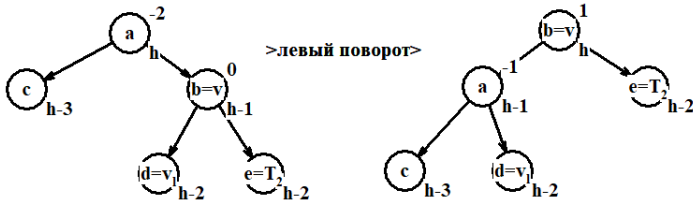
$$h(v_l) = h(T_2) \text{ или } h(v_l) = h(T_2) + 1.$$

Сольем дерево с корнем в v_l и T_2 с помощью стыковочной вершины v и подставим новое дерево на место старой вершины v_l (теперь корнем поддерева вместо вершины v_l станет вершина v).

Все поддерево, начинающееся с v , окажется сбалансированным по построению. Высота же дерева, начинающегося с v , увеличится на 1 по сравнению со старым поддеревом, начинавшимся с v_l , так как у вершины v есть два поддерева: старое и T_2 , которое не выше старого поддерева.

Итак, в результате изменений дерева у одной вершины w высота поддерева увеличилась на 1. Далее нам следует запустить стандартную процедуру балансировки дерева. Мы должны пройти ветвь от w до корня и в каждой вершине проверить баланс. Если он будет по модулю больше 1, то баланс в данной вершине следует скорректировать одним или двумя вращениями. Если при этом высота данного поддерева восстановится до значения до слияния деревьев, то далее проверку сбалансированности производить не надо (так как дерево T_1 до слияния было сбалансированным). В противном случае процесс проверки следует продолжить.

Отметим, что, в отличие от добавления к дереву одной вершины, в данном случае после уравновешения одной вершины процесс может не завершиться, так как, например, после первоначального объединения деревьев возможен следующий вариант:



Этот вариант симметричен рассмотренному нами ранее (баланс родителя вершины v , которая заменила вершину v_l после объединения деревьев, стал равен -2). Здесь баланс вершины b равен нулю, следовательно, после балансировки высота всего поддерева не изменится. Это означает, что по сравнению с состоянием дерева до объединения, высота поддерева, начинающегося с a , увеличилась на единицу и требуется продолжения балансировки дерева для вышележащих вершин.

Итак, алгоритм слияния двух сбалансированных деревьев позволяет сформулировать следующую теорему.

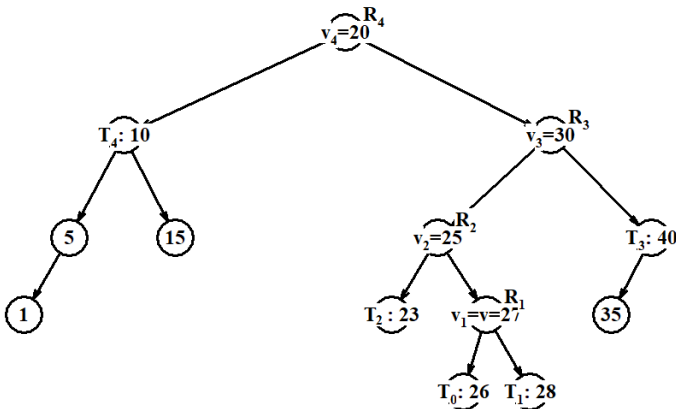
Теорема 9. Для двух сбалансированных деревьев поиска T_1 и T_2 , состоящих из N_1 и N_2 вершин и имеющих высоты h_1 и h_2 , и элемента v , таких что все элементы в T_1 меньше v и v меньше всех элементов в T_2 , слияние деревьев T_1 и T_2 с помощью стыковочного элемента v в одно сбалансированное дерево поиска T можно произвести за время $O(\log_2(N_1 + N_2))$ или за время $O(|h_1 - h_2|)$. Без стыковочного элемента указанные деревья T_1 и T_2 можно слить за время $O(\log_2(N_1 + N_2))$.

Отметим, что для слияния двух сбалансированных деревьев требуется сначала извлечь стыковочный элемент из одного из деревьев, поэтому эта операция требует большего времени, чем слияние деревьев с готовым стыковочным элементом.

Разбиение дерева по разбивающему элементу

Для данной вершины дерева v требуется разбить сбалансированное дерево поиска T на два сбалансированных дерева поиска T_1 и T_2 , таких что все элементы в T_1 меньше v и все элементы в T_2 больше v .

Как и для обычного дерева поиска, занумеруем вершины в ветви дерева от вершины v до корня: $\{v_1 = v, \dots, v_k = \text{root}\}$. Обозначим поддерева с корнем v_i через R_i . У каждого дерева R_i есть два поддерева. То из них, в котором нет вершины v_{i-1} , обозначим T_i . Пример такого дерева приведен на рисунке:



Алгоритм практически полностью совпадает с алгоритмом разбиения обычного дерева поиска (см. с 195). Только теперь нам следует пользоваться алгоритмом слияния деревьев для сбалансированных деревьев поиска. Пусть высота дерева T_0 равна s_0 . Сольём дерево T_0 с деревом T_{i_1} , результат назовём S_1 и сольём с деревом T_{i_2} (получая S_2), далее будем последовательно использовать деревья T с индексами i_3, \dots, i_K ($i_K \leq h$), получая деревья S_3, \dots, S_K . Будем считать $S_0 = T_0$. Таким образом, $S_{j-1} + T_{i_j} \rightarrow S_j$.

Высота дерева S_j равна либо $\max(h(S_{j-1}), h(T_{i_j}))$, либо $\max(h(S_{j-1}), h(T_{i_j})) + 1$.

Пусть R_i — деревья с корнями в вершинах v_i . Высота дерева R_i равна либо $h(T_i) + 1$, либо $h(T_i) + 2$. Высоты $h(R_i)$ строго возрастают.

Покажем по индукции, что высота дерева S_j равна либо $h(R_{i_j})$, либо $h(R_{i_j}) - 1$, либо $h(R_{i_j}) - 2$. Пусть данное свойство выполнено для $l < j$, тогда

$$h(S_j) = \max(h(S_{j-1}), h(T_{i_j})) \mid \max(h(S_{j-1}), h(T_{i_j})) + 1,$$

следовательно,

$$\begin{aligned} h(S_j) = & \max(h(S_{j-1}), h(R_{i_j}) - 1) \mid \max(h(S_{j-1}), h(R_{i_j}) - 2) \mid \\ & \max(h(S_{j-1}), h(R_{i_j}) - 1) + 1 \mid \max(h(S_{j-1}), h(R_{i_j}) - 2) + 1, \end{aligned}$$

следовательно, по индукции

$$\begin{aligned} h(S_j) = & (h(R_{i_j}) - 1) \mid ((h(R_{i_j}) - 1) \mid h(R_{i_j}) - 2) \mid \\ & h(R_{i_j}) \mid (h(R_{i_j}) \mid h(R_{i_j}) - 1), \end{aligned}$$

следовательно,

$$h(S_j) = h(R_{i_j}) \mid (h(R_{i_j}) - 1) \mid (h(R_{i_j}) - 2).$$

Здесь вертикальной чертой разделяются возможные варианты.

Оценим время работы всего алгоритма:

$$\begin{aligned} T = & O(|h(T_0) - h(T_{i_1})| + 1 + |h(S_1) - h(T_{i_2})| + 1 + \dots + \\ & + |h(S_{K-1}) - h(T_{i_K})| + 1) = \end{aligned}$$

$$\begin{aligned}
&= O(|h(T_0) - h(T_{i_1})| + |h(S_1) - h(T_{i_2})| + \dots + \\
&+ |h(S_{K-1}) - h(T_{i_K})|) + O(h) = \\
&= O(|h(T_0) - h(R_{i_1})| + 4 + |h(R_{i_1}) - h(R_{i_2})| + 4 + \dots + \\
&+ |h(R_{i_{K-1}}) - h(R_{i_K})| + 4) + O(h) = \\
&= O(|h(T_0) - h(R_{i_1})| + |h(R_{i_1}) - h(R_{i_2})| + \dots + \\
&+ |h(R_{i_{K-1}}) - h(R_{i_K})|) + O(h) =
\end{aligned}$$

(ввиду возрастания $h(R_i)$)

$$\begin{aligned}
&= O((h(R_{i_1}) - h(T_0)) + (h(R_{i_2}) - h(R_{i_1})) + \dots + \\
&+ (h(R_{i_K}) - h(R_{i_{K-1}}))) + O(h) = O(h).
\end{aligned}$$

Таким образом, $T = O(h) = O(\log_2 N)$, где N — количество вершин в суммарном дереве. Итак, верна следующая теорема.

Теорема 10. *Для данной вершины v сбалансированного дерева поиска T разбиение дерева T на два сбалансированных дерева поиска T_1 и T_2 , таких что все элементы в T_1 меньше v и все элементы в T_2 больше v , может быть произведено указанным алгоритмом за время $O(\log_2 N)$, где N — количество вершин дерева T .*

12.3. Красно-черные деревья

Красно-черными деревьями называют бинарные деревья поиска, у которых для каждой вершины добавляется дополнительное свойство: вершина является черной или красной. При этом требуется выполнение следующих свойств:

- корень дерева черный;
- у каждой красной вершины потомки черные;
- в любых двух ветвях от корня до листа количество содержащихся черных вершин одинаковое (здесь листом называется вершина с не более чем одним потомком).

Для простоты реализации в дерево добавляются фиктивные черные вершины: для каждой вершины дерева при отсутствии у нее по-

томка на место соответствующего потомка вставляется фиктивная черная вершина.

Вершины, отличные от фиктивных, называются внутренними. У листьев хотя бы один потомок фиктивный. При определении высоты дерева фиктивные вершины учитывать не будем.

Для задания одной вершины красно-черного дерева целых чисел в языке C можно использовать следующую структуру:

```
1 typedef struct SBTTree_  
2 {  
3     int IsRed;  
4     int value;  
5     struct SBTTree_ *par;  
6     struct SBTTree_ *left , *right ;  
7 } SBTTree ;
```

Здесь указатель `par` указывает на родительский элемент данной вершины, а `left` и `right` — на двух потомков, которых традиционно называют левым и правым. Целая переменная `IsRed` указывает, является ли данная вершина красной. Величина `value` называется ключом вершины.

Отступление на тему языка C. Битовые поля

В приведенном примере кажется весьма накладным использовать целую переменную для хранения всего одного бита информации. Можно попробовать отвести под эту переменную меньше памяти:

```
1 typedef struct SBTTreeX_  
2 {  
3     char IsRed;  
4     int value;  
5     struct SBTTreeX_ *par;  
6     struct SBTTreeX_ *left , *right ;  
7 } SBTTreeX ;
```

Однако из-за выравнивания в структурах для большинства современных машин размеры структур `SBTreeX` и `SBTree` окажутся равными.

Можно попробовать «отщипнуть» один бит для переменной `IsRed` из целой переменной с ключом данной структуры `value`, используя

битовые поля в структурах. Битовые поля в структурах — это переменные целого типа, при описании которых после имени переменной пишется двоеточие и вслед за ним количество битов, которые должны быть отведены под данную переменную. Например, в нашем случае можно определить вершину дерева следующим образом:

```

1 typedef struct SBTreel_
2 {
3     unsigned int IsRed :1;
4     unsigned int value :31;
5     struct SBTreel_ *par;
6     struct SBTreel_ *left , *right ;
7 } SBTreel ;

```

При этом следует понимать, что теперь каждая операция с членами структуры `IsRed` и `value` будет осуществляться довольно сложно. Действительно, например, чтобы изменить переменную `value`, требуется сначала извлечь ее из структуры (используя битовые операции), изменить, а затем поместить обратно.

Отметим, что данный подход применим далеко не всегда. Поля в структурах обязаны иметь тип `unsigned int`. В современных версиях языка C это требование немного ослаблено и вместо `unsigned int` часто можно использовать другие целые типы, но, например, тип `float` все равно использовать нельзя.

Отступление на тему языка C. Битовые операции

Язык C позволяет работать с битами в рамках возможностей, предоставляемых обычными ассемблерами. Для работы с битами используются следующие арифметические операции:

- арифметическое «и»: `&`,
- арифметическое «или»: `|`,
- арифметическое «не»: `~`,
- арифметическое «исключающее или»: `^`,
- сдвиг влево на `k` разрядов: `<< k`,
- сдвиг вправо на `k` разрядов: `>> k`.

C помощью этих операций можно осуществить базовые операции с битами:

проверить, отличен ли `k`-й бит целого числа `i` от 0: `(i & (1<<k))!=0`,

положить 1 в k -й бит целого числа i : $i |= (1 << k)$,
 положить 0 в k -й бит целого числа i : $i \&= \sim(1 << k)$,
 присвоить 1-й бит целого числа j k -му биту i :

$$i = ((j \& (1 << 1)) == 0) ? (i \& (\sim(1 << k))) : (i | (1 << k)).$$

Надо иметь в виду, что операция $a << b$ для любых целых переменных эквивалентна умножению a на 2^b . Поэтому побитовый результат данной операции (сдвиг влево с заполнением младших битов нулями) не зависит от знаковости переменной a . Данный простой факт является прямым следствием того, что знаковые целые числа, представленные в дополнительном коде, с арифметическими операциями образуют кольцо вычетов по модулю 2^d , где d — количество бит в представлении числа.

Со сдвигом вправо все хуже. В кольце вычетов нет, вообще говоря, операции деления, и поэтому сдвиг вправо $a >> b$, эквивалентный делению на 2^b , имеет разное представление для знаковых и беззнаковых чисел. Для беззнаковых чисел операция $a >> b$ эквивалентна сдвигу вправо с заполнением старших битов нулями, а для знаковых — сдвигу вправо с дублированием старшего бита. Простейшим выводом из этого является то, что следующий цикл будет вечным:

```
1 for (int a=-1234;a;a>>=1){/*...*/}
```

Для беззнаковых чисел данный цикл обязательно завершится. Настоящая неприятность возникает для целых чисел типа `char`. Для этих чисел нельзя точно сказать — знаковые они или нет. Например, наличие знака у таких чисел можно задавать ключами компилятора. Поэтому конечность выполнения подобного цикла для переменных типа `char` непредсказуема.

Большой неожиданностью является результат выполнения следующего тривиального кода:

```
1 int i=1,j=sizeof(int)*8;
2 cout<<(i<<j)<<" "<<((i<<(j/2))<<(j/2))<<endl;
```

И для компилятора Microsoft, и для компилятора gcc при компиляции без оптимизации (при компиляции без ключей или с ключом `-O0`) на экран будет неожиданно выведено `1 0`. При компиляции с оптимизацией мы увидим на экране строку `0 0`.

Можно долго рассуждать на тему данной странности, но проще вынести мораль: нельзя делать сдвиги на величину, больше или равную количеству битов сдвигаемой целой переменной. Иначе результат будет непредсказуемый.

Высота красно-черного дерева

На то, что в красно-черном дереве, состоящем из N вершин, высота h равна $\Theta(\log_2 N)$, указывает следующий факт: в каждой ветви дерева, начинающейся с корня дерева, не менее половины вершин черные (так как по определению красно-черного дерева за красной вершиной всегда следует черная и ветвь, начинающаяся с корня дерева, начинается с черной вершины). С другой стороны, в каждой ветви находится равное количество черных вершин (следует заметить, что, тем не менее, во всем красно-черном дереве черных вершин может быть меньше половины).

Назовем черной высотой дерева с корневой вершиной r максимальное количество черных вершин во всех ветвях, начинающихся в r и заканчивающихся в листьях, не считая саму вершину r . Будем обозначать ее $h_b(r)$.

Заметим, что требование черноты корня красно-черного дерева, вообще говоря, не является обязательным. Действительно, если не использовать это свойство в определении красно-черного дерева, то в таком дереве цвет корня дерева можно изменить с красного на черный с сохранением всех остальных свойств красно-черных деревьев. Будем называть дерево *красно-черным'*, если из определения красно-черного дерева убрать требование черноты корня. Легко показать, что любое поддерево красно-черного дерева является *красно-черным'*.

Верна следующая лемма.

Лемма. *В красно-черном дереве с черной высотой h_b количество внутренних вершин не меньше $2^{h_b+1} - 1$.*

Доказательство. Заметим, что смена цвета корня дерева не влияет на черную высоту дерева. Поэтому данную лемму можно доказать для красно-черных' деревьев. Будем доказывать лемму по индукции по обычной высоте красно-черного' дерева. Если рассмотреть дерево, состоящее из одного элемента, то для него лемма верна.

Рассмотрим внутреннюю вершину x . Пусть $h_b(x) = h$. Тогда если ее потомок p черный, то высота $h_b(p) = h - 1$, а если ее потомок красный, то $h_b(p) = h$. Таким образом, по предположению индукции в поддеревьях (а они тоже являются красно-черными' деревьями) содержится не менее $2^h - 1$ вершин, а во всем дереве, соответственно, не менее $(2^h - 1) + 1 + 2^h - 1 = 2^{h+1} - 1$ вершин. \square

Если обычная высота красно-черного дерева равна h , то черная высота дерева будет не меньше $h/2 - 1$, и по лемме количество внутренних вершин в дереве $N \geq 2^{h/2} - 1$. Прологарифмировав неравенство, имеем $h \leq 2 \log_2(N + 1)$.

Итак, учитывая, что для любого бинарного дерева $h > \log_2 N$, получаем, что доказана следующая теорема.

Теорема 11. *Для красно-черного дерева, имеющего N внутренних вершин, верна следующая оценка для его высоты: $h = \Theta(\log_2 N)$, или, более точно, $\log_2 N < h \leq 2 \log_2(N + 1)$.*

Добавление элемента в красно-черное дерево

Новая вершина вставляется в красно-черное дерева в два этапа.

На первом этапе вершина вставляется как в обычное дерево поиска (без фиктивных вершин). Новая вершина красится в красный цвет. Следует отметить, что в реальности фиктивных вершин может вообще не быть. Их наличие может обозначаться соответствующими нулевыми указателями у родительских вершин.

Добавление красной вершины x не меняет баланса дерева по черным вершинам. Так как потомки новой вершины фиктивные, то они черные по определению, что соответствует определению красно-черного дерева.

Единственная проблема, которая может возникнуть, это то, что у вставленной красной вершины x может оказаться красный родитель. Требуется изменить дерево, чтобы решить эту проблему. При преобразованиях дерева мы будем сохранять указанное свойство: у нас будет сохраняться балансировка по черным вершинам и единственная проблема будет заключаться в том, что у какой-то красной вершины x окажется красный родитель.

Итак, если $x \rightarrow \text{par}$ красная, то $x \rightarrow \text{par} \rightarrow \text{par}$ черная (так как единственная проблема — нестыковка $x \rightarrow \text{par}$ и x ; с другой стороны, у красной вершины может быть только черный родитель).

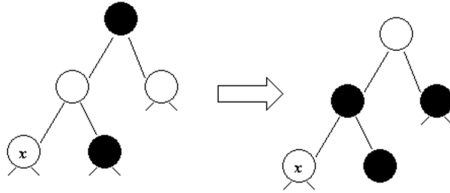
Будем называть вершину $x \rightarrow \text{par} \rightarrow \text{par} \rightarrow \text{next}$, где next — это left или right , *дядей* вершины x , если $x \rightarrow \text{par} \rightarrow \text{par} \rightarrow \text{next} \neq x \rightarrow \text{par}$.

Рассмотрим все возможные случаи.

0. Если вершина вставляется в пустое дерево, то она просто перекрашивается в черный цвет.

1. У вершины $x \rightarrow \text{par}$ нет родителя, т. е. эта вершина корневая. В таком случае мы просто перекрашиваем вершину $x \rightarrow \text{par}$ в черный цвет, и процесс завершается.

2. Дядя вершины x красный.



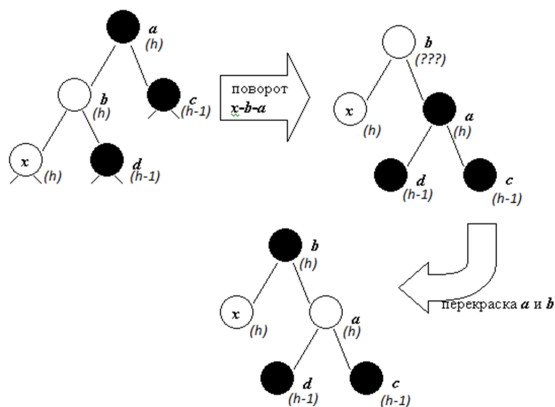
Перекрашиваем родителя, деда и дядю вершины x и рассматриваем в качестве вершины x ее деда: $x = x \rightarrow \text{par} \rightarrow \text{par}$. Таким образом, мы перенесли проблему выше по ветви дерева.

Осталось рассмотреть случаи, когда дядя вершины x черный.

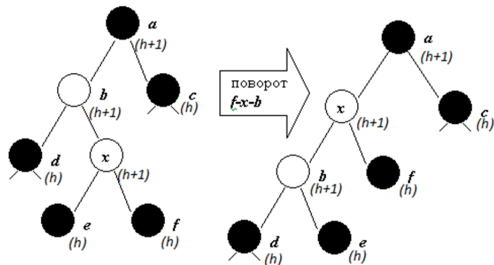
3. Дядя вершины x черный, x — левый потомок $x \rightarrow \text{par}$. Будем в скобках справа от вершины писать черную высоту поддерева (поддерево должно быть красно-черным'), начинающегося с данной вершины. Символами ??? будем обозначать дисбаланс высот (поддерево не является красно-черным').

В этом случае мы делаем правый поворот $x \rightarrow b \rightarrow a$ и в получившемся дереве перекрашиваем две вершины: a и b .

Вершина получившегося дерева черная, проблем с цветами нет, сохранилась черная высота дерева, начинающегося с корня, баланс черного сохранился. Таким образом, дерево сбалансировано. Последующая балансировка не требуется.



4. Дядя вершины x черный, x — правый потомок $x \rightarrow \text{par}$.



Делаем левый поворот $f-x-b$, и ситуация сводится к предыдущему случаю. Заметим, что при этом сохранилась черная высота дерева, начинающегося с корня, баланс черного сохранился.

Все случаи рассмотрены.

Итак, после добавления вершины процесс приведения дерева к красно-черному сводится к некоторому количеству процедур перекраски (не более h раз, где h — высота дерева) и к не более чем двум поворотам. После поворотов дерево не требует дальнейших изменений.

Итак, мы доказали следующую теорему.

Теорема 12. *Указанный алгоритм позволяет добавить вершину к красно-черному дереву за время $O(\log_2 N)$, где N — количество вершин в дереве.*

Однопроходное добавление элемента в красно-черное дерево

Отметим, что оценка максимальной высоты красно-черного дерева несколько больше оценки высоты сбалансированного дерева с тем же количеством вершин. В реальной практике высоты деревьев различаются несущественно (имеется в виду в среднем). Преимуществом красно-черных деревьев является то, что добавление вершин может быть осуществлено за один проход по соответствующей ветви дерева. В сбалансированных деревьях требуется два прохода: первый — для того, чтобы найти вершину, после которой следует вставить новую вершину, а второй — чтобы сбалансировать дерево.

Итак, все, что нам нужно — это не допустить в процессе поиска элемента, после которого будет вставлен новый элемент, реализации случая 2, так как случай 4 сводится к случаю 3, а последний завершает алгоритм. Это можно сделать при выборе листа, после которого следует вставить новую вершину. Нам следует обеспечить, чтобы у вставляемой вершины был либо черный родитель (тогда ничего больше делать не надо), либо черный дядя (тогда дерево можно сделать красно-черным за один или два поворота).

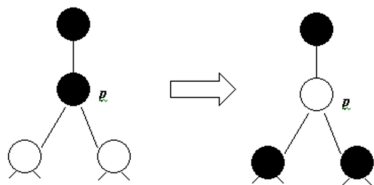
При поиске листа, после которого следует вставить новую вершину, мы сначала рассматриваем в качестве текущей вершины p корень дерева. Далее в качестве p рассматриваем один из потомков корня и т. д.

Пусть для определенности от вершины p мы переходим к вершине $p \rightarrow \text{left}$. Тогда нам следует обеспечить, чтобы, в случае если вершина $p \rightarrow \text{left}$ красная, вершина $p \rightarrow \text{right}$ была бы черной.

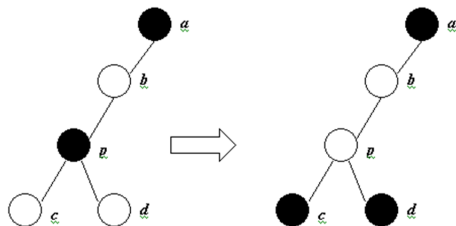
Рассмотрим все возможные случаи. Легко увидеть, что случаи, когда $p \rightarrow \text{left}$ черная или $p \rightarrow \text{right}$ черная, нас устраивают, поэтому следует рассматривать только случаи, когда оба потомка p красные.

0. p — корень дерева, оба потомка p красные. Тогда, все, что нужно сделать — это перекрасить обоих потомков корня p в черный цвет и перейти к рассмотрению следующей вершины $p \rightarrow \text{left}$.

1. Вершина $p \rightarrow \text{par}$ черная, оба потомка p красные. Тогда все, что нужно сделать — это перекрасить p и его потомков и перейти к рассмотрению следующей вершины $p \rightarrow \text{left}$.

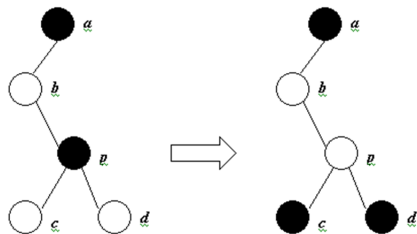


2. Вершина $p \rightarrow \text{par}$ красная, причем p — левый потомок $p \rightarrow \text{par}$, $p \rightarrow \text{par}$ — левый потомок $p \rightarrow \text{par} \rightarrow \text{par}$, оба потомка p красные (случай, когда оба потомка правые аналогичен).



Сначала мы перекрашиваем p и его потомков. Теперь мы попали в ситуацию, аналогичную случаю 3, рассмотренному выше (заметим, что правый потомок a должен быть черным, что обеспечено на предыдущем шаге прохода). Как было показано выше, за один поворот и одну перекраску проблему можно решить.

3. Вершина $p \rightarrow \text{par}$ красная, причем p — правый потомок $p \rightarrow \text{par}$, $p \rightarrow \text{par}$ — левый потомок $p \rightarrow \text{par} \rightarrow \text{par}$, оба потомка p красные (случай, когда p — левый потомок $p \rightarrow \text{par}$, $p \rightarrow \text{par}$ — правый потомок $p \rightarrow \text{par} \rightarrow \text{par}$ рассматривается аналогично).



Сначала мы перекрашиваем p и его потомков. Теперь мы попали в ситуацию, аналогичную случаю 4, рассмотренному выше. Как было показано выше, за два поворота и одну перекраску проблему можно решить.

Стоит отметить некоторую тонкость: несмотря на то, что нашей целью при проходе сверху вниз была ликвидация случая наличия двух красных братьев (после которых вставляется новый элемент!), мы все равно можем иметь после прохода ситуации двух красных братьев вдоль пройденной ветви (случай 3 при вставке элемента создает именно такую ситуацию!). Предлагается самостоятельно осознать, что это замечание не мешает нам решать поставленную задачу.

Итак, мы показали, что алгоритм добавления вершины к красно-черному дереву можно реализовать за один проход по соответствующей ветви дерева.

Удаление элемента из красно-черного дерева

Сначала мы удаляем вершину как в обычном дереве поиска.

Если у удаляемой вершины y всего один внутренний потомок x , то мы просто ставим x на место y . Если вершина y была красной, то проблем не возникает (черная длина дерева не изменяется). Если вершина y черная, а x красная, то проблем тоже нет: мы перекрашиваем вершину x , вставшую на место вершины y , в черный цвет и RB-свойства будут выполняться. Наконец, если обе вершины x и y черные, то нам придется присвоить вершине y двойную черноту. Как с ней бороться, будет ясно далее.

Если у удаляемой вершины y два внутренних потомка $w=y-\rightarrow$ right, $z=y-\rightarrow$ left, то мы извлекаем следующий элемент за y (минимальный в дереве с корнем w) и ставим его на место y . Вспомним, что минимальный элемент в дереве поиска всегда является листом.

Теперь все проблемы сместились к вершине, являющейся листом. Ситуацию, когда у листа есть один потомок, мы уже разобрали. Если же у листа нет потомков, то при его удалении вершина становится фиктивной, что не будет противоречить дальнейшему алгоритму. Если данная вершина была красной, то она просто перекрашивается в черный цвет (уже в качестве фиктивной вершины). Если же она была черной, то ей необходимо приписать двойную черноту.

Итак, задача сводится к следующей. Есть вершина в красно-черном дереве x с двойной чернотой. Все свойства красно-черного дерева выполняются. Требуется привести дерево к такому виду, что в нем все вершины будут просто черными или красными.

Далее мы будем производить некоторые манипуляции с окрестностью вершины x , которые будут или перебрасывать двойную черноту вверх по дереву, или приведут дерево к требуемому виду. Если в результате наших манипуляций корень дерева приобретет двойную черноту, то мы просто сделаем его черным, что завершит работу алгоритма.

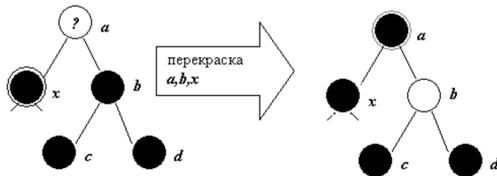
Рассмотрим различные варианты.

1. Брат x красный.
- 2—4. Брат x черный.
 2. Потомки брата x черные.
 3. Правый потомок брата x черный, левый — красный.
 4. Правый потомок брата x красный.
1. Брат x красный.



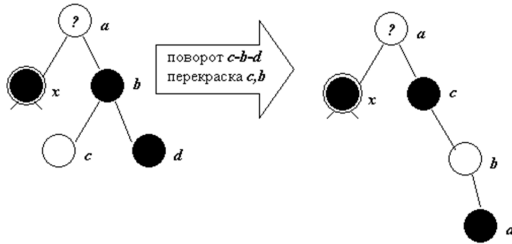
Вершина x остается с двойной чернотой, но получает черного брата. Ситуация сводится к вариантам 2—4, рассматриваемым далее.

2. Брат x черный, потомки брата x черные.



Одна чернота x и чернота b переходят к их родителю. Если родитель был красным, то процесс на этом завершается. Иначе рассматриваем далее в качестве вершины x вершину a .

3. Правый потомок брата x черный, левый — красный.



Делаем правый поворот $c-b-d$ и перекрашиваем вершины b и c . В результате получаем, что правый потомок брата x красный, т. е. приходим к случаю 4, рассматриваемому далее.

4. Правый потомок брата x красный, цвет левого не важен.



Делаем левый поворот $d-b-a$ и осуществляем указанную перекраску (x становится просто черной). При этом цвет корня дерева и вершины c не должны меняться.

Разберемся с балансировкой. Пусть $h_b(x) = h$ (не забываем, что в $h_b(x)$ не учитывается сама вершина x). Тогда $h_b(a) = h + 2$, $h_b(b) = h + 1 = h_b(d)$ = количество черных вершин в любой ветви, начинающейся с c . Простой проверкой получаем, что новое дерево является красно-черным.

Итак, мы завершили разбор операции удаления вершины в красно-черном дереве.

13. Python. Аналоги массивов

13.1. Python. Списки (lists). Срезы

Аналогами массивов в Python выступают *списки*. Квадратные скобки в языке выступают признаком списка. Например, квадратные скобки без содержимого между ними задают пустой список. Присвоить переменной *v* пустой список можно путем *v=[]*. Создать список из трех переменных можно следующим образом:

```
1 m=[1,2,3]
```

Список можно рассматривать как массив ссылок на переменные с индексами от 0 до $\text{len}(m)-1$, где *m* — имя списка. Типы переменных в списке могут быть различными.

Далее будет показано, что списки в Python организованы аналогично контейнеру *дек* в STL. Список в терминах языка C представляет собой массив указателей на объекты, т. е. в самом списке память отводится только под массив указателей на объекты, а сами объекты хранятся уже по адресам, заданным этими указателями. Убедиться в этом можно, добавляя к изначально пустому списку по одной целой переменной и выводя размер списка на экран с помощью функции `sizeof()` из модуля `sys`:

```
1 from sys import sizeof as sz
2 m=[];sz0=sz2=sz(m)
3 for i in range(1,78):
4     sz1=sz2;m.append(i);sz2=sz(m)
5     if (sz1!=sz2): print("%3.3d: %d"%(sz1,(sz1-sz0)//8))
6     else:         print("%3.3d"%(sz1),end=" ")
```

Функция `sizeof()` выводит размер непосредственно списка без учета данных, на которые указывают указатели из массива указателей списка, т. е. выводится размер массива указателей плюс размер каких-то служебных переменных. При изменении размера отведенной памяти после двоеточия выводится размер массива указателей (размер всей отведенной памяти минус размер памяти, отведенной на пустой список), деленный на размер указателя (8 байт), и происходит переход на следующую строку. В результате выполнения данного скрипта на экране появляется следующая информация:

```

1 056: 0
2 088 088 088 088: 4
3 120 120 120 120: 8
4 184 184 184 184 184 184 184 184: 16
5 248 248 248 248 248 248 248 248: 24
6 312 312 312 312 312 312 312 312: 32
7 376 376 376 376 376 376 376 376: 40
8 472 472 472 472 472 472 472 472 472 472 472 472: 52
9 568 568 568 568 568 568 568 568 568 568 568 568: 64
10 664 664 664 664 664 664 664 664 664 664 664 664: 76

```

Легко увидеть, что сначала отводится память под массив из четырех указателей, потом реаллокируется память под массив из восьми указателей, потом — из 16 и т. д. Мы наблюдаем ту же самую стратегию, что и при отведении памяти под вектор в STL при добавлении элементов по одному в конец вектора. Переотведение памяти происходит, когда массив указателей заполняется полностью. Цель данной стратегии — достичь амортизационной сложности добавления элемента $O(1)$.

Вставить значение в список можно не только в конец списка, но и в произвольную позицию с помощью метода списка `insert`. Например, вставить число 10 в начало списка можно следующей инструкцией:

```
1 m.insert(0, 10)
```

Удалить первый встречающийся элемент списка с заданным значением со схлопыванием остатка списка можно методом `remove`. Следующая команда удаляет из списка занесенный на предыдущей операции элемент:

```
1 m.remove(10)
```

Удалить элемент списка в заданной позиции со схлопыванием остатка списка можно методом `pop`. Следующая команда удаляет первый элемент списка:

```
1 m.pop(0)
```

Метод `pop()` без параметров удаляет последний элемент списка, т. е. просто уменьшает на единицу длину списка.

Добавить один список в конец другого можно методом `extend()`:

```
1 m1=[1,2,3]
2 m.extend(m1)
```

Из списка можно выбирать элементы с заданным шагом, образуя новые списки. Такая выборка называется *срезом*. Например, для списка

```
1 m=[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`m[0:5]` задает список из первых пяти элементов списка `m` (с индексами от 0 до 5, не включая 5);

`m[0:5:2]` выбирает из первых пяти элементов списка `m` элементы с шагом 2, т. е. данный срез задает список `[1, 3, 5]`.

Отрицательное число в границах списка задает номер элемента от конца списка (последний элемент списка имеет номер `-1`, предпоследний — `-2` и т. д.). Например, срез `m[-4:-1]` задает в нашем случае список `[6, 7, 8]`, поскольку элемент с индексом `-1` в срез не заносится.

`m[0:-2:2]` задает выборку из списка от первого до третьего с конца элемента включительно с шагом 2, т. е. список `[1, 3, 5, 7]`.

Срезы можно использовать, например, для эмуляции двумерных массивов, если исходные данные организованы как обычный список элементов двумерного массива, записанного по строкам. Например, пусть у нас есть список `m`, состоящий из `p*q` элементов, где `p` и `q` — некоторые целые переменные. Тогда данный список можно рассматривать как матрицу размерности $p \times q$ (p строк и q столбцов), где копия i -й строки задается срезом `m[i*q:(i+1)*q]`. Еще раз подчеркнем, что `m[i*q:(i+1)*q]` задает только **копию** элементов исходного списка `m` в понимании Python, т. е. присваивание значения элементу среза изменит только элементы среза, но изменение элемента среза изменит также и элемент исходного массива. Так, например, следующие действия изменят как срез, так и исходный массив:

```
1 m=[[1,2],[3,4]]
2 n=m[0:1]
3 n[0][0]=2
```

но следующий пример приведет к изменению только среза:

```
1 m=[1,2]
2 n=m[0:1]
3 n[0]=2
```

13.2. Python. Сортировки. Лямбда-функции

Метод `sort()` сортирует данный список по возрастанию, изменяя его:

```
1 m=[3,2,1]
2 m.sort()
```

Функция `sorted()` создает отсортированную по возрастанию копию списка, не меняя сам список:

```
1 m=[3,2,1]
2 m1=sorted(m)
```

Здесь стоит отвлечься на понятие *лямбда-функция* в языке Python. Вообще, лямбда-функцией в программировании называется функция, не имеющая имени, которая задается непосредственно в месте, где она необходима. Лямбда-функции в языке Python весьма тривиальны: в них можно записать только одно выражение, которое задает возвращаемое функцией значение. Чтобы задать лямбда-функцию, нужно написать ключевое слово **lambda**, список формальных переменных функции, двоеточие и выражение, задающее выходное значение функции. Например, лямбда-функция, возвращающая свой аргумент с противоположным знаком имеет вид:

```
1 lambda x:-x
```

Логично, что лямбда-функцию можно использовать как простую функцию:

```
1 y=(lambda x:-x)(10)
```

Лямбда-функцию можно присвоить переменной:

```
1 fun=(lambda x:-x)
```

Теперь `fun` можно использовать как соответствующую именованную функцию.

В функции сортировки `sorted()` и методе сортировки списка `sort` можно использовать параметр `key=fun`, где `key` — ключевое слово, а `fun` — имя функции, возвращающей ключ сортировки по значению элемента списка. Здесь имеется в виду, что функция сортировки сначала для каждого значения вычислит его ключ с помощью соответствующей функции `fun`, а собственно сортировка будет происходить по возрастанию вычисленных ключей. Например, получить отсортированную по убыванию копию списка можно следующим способом:

```
1 srt=(lambda x:-x)
2 m2=sorted(m, key=srt)
```

Отсортировать сам список по убыванию можно следующим способом:

```
1 m.sort(key=(lambda x:-x))
```

13.3. Python. Кортежи (tuples)

Кортежи представляют собой полные аналоги списков, но для них запрещены изменения внутри объекта и все методы и функции, изменяющие значения внутри кортежа. Круглые скобки в языке выступают признаком кортежа. Например, круглые скобки без содержимого между ними задают пустой кортеж. Присвоить переменной `v` пустой кортеж можно короткой командой `v=()`. Создать кортеж из трех переменных можно следующей командой:

```
1 m=(1,2,3)
```

Выражение `(x)` не задает кортеж, поскольку оно интерпретируется языком Python просто как переменная `x` в скобках. Задать кортеж из одной переменной можно выражением `(x,)`.

Поскольку кортеж нельзя изменять, к нему не применим метод `sort()`, однако функция `sorted`, примененная к кортежу, возвращает список, состоящий из отсортированных элементов исходного кортежа.

Срезы для кортежей работают и возвращают соответствующие срезы, также являющиеся кортежами.

Кортеж можно преобразовать в список:

```
1 n=(1,2,3); m=list(n)
```

Важной особенностью языка Python является то, что если при определении кортежа через набор переменных, разделенных запятыми, окруженный круглыми скобками, убрать круглые скобки, то оставшийся набор переменных, разделенных запятыми, все равно будет интерпретирован как кортеж, т. е. последний пример можно слегка сократить без изменения смысла:

```
1 n=1,2,3; m=list(n)
```

Запрет на изменение кортежа не распространяется на изменения переменных внутри кортежа. Поэтому выражение, активно используемое в Python для обмена местами двух переменных:

```
1 a,b=b,a
```

интерпретируется языком просто как присваивание одного кортежа другому кортежу. При этом `a` становится ссылкой на объект, на который ранее ссылалась переменная `b`, а `b` становится ссылкой на объект, на который ранее ссылалась переменная `a`. Например, выполнение скрипта

```
1 a=1; b=2; print(id(a),id(b)); a,b=b,a; print(id(a),id(b))
```

дает следующий вывод на экран:

```
1 140735193696696 140735193696728
2 140735193696728 140735193696696
```

Забегая вперед, надо отметить, что важным преимуществом кортежей перед списками является то, что кортежи могут выступать в качестве индексов словарей, речь о которых пойдет дальше, в то время как списки такой возможностью похвастаться не могут.

13.4. Python. Словари (dictionaries, dict)

Словари в языке Python представляют собой ассоциативные массивы — аналоги списков (массивов), наборы пар «ключ—значение»,

где значения могут ассоциироваться с целыми и вещественными переменными, строками, кортежами (и не могут ассоциироваться со списками и словарями).

Фигурные скобки в языке Python выступают признаком словаря. Например, фигурные скобки без содержимого между ними задают пустой словарь. Присвоить переменной `v` пустой словарь можно короткой командой `v={}`. Каждый элемент словаря задается ключом и значением. Например, создать словарь из трех элементов можно следующей командой:

```
1 m={1:" first ", 2:"second", 3:"third"}
```

Тогда выражение `m[1]` имеет значение «first», выражение `m[2]` — «second», `m[3]` — «third».

Добавлять элементы в словарь и заменять элементы в словаре можно простым присваиванием:

```
1 m={1:" first ", 2:"second", 3:"third" }
2 m[4]=" fourth "
```

Обращение к несуществующему элементу списка приведет к выбросу исключения `KeyError`. Однако у словаря присутствует метод `get`, который по ключу возвращает соответствующее значение, а при отсутствии в словаре ключа возвращает `None`:

```
1 if(m.get(5)==None): print("5 not defined")
```

Список ключей словаря можно получить с помощью метода `keys()`:

```
1 print(d.keys())
```

Список значений словаря можно получить с помощью метода `values()`:

```
1 d.values()
```

Метод `pop()` позволяет удалять пару «ключ—значение» по заданному ключу:

```
1 d.pop(3)
```

Python дает возможность легко работать с массивами (например, двумерными), организованными в виде словаря с ключами — кортежами, состоящими из (соответственно двух) целых переменных.

Например, следующий скрипт создает аналог двумерного массива размером 3×4 , находит его размерности и выводит их на экран:

```
1 d={}
2 for i in range(0,3):
3     for j in range(0,4):
4         d[i,j]=i*10+j
5 m=1+max(x[0] for x in d.keys())
6 n=1+max(x[1] for x in d.keys())
7 print(m,n)
```

Вопрос об эффективности работы с такими псевдомассивами остается за рамками данного обсуждения. Можно только сказать, что словари в Python реализованы с помощью хэш-таблиц, работу с которыми мы обсуждали ранее (см. [11]). Простейший вывод тогда заключается в том, что если в качестве ключей используются подряд идущие числа, то высока вероятность отсутствия коллизий. Поэтому работа с хэш-множествами в нашем последнем примере должна оказаться весьма эффективной.

13.5. Python. Решение системы линейных уравнений. Матрица Гильберта. Классы

Конечной целью данного раздела является написание функции решения невырожденной системы линейных уравнений $Ax = b$ (здесь A — заданная матрица размером $n \times n$, b — заданный вектор длины n , x — искомый вектор длины n). Для начала потребуем от нашей функции, чтобы она работала для невырожденной матрицы, состоящей из вещественных переменных, но будем иметь в виду, что в качестве типа переменных, из которых состоят матрица и вектор, может выступать не только тип `float`, но и новый тип, который мы впоследствии создадим.

Будем далее реализовывать обычный *метод Гаусса* [1]. Обычно под ним подразумевается следующий двухпроходный алгоритм: на первом (прямом) проходе с помощью элементарных преобразований мы приводим расширенную матрицу к верхнетреугольному виду, а на втором (обратном) проходе опять же с помощью элементарных преобразований приводим расширенную матрицу к виду, в котором матрица становится единичной. Тогда правая часть будет представ-

лять собой решение системы линейных уравнений. Для начала упростим задачу: будем предполагать, что диагональные элементы, возникающий при прямом проходе ненулевые. Тогда функция решения системы линейных уравнений примет довольно простой вид:

```

1  def gauss(a,b):
2  #ищем n=размер матрицы:
3      n=1+max(x[0] for x in a.keys())
4      for i in range(n):#проход сверху вниз:
5          for j in range(i+1,n):
6              K=a[j,i]/a[i,i]; b[j]=b[j]-b[i]*K
7              for k in range(i,n): a[j,k]=a[j,k]-a[i,k]*K
8      for i in range(n-1,-1,-1):#проход снизу вверх:
9          b[i]=b[i]/a[i,i]
10         for j in range(i-1,-1,-1): b[j]=b[j]-b[i]*a[j,i]
11     return b

```

Будем считать, что индексы матрицы принимают значения $0 \leq i, j < n$.

В численных методах под методом Гаусса обычно подразумевается вышеприведенная процедура, в которой для каждого i сначала ищется элемент $A_{j,i}$ для $i \leq j < n$ с максимальным модулем. Пусть найденный элемент имеет индекс $j_{\max,i}$. Далее перед обнулением элементов i -го столбца, находящихся ниже диагонали, происходит обмен местами строк расширенной матрицы с номерами i и j_{\max} . Это гарантирует нам, что элемент $A_{i,i}$ будет иметь максимальный модуль среди всех элементов, расположенных строго под ним. А это, в свою очередь, обеспечит условие $|K| \leq 1$, где K — коэффициент, на который умножается i -я строка при вычитании ее из j -й строки. Мы получаем минимально возможную скорость роста модулей чисел при приведении матрицы к верхнетреугольному виду в методе Гаусса. Наш скрипт немного усложнится:

```

1  def gauss(a,b):# предполагаем, что a квадратная
2  n=1+max(x[0] for x in a.keys())#ищем размер матрицы
3  for i in range(n):#прямой ход метода Гаусса
4      jmax=i #ищем макс. по модулю элемент в первом столбце:
5      for j in range(i,n):
6          if (Abs(a[j,i])>Abs(a[jmax,i])):jmax=j
7      b[i],b[jmax]=b[jmax],b[i]#меняем местами строки i и jmax
8      for j in range(i,n): #в расширенной матрице

```

```

9     a[i, j], a[jmax, j]=a[jmax, j], a[i, j]
10    for j in range(i+1, n): #для строк подматриц, кроме первой
11        K=a[j, i]/a[i, i]      #хочется обнулить a[j, i]
12        b[j]=b[j]-b[i]*K
13        for k in range(i, n): #для всех элементов j-й строки:
14            a[j, k]=a[j, k]-a[i, k]*K
15    for i in range(n-1, -1, -1): #обратный ход метода Гаусса
16        b[i]=b[i]/a[i, i]      #в уме: получаем: a[i, i]=1
17        for j in range(i-1, -1, -1): #обнуляем столбец над a[i, i]
18            b[j]=b[j]-b[i]*a[j, i]
19    return b

```

Здесь вместо встроенной функции `abs()` использована собственная функция `Abs()`, которую мы определим позднее. Для вещественных чисел она будет просто вызывать встроенную функцию `abs()`, поэтому пока можно не заострять на ней внимание.

Для отладочных целей нам была бы полезна функция аккуратно-го вывода на экран матрицы (в нашем случае — словаря) и вектора (списка) вместе со строкой комментария, и мы начнем знакомиться с языком Python как объектно-ориентированным языком. Мы не можем сказать, что Python обладает свойством *полиморфизма* в обычном смысле этого слова: в нем нет прямой поддержки перегрузки функций на основе типов аргументов, ведь тип аргументов функции выясняется только при ее вызове. Однако можно использовать банальный условный оператор для выяснения типа переменной, например: `if(type(a)==list)...` Таким образом, мы можем написать отдельный код для вывода списка и отдельный — для словаря, индексированного кортежем, состоящим из двух целых переменных:

```

1    def show(a, text=""):
2        print(text) #вывод текста комментария
3        if (type(a)==list): #если надо вывести список (вектор):
4            print("(", end="")
5            for x in a: print(str(x), end=" ")
6            print(")")
7        elif (type(a)==dict): #если надо вывести словарь (матрицу):
8            n=1+max(x[0] for x in a.keys()) #размер матрицы
9            l=[0]*n #список длины n нулевых значений
10           for i in range(0, n): #ищем l[j] = ширину j-го столбца

```

```

11     for j in range(0, n): l[j]=max(l[j], len(str(a[i, j])))
12     for i in range(0, n):# j-й столбец в поле ширины l[j]
13         for j in range(0, n):
14             print("%*s"%(l[j], str(a[i, j])), end=" ")
15     print()

```

Здесь для матрицы (словаря) мы вычисляем ширину каждого j -го столбца матрицы как максимум размеров текстового представления элементов матрицы: `len(str(a[i, j]))`. Вывод каждого элемента матрицы осуществляется по аналогии с выводом переменных в языке C с использованием символа `*` в строке формата. Далее в строке с выводимыми переменными этому символу должна соответствовать целая переменная, задающая значение данного поля. В нашем случае вывод текстовой переменной осуществляется в поле ширины `l[j]` с помощью команды `print("%*s"%(l[j],str(a[i, j])) ...`.

Также нам понадобится функция умножения матрицы на вектор:

```

1 def mlt(a, b):
2     n=1+max(x[0] for x in a.keys())
3     c=[0]*n #[0. if (type(a[0,0])==float) else Frac()]*n
4     for i in range(0, n):
5         for j in range(0, n):
6             c[i]=c[i]+a[i, j]*b[j]
7     return c

```

Комментарии в данном коде подчеркивают, что, хотя здесь `c` является списком целых переменных, в дальнейшем мы будем использовать элементы этого списка также для сложения с другим типом `Frac`, который будет определен ниже.

Наконец, напишем простой тест для реализованного метода Гаусса, состоящий в создании некоторого вектора b и матрицы A , вычислении их произведения $c = Ab$ и решении системы линейных уравнений $Av = c$.

```

1 def main():
2     A={}; b=[1, 2, 3]; n=len(b)
3     for i in range(0, n):
4         for j in range(0, n):
5             A[i, j]=(i+1)**j
6     c=mlt(A, b)

```

```
7     show(c, "A*b=")
8     v=gauss(A, c)
9     show(v, "rez=")
10
11 main()
```

Далее легко визуально сравнить полученное решение v с исходным вектором b .

Попробуем применить написанную реализацию метода Гаусса для нашего собственного класса рациональных дробей.

Класс `Classname` определяется в языке Python с помощью инструкции `class Classname:` с последующим определением элементов и методов класса. Элементы и методы класса записываются с соответствующим отступом в соответствии с правилами создания блоков в языке Python. Имена элементов класса просто записываются в очередной строке блока класса, возможна инициализация их значений обычным присваиванием:

```
1 class Frac:
2     a=0
3     b=1
```

На самом деле (если углубляться в подробности) в языке Python существует различие между элементами (атрибутами) *класса* и атрибутами *экземпляра класса*. Атрибут класса Python в каком-то смысле является аналогом статического элемента класса языка C++. Атрибут экземпляра класса Python является аналогом обычного элемента класса C++. Заданные нами элементы a и b формально являются атрибутами класса, т. е. они должны быть общими для всех экземпляров класса `Frac`. Но в силу особенностей модели памяти Python эта разница весьма расплывчата. При присваивании атрибутам класса новых значений атрибуты класса сразу превращаются в атрибуты экземпляра класса. Поэтому в дальнейшем мы не будем разделять два описанных понятия и будем просто говорить про элементы класса.

Все методы класса должны иметь своим первым параметром ключевое слово `self`, являющееся аналогом слова `this` в языке C++. В отличие от языка C++, в языке Python обращение к элементам класса

из методов класса может происходить только с использованием ключевого слова `self`. Например:

```

1 class Frac:
2     a=0
3     b=1
4     def Show(self): print(self.a, self.b)
5 x=Frac(); x.Show()
```

Все специфические методы, переопределяющие стандартные операторы, в языке Python задаются в виде функций, в именах которых в начале и конце стоят по два символа подчеркивания. Например, конструктор класса задается функцией `__init__`.

Для работы с дробями нам потребуется операция сокращения дробей. Функцию сокращения дроби проще всего реализовать с помощью функции `math.gcd` — *greatest common divisor*.

```

1 from math import gcd as gcd
2 def shorter(a,b):
3     if (b<0): a*=-1;b*=-1
4     c=gcd(a,b)
5     if (c): a//=c;b//=c
6     return a, b
```

Операторы преобразования типа от типа данного класса к типу `type` задаются с помощью метода с именем `__type__`.

Для обеспечения возможности вывода переменной нового типа с помощью функции `print()` достаточно задать в классе оператор преобразования данного типа к типу строки, т. е. должен быть определен метод `__str__`.

Имена методов, переопределяющих арифметические операции, легко запомнить: оператор `+` задается методом `__add__`, оператор `-` — методом `__sub__`, оператор `*` — методом `__mul__`, оператор `//` — методом `__div__`, оператор `/` — методом `__truediv__`. При этом левый операнд операции соответствует `self`, первому аргументу метода, а правый операнд — второму аргументу.

На самом деле если мы хотим использовать уже написанную функцию умножения матрицы на вектор, то мы столкнемся с проблемой: первоначально значениями списка `c` являются целые переменные, нам надо выполнять операцию `c[i]=c[i]+A[i,j]*b[j]`, предпола-

гающую сложение целого числа $c[i]$ и дроби $A[i,j]*b[j]$, но первому операнду (`self`) в методе `__add__` класса `Frac` соответствует дробь, и сложить целое число и дробь с помощью этого метода оказывается невозможным. Выходом из этой ситуации в языке Python являются *отраженные арифметические операторы*. Отвечающие им методы аналогичны описанным, но `self` в них соответствует правому операнду бинарного оператора, а второй аргумент — левому. Имена методов отраженных операторов аналогичны обычным, но перед содержательной частью имени добавляется буква «r». Например, отраженному оператору сложения соответствует метод `__radd__`.

Таким образом, мы получаем следующее определение класса для работы с рациональными дробями.

```

1 class Frac:
2     a=0#числитель
3     b=1#знаменатель
4     def __init__(self, a=0,b=1):#конструктор
5         a,b=shorter(a,b)
6         self.a=a; self.b=b
7     def __str__(self):
8         return "("+str(self.a)+" "+str(self.b)+")"
9     def __float__(self):
10        return self.a/self.b
11    def __add__(self, x):#self+x
12        if (type(x)==int):x=Frac(x)#могу складывать дробь и целое
13        a=self.a*x.b+self.b*x.a; b=self.b*x.b
14        return Frac(a,b)
15    def __radd__(self, x):#x+self
16        if (type(x)==int):x=Frac(x)#могу складывать целое и дробь
17        a=self.a*x.b+self.b*x.a; b=self.b*x.b
18        return Frac(a,b)
19    def __sub__(self, x):#self-x
20        if (type(x)==int):x=Frac(x)
21        a=self.a*x.b-self.b*x.a; b=self.b*x.b
22        return Frac(a,b)
23    def __mul__(self, x):#self*x
24        if (type(x)==int):x=Frac(x)
25        a=self.a*x.a; b=self.b*x.b
26        return Frac(a,b)

```

```

27     def __truediv__(self, x):#self/x
28         if (type(x)==int):x=Frac(x)
29         a=self.a*x.b; b=self.b*x.a
30         return Frac(a,b)

```

Здесь для дробей определены операторы сложения, вычитания, умножения, деления. Также для примера показано, как определить оператор сложения дроби и целого числа и как определить оператор вычитания целого числа из дроби. Отметим, что во всех операциях будет происходить сокращение полученных дробей, поэтому числитель и знаменатель результата любой операции будут взаимно просты.

Мы до сих пор не определили функцию Abs(), которая использовалась в методе Гаусса, сделаем это:

```

1  def Abs(x):
2      if (type(x)==float):return abs(x)
3      if (type(x)==int):return abs(x)
4      if (type(x)==Frac):return (abs(x.a)/x.b)

```

Легко написать тест для решения системы линейных уравнений в дробях, аналогичный приведенному выше.

```

1  def main():
2      a={}; b=[Frac(1,1),Frac(2,1),Frac(3,1)]; n=len(b)
3      for i in range(0,n):
4          for j in range(0,n):
5              a[i,j]=Frac((i+1)**j,1)
6      show(a,"A=")
7      c=mlt(a,b)
8      show(c,"c=")
9      v=gauss(a,c)
10     show(v,"v=")
11
12 main()

```

Реализовав класс Frac, мы получили возможность производить вычисления без ошибок округления. Значимость этого можно проиллюстрировать на решении системы линейных уравнений с матрицей Гильберта. Эта матрица имеет весьма невзрачный вид: $a_{i,j} = 1./(i+j+1)$ (при индексации элементов матрицы с нуля), можно доказать, что она невырождена. Однако решение даже сравнительно

небольших систем линейных уравнений с этой матрицей стандартным методом Гаусса, использующим обычные вещественные числа с плавающей точкой, заканчивается провалом. Причина этого — плохая обусловленность матрицы Гильберта (см. [1]). Напишем простой тест для решения данной системы линейных уравнений:

```

1 def main():
2     n=14; a={}; b=[]
3     Type=float
4     # матрица Гильберта
5     for i in range(0,n):
6         b.append(Type(1))
7         for j in range(0,n):# (a?b:c) <=> b if(a) else c
8             a[i,j]=1/(i+j+1) if Type==float else Frac(1,i+j+1)
9     show(b,"b=")
10    c=mlt(a,b) # show(c,"c=a*b=")
11    rez=gauss(a,c)
12    show(rez,"rez=")

```

В данном тесте используется переменная `Type` для задания вида теста: для `Type=float` реализуется тест в вещественных числах, а для `Type=Frac` реализуется работа с созданным классом дробей. Изначально задается вектор `b` из единиц размером `n=14` чисел. Происходит умножение матрицы Гильберта на этот вектор, и результат помещается в переменную `c`. Далее решается система линейных уравнений с матрицей Гильберта и вектором `c` в правой части. На экран выводится исходный вектор `b` и результат решения системы линейных уравнений. Оказывается, что при работе с вещественными числами при размере матрицы `n=10` результат оказывается вполне приемлемым:

```

1 b=
2 (1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 )
3 rez=
4 (0.9999999990954135 1.0000000776380038 0.9999983542947577
5 1.0000149062956762 0.9999291061364417 1.0001944259708704
6 0.9996816350152051 1.000307148615796 0.9998389814644577
7 1.000035366078449 )

```

Однако уже при $n=13$ система линейных уравнений решается некорректно:

```
1 b=  
2 (1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 )  
3 rez=  
4 (1.000000068924765 0.9999892239930845 1.0004135381103192  
5 0.9931474946926484 1.0612462082163256 0.6692612444601342  
6 2.1490741314491006 -1.6541711897923186 5.118548079749099  
7 -3.2430499393203998 3.7830048956575983 -0.05179281738579618  
8 1.174329116501686 )
```

Вполне ожидаемо при работе с дробями результат оказывается точным. Например, при $n=100$ система линейных уравнений на современном компьютере решается за время порядка секунды и выдает точное решение. Поскольку время решения системы линейных уравнений равно $\Theta(n^3)$, то уже для $n=200$ Python решает данную систему линейных уравнений в дробях за время порядка 10 секунд. Для сравнения, если бы мы написали метод решения системы линейных уравнений на языке C/C++, то даже для $n=1000$ система линейных уравнений решалась бы на вещественных числах с плавающей точкой в режиме реального времени (т. е. визуально время решения задачи было бы нулевым). Безусловно, решение системы линейных уравнений в дробях в реальной практике не особо востребовано, поэтому приведенный пример является по большому счету чисто демонстрационным.

13.6. Python. Неприятности. Изменение переменных. Параметры функций по умолчанию

Поговорим немного о неприятностях, проистекающих из сущности присваивания в языке Python. Мы уже говорили, что присваивание в Python фактически является присваиванием ссылок, а не переменных. Для иллюстрации возможных проблем рассмотрим функцию умножения матрицы на вектор, которую мы ввели в предыдущем разделе. Попробуем ее немного «улучшить».

Сложение в функции `mlt()` выглядит для нас несколько раздражающе, поскольку, имея опыт работы с языком C, мы, безусловно,

хотели бы заменить оператор `+` оператором `+=`. Мы можем попробовать сделать это так:

```

1 def mlt(a, b): #работает для списков целых и дробей
2     n=1+max(x[0] for x in a.keys())
3     c=[Frac() if (type(b[0])==Frac) else 0]*n
4     for i in range(0, n):
5         for j in range(0, n):
6             c[i]+=a[i, j]*b[j]
7     return c

```

Любой арифметический оператор вида `op=` может быть определен в классе в виде функции с именем `__iFun__`, где `Fun` — имя функции, соответствующей данному оператору. Например, оператору `+=` соответствует метод класса с именем `__iadd__`:

```

1 def __iadd__(self, x): #self+=x
2     if (type(x)==int): x=Frac(x) #для сложения дроби и целого
3     a=self.a*x.b+self.b*x.a; b=self.b*x.b
4     self.a=a; self.b=b
5     return Frac(a, b)

```

Мы без проблем можем добавить данный метод к нашему классу дроби. После чего с удивлением (или без такового) убедимся в том, что написанная нами программа решения системы линейных уравнений работает неправильно. Точнее, она не будет правильно работать при работе с дробями, но при работе с вещественными числами проблем у нас не будет.

Действительно, начальный вектор `s` состоит из ссылок на одну и ту же переменную (`Frac()` или `0`). Оператор `+=` для класса (но не для целых или вещественных переменных!) осуществляет изменение переменной данного типа, не создавая новой переменной. В результате при работе с дробями все операции `c[i]+=a[i,j]*b[j]` для всех `i` приводят к изменению одной и той же переменной. При работе с вещественными числами операция прибавления изменяет разные переменные.

С проблемой можно справиться, например создав в начале процедуры умножения вектор, состоящий из различных объектов:

```

1 def mlt(a, b):
2     c = []; n=1+max(x[0] for x in a.keys())

```

```

3   for i in range(n):
4       c.append(Frac() if (type(b[0])!=Frac) else 0)
5   for i in range(0,n):
6       for j in range(0,n): c[i]+=a[i,j]*b[j]
7   return c

```

Другим вариантом решения проблемы является отказ от изменения объекта в функции `__iadd__()`. Если мы сократим эту функцию:

```

1   def __iadd__(self, x): #self+=x
2       if (type(x)==int): x=Frac(x) #для сложения дроби и целого
3       a=self.a*x.b+self.b*x.a; b=self.b*x.b
4       return Frac(a, b)

```

то операция `c[i]+=a[i,j]*b[j]` будет изменять только данный элемент списка, не меняя его первоначальные копии.

Несмотря на возможность решения указанной проблемы, мы видим, насколько опасным может быть дословное перенесение логического математического алгоритма в язык Python. В больших проектах подобные ошибки могут привести к катастрофе.

Еще одной причиной возможных ошибок в Python является попытка изменения переменных, передаваемых в функцию по умолчанию. Здесь проблема проистекает из весьма странного (отличающегося от привычного для других языков) понимания переменных по умолчанию в функциях. Значение, задаваемое по умолчанию для параметра функции, вычисляется ровно один раз при создании функции. Далее при каждом вызове функции с пропущенным параметром происходит (уже привычное) присваивание ссылки на созданное значение формальному параметру функции, т. е. при каждом таком вызове функции формальный параметр оказывается ссылкой на одну и ту же переменную, независимо от ее изменения внутри функции. Для целых или вещественных типов это не приводит к проблемам, так как эти типы неизменяемые (при присваивании нового значения переменным такого типа создается новый объект с измененным значением и перебрасывается ссылка). Для классов, списков и словарей это не так, их можно изменять. Для них при последующих вызовах функции с пропущенным параметром мы получим формальный параметр, являющийся ссылкой на возможно измененную при предыдущих вызовах переменную.

Например, для созданного нами класса дроби определим функцию, увеличивающую значение дроби на 1:

```
1 def Inc(x=Frac()): x+=Frac(1,1); return x
```

При ее вызове без параметра

```
1 print(Inc()); print(Inc()); print(Inc())
```

мы получим увеличивающиеся значения:

```
1 (1,1)
2 (2,1)
3 (3,1)
```

что совершенно не соответствует поведению, например, языка C++ в подобной ситуации.

Отсюда следует простой вывод: нельзя без четкого понимания своих действий изменять значения переменных, передаваемых в функцию с помощью параметров по умолчанию.

13.7. Лямбда-функции: Python vs C++ без захвата vs C++ с захватом

Сравним лямбда-функции в языках Python и C++.

Прежде всего надо отметить, что Python допускает в лямбда-функции всего одно возвращаемое выражение, в то время как в языке C++ лямбда-функция имеет вполне полноценное тело, допускающее код любой длины, в том числе создание новых переменных. Фактически, для Python все выходы за указанные рамки связаны с какими-либо трюками, несколько расширяющими возможности языка. Например, Python не дает доступа к глобальным переменным непосредственно в лямбда-функциях, но, как уже отмечалось, из лямбда-функции можно вызывать обычную функцию, которая уже сможет иметь доступ к глобальным переменным. Также лямбда-функции в Python имеют доступ к обычным локальным переменным, видимым в области работы лямбда-функции. В следующем примере лямбда-функция имеет прямой доступ к значению внешней переменной `d`:

```
1 d=10
2 f = lambda x: x+d
```

```
3 print ( f ( 1 ) )
4 d=11
5 print ( f ( 1 ) )
```

При изменении переменной `d` значение, возвращаемое лямбда-функцией, меняется. Данный пример приведет к следующему выводу на экран:

```
1 11
2 12
```

Существует возможность фиксации внутри лямбда-функции значения переменной, заданной снаружи лямбда-функции, при определении функции. Следующий пример очень похож на предыдущий, но переменная `d` является параметром функции, заданным по умолчанию, и вычисляется один раз, так что изменение внешней переменной `d` после определения лямбда-функции уже никак не сказывается на внутренней переменной `d` внутри функции:

```
1 d=10
2 f = lambda x, d=d: x+d
3 print ( f ( 1 ) )
4 d=11
5 print ( f ( 1 ) )
```

Скрипт приводит к следующему выводу на экран:

```
1 11
2 11
```

Более того, локальную переменную `d` внутри лямбда-функции можно менять, используя кортеж для обеспечения возможности задания нескольких выражений внутри функции:

```
1 d=9
2 f = lambda x, d=d: (d:=d+1, x+d) [1]
3 print ( f ( 1 ) )
4 d=10
5 print ( f ( 1 ) )
```

На экране снова

```
1 11
2 11
```

Изменить внешнюю переменную из лямбда-функции не удастся. Следующий пример задает всего лишь внутреннюю локальную переменную `y` внутри функции и не меняет внешнее значение `y`:

```
1 y=10
2 fun=lambda x:(y:=1,print(x+y))[0]
3 print(fun(1))
```

На экране

```
1 2
2 1
```

Изменить глобальную переменную можно из обычной функции, вызываемой из лямбда-функции. Например, можно подсчитать, сколько раз вычисляется ключ значения при выполнении сортировки:

```
1 import random
2 n=0
3 def inc(): global n; n+=1
4 m=[]
5 for i in range(1000):m.append(random.random())
6 m.sort(key=lambda x:(x,inc())[0])
7 print(n)
```

Мы увидим, что ключ сортировки вычислялся 1000 раз, из чего следует, что при выполнении сортировки сначала создается массив ключей, а собственно сортировка в дальнейшем исходный массив для сравнения элементов не использует.

Язык C++ намного более гибок в плане работы с лямбда-функциями. Прежде всего заметим, что лямбда-функции в C++ бывают двух видов: без захвата и с захватом, и это совершенно разные сущности.

Рассмотрим простой пример из языка C++ сортировки массива переменных с помощью стандартной функции языка C `qsort()`.

```
1 #include<iostream>
2 #include<stdlib.h>
3 using namespace std;
4 int cmp(const void*a,const void *b)
5 {
6     if (*(int*)a>*(int*)b) return 1;
7     if (*(int*)a<*(int*)b) return -1;
8     return 0;
9 }
10 int main(void)
11 {
12     {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m));
13     for (int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
14     qsort(m,n,sizeof(*m),cmp);
15     for (int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
16     }
17     return 0;
18 }
```

Четвертым параметром в функцию `qsort` передается указатель на функцию, возвращающую результат сравнения двух переменных, передаваемых в функцию через указатели типа `void*` на эти переменные. При сортировке по возрастанию значение, возвращаемое функцией, должно быть положительным для случая знака *больше*, отрицательным — для знака *меньше* и равным нулю для равных переменных.

Лямбда-функции в языке C++ записываются как последовательность скобок: `[]()`. Здесь в круглых скобках записываются параметры функции в обычной нотации, а в фигурные скобки помещается тело функции. О квадратных скобках речь пойдет позднее. Тип возвращаемого параметра указывать не обязательно; он может определяться по типу выражения, возвращаемого функцией (в операторе `return`). При необходимости тип возвращаемого значения `type` указывается следующим образом: `[]() ->type`.

В приведенном выше примере функцию сравнения вполне можно заменить лямбда-функцией:

```

1 #include<iostream>
2 #include<cstdlib>
3 using namespace std;
4 int main(void)
5 {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m));
6   for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
7   qsort(m,n,sizeof(*m),[](const void *a,const void *b)
8     {if(*(int*)a<*(int*)b)return -1;
9     if(*(int*)a>*(int*)b)return 1;
10    return 0;});
11   for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
12   return 0;
13 }
```

Из этого примера мы можем сделать вывод, что лямбда-функция является обычным указателем на функцию (функция `qsort` языка C требует указателя на функцию в четвертом параметре!). Однако все оказывается не так просто. Лямбда-функция реализуется как обычный указатель на функцию только в случае лямбда-функции *без захвата*. Отсутствие содержимого в квадратных скобках как раз и говорит о том, что наша функция является лямбда-функцией без захвата.

В частности, мы можем задать аналогичную лямбда-функцию в виде шаблона (и она тоже будет интерпретироваться в конкретном месте как обычный указатель на функцию!):

```

1 template<class T> auto cmp = [](const void *a,const void *b)
2 {if(*(T*)a<*(T*)b)return -1; if(*(T*)a>*(T*)b)return 1;
3   return 0;};
4 int main(void)
5 {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m));
6   for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
7   qsort(m,n,sizeof(*m),cmp<int>);
8   for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
9   return 0;
10 }
```

Однако здесь использование лямбда-функции уже явно неоправданно. Отметим, что в данных примерах мы фактически оставались на уровне языка C.

Мы можем использовать в качестве сортировки алгоритм STL `sort()`, который на самом деле является шаблоном функции. Он может иметь либо два, либо три параметра. Первые два параметра — итераторы/указатели на начало и пустой конец сортируемого массива. Параметр шаблона указывать не обязательно, так как компилятор, как правило, может вывести его из типа элементов, содержащихся в массиве. Надо отметить, что типом, параметризующим шаблон, является тип итератора/указателя, который передается в параметры шаблона. В качестве третьего параметра можно передать указатель на функцию, заменяющую операцию *меньше*. В качестве параметров этой функции надо передавать значения или ссылки на значения элементов сортируемого массива. Два следующих примера эквивалентны:

```
1 #include<iostream>
2 #include<algorithm>
3 #include<cstdlib>
4 using namespace std;
5 bool lt(int a,int b){return(a<b);}
6 int main(void)
7 {
8     {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m));
9       for(int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
10      sort(m+0,m+n,lt); //использование указателя на функцию
11      for(int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
12     } cout<<"=====\n";
13     {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m));
14       for(int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
15       //использование лямбда-функции:
16       sort(m+0,m+n,[](int a,int b){return a<b;});
17       for(int i=0;i<n;i++){cout<<m[i]<<" ";} cout<<endl;
18     } cout<<"=====\n";
19     return 0;
20 }
```

Существенным недостатком использования указателей на функции является невозможность пробрасывания вместе с указателем на функцию каких-либо дополнительных параметров, которые необходимы для вычислений внутри данной функции. При возникновении потребности в таких параметрах не остается ничего, кроме как перебрасывать их через глобальные переменные. А это уже является в программировании признаком чертовски дурного тона и приводит к потенциально большим проблемам при масштабировании программы.

Решением данной проблемы является использование лямбда-функций с захватом. Признаком захвата является появление переменных в квадратных скобках лямбда-функции. В этом случае лямбда-функция превращается в *функциональный объект*, т. е. в именованную переменную, к которой применим оператор *круглые скобки*. Внешне указатель на функцию используется так же, как функциональный объект: и к тому, и к тому можно приписать круглые скобки с требуемыми параметрами. Но указатель на функцию в используемой нами архитектуре представляет собой обычный указатель, в то время как функциональный объект представляет собой объект, внутри которого хранятся ссылки на захватываемые переменные и/или значения захватываемых переменных. Отсюда вытекают ограничения на использование указанных сущностей. И функциональные объекты, и указатели на функции могут взаимозаменяемо использоваться в шаблонах, где нет, фактически, никакого контроля за типом переменных до использования шаблона. Но в случае параметров обычных функций типы переменных должны описываться явно, и получается, что нельзя использовать в одной функции в одном параметре одновременно на выбор указатель на функцию или функциональный объект. Например, функция `qsort()` допускает использование указателя на функцию (либо в виде обычного указателя на функцию, либо в виде лямбда-функции без захвата), но в эту функцию нельзя передавать лямбда-функцию с захватом.

В следующем примере мы попробуем посчитать, сколько раз вызывается функция сравнения в сортировке, осуществляемой алгоритмом `sort()`. Лямбда-функция, передаваемая для определения операции сравнения, будет захватывать целую переменную, которая будет увеличиваться на 1 при каждом вызове лямбда-функции. Сим-

вол &, написанный перед параметром `ncmp` в квадратных скобках лямбда-функции, служит признаком того, что параметр *захватывается по ссылке*, т. е. в неименованном объекте лямбда-функции хранится ссылка (указатель) на данную переменную:

```

1 #include<iostream>
2 #include<algorithm>
3 #include<cstdlib>
4 using namespace std;
5 int main(void)
6 {
7     {int m[]={3,2,1,3,2},n=(int)(sizeof(m)/sizeof(*m)),ncmp=0;
8       for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
9       sort(m+0,m+n,[&ncmp](int a,int b){ncmp++;return a<b;});
10      for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
11      cout<<"the number of comparisons="<<ncmp<<endl;
12    }
13    return 0;
14 }
```

В качестве подтверждения вышеприведенных слов о внутреннем строении лямбда-функции, можно вывести размер данного функционального объекта:

```

1 auto f=[&ncmp](int a,int b){ncmp++;return a<b;};
2 cout<<"sizeof(cmp)="<<sizeof(f)<<endl;
```

Старый компилятор Microsoft, к сожалению, пугался данной просьбы и не компилировал соответствующий код, но последний компилятор Microsoft и компилятор `gcc` успешно справляются с поставленной задачей. Ожидаемо выясняется, что размер данного функционального объекта равен размеру одного указателя (если бы мы захватывали две переменные по ссылке, то размер был бы равен суммарному размеру двух указателей). При использовании компилятора `gcc` на экран будет выведено

```

1 sizeof(cmp)=8
```

поскольку `gcc` по умолчанию создает 64-битное приложение.

При компиляции данного примера, возможно, потребуется указать одну из последних версий языка C++ для компилятора `gcc`:

```
1 g++ -std=c++2a q.cpp
```

По умолчанию мы используем 32-битный компилятор Microsoft, поэтому для него на экран будет выведено 4.

Если мы попробуем не писать символ `&` перед именем переменной в квадратных скобках, то это будет служить признаком того, что переменная захватывается по значению, т. е. в функциональном объекте будет храниться копия соответствующей переменной. При этом переменная объявляется `const` и ее нельзя изменять. Следующий пример не скомпилируется:

```
1 auto f=[ncmp](int a,int b){ncmp++;return a<b;};
2 cout<<"sizeof(cmp)="<<sizeof(f)<<endl;
```

Однако, как обычно, если что-то делать нельзя, но очень хочется, то это все же сделать можно. Ключевое слово `mutable` будет у нас встречаться и в другом контексте. Оно говорит о том, что `const`-переменные можно изменять. Например, если его написать перед описанием элемента в классе, то, даже если переменная типа данного класса будет объявлена как `const`, данный элемент все равно будет можно изменять. В данном примере разрешается менять переменную `ncmp`, являющуюся экземпляром данного функционального объекта (слово `mutable` пишется после списка параметров лямбда-функции):

```
1 auto f=[ncmp](int a,int b)mutable{ncmp++;return a<b;};
2 cout<<"sizeof(cmp)="<<sizeof(f)<<endl;
```

На экран для компилятора `g++` будет выведен размер этой переменной:

```
1 sizeof(cmp)=4
```

Можно попросить автоматически захватывать по ссылке все переменные, к которым обнаружится доступ внутри лямбда-функции:

```
1 auto f=[&](int a,int b){ncmp++;return a<b;};
2 cout<<"sizeof(cmp)="<<sizeof(f)<<endl;
```

При этом на экран выведется то же самое значение, что было при обычном захвате по ссылке переменной `ncmp`. Переменные, к которым не было обращений, захвачены не будут.

Аналогично можно попросить захватывать по значению все переменные, к которым обнаружится доступ внутри лямбда-функции:

```
1 auto f=[=](int a,int b)mutable{ncmp++;return a<b;};  
2 cout<<" sizeof (cmp)="<<sizeof(f)<<endl;
```

Внутри квадратных скобок можно через запятую указывать на захват нескольких переменных. В том числе, например, можно объявить с помощью символа `&` о захвате всех переменных по ссылке, а потом через запятую перечислить несколько выделенных переменных, которые должны быть захвачены по значению, или наоборот.

14. В-деревья

До сих пор мы рассматривали только бинарные деревья. Теперь нас интересуют деревья, имеющие бóльшую степень ветвления. Хотелось бы, чтобы они обладали свойством, аналогичным свойству сбалансированности в сбалансированных деревьях. Так мы приходим к В-деревьям. Отметим, что В-деревья являются основным инструментом построения некоторых прогрессивных файловых систем (ReiserFS, JFS, XFS, Btrfs).

14.1. В-деревья

В В-деревьях в каждой вершине может содержаться несколько элементов (ключей). Высота дерева определяется как максимальное количество вершин в ветвях. Будем далее рассматривать случай, когда все элементы (ключи) в дереве различны.

В-дерево степени n определяется следующим образом:

— каждая вершина дерева, кроме корня, содержит от $n - 1$ до $2n - 1$ элементов (ключей) и от n до $2n$ ссылок на дочерние элементы (кроме листьев, ссылок на дочерние элементы не имеющих); корень дерева содержит не более $2n - 1$ элементов (ключей) и не более $2n$ ссылок на дочерние элементы;

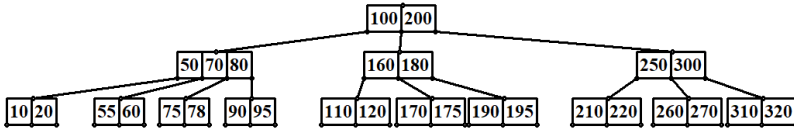
— длины всех ветвей от корня до листа совпадают;

— элементы в каждой вершине упорядочены по возрастанию;

— если в вершине содержится k элементов, то в ней содержится $k + 1$ ссылок на дочерние вершины (кроме листьев, ссылок на дочерние вершины не содержащих);

— говорят о следующих соотношениях между элементами в вершине $\{v_0, \dots, v_{k-1}\}$ и ссылками на дочерние вершины $\{T_0, \dots, T_k\}$: ссылка T_i располагается до элемента v_i ($0 \leq i < k$), после элемента v_{i-1} ($0 < i \leq k$), между элементами v_{i-1} и v_i ($0 < i < k$);

— все элементы в поддереве T_i , ссылка на которое расположена после некоторого элемента v_{i-1} , больше v_{i-1} : $T_i > v_{i-1}$; все элементы в поддереве T_i , ссылка на которое расположена до некоторого элемента v_i , меньше v_i : $T_i < v_i$.

Пример В-дерева степени $n = 3$

Как правило, В-деревья имеют достаточно большие степени. Например, их задают, исходя из того, что одна вершина должна занимать один блок на диске.

По аналогии с бинарными деревьями будем использовать для вершин дерева термины *родитель*, *потомок*, *брат*, причем братьями будем называть только два соседних элемента на одном уровне, имеющих одного родителя.

14.2. Высота В-дерева

Получим оценку высоты В-дерева степени n через количество элементов в нем.

Корень дерева содержит не менее одного элемента. На втором уровне содержится не менее двух вершин, а в каждой вершине — не менее $n - 1$ элементов. На каждом следующем уровне количество вершин увеличивается не менее чем в n раз (так как каждая вершина имеет не менее n потомков). Таким образом, на k -м уровне будет не менее $2n^{k-2}$ вершин для $k > 1$ и, соответственно, не менее $2(n - 1)n^{k-2}$ элементов. В итоге получаем оценку на количество элементов N в дереве высоты h :

$$N > 1 + \sum_{k=2}^h 2(n - 1)n^{k-2} = 1 + \frac{2(n - 1)(n^{h-1} - 1)}{n - 1} = 2n^{h-1} - 1.$$

Учитывая то, что оценка сверху на число элементов в дереве получается аналогичным образом, можем сформулировать следующую теорему.

Теорема 13. Для непустого В-дерева степени n высоты h , содержащего N элементов, верна оценка для высоты

$$h = \Theta(\log_n(N)).$$

Точная оценка выглядит следующим образом:

$$h < \log_n \left(\frac{N}{2} + 1 \right) + 1.$$

14.3. Поиск элемента в В-дереве. Определение шаблона метода для вложенного в класс класса.

Алгоритм `lower_bound`

Работу с В-деревом степени N на языке C++ начнем с задания шаблона класса В-дерева и определения вершины дерева внутри данного класса:

```

1 template<class T, int N=3>class BTree{public :
2   struct Node{T v[2*N]; Node* b[2*N]={0}, *par=nullptr;
      size_t n=0;
3   Node() {} //=default
4   Node(initializer_list<T> l, Node *p=nullptr)
5   { for (auto&x:l) {v[n++] = x;}
6     sort(v, v+n, [] (const T&a, const T&b) {return a<b;}); par=p;}
7 }*r=new Node();
```

Шаблон параметризуется типом T данных, хранимых в дереве, и степенью дерева N . Для хранения элементов дерева создадим массив на единицу большей, чем требуется, размерности (бóльшая размерность массива нам понадобится в дальнейшем). Также зададим в вершине массив указателей на потомков, указатель на родителя и переменную n , которая будет хранить количество элементов в дереве. Используя C++11, мы можем проинициализировать указатели и целую переменную нулями сразу в момент описания переменных. Отметим, что проверять, является ли вершина В-дерева листом, можно с помощью оператора `b[0]==nullptr`.

Также зададим пару конструкторов класса. Конструктор по умолчанию придется задать пустым (жесткие ключи компилятора могут помешать использовать задание конструктора с помощью ключевого слова `default`). Для простой инициализации вершины списком значений зададим конструктор с параметром `initializer_list<T> l`. Здесь мы использовали алгоритм `sort` для упорядочивания элементов массива. Третьим параметром алгоритма сортировки выступает лямбда-функция, которая задает способ сравнения элементов массива. В дан-

ном случае ее можно было бы не использовать (по умолчанию сортировка происходит по возрастанию), но теоретически мы можем использовать другой способ сравнения элементов дерева и скорректировать лямбда-функцию соответствующим образом. Например, если требуется работать с деревом целых чисел в кольце вычетов по некоторому модулю p , то сортировка будет выглядеть следующим образом:

```
1 sort(v, v+n, [])(const T&a, const T&b){return (a%p)<(b%p);});
```

Поиск вершины, содержащей заданный элемент (или элемент с ключом, равным заданному), осуществляется аналогично поиску в двоичном дереве поиска, но для каждой вершины процедура поиска данного элемента более сложная, чем для случая дерева поиска. Удобно использовать алгоритм STL `lower_bound`. Он возвращает итератор, указывающий на минимальный элемент, больший или равный заданному элементу. Алгоритм получает на вход три параметра: итератор, указывающий на начало массива, итератор, указывающий на пустой конец массива, искомый элемент. Смысл данного алгоритма заключается в следующем: он позволяет найти место в упорядоченном массиве, куда надо вставлять новый элемент (с сохранением упорядоченности массива). Тогда поиск элемента, равного v , в В-дереве с корнем r можно оформить в виде следующего метода класса дерева:

```
1 T* find(const T&x, Node*n=nullptr){if(n==nullptr)n=r;
2 auto it=lower_bound(n->v, n->v+n->n, x);
3 if(it<n->v+n->n && *it==x)return it;
4 else if(n->b[0]==nullptr)return nullptr;
5 return find(x, n->b[it-n->v]);}
```

Данную функцию мы можем спокойно, не боясь переполнения стека, делать рекурсивной, так как глубина В-дерева большой не бывает. Тест для данного метода можно записать в следующем виде:

```
1 int main(void)
2 {BTree<int> t; *t.r={100,200}; cout<<t<<endl;
3 t.r->b[0]=new BTree<int>::Node({50,70,80}, t.r);
4 t.r->b[1]=new BTree<int>::Node({160,180}, t.r);
5 t.r->b[2]=new BTree<int>::Node({250,300}, t.r);
```

```

6  for (auto x: {50, 60, 70, 80, 90, 160, 250, 260})
7  { cout << x << (t.find(x) ? " found": " not found") << endl; }
8  return 0;
9  }

```

Здесь мы вручную создали В-дерево, состоящее из двух слоев дерева, представленного на рисунке на с. 255. Для работоспособности данного теста нам необходим оператор << вывода дерева на экран. Простейшая реализация данного оператора может выглядеть следующим образом:

```

1  template<class T, int N=3>
2  ostream &operator<<(ostream&cout, const BTree<T,N> &t)
3  { cout << "{ " << *t.r << " "; return cout; }

```

В свою очередь, эта функция требует наличия оператора << для структуры Node, выводящего на экран поддерево с данной корневой вершиной. Для конкретизации шаблона В-дерева <int, 3> оператор вывода для Node можно переопределить как глобальную функцию

```

1  ostream &operator<<(ostream&cout, const BTree<int, 3>::Node &n)
2  { cout << "("; for (size_t i=0; i<n.n; i++){ cout << n.v[i] << " "; }
3  if (n.b[0] == nullptr) cout << ")" << endl;
4  else
5  { cout << " "; for (size_t i=0; i<=n.n; i++){ cout << *n.b[i]; } }
6  return cout;
7  }

```

Безусловно, данный подход определения оператора << является довольно убогим, ведь мы определили оператор только для параметров шаблона класса дерева <int, 3>. Логично бы было определить данный оператор в виде шаблона по аналогии с определением оператора << для всего дерева. Однако здесь мы сталкиваемся с трудностями, основа которых лежит в глубоких дебрях семантики языка C++. Не удастся просто переопределить данную функцию как шаблон. Решение данной проблемы весьма неожиданно. Мы можем определить данную функцию **внутри** класса дерева. Однако любой оператор, определяемый внутри класса, интерпретируется как оператор с количеством аргументов, на единицу большим, чем количество аргументов соответствующим функции (в качестве первого аргумента

выступает **this**). Объяснить компилятору наше нежелание интерпретировать аргументы оператора подобным образом можно с помощью ключевого слова **friend** перед определением функции. Здесь данный спецификатор не будет иметь никакого отношения к интерпретации публичности членов класса. Единственная его цель — дать возможность поместить определение шаблона некоторой глобальной функции внутри класса. Таким образом, метод класса получит вид:

```

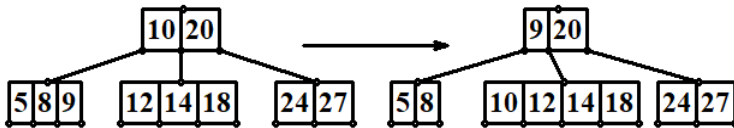
1  friend ostream &operator<<(ostream&cout , Node &n)
2  { cout<<" "; for (size_t i=0;i<n.n;i++){ cout<<n.v [ i]<<" ";}
3    if (n.b[0]== nullptr) cout<<" "<<endl;
4    else
5    { cout<<" "; for (size_t i=0;i<=n.n;i++){ cout<<*n.b [ i];}}
6    return cout;
7  }

```

14.4. Вставка элемента в В-дерево

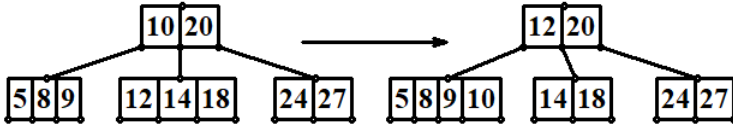
Введем четыре основных операции с В-деревом.

1. Перебрасывание элемента к правому брату. Название операции несколько условно. Операция состоит в том, чтобы правый элемент вершины V переместить на место элемента p в родительской вершине, для которого V является левым потомком, но предварительно элемент p перебросить в начало элементов правого брата V :



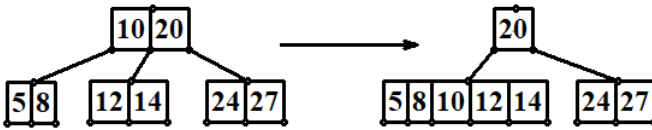
Перевос вправо элемента 9

2. Перебрасывание элемента к левому брату. Данная операция обратна предыдущей. Она заключается в том, что левый элемент вершины V перемещается на место элемента p в родительской вершине, для которого V является правым потомком, но предварительно элемент p перебрасывается в конец элементов левого брата V :



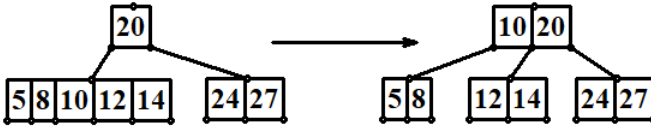
Переворот влево элемента 12

3. Объединение братьев. Данная операция объединяет поддеревья T_i и T_{i+1} с помощью стыковочного элемента v_i в родительской вершине. При этом корни поддеревьев T_i и T_{i+1} объединяются и элемент v_i вставляется между ними. Количество элементов родительской вершины и поддеревьев, выходящих из нее, уменьшается на единицу:

Объединение вершин $\{5, 8\}$ и $\{12, 14\}$

Если родительской вершиной был корень и в нем до объединения был всего один элемент, то корень перемещается к двум новым объединенным вершинам (один элемент в корне соответствует ровно двум потомкам).

4. Разбиение вершины по некоторому элементу. Данная операция является обратной к предыдущей. Она разбивает вершину — корень поддерева T_i на две вершины, которые становятся корнями поддеревьев T_i и T_{i+1} (индексация всех последующих поддеревьев T_{i+1}, T_{i+2}, \dots смещается). Корни поддеревьев T_i и T_{i+1} состоят из элементов вершины — изначального корня поддерева T_i , расположенных соответственно до и после разбивающего элемента w_k . Сам элемент w_k вставляется в родителя поддеревьев T_i и T_{i+1} в позицию i :



Разбиение вершины $\{5, 8, 10, 12, 14\}$ разбивающим элементом 10

Легко увидеть, что все четыре описанные операции могут переводить В-дерево степени n в дерево, которое уже не будет считаться В-деревом степени n (количество элементов в вершинах может стать меньше $n - 1$ или больше $2n - 1$). Поэтому при работе с В-деревьями надо следить за тем, чтобы их свойства при применении данных операций не нарушались.

Разберем сначала двухпроходный алгоритм вставки элемента в В-дерево. На первом проходе мы ищем лист, куда можно вставить новый элемент (элемент всегда будет вставляться в лист), а потом, если количество элементов в вершине окажется слишком большим, нам придется разбивать вершину и переносить разбивающий элемент в родителя вершины. Это может переполнить родителя, мы исправим ситуацию. И т. д. В худшем случае мы дойдем снизу вверх до корня и, если он окажется переполненным, разобьем его на две вершины и у дерева появится новый корень. Таким образом, двухпроходный алгоритм вставки элемента w в В-дерево сводится к следующим шагам.

1. Как и в дереве поиска, сначала ищется лист (вершина без потомков), в который можно вставить новый элемент (это первый проход по дереву). При этом если вставляемый элемент найдется при первом проходе по дереву (уже есть в дереве), то вся процедура вставки завершается неуспехом. Иначе процедура продолжается.

2. Если найденный лист V не заполнен (количество элементов после вставки меньше $2n - 1$), то новый элемент вставляется в лист V и на этом процедура завершается.

3. Иначе найденный лист содержит $2n - 1$ элемент. Вставляем w в найденную вершину (заметим, что при этом вершина переполняется и в ней оказывается $2n$ элементов).

4. В элементах данной вершины находим медиану x , и вершина разбивается на две вершины по $n - 1$ и n элементов соответственно

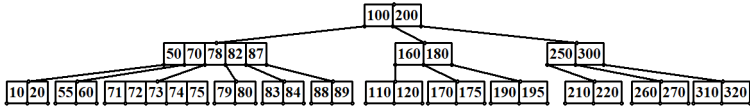
($2n = (n-1)+1+(n)$), причем элементы в первой вершине V_- должны быть меньше x , а во второй V_+ — больше x (процедура разбиения вершины описана выше).

5. Элемент x вставляется в массив элементов в родительской вершине между элементами, между которыми находилась ссылка на вершину V . Ссылки на вершины V_- и V_+ должны располагаться непосредственно слева и справа от x .

6. Если теперь в родительской вершине количество элементов остается меньше $2n$, то на этом процедура завершается. Иначе процедура разбиения вершины рекурсивно применяется для родителя (переходим к шагам 4, 5, 6). В этом заключается второй проход по дереву.

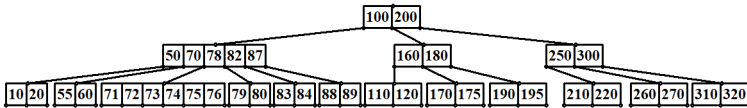
В худшем случае процедура будет последний раз применена для корня дерева и после его разбиения дерево увеличит свою высоту на 1.

Приведем пример. В следующее В-дерево степени 3 требуется вставить элемент со значением 76.



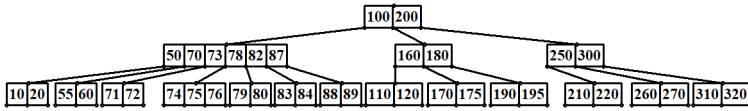
Пример В-дерева степени $n = 3$ для вставки элемента 76

На шагах 1–3 алгоритма находим лист для вставки (он заполнен) и вставляем в него значение 76.

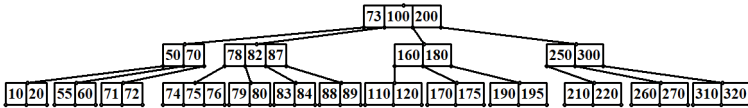


Пример В-дерева степени $n = 3$

На шагах 4, 5 алгоритма разбиваем переполненную вершину $\{71, 72, 73, 74, 75, 76\}$ на вершины $\{71, 72\}$ и $\{74, 75, 76\}$ и вставляем медиану массива 73 в вершину $\{50, 70, 78, 82, 87\}$.

Пример В-дерева степени $n = 3$

Поскольку мы опять получили переполненную вершину, повторяем шаги 4, 5, т. е. опять разбиваем переполненную вершину на две вершины и разбивающий элемент переносим в родителя (корень дерева в нашем случае):

Пример В-дерева степени $n = 3$

Процедура вставки завершена.

Надо заметить, что мы воспользовались возможностью работы с переполненным деревом (в описанной ранее структуре вершины В-дерева под массив элементов отведено $2n$ мест). В конкретной реализации дерева может случиться так, что этой вольности мы себе позволить не сможем (например, в реализации файловой системы блок диска может быть полностью отдан под используемые данные вершины и расширить его просто нет возможности). В этом случае алгоритм несколько усложнится, поскольку лишнее значение массива элементов в процессе работы придется хранить в памяти. Вообще, надо иметь в виду, что все файловые системы, основанные на В-деревьях, имеют высокую скорость работы, но и весьма существенный недостаток: в процессе перестройки файловая система оказывается весьма уязвимой и в случае сбоя питания может легко перейти в некорректное состояние. Восстановление файловой системы оказывается весьма проблематичным.

Алгоритм вставки элемента в В-дерево несложно сделать однопроходным. Для этого достаточно обеспечить незаполненность нижней вершины дерева в процессе поиска листа для вставки элемента. При первом проходе по дереву при обнаружении заполненной вершины (не листа) мы сразу разбиваем ее на две вершины и медиану массива элементов вершины сразу переносим в родителя. Тогда

при вставке элемента в лист при возникновении переполнения мы разбиваем лист на две вершины, на чем алгоритм вставки элемента завершается. Таким образом, алгоритм однопроходного добавления элемента в В-дерево имеет следующий вид.

1. Как и в дереве поиска, сначала ищется лист, в который можно вставить новый элемент. Если заполненным оказывается корень дерева, то он сразу разбивается на две вершины (алгоритм разбиения описан выше) и у дерева появляется новый корень. Если в процессе поиска листа встречается заполненная вершина (не лист), то она разбивается на две вершины и медиана массива элементов вставляется в родительскую вершину.

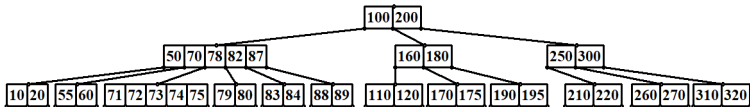
2. Если найденный лист V не заполнен (количество элементов после вставки меньше $2n - 1$), то новый элемент вставляется в лист V и на этом процедура завершается.

3. Иначе найденный лист V содержит $2n - 1$ элемент; вставляем в него w (при этом лист переполняется и в нем оказывается $2n$ элементов).

4. В элементах данного листа V находим медиану x , и лист разбивается на два листа по $n - 1$ и n элементов соответственно ($2n = (n - 1) + 1 + (n)$), причем элементы в первом листе V_- должны быть меньше x , а во втором V_+ — больше x (процедура разбиения вершины описана выше).

5. Элемент x вставляется в массив элементов в родительской вершине листа V между элементами, между которыми находилась ссылка на лист V . Ссылки на вершины V_- и V_+ должны расположиться непосредственно слева и справа от x . Поскольку (в силу пункта 1) заполненных вершин не появилось, алгоритм вставки завершается.

Приведем пример однопроходной вставки элемента в В-дерево. В следующее В-дерево степени 3 требуется вставить элемент со значением 76.

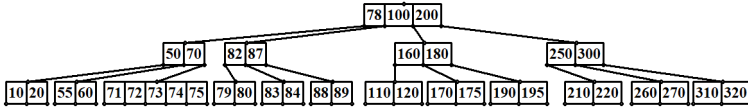


Пример В-дерева степени $n = 3$ для вставки элемента 76

На шаге 1 алгоритма при поиске листа для вставки элемента мы идем по ветви

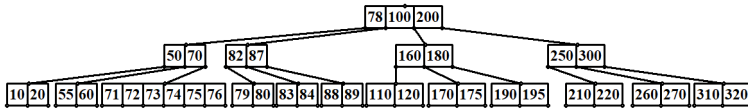
$$\{100, 200\} \rightarrow \{50, 70, 78, 82, 87\} \rightarrow \{71, 72, 73, 74, 75\}.$$

В процессе прохода мы обнаруживаем заполненную вершину (не лист) $\{50, 70, 78, 82, 87\}$ и разбиваем ее по элементу 78.



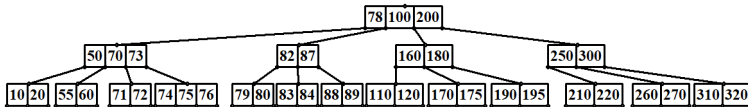
В-дерево после разбиения вершины

На шаге 3 алгоритма находим лист для вставки (он заполнен) и вставляем в него значение 76.



В-дерево непосредственно после вставки элемента в лист

На шагах 4, 5 алгоритма разбиваем переполненный лист $\{71, 72, 73, 74, 75, 76\}$ на вершины $\{71, 72\}$ и $\{74, 75, 76\}$ и вставляем медиану массива 73 в вершину $\{50, 70\}$. Процесс вставки завершен.



В-дерево после вставки элемента 76

Реализация процедуры однопроходной вставки элемента в В-дерево довольно коротка, но вместе с тем достаточно сложна. Предварительно создадим метод класса `BTree`, позволяющий вставлять элемент в обычный массив произвольного типа. Оказывается, в STL нет подобной процедуры (для контейнера `vector` такая процедура есть, но `vector` предполагает динамическое отведение памяти, чем мы не хотели бы пользоваться, так как наша реализация основана на простом массиве фиксированной длины). Создадим соответствующий метод как шаблон внутри шаблона класса дерева. Метод будет иметь следующий вид:

```

1 template<class U>void Insert (U*e,U*i ,const U&x)
2 {for (auto p=e;p>i;--p){p[0]=p[-1];} *i=x;}

```

Здесь e — указатель на пустой конец массива, в который вставляется элемент, i — указатель на место, куда вставляется элемент x . Предполагается, что данный метод увеличивает длину массива на единицу. Увеличение переменной, в которой хранится длина массива, должно произойти вне данной процедуры.

Сама процедура вставки довольно проста:

```

1 bool insert (const T&x, Node *n=nullptr){
2   if (n==nullptr)n=r; //если нет n, то вставить в дерево
3   if (split (n))n=n->par; //разбить вершину, если она заполнена
4   auto p=lower_bound (n->v, n->v+n->n, x); //найти место вставки
5   //выйти, если элемент есть в массиве:
6   if (p<n->v+n->n&&*p==x)return false;
7   if (n->b[0]== nullptr) //если лист, то вставить
8   {Insert (n->v+n->n, p, x); n->n++; split (n);}
9   else //иначе перейти к вставке в поддерево
10  return insert (x, n->b[p-n->v]);
11  return true;
12 }

```

При вставке элемента во все дерево мы будем использовать метод с одним параметром. Метод `split ()` должен разбивать вершину, если это требуется. Если разбиение вершины состоялось, то метод `split ()` должен вернуть `true`, в противном случае — `false`. Если разбиение вершины произошло, надо откатиться к родителю данной вершины ($n=n->par$) и повторить процедуру поиска места вставки, так как непонятно, в какую из двух появившихся вершин надо вставлять элемент x . Если вершина является листом, то надо просто вставить элемент в эту вершину и при необходимости разбить ее на две вершины. Как указывалось ранее, в родительской вершине будет место для вставки одного элемента.

Реально сложным оказывается метод `split ()` для разбиения (при необходимости) вершины. Он имеет следующий вид:

```

1 bool split (Node *n)
2 {if (n->n<2*N-1)return false; //выйти, если все и так хорошо

```

```

3 //создаем новую вершину n2 с правой половиной *n
4 Node *n2=new Node(); size_t n1=n->n/2;//новый размер *n
5 for (size_t i=n1+1;i<n->n;i++,n2->n++)
6 {//заполняем новую вершину n2:
7   n2->v[n2->n]=n->v[i]; n2->b[n2->n]=n->b[i];
8   if (n2->b[n2->n]) n2->b[n2->n]->par=n2;
9 }
10 n2->b[n2->n]=n->b[n->n];
11 if (n2->b[n2->n]) n2->b[n2->n]->par=n2;
12 n->n=n1;//= задаем новый размер *n
13 if (n->par==nullptr){r=new Node({n->v[n->n]});//для корня
14   r->b[0]=n;r->b[1]=n2;n->par=n2->par=r;}//новый корень r
15 else
16 {size_t i=(size_t)//если не корень,то найти место разбиения:
17   lower_bound(n->par->v,n->par->v+n->par->n,n->v[0])-n->par->v;
18   //вставляем медиану n->v[n->n] в родителя:
19   Insert(n->par->v+n->par->n,n->par->v+i,n->v[n->n]);
20   //вставляем новую вершину n2 с поддеревом в родителя:
21   Insert(n->par->b+n->par->n+1,n->par->b+i+1,n2);
22   n->par->n++; n2->par=n->par;
23 }
24 return true;
25 }

```

Здесь при необходимости (если заполнена) вершина `*n` разбивается на две половины с длинами `n1=n->n/2` и `n->n-n1-1` по медиане `n->v[n1]`. Первая половина остается в вершине `*n`, а вторая помещается в новую вершину `*n2`. Элемент `n->v[n1]` переносится в родителя (если он есть), что увеличивает длину родителя на 1. Отсутствие родителя говорит о том, что вершина является корнем дерева, поэтому надо создать новый корень (`r=new Node({n->v[n->n]})`) и прикрепить к нему два потомка: укоротившуюся старую вершину `*n` (старый корень) и новую `*n2`.

Полный код шаблона класса и прилагающейся функции для оператора вывода дерева на экран имеет следующий вид:

```

1 template<class T,int N=3> class BTree{public:
2   struct Node{
3     T v[2*N]; Node *b[2*N+1]={0},*par=nullptr;size_t n=0;

```

```

4   Node()=default;
5   Node(initializer_list<T> l)
6   {for(auto &x:l){v[n++]=x;}
7     sort(v,v+n,[](const T&a,const T&b){return a<b;});}
8   }*r=new Node();
9   BTree(){}
10  ~BTree(){/*для упрощения не реализован*/}
11  //-----
12  template<class U>void Insert(U*e,U*i,const U&x)
13  {for(auto p=e;p>i;--p){p[0]=p[-1];} *i=x;}
14  //-----
15  bool split(Node *n){if(n->n<2*N-1)return false;
16    Node *n2=new Node(); size_t n1=n->n/2;
17    for(size_t i=n1+1;i<n->n;i++,n2->n++)
18    {
19      n2->v[n2->n]=n->v[i]; n2->b[n2->n]=n->b[i];
20      if(n2->b[n2->n])n2->b[n2->n]->par=n2;
21    }
22    n2->b[n2->n]=n->b[n->n];
23    if(n2->b[n2->n])n2->b[n2->n]->par=n2;
24    n->n=n1;
25    if(n->par==nullptr)
26      {r=new Node({n->v[n->n]});
27        r->b[0]=n;r->b[1]=n2; n->par=n2->par=r;}
28    else
29      {size_t i=(size_t)
30        lower_bound(n->par->v,n->par->v+n->par->n,n->v[0])-n->par->v;
31        Insert(n->par->v+n->par->n,n->par->v+i,n->v[n->n]);
32        Insert(n->par->b+n->par->n+1,n->par->b+i+1,n2);
33        n->par->n++; n2->par=n->par;
34      }
35    return true;
36  }
37  //-----
38  bool insert(const T&x,Node *n=nullptr){if(n==nullptr)n=r;
39    if(split(n))n=n->par;
40    auto p=lower_bound(n->v,n->v+n->n,x);
41    if(p<n->v+n->n&&*p==x)return false;
42    if(n->b[0]==nullptr)

```

```

43 {Insert(n->v+n->n,p,x);n->n++;split(n);}
44 else return insert(x,n->b[p-n->v]);
45 return true;
46 }
47 //-----
48 friend ostream &operator<<(ostream&cout,Node &n)
49 {cout<<" ";for(size_t i=0;i<n.n;i++){cout<<n.v[i]<<" ";}
50 if(n.b[0]==nullptr)cout<<" "<<endl;
51 else {cout<<" ";for(size_t i=0;i<=n.n;i++){cout<<*n.b[i];}}
52 return cout;}
53 };
54 //=====
55 template<class T,int N=3>
56 ostream &operator<<(ostream&cout,const BTree<T,N> &t)
57 {cout<<" {"<<*t.r<<" ";return cout;}
58 //=====

```

Тест для данного дерева может иметь простой вид:

```

1 int main(void)
2 {BTree<int> t; //выводим вставляемый x и полученное дерево:
3 for(int x=1;x<35;x++)
4 {cout<<"+"<<x<<endl; t.insert(x); cout<<t<<endl;}
5 return 0;
6 }

```

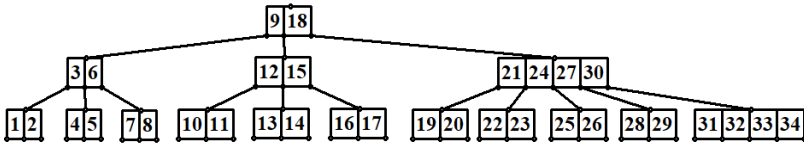
Последнее выводимое дерево будет отображено на экране в виде

```

1 {(9 18 )(3 6 )(1 2 )
2 (4 5 )
3 (7 8 )
4 (12 15 )(10 11 )
5 (13 14 )
6 (16 17 )
7 (21 24 27 30 )(19 20 )
8 (22 23 )
9 (25 26 )
10 (28 29 )
11 (31 32 33 34 )
12 }

```

что соответствует (если разобраться с форматом вывода) следующему дереву:



Созданное дерево из целых чисел от 1 до 34

14.5. Удаление элемента из В-дерева

Удаление элемента из В-дерева степени n происходит аналогично удалению вершины дерева поиска. Алгоритм сводится к следующим шагам.

1. Если вершина, из которой удаляется элемент, не является листом, то ищем минимальный элемент из правого поддерева данной вершины, и задача сводится к удалению элемента из листа. Для листа сразу переходим к шагу 2.

2. Из листа V элемент удаляется обычным способом (как удаляется элемент из массива). Если размер листа V после этого не становится меньше $n - 1$, то процесс удаления элемента завершается.

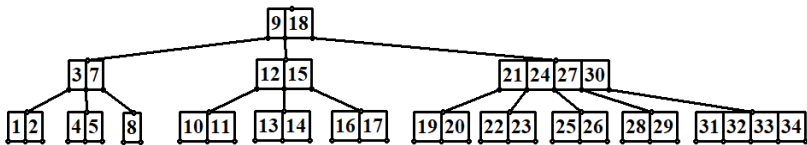
3. Иначе если у вершины V есть брат, размер которого больше $n - 1$, то ближайший элемент из этого брата перебрасывается в сторону листа V (алгоритмы перебрасывания элемента влево и вправо описаны выше). Алгоритм завершается.

4. Наконец, если каждый из братьев вершины V имеет размер $n - 1$, то происходит слияние вершины V с любым из братьев (алгоритм слияния братьев описан выше). Процесс слияния братьев сопровождается изыманием элемента из родительской вершины V . Возникает вершина корректного размера $(n - 1) + 1 + (n - 1) = 2n - 1$.

5. Если родительская вершина вершины V после изымания элемента имеет размер не меньше $n - 1$, то процесс завершается. Иначе если родительская вершина вершины V является корнем дерева и в нем остается не менее одного элемента, то процесс удаления завершается. Иначе если в родительской вершине вершины V в результате не остается элементов, то это значит, что до удаления в родителе вершины V содержался всего один элемент и он был корнем, следо-

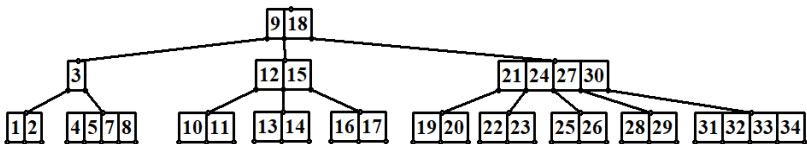
вательно, после слияния V с братом на данном уровне осталась всего одна вершина дерева, которая назначается новым корнем. На этом процесс удаления завершается. Иначе (родитель вершины V не корень) в качестве V рассмотрим родительскую вершину вершины V и процесс рекурсивно переходит к шагу 3.

Легко видеть, что алгоритм, вообще говоря, является двухпроходным. Например, удаление элемента 6 из последнего приведенного дерева происходит следующим способом: сначала (на шаге 1) мы находим вершину с элементом 6 и обнаруживаем, что данная вершина не является листом. Заменяем значение 6 минимальным элементом из поддерева, выходящего справа от данной вершины. Этим элементом оказывается элемент 7. На шаге 2 удаляем старый элемент 7 из листа $\{7,8\}$, после чего лист оказывается слишком коротким:



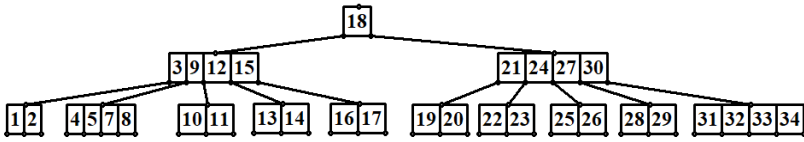
Дерево после удаления одной вершины из листа

У вершины $\{8\}$ нет брата длины, большей $n - 1$, поэтому шаг 3 пропускается. Далее на шаге 4 сливаем братьев $\{4,5\}$ и $\{8\}$ с помощью стыковочного элемента 7:



Дерево после объединения двух листьев

Поскольку вершина $\{3\}$ стала слишком короткой, мы далее рассматриваем в качестве текущей вершину $\{3\}$ и переходим к шагу 3. У данной вершины нет достаточно длинного брата, поэтому мы сливаем вершину $\{3\}$ с братом $\{12,15\}$ с помощью стыковочного элемента 9:



Дерево после объединения двух листьев

Родителем данной вершины является непустой корень, поэтому алгоритм завершается.

14.6. V^+ -деревья

Интересной и очень важной модификацией V -деревьев являются V^+ -деревья. Основное отличие V -дерева от V^+ -дерева состоит в следующем. В V -дереве в каждой вершине хранится массив элементов, эти элементы и являются данными, хранящимися в V -дереве. В V^+ -дереве сами элементы хранятся только в листьях. В каждой вершине, не являющейся листом, хранится массив ключей элементов, хранящихся в дереве (точнее, копии ключей элементов, лежащих на нижнем уровне). В вершинах, находящихся выше листьев, задаются ключи v_i , равные минимальному значению соответствующего следующего за ключом поддерева T_{i+1} . Поскольку все элементы данных хранятся только в листьях, находящихся на одном уровне дерева, листья легко связать в список, что существенно упрощает процедуру перебора элементов.

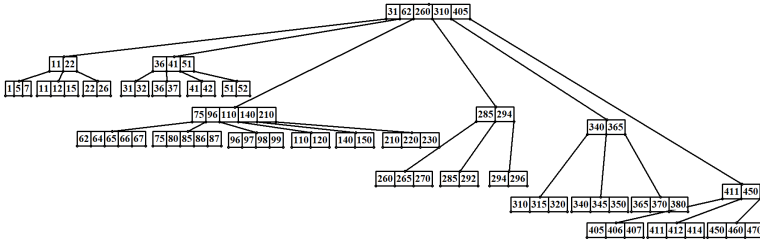
Приведем далее формальное определение V^+ -дерева, выделив шрифтом формальные отличия между V -деревьями и V^+ -деревьями.

V^+ -дерево степени n определяется следующим образом:

- каждая вершина дерева, кроме корня, содержит от $n - 1$ до $2n - 1$ элементов (ключей) и от n до $2n$ ссылок на дочерние элементы (кроме листьев, ссылок на дочерние элементы не содержащих); корень дерева содержит не более $2n - 1$ элементов (ключей) и не более $2n$ ссылок на дочерние элементы; **все вершины, расположенные не на нижнем уровне (не листья), содержат только ключи элементов, хранящихся в дереве; в листьях хранятся сами данные;**

- длины всех ветвей от корня до листа совпадают;

- элементы в каждой вершине упорядочены по возрастанию;
- если в вершине содержится k элементов, то в ней содержится $k + 1$ ссылок на дочерние вершины (в листьях ссылок на дочерние вершины нет);
- говорят о следующих соотношениях между ключами в вершине $\{v_0, \dots, v_{k-1}\}$ и ссылками на дочерние вершины $\{T_0, \dots, T_k\}$: ссылка T_i располагается до элемента v_i ($0 \leq i < k$), после элемента v_{i-1} ($0 < i \leq k$), между элементами v_{i-1} и v_i ($0 < i < k$);
- все элементы в поддереве T_i , ссылка на которое расположена после некоторого ключа v_{i-1} , **больше или равны v_{i-1} : $T_i \geq v_{i-1}$, причем v_{i-1} является минимальным ключом элементов, хранящихся в T_i** ; все элементы в поддереве T_i , ссылка на которое расположена до некоторого ключа v_i меньше v_i : $T_i < v_i$.

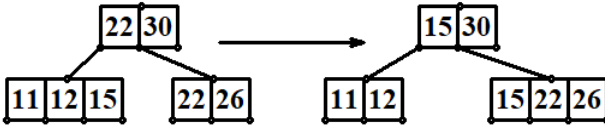
Пример B^+ -дерева степени $n = 3$

Поиск вершины, содержащей ссылку на заданный элемент, осуществляется аналогично поиску в В-дереве, с той лишь разницей что искать надо в любом случае вплоть до листа.

Добавление элемента в дерево и удаление элемента из дерева выполняются по аналогии с В-деревьями, но четыре основные операции, использовавшиеся для В-деревьев, определяются несколько иначе.

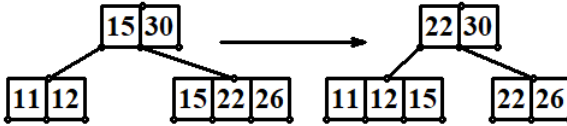
1. Перебрасывание элемента к правому брату. Если вершина V лежит на нижнем уровне дерева, то операция состоит в том, что правый элемент вершины V перемещается в начало правого брата исходной вершины (заметим, что на одном уровне могут быть либо только ключи, либо значения элементов); на место элемента p в родительской вершине, для которого V является левым потомком, надо поместить ключ перенесенного элемента. Если вершина V лежит

выше нижнего уровня дерева, то перебрасывание к правому брату осуществляется в точности также, как и для B -дерева.



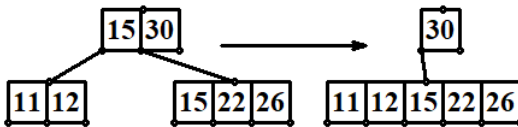
Переброс вправо элемента 15, лежащего в листе дерева

2. Перебрасывание элемента к левому брату. Данная операция обратна предыдущей. В случае, когда V является листом, она заключается в том, что левый элемент вершины V перемещается в конец левого брата; на место элемента p в родительской вершине, для которого V является правым потомком, надо поместить ключ первого элемента новой вершины V . Если вершина V лежит выше нижнего уровня дерева, то перебрасывание к левому брату осуществляется в точности также, как и для B -дерева.



Переброс влево элемента 15, лежащего в листе дерева

3. Объединение братьев. Данная операция объединяет поддеревья T_i и T_{i+1} с помощью стыковочного элемента (ключа) v_i в родительской вершине. При этом, если корни поддеревьев T_i и T_{i+1} являются листьями, то они просто объединяются, а элемент v_i удаляется. Количество поддеревьев, выходящих из родительской вершины, уменьшается на единицу вместе с количеством элементов родительской вершины:

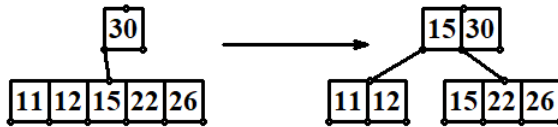


Объединение листьев $\{11, 12\}$ и $\{15, 22, 26\}$

Если же корни поддеревьев T_i и T_{i+1} листьями не являются, то объединение поддеревьев осуществляется в точности также, как и в случае обычных В-деревьев.

Если родительской вершиной был корень и в нем до объединения был всего один элемент, то корень перемещается к двум новым объединенным вершинам (один элемент в корне соответствует ровно двум потомкам).

4. Разбиение вершины по некоторому элементу. Данная операция является обратной к предыдущей. Она разбивает корень поддерева T_i на две вершины, которые становятся корнями поддеревьев T_i и T_{i+1} (индексация всех последующих поддеревьев T_{i+1} , T_{i+2} , ... смещается). Корень поддерева T_i после разбиения состоит из элементов изначального корня, расположенных до разбивающего элемента w_k . В случае, если корень поддерева T_i является листом, то после разбиения корень поддерева T_{i+1} будет состоять из самого элемента w_k и элементов изначального корня, расположенных после w_k . Ключ элемента w_k вставляется в родителя поддеревьев T_i и T_{i+1} в позицию i . Если же корень поддерева T_i листом не является, то разбиение вершины происходит в точности также, как и в В-дереве.



Разбиение листа $\{11,12,15,22,26\}$ разбивающим элементом 15

С так модифицированными операциями добавление элемента в B^+ -дерево и удаление элемента из B^+ -дерева осуществляются точно так же, как и для В-дерева.

14.7. Кратко обо всех основных контейнерах STL.

Работа с пирамидой

В этом разделе дается краткий обзор всех контейнеров STL. Мы перечислим основные контейнеры STL и напишем по несколько строк тестирующего кода для каждого из них.

Весь содержательный тест будет погружен в функцию `main()`, имеющую следующее обрамление:

```

1 #include<iostream>
2 #include<vector>
3 #include<stack>
4 #include<queue>
5 #include<deque>
6 #include<set>
7 #include<map>
8 #include<string>
9 using namespace std;
10 int main(void)
11 { //Текст тестов
12     return 0;
13 }

```

Легко понять, ради описания какой структуры данных используется каждый include-файл. Тест для каждой структуры данных разместим в отдельном блоке. Далее будет приведено содержимое блока для каждого контейнера.

Вектор:

```

1 vector<int> s={1,2}; for(int x:{3,4,5})s.push_back(x);
2 while(!s.empty())
3 {cout<<s.front()<<" "; s.erase(s.begin());} cout<<endl;

```

Здесь показан вариант работы с вектором как с очередью.

Стек:

```

1 stack<int> s; for(int x:{1,2,3,4,5})s.push(x);
2 while(!s.empty()){cout<<s.top()<<" "; s.pop();} cout<<endl;

```

Очередь:

```

1 queue<int> s; for(int x:{1,2,3,4,5})s.push(x);
2 while(!s.empty()){cout<<s.front()<<" ";s.pop();} cout<<endl;

```

Приоритетная очередь с заданием операции сравнения:

```

1 auto cmp=[](int a,int b){return a>b;};
2 priority_queue<int, vector<int>,decltype(cmp)> s(cmp);
3 for(int x:{2,1,3,4,5})s.push(x);
4 while(!s.empty()){cout<<s.top()<<" "; s.pop();} cout<<endl;

```

Основное свойство приоритетной очереди заключается в том, что после каждого добавления элемента происходит автоматическое переупорядочивание массива таким образом, что минимальный элемент оказывается в начале очереди. При этом, если это необходимо, операция сравнения *меньше* может быть задана описанным выше способом. Таким образом, в приведенном примере после добавления всех элементов мы сможем извлекать их из приоритетной очереди в порядке возрастания.

Отметим, что процедура добавления элемента в приоритетную очередь не требует полного переупорядочивания массива внутри очереди. В [11] мы подробно разбирали процедуру упорядочивания массива HeapSort и используемое при этом понятие *пирамида*. Именно с помощью *пирамиды* происходит организация приоритетной очереди. В результате мы получаем, что время работы операций добавления элемента в приоритетную очередь и удаления элемента из приоритетной очереди равно $\Theta(\log_2 n)$, где n — количество элементов в контейнере.

Дек:

```

1 deque<int> s={3,4,5}; for (int x:{1,2}) s.push_back(x);
2 while (!s.empty())
3 {cout<<s.front()<<" "; s.pop_front();} cout<<endl;

```

Здесь показан вариант работы с деком как с очередью. Напомним, что с деком можно работать как с вектором.

Множество с заданием операции сравнения:

```

1 auto cmp=[](int a,int b){return a>b;};
2 set<int,decltype(cmp)> s(cmp);
3 for (int x:{1,2,3,4,5,1,2,3,4,5}) s.insert(x);
4 for (int x:{1,2,3,4,5})
5 if (s.find(x)!=s.end()) cout<<x<<" found\n";
6 for (set<int>::iterator it=s.begin();it!=s.end();++it)
7 {cout<<*it<<" ";} cout<<endl;

```

Задание операции сравнения здесь привело к тому, что стандартный перебор элементов будет происходить по убыванию.

Мультимножество с заданием операции сравнения:

```

1 auto cmp=[](int a,int b){return a>b;};

```

```

2 multiset<int, decltype(cmp)> s(cmp);
3 for (int x:{1,2,3,4,5,1,2,3,4,5}) s.insert(x);
4 for (int x:{1,2,3,4,5})
5     if (s.find(x)!=s.end()) cout<<x<<" found\n";
6 for (multiset<int>::iterator it=s.begin(); it!=s.end(); ++it)
7     {cout<<*it<<" "; cout<<endl;

```

В мультимножество заносятся все вставляемые элементы, в то время как в множество добавляются только различные элементы.

Ассоциативный контейнер `std::map` с целой переменной в качестве ключа и строкой в качестве значения с заданием операции сравнения:

```

1 auto cmp=[](int a,int b){return a>b;};
2 map<int, string, decltype(cmp)> s(cmp); char ss[10];
3 for (int x:{1,2,3,4,5,1,2,3,4,5})
4     s.insert(pair<int, string>(x, itoa(x*x, ss, 10)));
5 for (int x:{1,2,3,4,5}) if (auto it=s.find(x); it!=s.end())
6     cout<<"("<<it->first<<" , "<<it->second<<" ) found\n";
7 for (auto it=s.begin(); it!=s.end(); ++it)
8     {cout<<it->first<<" , "<<it->second<<" "; cout<<endl;

```

Здесь используется функция `itoa`, преобразующая целое число в строку в указанной системе счисления. Напомним, что использование оператора `if` с определением локальной переменной требует задания ключа компилятора, указывающего версию языка C++:

```

1 g++ -std=c++17 q.cpp

```

В приведенном примере при замене `map` на `multimap` мы получим возможность заносить в контейнер элементы с одинаковыми ключами; `map` запрещает подобную возможность.

Наконец, опишем возможности STL по работе с *пирамидой* (она же *куча*, она же *heap*). Определение пирамиды и подробности соответствующих алгоритмов были даны в [11].

Создать пирамиду из массива можно с помощью следующего простого кода:

```

1 vector<int> v={1,2,3,4,5,1,2,3,4,5};
2 make_heap(v.begin(), v.end());

```

Добавление элементов в пирамиду по одному элементу возможно с помощью функции `push_heap`. Предполагается, что массив, передаваемый в функцию, обладает свойствами пирамиды для всех элементов, кроме последнего. Удаление элемента с вершины пирамиды осуществляется функцией `pop_heap`. Приведем код для теста этих функций.

```

1  vector<int> v;
2  for (int x:{1,2,3,4,5,1,2,3,4,5})
3  {v.push_back(x); push_heap(v.begin(),v.end());
4    for(auto it=v.begin();it!=v.end();++it)
5    {cout<<*it<<" ";} cout<<endl;
6  }
7  while(!v.empty())
8  {
9    pop_heap(v.begin(),v.end()); v.pop_back();
10   for(auto it=v.begin();it!=v.end();++it)
11   {cout<<*it<<" ";} cout<<endl;
12 }

```

В данном примере содержимое пирамиды распечатывается после добавления и удаления каждого элемента. Отметим, что функции `push_heap` и `pop_heap` сами по себе физический размер массива (вектора) не изменяют и это приходится делать внешними средствами. Осталось добавить, что функцию сравнения в операциях с пирамидой можно задавать самостоятельно (по умолчанию в вершине пирамиды хранится максимальный элемент массива). Например, если требуется, чтобы в вершине пирамиды хранился минимальный элемент массива, надо добавить соответствующую функцию сравнения:

```

1  vector<int> v={1,2,3,4,5,1,2,3,4,5};
2  auto cmp=[](int a,int b){return a>b;};
3  make_heap(v.begin(),v.end(),cmp);

```

Для функций `push_heap` и `pop_heap` надо поступить аналогично:

```

1  vector<int> v; auto cmp=[](int a,int b){return a>b;};
2  for (int x:{1,2,3,4,5,1,2,3,4,5})
3  {v.push_back(x); push_heap(v.begin(),v.end(),cmp);
4    for(auto it=v.begin();it!=v.end();++it)
5    {cout<<*it<<" ";} cout<<endl;

```

```

6 }
7 while (!v.empty())
8 {
9     pop_heap(v.begin(), v.end(), cmp); v.pop_back();
10    for (auto it=v.begin(); it!=v.end(); ++it)
11        {cout<<*it<<" ";} cout<<endl;
12 }

```

На самом деле функция `push_heap` позволяет не только добавлять элемент к пирамиде, но и корректировать готовую пирамиду после увеличения ее любого элемента (в случае стандартного построения пирамиды). Чтобы скорректировать готовую пирамиду, расположенную в векторе `v`, после увеличения элемента с индексом `i`, достаточно применить процедуру

```

1 push_heap(v.begin(), v.begin()+i+1);

```

Легко увидеть, что коррекция до состояния пирамиды данного массива в указанном диапазоне индексов не испортит свойства пирамиды для индексов на всем векторе `v`. Для понимания этого достаточно вспомнить организацию процедуры `Heapify`, описанной в [11] (именно эту процедуру выполняет функция `push_heap()`).

Действительно, один шаг коррекции пирамиды сводится к следующему. Рассмотрим элементы массива с индексами $p = (i - 1)/2$, $q = (i - 1)/2 * 2 + 1$, $r = (i - 1)/2 * 2 + 2$ (деление целочисленное; в начальный момент i — индекс модифицируемого элемента). Эта тройка индексов соответствует родительскому элементу и двум его потомкам, один из которых равен i . Допустим для однозначности, что $i = q$. Если $v[p] \geq v[q]$, то процесс коррекции завершился, так как свойство $v[p] \geq v[r]$ выполнялось изначально. Иначе максимальным элементом этой тройки является $v[q]$, и данная процедура меняет местами $v[p]$ и $v[q]$. После этого выполнится свойство $v[p] \geq v[q]$, и тем более будет выполняться свойство $v[p] \geq v[r]$. Далее происходит присваивание $i = p$ и этот же шаг процедуры повторяется с новой тройкой элементов (фактически мы идем вверх по ветви дерева).

В следующем примере пирамида строится в обратном порядке (именно это нам потребуется в дальнейшем), потом для каждого элемента массива его значение уменьшается и производится коррекция пирамиды с помощью соответствующего выполнения функции

`push_heap()`. После каждой коррекции проверяется, что полученный массив обладает свойствами пирамиды:

```
1  vector<int> v={2,1,2,3,4,5,2,1,2,3,4,5};
2  auto cmp=[](int a,int b){return a>b;};
3  make_heap(v.begin(),v.end(),cmp);
4  for(auto x:v){cout<<x<<" ";} cout<<endl;
5  for(size_t i=v.size()-1;i>0;i--)
6  {v[i]-=3; push_heap(v.begin(),v.begin()+i+1,cmp);
7    cout<<(is_heap(v.begin(),v.end(),cmp)?"ОК: ":"ERROR: ");
8    for(auto x:v){cout<<x<<" ";} cout<<endl;
9  }
```

15. Графы

Ранее мы уже ввели стандартное понятие *графа* как пары, состоящей из множества *вершин* произвольной природы и семейства пар вершин (*ребер*): $G = (V, E)$. Напомним, что мы обсуждаем лишь конечные графы, т. е. графы, в которых количество вершин и ребер конечно. Как и ранее, количество элементов в некотором множестве X будем обозначать через $\#X$.

Вершина графа $v \in V$ и ребро $e \in E$ называются *инцидентными*, если $v \in e$, т. е. вершина является одной из вершин ребра. Понятно, что *ребро*, *инцидентное паре вершин*, соединяет именно эти две конкретные вершины, и у нас появляется *пара вершин*, *инцидентная ребру*. Ребро графа называется *кратным*, если для пары вершин, инцидентных ребру, существует более одного ребра, инцидентного этой паре вершин. Ребро называется *петлей*, если вершины, инцидентные ребру, совпадают.

Две вершины графа $v, w \in V$ называются *смежными*, если они принадлежат одному ребру графа. Аналогично вводится понятие *смежных ребер*: ребра называются смежными, если у них есть общая вершина.

Имеет смысл одинаково, $u \sim v$, обозначать инцидентность и смежность объектов, так как по типу объектов легко понять, какое из введенных понятий используется (например, если для $u, v \in V$ пишется, что $u \sim v$, то здесь имеется в виду смежность вершин графа).

Как и ранее, последовательность вершин $\{p_0 = a, \dots, p_k = b\}$ ($p_i \in V$) называется *путем* от вершины a до вершины b , если каждая пара подряд идущих в последовательности вершин является ребром графа. В этом случае будем говорить, что в графе *можно добраться от вершины a до вершины b за k шагов*.

Граф называется *связным*, если для любых двух вершин графа a и b существует путь по ребрам графа от a до b , т. е. существует последовательность $\{x_1, \dots, x_n\}$ вершин графа, такая что $x_1 = a$, $x_n = b$ и для всех i пара (x_i, x_{i+1}) ($1 \leq i < n$) инцидентна некоторому ребру графа.

Связной компонентой графа называется нерасширяемый связный подграф данного графа.

Алгоритмы, связанные с графами, широко используются в различных областях вычислительной математики. В частности, любая

сетка, используемая при решении систем дифференциальных уравнений, задает граф. А это приводит к необходимости решения различных соответствующих задач. Например, в качестве простейшей задачи здесь можно рассмотреть задачу быстрого нахождения ячейки сетки, которой принадлежит заданная точка ([3], [18]).

15.1. Путь в графе с минимальным количеством шагов. Алгоритм волны

Пусть задан некоторый конечный граф $G = (V, E)$ и две вершины этого графа $a, b \in V$. Поставим задачу построить путь от a до b с минимальным количеством шагов. Стандартным решением данной задачи является *алгоритм волны*.

Будем называть *волной степени n* множество вершин графа S_n , таких что до них можно добраться от вершины a за n шагов, а за меньшее количество шагов добраться нельзя. Понятно, что $S_0 = \{a\}$.

Введем множество S'_n следующим образом.

$$S'_0 = \{a\}.$$

Для $i > 0$ определим S'_i как множество вершин $p \in V$, таких что для каждой p есть смежная вершина из S'_{i-1} и p не принадлежит ни одному S'_j для $j < i$.

Теорема 1. Для любого $i \in \mathbb{Z}^+$ $S_i = S'_i$.

Доказательство. Теорема легко доказывается по индукции.

Для $i = 0$ имеем $S'_0 = S_0 = \{a\}$.

Пусть $S'_i = S_i$ для всех $i \leq n$. Докажем данный факт для $i = n+1$.

Докажем, что $S'_{n+1} \subset S_{n+1}$. Действительно, до точек из S'_{n+1} можно добраться за $n+1$ шаг по построению. За меньшее количество шагов добраться нельзя, так как для любого $i < n$ выбранные точки не принадлежат $S'_i = S_i$.

Докажем, что $S_{n+1} \subset S'_{n+1}$. Допустим, что найдется вершина графа x , которую мы не включили в созданное множество S'_{n+1} , но $x \in S_{n+1}$. По условию $x \in S_{n+1}$ имеем, что существует путь $p_0 = a, \dots, p_{n+1} = x$ от вершины a до x в $n+1$ шагов, а значит, у вершины x есть смежная вершина p_n , до которой можно добраться от вершины a за n шагов. Быстрее до вершины p_n добраться нельзя, так как если бы это было возможно, то и до вершины x можно было бы добраться быстрее, чем за $n+1$ шагов, а это уже противоречит

тому, что $x \in S_{n+1}$. Значит, $p_n \in S_n = S'_n$, а из этого и смежности вершин p_n и p_{n+1} сразу же вытекает, что $p_{n+1} \in S'_{n+1}$. \square

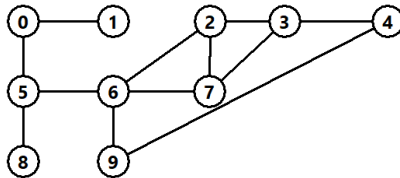
Из данной теоремы сразу же вытекает алгоритм поиска кратчайшего пути от вершины a до вершины b . Создадим два множества вершин V_0 и V_1 . В V_0 будем хранить волну текущей степени, а в V_1 будем собирать вершины волны следующей степени. Изначально $V_0 = \{a\}$. После нахождения полной волны следующей степени поменяем местами множества V_0 и V_1 и перейдем к поиску следующей волны. Если в какой-то момент создания множества S_{n+1} в множество V_1 попадет вершина b , то мы получим, что до вершины b можно добраться от вершины a за $n + 1$ шагов и быстрее добраться нельзя. Если множество $V_1 = S_{n+1}$ окажется пустым, то это значит, что мы полностью исчерпали связную компоненту, содержащую a , и пути от a до b не существует.

В структуре каждой вершины графа нам надо будет хранить две переменные: `used` — признак того, что мы уже побывали в данной вершине при создании очередной волны, и `ib` — номер вершины, откуда мы пришли в данную вершину из предыдущей волны. Последняя переменная пригодится нам, когда, дойдя в очередной волне до вершины b , мы будем строить обратный путь к вершине a .

Исходные данные графа будем считывать из файла, в котором последовательно заданы номера вершин графа `ib` и `ie`, между которыми будет строиться путь, количество вершин графа, пары номеров вершин ребер графа. Например, файл (далее он будет называться `e.txt`)

1	0	4	10																					
2	0	1	0	5	2	3	3	4	5	8	5	6	6	9	6	7	6	2	7	2	7	3	9	4

задает следующий граф (внутри кругов с изображением вершин написаны номера вершин).



Граф, в котором строится путь с наименьшим количеством ребер от вершины 0 до вершины 4

Заголовок программы с заданием структуры вершины будет иметь вид:

```

1 #include<iostream>
2 #include<fstream>
3 #include<vector>
4 #include<algorithm>
5 using namespace std;
6 struct Vertex{int ib=-1; bool used=false};

```

Для перебора смежных вершин графа создадим вектор векторов a (от слова *adjacent*) индексов вершин, в котором вектор $a[i]$ задает вектор индексов вершин, смежных i -й вершине. Такое представление графа называется *списком смежных вершин*. Из-за неориентированности графа при вводе пары индексов i, j , задающих очередное ребро, нам придется в вектор $v[i]$ заносить индекс вершины j , а в вектор $v[j]$ — индекс вершины i .

Основная функция, решающая задачу поиска пути от вершины a до вершины b за минимальное количество шагов, будет иметь вид:

```

1 int main(void)
2 {vector<Vertex> p; vector<vector<int>> a; int ib,ie,n;
3 {int i,j;//вводим ib, ie, n и индексы вершин ребер графа:
4 ifstream f("e.txt"); f>>ib>>ie>>n; cout<<"n"<<n<<endl;
5 p.resize(n); a.resize(n);//массивы вершин и смежных вершин
6 //создаем список смежных вершин:
7 while(f>>i>>j){a[i].push_back(j);a[j].push_back(i);}
8 }
9 vector<int> V0={ib},V1; p[ib].used=true;
10 for(;!V0.empty();swap(V0,V1))
11 {V1.clear();//по очередной волне V0 создаем след.волну V1
12 for(int i:V0)for(int j:a[i])if(!p[j].used)
13 {V1.push_back(j); p[j].ib=i; p[j].used=true;
14 if(j==ie)goto le;//если встретили конец пути, то выходим
15 }
16 //если следующая волна пустая, то пути не существует
17 cout<<"path not found"<<endl;return 0;
18 le:; vector<int> w;//идем назад по ссылкам на пред. вершину
19 for(int i=ie;i>=0;i=p[i].ib)w.push_back(i);
20 reverse(w.begin(),w.end());//разворачиваем массив

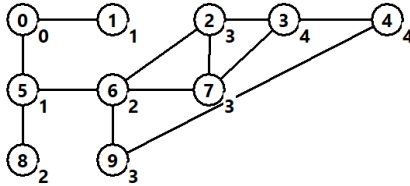
```

```

21 for (int x:w) {cout<<x<<" ";} cout<<endl; //распечатываем путь
22 return 0;
23 }

```

На следующем рисунке справа снизу от каждой вершины графа с предыдущего рисунка приписан номер волны, в которую попадает вершина. Таким образом, $\{S_0\} = p_0$, $\{S_1\} = p_1, p_5$ и т. д.



**Граф с номером волны для каждой вершины
при выходе волны из вершины p_0**

Здесь следует отметить, что существуют различные варианты удобного представления графа в памяти ЭВМ. В рассмотренной задаче хорошо зарекомендовал себя *список смежных вершин*.

Фактически в предыдущей задаче граф был задан в файле в виде *массива ребер*, где для каждого ребра задавались номера его вершин. Сам по себе он нам не понадобился, поэтому мы его использовали для создания списка смежных ребер, но не сохраняли в программе.

Далее мы будем использовать *список инцидентных ребер*, представляющий собой вектор векторов, в каждом элементе которого $p[i][j]$ лежит номер ребра, инцидентного вершине с индексом i .

Иногда при работе с графом бывает полезным использование *матрицы смежности*, в элементах $m[i][j]$ которой хранится количество ребер, инцидентных паре вершин (i, j) .

Также используется *матрица инцидентности*, в элементах которой $p[i][j]$ хранится 1, если вершина с номером i инцидентна ребру с номером j , и 0 при отсутствии инцидентности.

Легко понять, когда оказывается полезной каждая из этих структур данных.

Можно задаться вопросом о времени работы алгоритма волны. Легко видеть, что без дополнительных предположений это время нельзя оценить с помощью функции от $n = \#V$, так как количество ребер, инцидентных вершине, не ограничено (здесь имеется в виду,

что одна пара вершин может иметь неограниченное количество инцидентных ребер). Тривиальна следующая теорема.

Теорема 2. *Верны следующие оценки для графа, состоящего из N вершин и M ребер.*

1. *Алгоритм волны работает за время $O(N + M)$.*
2. *Если в графе отсутствуют петли и кратные ребра, то алгоритм волны работает за время $O(N^2)$.*
3. *Если каждая вершина графа имеет не более M_1 инцидентных ребер, то алгоритм волны работает за время $O(M_1 \cdot N)$.*

Доказательство теоремы опирается на то, что время работы алгоритма волны равно $O(M + N)$, где N — количество вершин, M — количество ребер в графе (точнее, не ребер, а пар смежных вершин, но в условиях теоремы это одно и то же). Последний факт следует из того, что каждое ребро при взятии смежной вершины в алгоритме может встретиться не более двух раз.

Чтобы получить более оптимистичную оценку, введем несколько новых понятий.

15.2. Планарные графы. Формула Эйлера

Пусть дан некоторый граф $G = (V, E)$, пусть каждой его вершине поставлена в соответствие точка в некотором евклидовом пространстве, причем точки, соответствующие различным вершинам, не совпадают. Пусть каждому ребру графа поставлена в соответствие некоторая непрерывная кривая, соединяющая соответствующие вершины графа, причем кривые, соединяющие различные пары точек не пересекаются нигде, кроме вершин графа. Такое представление графа называется его *геометрическим представлением*.

Теорема 3. *Для любого конечного графа существует его геометрическое представление в \mathbb{R}^3 .*

Доказательство. Рассмотрим произвольный отрезок $[a, b] \subset \mathbb{R}^3$. Поставим в соответствие всем вершинам графа различные точки на отрезке $[a, b]$. Поставим в соответствие каждому ребру графа свою плоскость, проходящую через $[a, b]$. Плоскости, соответствующие ребрам, пересекаются только вдоль отрезка $[a, b]$, поэтому в каждой плоскости можно нарисовать кривую, соединяющую вершины, инцидент-

ные соответствующему ребру, которая не пересекается с другими построенными кривыми нигде, кроме как в вершинах графа. \square

Граф называется *планарным*, если существует его геометрическое представление в \mathbb{R}^2 . *Плоской укладкой* планарного графа называется такое его геометрическое представление, при котором каждое ребро представляется отрезком на плоскости.

Верна следующая известная теорема [18], мы приведем ее без доказательства.

Теорема 4. *Для любого конечного планарного графа без кратных ребер и петель существует его плоская укладка.*

Иногда в литературе последнее утверждение выступает в виде определения планарного графа, а не его свойства.

Геометрическое представление планарного графа разбивает плоскость на некоторое количество связных областей (одна из них бесконечна). Эти области называются *гранями*.

Связный граф без циклов называется *деревом* или *неориентированным деревом*. Для неориентированного дерева вершины, инцидентные не более чем одному ребру дерева, называются *конечными вершинами*. Иногда в литературе такие вершины называют *листьями*, но нам такое определение листьев неудобно (сравните с данным определением листьев для бинарных деревьев). Отметим еще раз, что понятия ориентации ребер в этом определении нет!

Ориентированным деревом называется ориентированный граф, являющийся деревом, для которого ровно одна вершина, называемая *корнем*, имеет нулевую степень захода (*степень захода* — количество ребер, заходящих в вершину), а все остальные вершины имеют степень захода, равную 1. Для случая ориентированного графа *конечной вершиной* называется вершина, имеющая нулевую степень исхода (*степень исхода* — количество ребер, выходящих из вершины). Это определение отличается от определения для неориентированного дерева!

Как уже ранее упоминалось, ориентированное дерево можно определить просто как связный ориентированный граф без циклов. Тогда легко доказать теорему о том, что в таком дереве ровно одна вершина (называемая *корнем*) имеет нулевую степень захода, а все остальные вершины имеют степень захода, равную 1.

Лемма 1. *Любое конечное неориентированное дерево имеет плоскую укладку.*

Доказательство. Возьмем произвольную вершину дерева. Поместим ее в точку плоскости с координатами $(0,0)$ и назовем корнем дерева. Пусть этой вершине инцидентно k ребер. Поместим вторые вершины, инцидентные данным ребрам, в точки $(i, -1)$. Каждой из вновь размещенных точек поставим в соответствие полосу плоскости со сторонами, параллельными оси OY , не имеющую общих точек с полосами соседних точек.

Пусть для каждой вершины x , помещенной на плоскость в точку с координатой y , равной $-h$, найдутся l_x парных ей вершин, которые еще не размещены на плоскости. Будем называть эти вершины потомками вершины x , а вершину x — их родителем. Разобьем полосу, соответствующую вершине x , на l_x вертикальных непересекающихся полос и разместим в них потомков x на координате $y = -h - 1$.

Для потомков x рекурсивно выполним процедуру, описанную в предыдущем абзаце.

Условие отсутствия циклов гарантирует, что вершина x будет соединена с новыми вершинами только одним ребром и что в графе не будет ребер, соединяющих вершину x с уже размещенными вершинами. Таким образом, для каждой вершины, размещенной на плоскости, будут реализованы все ребра графа, ей инцидентные.

Условие связности гарантирует, что после окончания этого алгоритма все точки будут размещены на плоскости. Действительно, пусть вершина b принадлежит графу, тогда из связности графа следует, что существует путь по ребрам графа от корня дерева до b . Мы показали, что для вершин, размещенных на плоскости, реализуются все ребра графа, из чего по индукции получаем, что все вершины пути, соединяющего корень дерева и b , будут размещены на плоскости. \square

Доказательство леммы 1 одновременно служит конструктивным доказательством того, что верна следующая лемма.

Лемма 2. *В любом конечном дереве можно ввести ориентацию, т. е. для ребер произвольного дерева можно задать ориентацию так, что дерево станет ориентированным.*

Верна следующая известная теорема.

Теорема 5 (формула Эйлера). Пусть задан связный планарный граф, имеющий в некотором геометрическом представлении p вершин, q ребер, r граней. Тогда верна формула $p - q + r = 2$.

Сначала докажем следующую лемму.

Лемма 3. Если в некотором графе содержится p вершин и q ребер, то граф состоит из не менее чем $p - q$ связных компонент. Если в графе нет циклов, то в графе ровно $p - q$ связных компонент. Если в графе есть циклы, то в графе строго больше $p - q$ связных компонент.

Доказательство. Любой конечный граф $G = (V, E)$ без циклов может быть получен из графа $G_0 = (V, \emptyset)$ путем последовательного добавления ребер, при этом каждый промежуточный граф не будет содержать циклов (сначала мы можем из исходного графа исключать по одному все ребра до их полного удаления, а потом из получившегося графа $G_0 = (V, \emptyset)$ получить исходный граф добавлением ребер в обратном порядке). Для графа G_0 данная лемма выполняется (количество связных компонент равно количеству вершин).

Рассмотрим случай отсутствия циклов. При добавлении каждого ребра в граф G_0 мы либо соединяем две связные компоненты и при этом лемма остается верной (при добавлении каждого ребра величина $p - q$ уменьшается на 1 и количество связных компонент тоже уменьшается на 1), либо обе соединяемые вершины принадлежат одной связной компоненте. Но в последнем случае перед соединением существовал путь от одной вершины к другой, и добавление ребра образовало цикл, что противоречит отсутствию циклов в исходном графе.

Если разрешить наличие циклов, то в предыдущем доказательстве станет возможным соединение вершин из одной связной компоненты, но тогда количество связных компонент будет строго больше $p - q$ (при добавлении ребра величина $p - q$ уменьшится на 1, а количество связных компонент не изменится). Лемма доказана. \square

Следствием леммы 3 является простой критерий наличия циклов в графе.

Условие, что $p - q$ равно количеству связных компонент в графе, равносильно отсутствию циклов в графе. В частности, для связ-

ного графа условие $p - q = 1$ равносильно тому, что в графе нет циклов.

Доказательство формулы Эйлера. Исходя из леммы, получаем, что теорема Эйлера верна для деревьев. Действительно, по лемме 1 каждое дерево с конечным числом вершин имеет плоскую укладку. Количество граней в этой укладке равно 1 (иначе существует хотя бы одна конечная грань, а значит, существует последовательность ребер на ее границе, образующая цикл). Количество граней в геометрическом представлении дерева равно 1, и по лемме 3 получаем

$$(p - q) + r = 1 + 1 = 2.$$

Рассмотрим произвольный связный граф с циклами с q_0 ребрами. По лемме 3 получаем, что $p - q_0 > 1$. Зафиксируем p и предположим, что для всех $q < q_0$ теорема доказана. Заметим, что для случая $p - q = 1$ мы уже доказали теорему. По предположению в графе есть цикл, возьмем одно ребро в этом цикле и исключим из графа. Ребро было в цикле графа, поэтому связность нарушена не была. Так как ребро содержалось в цикле, то оно было общим для двух граней, поэтому при исключении ребра количество граней и количество ребер уменьшились на 1. По предположению индукции для урезанного графа теорема выполняется, следовательно, она выполняется и для исходного графа. \square

Рассмотрим планарный граф без кратных ребер и петель, в каждой связной компоненте которого есть хотя бы три ребра. В его плоской укладке для каждой грани с номером i количество ребер на ее границе q_i не меньше трех. Таким образом,

$$\sum q_i \geq 3r.$$

С другой стороны, каждое ребро принадлежит не более чем двум граням, поэтому

$$2q \geq \sum q_i.$$

В итоге

$$2q \geq 3r, \quad r \leq \frac{2}{3}q.$$

Тогда по формуле Эйлера, примененной к i -й связной компоненте графа, получаем

$$q_i = p_i + r_i - 2 \leq p_i + \frac{2}{3}q_i - 2, \quad q_i \leq 3p_i.$$

Перебрав все варианты связных графов с менее чем тремя ребрами, легко увидеть, что последняя формула выполняется и для таких графов. Значит, и для всего графа верно

$$q \leq 3p.$$

Из последнего соотношения и из того, что время работы алгоритма волны равно $O(M + N)$, вытекает следующий результат.

Теорема 6. *Для случая планарного графа без кратных ребер и петель, состоящего из N вершин, алгоритм волны работает за время $O(N)$.*

Отметим, что параллельно мы получили оценки количества ребер и граней через количество вершин в планарном графе.

Теорема 7. *Для случая планарного графа без кратных ребер и петель, имеющего в плоском представлении p вершин, q ребер, r граней, верны оценки*

$$q \leq 3p,$$

$$r \leq 2p.$$

Случай, когда в графе имеются связные компоненты с количеством ребер меньше трех, легко рассмотреть отдельно, учитывая, что

- количество ребер для каждой такой компоненты строго меньше количества ее вершин;
- при добавлении к графу отдельной связной компоненты с количеством ребер меньше трех количество граней не увеличится.

Случай, когда в планарном графе без кратных ребер и петель имеются связные компоненты с количеством ребер меньше трех, легко рассмотреть отдельно. Действительно, есть только три возможных вида таких компонент: отдельная вершина; отдельное ребро; вершина, инцидентная двум ребрам. Легко увидеть, что в каждом из этих трех случаев количество ребер компоненты строго меньше количества ее вершин и при добавлении к графу такой связной компоненты количество граней не увеличится. Поэтому при добавлении таких связных компонент теорема 7 остается верной.

15.3. Поиск кратчайшего пути в графе. Алгоритм Дейкстры (Dijkstra's algorithm)

Пусть дан некоторый конечный граф $G = (V, E)$, состоящий из N вершин, и две его вершины $a, b \in V$. Пусть для каждого ребра e графа задано неотрицательное вещественное число $l(e)$, которое будем называть длиной ребра. Здесь слово *длина* не имеет никакого отношения к понятию *расстояния*. Требуется найти путь между вершинами графа a и b минимальной длины, т. е. требуется найти последовательность вершин графа $\{x_1, \dots, x_n\}$, где $x_i \in V$, $x_1 = a$, $x_n = b$, $(x_i, x_{i+1}) \in E$ с минимально возможной суммой $\sum_i l(x_i, x_{i+1})$.

Стандартным решением данной задачи является *алгоритм Дейкстры*.

Основная идея алгоритма заключается в следующем.

Пусть Q_n — множество, состоящее из n ближайших вершин к вершине a вместе с длинами пути от a до этих вершин, т. е. в каждом элементе множества $q_i \in Q_n$ содержится номер соответствующей вершины x_i и длина l_i пути от a до этой вершины: $q_i = (x_i, l_i)$. Можно предполагать, что последовательность $\{l_i\}$ является неубывающей.

Утверждается, что *ближайшая к a вершина графа из вершин, не внесенных в Q_n , задается следующим соотношением:*

$$\operatorname{argmin}(v, w \in V, w \in Q_n, v \notin Q_n, (w, v) \in E: l(w) + |(w, v)|); \quad (*)$$

здесь и далее соотношения $w \in Q_n, v \notin Q_n$ говорят о том, что w встречается среди вершин, внесенных в Q_n , а v — нет; длина ребра (w, v) обозначается $|(w, v)|$. Таким образом, элемент q_{n+1} состоит из вершины v , на которой достигается указанный минимум, и соответствующей длины $l_{n+1} = l(w) + |(w, v)|$. Отметим, что минимум вычисленного расстояния может достигаться одновременно на нескольких вершинах, и тогда в качестве следующей вершины выбирается любая из них.

Доказательство. Допустим, что алгоритм $(*)$ не находит ближайшую к a вершину графа из вершин, не внесенных в Q_n , т. е. существует вершина $s \notin Q_n$ с минимальной длиной реального пути от a до s меньше величины, найденной в $(*)$. Пусть кратчайший путь от a до s проходит по последовательности вершин графа $\{x_1 = a, x_2, \dots, x_n, x_{n+1} = s\}$. Выберем в этой последовательности

элемент x_k , такой что все элементы до него принадлежат Q_n , а сам он не принадлежит Q_n . Такой элемент существует и $k > 1$, так как x_1 принадлежит Q_n , а x_{n+1} не принадлежит Q_n . Итак, $x_{k-1} \in Q_n$, $x_k \notin Q_n$, по допущению длина пути $\{a, x_2, \dots, x_k\}$ меньше величины, найденной в (*), что сразу приводит к противоречию с алгоритмом вычисления (*). \square

При поиске значения (*) формально требуется перебрать все вершины $w \in Q_n$ и для каждой из них перебрать все смежные вершины. Будем далее предполагать, что в графе отсутствуют кратные ребра и петли, тогда процедура (*) требует выполнения $O(N^2)$ операций, из чего следует, что суммарное время работы алгоритма равно $O(N^3)$.

На самом деле для выполнения процедуры (*) можно обойтись $O(N)$ операциями. Для этого заметим, что за один поиск значения (*) к множеству Q_n добавляется всего одна точка и поэтому на следующем шаге значения l_x изменятся только у вершин, смежных этой новой точке. Таким образом, для пересчета в (*) всех значений l_x требуется пересчитать l_x только для точек, смежных одной точке, полученной при предыдущем выполнении (*). В силу введенных предположений это можно сделать за $O(N)$ операций. Искать же минимум l_x можно по всем вершинам графа, не принадлежащим Q_n , что тоже требует $O(N)$ операций. Таким образом, процедуру (*) можно реализовать за $O(N)$ операций.

Для реализации алгоритма Дейкстры следует добавить в каждую вершину w графа вещественное число l_w , характеризующее длину кратчайшего пути от вершины a до вершины w и логическую переменную s_w , указывающую, посчитана ли на данный момент эта кратчайшая длина пути.

Для упрощения дальнейшей жизни (т. е. для поиска собственно кратчайшего пути при определенных значения l_w) можно в каждом элементе также хранить ссылку на вершину, из которой мы в данную вершину пришли. Для текущей вершины w будем эту ссылку называть back_w . Таким образом, элементы Q_i представляются в виде $q = (w, l_w, s_w, \text{back}_w)$.

Будем предполагать, что конкретная техника позволяет нам инициализировать все l_w значением, равным плюс бесконечность (+INF), что мы и сделаем. Все s_w инициализируем значением, равным FALSE.

Введем переменную c , в которой будем хранить номер последней вершины, добавленной в множество вершин с найденной минимальной длиной пути до вершины. Тогда алгоритм будет выглядеть следующим образом.

Алгоритм Дейкстры

$c = a; s_c = \text{TRUE}; l_c = 0; \text{back}_c = \text{NULL} // Q_0 = \{(a, 0, \text{true}, \text{NULL})\}$

Вечный цикл

Для всех вершин v , смежных c

Если $s_v == \text{FALSE}$ и ($l_v == +\text{INF}$ или $l_c + |(c,v)| < l_v$), то
 $l_v = l_c + |(c,v)|; \text{back}_v = c$

$l = +\text{INF}$

Для всех вершин v графа

Если $s_v == \text{FALSE}$ и $l_v < l$, то $l = l_v; z = v$

Если $l == +\text{INF}$, то ВЫЙТИ// в графе не осталось элементов

Если $z == b$, то ВЫЙТИ // дошли до конца пути

$s_z = \text{TRUE}; c = z // Q_{n+1} = Q_n \cup \{(z, l, \text{true}, \text{back}_z)\}$

Конец вечного цикла

Если после завершения первой части алгоритма $l == +\text{INF}$, то это означает, что от a до b дойти по ребрам невозможно. Если же вершина b достигнута, мы можем по ссылкам back_w добраться от b до a .

Можно написать тот же самый алгоритм в виде программы на C++ с использованием STL (будем считать значения не менее $1.e100$ бесконечностью, хотя это не является хорошей практикой). Заголовков программы вместе с определениями структур для вершины и ребра графа и переопределением оператора $>>$ для ввода номеров вершин ребра и его длины из файла выглядят следующим образом:

```

1 #include<stdio.h>
2 #include<iostream>
3 #include<algorithm>
4 #include<fstream>
5 #include<vector>
6 using namespace std;
7 struct Vertex{double l=1.e100; int iback=-1,s=0;};
8 struct Edge{int i0,i1; double d;
9             int operator()(int i){return i==i0?i1:i0;}};

```

```

10 ifstream &operator>>(ifstream &f, Edge &p)
11 {f>>p.i0>>p.i1>>p.d; return f;}

```

Оператор (`>>`) для ребра возвращает номер одной вершины ребра по заданному номеру другой вершины ребра.

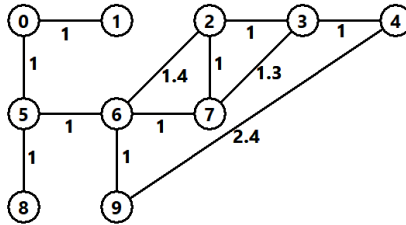
Описание графа в виде массива ребер с длинами будет считываться из файла `e.txt`. В первой строке этого файла будут заданы номера вершин графа, являющихся концами искомого пути, и количество вершин в графе. Например, файл может выглядеть следующим образом:

```

1 0 4 10
2 0 1 1 0 5 1 2 3 1 3 4 1 5 8 1 5 6 1
3 6 9 1 6 7 1 6 2 1.4 7 2 1 7 3 1.3 9 4 2.4

```

Здесь описан следующий граф:



Граф с длинами ребер, в котором строится кратчайший путь от вершины 0 до вершины 4

Функция, в которой вводится информация по графу и ищется кратчайший путь от вершины с номером `a` до вершины с номером `b`, имеет следующий вид:

```

1 int main(void)
2 {vector<Vertex> p; vector<Edge> e; vector<vector<int>> n;
3 int a,b,j,N; double l; int ipt; Edge E;
4 //—
5 //ввод вектора ребер с номерами вершин ребра и его длиной
6 //в формате i0 i1 d; в первой строке задаются a b кол. вершин
7 ifstream f("e.txt"); f>>a>>b>>N; n.resize(N); p.resize(N);
8 for(int i=0;f>>E;i++)//параллельно создаем список инц.ребер:
9 {e.push_back(E); n[E.i0].push_back(i); n[E.i1].push_back(i);}
10 //— ipt – номер очередной точки, добавляемой к Qn:

```

```

11  for (ipt=a, p[a].l=0, p[a].s=1; ipt!=b;)
12  {//мы сознательно не проверяем, что p[j].s==0:
13    for (auto i:n[ipt]) if ((l=p[ipt].l+e[i].d)<p[j=e[i](ipt)].l)
14      {p[j].iback=ipt; p[j].l=l;} //обновляем длины путей около ipt
15    l=1.e100; ipt=-1;
16    for (size_t i=0; i<p.size(); i++) if (p[i].s==0&& p[i].l<l)
17      {ipt=i; l=p[i].l;} //ищем мин.длины путей до смежных вершин
18    if (ipt<0) goto le; //если не нашли ничего, то пути не суц.
19    p[ipt].s=1; //признак того, что p[ipt] принадлежит Qn
20  }
21  //—
22  {cout<<"l="<<p[b].l<<endl; vector<int> w; //ищем обратный путь
23    for (int k=b; k>=0; k=p[k].iback) w.push_back(k);
24    reverse(w.begin(), w.end()); //разворачиваем путь
25    for (int i:w) {cout<<i<<" "; cout<<endl; //распечатываем путь
26  }
27  return 0;
28  } le:
29  cout<<"path not found"<<endl;
30  return 0;
31  }

```

В процедуре строится список инцидентных ребер n . Стоит обратить внимание на то, что в 13-й строке кода должны обновляться длины путей до вершин, смежных очередной добавляемой вершине ipt , которые не принадлежат текущему множеству Q_n , т. е. в операторе `if` формально должно проверяться условие `p[j].s==0`. Однако мы этого не делаем, так как если некоторая вершина принадлежит Q_n , то в ней уже лежит длина кратчайшего пути до данной вершины, поэтому длина пути до нее не может быть больше длины пути до только что добавленной к Q_n вершины с индексом ipt .

В алгоритме Дейкстры каждое ребро графа при переборе ребер, инцидентных текущей вершине, затрагивается дважды: в первый раз — когда номер текущей вершине равен первому индексу ребра, а второй — когда номер текущей вершины равен второму индексу ребра; таким образом, суммарно перебор ребер занимает время $O(M)$, где M — количество ребер графа. Для случая графа без кратных ребер и петель $M < N^2$, где N — количество вершин графа. Кроме того, на каждом шаге алгоритма надо находить минимум длин путей по вершинам, смежным вершинам из Q_n , но не принадлежащим Q_n ,

что занимает время $O(N)$. Поскольку это надо делать в худшем случае для всех вершин графа, то суммарное время выполнения этой части алгоритма равно $O(N^2)$. Таким образом, верна следующая теорема.

Теорема 8. *Для случая графа без кратных ребер и петель, состоящего из N вершин, алгоритм Дейкстры работает за время $O(N^2)$.*

Для планарного графа без кратных ребер и петель $M = O(N)$, поэтому, если бы мы умели находить минимум длин путей быстрее, чем за время $O(N)$, для этого вида графов мы бы получили более эффективный алгоритм. Быстро находить минимум возможно при использовании структуры данных *пирамида*, которую мы обсуждали ранее. Если опустить подробности, то в этом и состоит *модифицированный алгоритм Дейкстры*. Теперь мы будем хранить вершины графа, не занесенные в Q_n , в виде пирамиды. Точнее, в пирамиде будут лежать ссылки (указатели) на вершины графа, расположенные в векторе (массиве) вершин графа. Длины путей до соответствующих вершин, как всегда, будут лежать в вершинах графа в массиве вершин графа. Для пирамиды операции добавления элементов, удаления вершины пирамиды, нахождения минимума (если пирамида расположена по убыванию), уменьшения значения элемента в пирамиде (если пирамида расположена по убыванию) выполняются за время $O(\log(n))$. А больше никаких операций с вершинами графа при нахождении минимума элементов и при модификации множества вершин, не входящих в Q_n , нам выполнять не надо.

К сожалению, есть тонкость, которая при новом подходе существенно усложнит нам жизнь. В алгоритме Дейкстры при операции

Если $s_v == \text{FALSE}$ и ($l_v == +\text{INF}$ или $l_c + |(c,v)| < l_v$), то
 $l_v = l_c + |(c,v)|$; $\text{back}_v = c$

в случае $l_v \neq +\text{INF}$ значение l_v должно уменьшиться, а для этого нам теперь потребуется не только изменить значение l_v в вершине графа, но и скорректировать пирамиду. Эту коррекцию несложно выполнить (с помощью введенной нами ранее операции `Heapify` или с помощью функции `STL push_heap`), но для этого надо знать положение изменяемого элемента в векторе, в котором хранится пирамида. В рамках рассмотренного нами ранее алгоритма эту информацию мы не можем получить напрямую. В любом случае в изначальном

векторе вершин v в каждой вершине с индексом i нам придется хранить номер данной вершины в пирамиде — $v[i].ih$. Вопрос заключается в том, как корректировать данные индексы $v[i].ih$ при коррекции пирамиды. Можно предложить два варианта решения данной проблемы. Мы можем корректировать индексы в пирамиде для элементов, хранящихся в массиве исходных вершин, напрямую при коррекции пирамиды, вместе с коррекцией положения элементов в пирамиде. Это можно реализовать с помощью соответствующего дополнения созданной нами в [11] процедуры `Heapify`. Второй вариант — использовать стандартную функцию `STL push_heap`. В этом случае мы можем переопределить оператор присваивания элементов пирамиды таким образом, чтобы при присваивании элемента пирамиды p_i , в котором содержался индекс вершины графа j , элементу пирамиды p_k , в котором содержался индекс вершины графа l , происходил бы обмен местами значений $v[j].ih$ и $v[l].ih$.

Разберемся с первым из предлагаемых вариантов решения задачи. Заголовок файла с функцией, решающей задачу, и с описанием структуры вершины будет выглядеть следующим образом.

```

1 #include<iostream>
2 #include<fstream>
3 #include<vector>
4 #include<algorithm>
5 using namespace std;
6 struct Vertex{int  iback=-1,ih=-1; double l=1.e100;
7             Vertex(int  i=-1,int  j=-1){iback=i; ih=j; }};

```

Здесь в структуру добавлен элемент `ih`, в котором будет храниться индекс данной вершины в пирамиде.

Для работы с пирамидой напишем свои функции `push_heap()` и `pop_heap()`.

В функцию `push_heap()` будем передавать вектор `b` с пирамидой, в которой будут лежать индексы вершин, вектор `v` вершин дерева и индекс `i` добавляемого в пирамиду элемента. Будем считать, что пирамида организована по возрастанию (т. е. элемент с индексом $(i-1)/2$ не превосходит элемента с индексом i). По аналогии с одноименной функцией из `STL` предполагается, что вектор с пирамидой организован как пирамида до (не включая) элемента с индексом i . Функция должна включить элемент с индексом i в пирамиду. Дан-

ная функция не должна изменять размер векторов, так как с помощью данной функции можно либо добавлять элемент в пирамиду (в этом случае индекс i будет последним в векторе с пирамидой), либо подправлять пирамиду, если значение элемента с индексом i в ней уменьшилось.

В функцию `pop_heap()` будем передавать вектор с пирамидой, в которой будут лежать индексы вершин, и вектор вершин дерева. Функция должна менять местами первый и последний элементы вектора с пирамидой и подправлять вектор с пирамидой с урезанной на 1 длиной до полной пирамиды.

Реализация функций `push_heap()` и `pop_heap()` аналогична их реализации в [11], но при обмене местами элементов пирамиды надо также менять местами ссылки на эти элементы в массиве вершин (значения `v[*].ih`).

```

1 #define iL (2*i)+1 /* индекс левого потомка индекса i */
2 #define iR (2*i)+2 /* индекс правого потомка индекса i */
3 #define iB (i-1)/2 /* индекс родителя индекса i */
4 #define Swap(a,b) {swap(a,b); swap(v[a].ih,v[b].ih);}
5 void pop_heap(vector<int> &b, vector<Vertex> &v)
6 {size_t i=0,n=b.size()-1; Swap(b.front(),b.back());
7  while(iR<n)//пока есть два потомка
8  {
9    if(v[b[iL]].l>=v[b[i]].l&&v[b[iR]].l>=v[b[i]].l)return;
10   if(v[b[iL]].l>v[b[iR]].l)
11    {Swap(b[i],b[iR]); i=iR;}//если мин. эл-т имеет индекс iR
12   else
13    {Swap(b[i],b[iL]); i=iL;}//если мин. эл-т имеет индекс iR
14  }//для одного оставшегося потомка:
15  if(iL<n && v[b[iL]].l<v[b[i]].l)Swap(b[i],b[iL]);
16 }
17 void push_heap(vector<int> &b, vector<Vertex> &v,size_t i)
18 {
19  for(;i>0;i=iB)//идем вверх по ветке дерева
20  {
21   if(v[b[iB]].l<=v[b[i]].l)return;//если все хорошо, то выход
22   Swap(b[i],b[iB]);//иначе меняем местами тек.эл-т и пред.
23  }
24 }
```

Осталось реализовать сам модифицированный алгоритм Дейкстры. Будем хранить пирамиду в векторе b . Ребра графа вообще хранить не будем, так как сразу создадим список смежных вершин n . Придется отдельно хранить длины ребер: для ребра, инцидентного вершинам i и $n[i][j]$, длина ребра будет лежать в $L[i][j]$. Функция с реализацией модифицированного алгоритма Дейкстры будет иметь следующий вид (формат описания графа такой же, как на с. 296):

```

1  int main(void)
2  {int iv0, iv1, nv; vector<Vertex> v; vector<vector<int>> n;
3   vector<vector<double>> L;
4   ifstream f("e.txt"); f>>iv0>>iv1>>nv; int i0, i1; double l;
5   v.resize(nv); n.resize(nv); L.resize(nv);
6   while(f>>i0&&f>>i1&&f>>l)//создаем список смежных вершин
7   {n[i0].push_back(i1); if(1)n[i1].push_back(i0);
8     L[i0].push_back(l); if(1)L[i1].push_back(l);}
9   }cout<<"n"<<v.size()<<endl;
10  //— b — пирамида; первым в b заносится начало пути
11  vector<int> b; b.push_back(iv0); v[iv0].l=0; v[iv0].ih=0;
12  while(!b.empty())
13  {//ip=b[0] — индекс следующей ближайшей вершины
14   int ip=b[0], j; pop_heap(b, v); b.pop_back();
15   if(ip==iv1)goto lOk;//если дошли до конца, то уходим
16   for(size_t i=0; i<n[ip].size(); i++)//для всех соседей ip
17     if(double l; v[j=n[ip][i]].l>(l=v[ip].l+L[ip][i]))
18     {//если до соседа ip есть более короткий путь:
19      v[j].l=l; v[j].iback=ip;
20      //если мы еще там не были, то добавляем вершину в пирамиду:
21      if(v[j].ih==-1){v[j].ih=b.size(); b.push_back(j);}
22      push_heap(b, v, v[j].ih);//в любом случае правим пирамиду
23    }
24  }cout<<"path not found"<<endl;return 0;
25  //—
26  lOk:vector<int> p;//строим обратный путь:
27  for(int i=iv1; i!=iv0; i=v[i].iback)p.push_back(i);
28  p.push_back(iv0); reverse(p.begin(), p.end());//разворачиваем
29  for(int x:p){cout<<x<<" ";} cout<<endl;//распечатываем путь
30  return 0;
31  }

```

Хочется обратить внимание на 21-ю строку кода. При добавлении элемента в пирамиду (если вершина, смежная вершинам из Q_n , встретилась в первый раз, т. е. $v[j].ih == -1$) задается значение $v[j].ih = b.size()$ (последнее место в массиве пирамиды) и функция `push_heap()` добавляет индекс в пирамиду. Если же индекс встретившейся вершины уже есть в пирамиде, то значение длины пути до вершины с индексом $v[j].ih$ уменьшается ($v[j].l = 1$) и та же функция `push_heap()` служит уже для коррекции положения элемента с индексом $v[j].ih$ с уменьшившимся значением l в пирамиде. Эту возможность функции `push_heap()` мы обсуждали ранее.

Также обратим внимание на условие `if(1)` в строках 7, 8. Если заменить его на условие `if(0)`, то мы получим решение той же задачи для ориентированного графа.

По построению алгоритма время работы модифицированного алгоритма Дейкстры равно $O(\#E) + O(N \log N)$, где $\#E$ — количество ребер графа, N — количество вершин графа. Здесь первая часть суммы оценивает модификацию значений длин путей, хранящихся в вершинах, поскольку в данной части алгоритма каждое ребро затрагивается не более чем два раза. Второе слагаемое оценивает время нахождения минимальной длины пути за N шагов алгоритма с учетом того, что на каждом шаге минимум длин путей находится за время $O(\log(N))$. Для планарного графа без кратных ребер и петель $\#E = O(N)$. Все сказанное позволяет сформулировать следующую теорему.

Теорема 9. *Для случая планарного графа без кратных ребер и петель, состоящего из N вершин, модифицированный алгоритм Дейкстры работает за время $O(N \log(N))$.*

Модифицировав алгоритм, мы получили лучшую оценку для времени работы алгоритма Дейкстры для планарного графа без кратных ребер и петель.

15.4. Модифицированный алгоритм Дейкстры на основе STL

Более интересна вариация алгоритма Дейкстры, которую мы будем называть *модифицированный алгоритм Дейкстры на основе STL*. В данной вариации алгоритма в каждом элементе пирамиды

будет лежать, кроме ссылки на вершину графа из массива вершин графа, длина пути от исходной вершины a до данной вершины графа (эта длина также будет лежать в вершинах графа в массиве вершин). Таким образом, длина пути будет храниться сразу в двух местах. При добавлении вершины в пирамиду длина пути до данной вершины в вершине из массива вершин и в элементе пирамиды будут равны. Если станет необходимо уменьшить длину пути до данной вершины, то мы просто проигнорируем то, что в пирамиде уже есть ссылка на данную вершину, и снова добавим в пирамиду ссылку на ту же вершину с новой длиной пути до данной вершины. При этом длина пути до данной вершины в массиве вершин скорректируется соответствующим образом. Отметим, что при таком поведении количество элементов в пирамиде не будет больше числа ребер в графе, т. е. для планарного графа без кратных ребер и петель размер пирамиды будет $O(N)$. Поэтому теорема 9 будет верна и для данной модификации алгоритма Дейкстры. Можно говорить, что в пирамиде есть *корректные* элементы (те, для которых длина пути до вершины, заданная в пирамиде, совпадает с длиной пути до вершины, заданной в массиве вершин) и *некорректные* элементы (где длины путей до вершины в пирамиде и в массиве вершин не совпадают). Теперь при поиске минимального элемента пирамиды мы будем извлекать элементы из начала пирамиды, игнорируя некорректные элементы. Первый встреченный корректный элемент в начале пирамиды будет элементом с минимальной длиной пути до данной вершины.

Здесь стоит обратить внимание на некоторую неприятность, связанную с тем, что при сравнении длин путей нам придется, фактически, сравнивать на равенство вещественные числа. Но это противоречит рассуждениям из [11] и информации о стандарте представления вещественных чисел с плавающей точкой [17], из которых вытекает в общем случае запрет на такие действия. Нам остается только надеяться (на самом деле, обоснованно), что причины этого запрета не распространяются на данную задачу.

Отметим некоторую тонкость при доказательстве оценки времени работы данной модификации алгоритма. Тонкость связана с тем, что мы не учли, что на каждом шаге алгоритма можно много раз (вообще говоря, $O(N)$) доставать некорректные вершины из пирамиды (в исходном модифицированном алгоритме Дейкстры минимум длин

всегда лежал непосредственно в ее вершине). Однако легко увидеть, что количество занесений элементов в пирамиду не превосходит $\#E$, поэтому и извлечений элементов из пирамиды во всем алгоритме не может быть больше, чем $\#E$. Таким образом, для планарного графа без кратных ребер и петель во всем алгоритме количество извлечений элементов из пирамиды равно $O(N)$, и теорема 9 будет выполняться также для данной модификации алгоритма Дейкстры.

Заголовок файла с реализацией алгоритма будет иметь следующий вид:

```

1 #include<iostream>
2 #include<fstream>
3 #include<vector>
4 #include<list>
5 #include<algorithm>
6 using namespace std;
7 //— вершина для массива вершин:
8 struct SPoint{int iback=-1; double l=1.e100;
9     SPoint(int i=-1){iback=i;}};
10 struct SP{int i; double l; //вершина для пирамиды
11     SP(int i=-1, double l=1.e100){this->i=i; this->l=l;}};
12 bool operator<(const SP&a, const SP&b){return a.l>b.l;}
```

Здесь мы реализовали две структуры. Первая из них служит для хранения информации о вершине графа в массиве вершин графа. Вторая будет использоваться для хранения индекса и текущей длины пути в пирамиде.

Так же как и в предыдущей реализации, будем хранить пирамиду в векторе v . Ребра графа вообще хранить не будем, так как сразу создадим список смежных вершин n . Длины ребер будем хранить в следующем виде: для ребра, инцидентного вершинам i и $n[i][j]$, длина ребра будет равна $L[i][j]$. Функция с реализацией модифицированного алгоритма Дейкстры для STL будет иметь следующий вид:

```

1 int main(void)
2 {int iv0, iv1, nv; vector<SPoint> v; //вектор вершин
3   vector<vector<int>> n; //список смежных вершин
4   vector<vector<double>> L; //L[i][j] - длина ребра (i, n[i][j])
5   {ifstream f("e.txt"); f>>iv0>>iv1>>nv; int i0, i1; double l;
6     v.resize(nv); n.resize(nv); L.resize(nv);
```

```

7   while(f>>i0&&f>>i1&&f>>l)//ввод графа и списка смежн.вершин
8   {n[i0].push_back(i1); if(1)n[i1].push_back(i0);
9     L[i0].push_back(1);  if(1)L[i1].push_back(1);}
10  }
11  // b – пирамида; изначально в ней только вершина начала пути
12  vector<SP> b; b.push_back(SP(iv0,0)); v[iv0].l=0;
13  while(!b.empty())//b – пирамида
14  {int ip,j; SP B;
15   do{if(b.empty())goto 10; B=b[0]; ip=B.i;
16     pop_heap(b.begin(),b.end()); b.pop_back();
17     }while(B.l>v[ip].l);//пока вершина не станет корректна
18   if(ip==iv1){goto 10Ok;} double l; //если дошли до конца...
19   for(size_t i=0;i<n[ip].size();i++)//перебираем смежн.вершины
20   if(v[j=n[ip][i]].l>(l=v[ip].l+L[ip][i]))
21   {b.push_back(SP(j,l)); push_heap(b.begin(),b.end());
22     v[j].l=l; v[j].iback=ip;}
23 } 10: cout<<"path not found"<<endl;return 0;
24 //
25 10Ok:vector<int> p;//идем от конца пути к началу по ссылкам:
26 for(int i=iv1;i!=iv0;i=v[i].iback)p.push_back(i);
27 p.push_back(iv0); reverse(p.begin(),p.end());//разворот пути
28 for(int x:p){cout<<x<<" ";} cout<<endl;//распечатываем путь
29 return 0;
30 }

```

Содержательный код алгоритма Дейкстры здесь занимает строки от 12-й до 23-й. Если избавиться от вынужденных переносов строк, то мы получим код длиной в 9 строк, что весьма немного.

В предыдущем разделе предлагалось два варианта реализации модифицированного алгоритма Дейкстры, и мы рассмотрели один из них. У второго предложенного варианта решения задачи оказывается более короткий код, но он опирается на некоторые (весьма логичные!) внутренние особенности реализации функций работы с пирамидой, поэтому данную реализацию нужно рассматривать скорее как экзотику (работоспособную!), нежели рекомендованную к употреблению реализацию алгоритма. В данном варианте решения задачи мы будем, так же как и в последнем случае, использовать функции работы с пирамидой, реализованные в STL. Будем считать дан-

ные, описывающие граф, в том же формате, что и в предыдущей реализации.

Заголовок файла с функцией, решающей задачу, будет иметь следующий вид:

```

1 #include<iostream>
2 #include<fstream>
3 #include<vector>
4 #include<algorithm>
5 using namespace std;
6 struct Vertex{int iback=-1,ih=-1; double l=1.e100;
7     Vertex(int i=-1,int j=-1){iback=i; ih=j;}};
8 vector<Vertex> v;
9 struct SP{int i=-1; SP(int i=-1){this->i=i;}
10     SP& operator=(const SP& r);
11     bool operator<(const SP&b
12     {if(v[i].l>v[b.i].l){return true;} return false;}};
13 vector<SP> b;
14 SP& SP::operator=(const SP& r)
15 {auto i1=this-b.data(), i2=&r-b.data();
16     if(i1>=0&&i1<static_cast<decltype(i1)>(b.size())&&
17         i2>=0&&i2<static_cast<decltype(i2)>(b.size()))
18         swap(v[i].ih,v[r.i].ih);
19     i=r.i;
20     return *this;
21 }
```

Пирамида будет строиться из элементов типа SP, в которых нет ничего, кроме одного целого числа. Задание оператора < для элементов данного типа обеспечивает сортировку пирамиды по убыванию, т. е. сортировка пирамиды будет происходить в точности так же, как и в варианте решения из предыдущего раздела (при сравнении элементов $b[i]$ и $b[j]$ реально будут сравниваться величины $v[b[i]].ih$ и $v[b[j]].ih$). Создание типа-обертки над целым числом SP нужно также для переопределения операции присваивания элементов в пирамиде (фактически в пирамиде лежат только индексы элементов массива вершин). Мы (наивно) предполагаем, что в алгоритмах STL `pop_heap` и `push_heap` фактически используются только операции `swap()` над элементами пирамиды (как потом окажется, в одном месте это окажет-

ся неверным). По крайней мере в нашей реализации одноименных функций, приведенной выше, для перемещения элементов использовалась только созданная нами функция `Swap()`. При использовании `swap()` только один раз происходит присваивание элемента пирамиды элементу пирамиды, в двух других случаях в присваивании используется временная переменная. При реализации присваивания (мы переопределяем этот оператор) мы можем отлавливать случай присваивания элемента пирамиды элементу пирамиды, и в данном переопределенном операторе будем менять местами ссылки на номера элементов в пирамиде, расположенные в массиве вершин, `v[i].ih` и `v[r.i].ih`, где `i` — значение индекса вершины, расположенного в пирамиде, которому присваивается значение, а `r.i` — значение индекса вершины, расположенного справа от знака присваивания в перестройке пирамиды. Для отлавливания данного случая мы берем разность указателя на текущий (которому нужно присвоить) элемент пирамиды и на первый элемент пирамиды. Если данная разность не меньше нуля и меньше длины массива пирамиды и если аналогичные соотношения выполняются для адреса присваиваемого элемента, то мы находимся внутри пирамиды, иначе происходит либо присваивание временной переменной (через которую осуществляется `swap`), либо присваивание элементу пирамиды от временной переменной (см. строки 16, 17 приведенного выше кода). Мы ошибаемся в нашем предположении только в одном месте: в начале реализации функции `swap` не обязательно менять местами первый и последний элементы массива (для извлечения первого элемента из пирамиды), достаточно просто присвоить последний элемент массива вершине пирамиды, что и делает функция `STL pop_heap()`. При этом последний элемент массива оказывается уже за границами реальной пирамиды и вполне возможно его присваивание временной переменной, а потом — присваивание временной переменной первому элементу массива с пирамидой. Поэтому нам придется после выполнения `pop_heap()` вручную корректировать индекс элемента в пирамиде, расположенный в соответствующем элементе массива вершин: `v[b[0].i].ih=0`.

Код функции, решающей задачу, примет следующий вид:

```

1  int main(void)
2  {int iv0, iv1, nv;  vector<vector<int>> n;
3  vector<vector<double>> L;
```

```

4  {ifstream f("e.txt"); f>>iv0>>iv1>>nv; int i0,i1; double l;
5  v.resize(nv); n.resize(nv); L.resize(nv);
6  while(f>>i0&&f>>i1&&f>>l)//создаем список смежных вершин
7  {n[i0].push_back(i1); if(1)n[i1].push_back(i0);
8  L[i0].push_back(l); if(1)L[i1].push_back(l);}
9  }cout<<"n="<<v.size()<<endl;
10 //— b — пирамида; в начале в b заносится начало пути
11 b.push_back(iv0); v[iv0].l=0; v[iv0].ih=0;
12 while(!b.empty())
13 {//ip=b[0] = индекс следующей ближайшей вершины
14  int ip=b[0].i,j; pop_heap(b.begin(),b.end());
15  v[b[0].i].ih=0; b.pop_back();
16  if(ip==iv1)goto lOk;//если дошли до конца, то уходим
17  for(size_t i=0;i<n[ip].size();i++)//для всех соседей ip
18  if(double l; v[j=n[ip][i]].l>(l=v[ip].l+L[ip][i]))
19  {//если до соседа ip есть более короткий путь, чем было:
20  v[j].l=1; v[j].iback=ip;
21  //если мы еще там не были, то добавляем вершину в пирамиду:
22  if(v[j].ih==-1){v[j].ih=b.size();b.push_back(j);}
23  //в любом случае правим пирамиду:
24  push_heap(b.begin(),b.begin()+v[j].ih+1);
25  }
26  }cout<<"path not found"<<endl;return 0;
27 //—
28 lOk:vector<int> p;//строим обратный путь:
29 for(int i=iv1;i!=iv0;i=v[i].iback)p.push_back(i);
30 p.push_back(iv0); reverse(p.begin(),p.end());//разворачиваем
31 for(int x:p){cout<<x<<" ";} cout<<endl;//распечатываем путь
32 return 0;
33 }

```

Отметим, что векторы пирамиды b и вершин графа v нам пришлось вынести в глобальные переменные, чтобы иметь к ним доступ из функции присваивания. В реальной программе мы можем, например, погрузить функцию, реализующую алгоритм Дейкстры, в отдельный класс и в него же погрузить эти переменные. Данный подход обеспечит надлежащий уровень инкапсуляции.

Осталось заметить, что данная версия алгоритма по сути ничем принципиально не отличается от базового модифицированного алгоритма Дейкстры. Таким образом, для данного алгоритма автоматически выполняется теорема 9.

Список использованных источников

1. Бахвалов Н. С., Жидков Н. П., Кобельков Г. М. *Численные методы*. М.: Изд-во Моск. ун-та, 2015. 639 с. ISBN 978-5-9963-2616-7.
2. Богачёв К. Ю. *Основы параллельного программирования*. М.: Лаборатория знаний, 2020. 345 с. ISBN 978-5-00101-758-5.
3. Василевский Ю. В., Данилов А. А., Липников К. Н., Чугунов В. Н. *Автоматизированные технологии построения неструктурированных расчетных сеток*. Т. 4. М.: Физматлит, 2016. 216 с. ISBN 978-5-9221-1730-2.
4. Керниган Б., Ритчи Д. *Язык программирования С*. М.: Вильямс, 2015. 304 с. ISBN 978-5-8459-1975-5.
5. Боресков А. В., Харламов А. А. *Основы работы с технологией CUDA*. М.: ДМК Пресс, 2010. 233 с. ISBN 978-5-94074-578-5.
6. Валединский В. Д., Корнев А. А. *Методы программирования в задачах и примерах на C/C++*. М.: Изд-во Моск. ун-та, 2023. 413 с. ISBN 978-5-19-011927-5.
7. Кнут Д. *Искусство программирования для ЭВМ*. Т. 1—3. М.: Мир, 1996—1998.
8. Кормен Т., Лейзерсон Ч., Ривест Р. *Алгоритмы. Построение и анализ*. М.: МЦНМО, 1999. 960 с. ISBN 5-900916-37-5.
9. Кронрод М. А. Оптимальный алгоритм упорядочения без рабочего поля // *Докл. АН СССР*. 1969. Т. 186, № 6. С. 1256—1258.
10. Плещинский Н. Б., Плещинский И. Н. *Многопроцессорные вычислительные комплексы. Технологии параллельного программирования*. Казань: Казанский фед. ун-т, 2018. 80 с.
11. Староверов В. М., Шевелева А. В. *Лекции по курсу «Работа на ЭВМ и программирование»*. Ч. 1. М.: Изд-во Моск. ун-та, 2024. 317 с. ISBN 978-5-19-012046-2.
12. Степанов А., Менг Ли. *Руководство по стандартной библиотеке шаблонов (STL)*. М.: Московский государственный институт радиотехники, электроники и автоматики (Технологический Университет), 1999.

13. Столмен Р., Пеш Р., Шебс С. и др. *Отладка с помощью GDB. Отладчик гни уровня исходного кода. Восьмая редакция, для GDB версии 5.0.* Free Software Foundation, 2000. Пер. Дмитрия Сиваченко (2000). ISBN 1-882114-77-9.
14. Таненбаум Э., Бос Х. *Современные операционные системы.* 4-е изд. СПб.: Питер, 2022. 1120 с. ISBN 978-5-4461-1155-8.
15. Уилсон М. *Расширение библиотеки STL для C++. Наборы и итераторы.* М.: ДМК Пресс, 2008. 608 с. ISBN 978-5-94074-442-9.
16. Хуанг Т. *Обработка изображений и цифровая фильтрация.* М.: Мир, 1979. 320 с.
17. *IEEE Standard for Floating-Point Arithmetic.* IEEE Std. 754-2019. IEEE, 2019. 84 pp. doi:10.1109/IEEESTD.2019.8766229. ISBN 978-1-5044-5924-2.
18. Препарата Ф., Шеймос М. *Вычислительная геометрия: Введение* / Под ред. М. Баяковского. Пер. с англ. С. А. Ви́чеса, М. М. Комарова. М.: Мир, 1989. 478 с. ISBN 5-03-001041-6.
19. Stroustrup B. *The C++ Programming Language* (4th Edition). Addison-Wesley, 2013. 1360 pp. ISBN 978-0321563842.
20. Mattson T. G., Yun (Helen) He, Koniges A. E. *The OpenMP Common Core.* The MIT Press, 2019. 320 pp. ISBN 978-02-625-3886-2.