Содержание

1. Введение

Данное учебное пособие является конспектами лекций второго полугодия курса «Работа на ЭВМ и программирование», читаемого в течение ряда лет на механико-математическом факультете МГУ им. М.В. Ломоносова. Курс является логическим продолжением лекций, читаемых на механико-математическом факультете в предыдущем семестре [12].

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);

- разбор языка Python в плоскости работы с массивами и объектно-ориентированного программирования (в понимании Python);

- введение в теорию графов на уровне разбора различных модификаций алгоритма Дейкстры.

Разбор понятия *наследование* не входит в данный курс, поскольку ему посвящен следующий семестр обучения. На данный момент это направление не вмещается в курс лекций и разбирается только на семинарских занятиях после окончания лекций. Все утверждения данного курса верны для структур/классов, в которых не используется понятие *наследования*, что нигде далее упоминаться не будет.

Для проверки уровня понимания курса к концу семестра слушатель должен уметь пользоваться следующими понятиями языка C++ (ver.11):

0) правила трех, пяти, нуля (это основа!);

1) семантика перемещения и rvalue-ссылки (зачем числу 5 может понадобиться что-то присваивать?);

2) лямбда-функции (двух типов);

3) "умные" указатели (например, надо знать, как их использовать в списках, и почему это делать нельзя);

4) range-based циклы + initializer_list (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 должен свободно владеть этими терминами и не только объяснять их на собеседованиях, но и четко знать, как использовать эти понятия в прикладном программировании.

2. Простые и сложные классы. Правило трех. Исключения

Предполагается, что читатель усвоил материал предыдущей части курса [12], в частности, в плане начального (*вульгарного*) понимания языка C++. Т.е. учащийся знает общий синтаксис создания и использования структур и классов в языке C++, шаблонов, переопределения операторов, в частности, операторов используемых для ввода/вывода. Разберемся с этими темами более подробно. Данная глава посвящена детальному разбору понятий *простые* и *сложные классы* и особенностям работы с данными объектами.

Напомним, что функции класса принято называть *методами*, а данные внутри класса принято *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*.

Поэтому везде далее мы не будем подразделять понятия *структур* и *классов* и везде для краткости будем использовать понятие *класса*, если это не будет приводить к недопониманиям.

Под *приватностью* подразумевается невозможность обращения вне классов в элементам класса, а под *публичностью*, соответственно, — 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 A и B, причем внутри класса A содержится указатель на класс B, а внутри класса B содержится указатель на класс A:

```
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 нет описания до данной позиции, что пока является синтаксической ошибкой. Решить проблему можно с указанием до вышеупомянутой конструкции того, что A и B 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 A и B надо указывать, соответственно, что данные имена являются именами структур:

```
1 struct A; struct B;
2 struct A
3 {B *x;
```

```
};
struct B
{A *x;
};
```

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:

```
struct A; struct B;
struct A
{B *x;
 void f();
};
struct B
{A *x;
 void f();
};
void A::f(){x=nullptr;}
void B::f(){x=nullptr;}
```

2.1. Простые классы

Простыми структурами/классами называются структуры/классы, не подразумевающие отведения каких-либо ресурсов внутри своих объектов. Т.е. простые структуры содержать только элементы базовых типов.

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

Надо понимать, что простые классы требуют весьма ограниченного функций, необходимых для их реализации. Например, для работы с классом *вектор*

```
struct Vector
{int v[N];
 ...
};
```

в простейшем случае достаточно определить 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++){cout<<v.v[i]<<" ";}
       cout<<")"; return cout;}
```

Полный код примера выглядит следующим образом:

```
1  #include<iostream>
2  #include<fstream>
3  #include<string>
4  #include<stdlib.h>
5  #include<string.h>
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++){cout<<v.v[i]<<" ";}
       cout<<")"; return cout;}
16 int main(void)
17 {Vector v; int m[N]={0,1,2,3,4,5,6,7,8,9};
18  for(int i=0;i<N;i++)v[i]=m[i];
19  cout<<"v="<<v<<endl;
20 return 0;
21 }
```

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

```
gcc -c -D N=20 q.c
```

то макроопределение 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, т.к., 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++ используется понятие *исключений*. Про исключения надо знать, что их можно *выбрасывать* и *ловить*. При выбрасывании исключения указывается переменная/значение определенного типа, содержащая информацию о соответствующей ошибке. Далее происходит выход изо 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 соответствующую ошибку. При выходе за границы массива будем выбрасывать исключение данного типа с описанием ошибки:

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

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

```
try{
for(int i=0;i<=N;i++)v[i]=m[i];
}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 данного типа, далее (только в случае поимки исключения!) происходит выполнение блока, следующего за инструкцией *catch*, после чего продолжение программы происходит обычным способом.

Далее следует упомянуть еще одно правило хорошего тона объектно-ориентированного программирования: 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. При этом станет невозможным прямое обращение к элементам класса и вышеприведенный вариант оператора вывода содержимого класса на экран станет неработоспособным. Его можно заменить следующим образом:

```
1 ostream &operator<<(ostream&cout,const Vector &v)
2 {cout<<"(";for(int i=0;i<N;i++){cout<<v[i]<<"
      ";}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*, т.к. 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){throw
       SErr("vector: bad index");}return v[i];}
```

Здесь первая спецификация *const* гарантирует неизменяемость значения по возвращаемой ссылке (т.е. данный оператор нельзя использовать слева от знака присваивания), а второй — 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{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 следует неизбежность 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<stdlib.h>
5  #include<string.h>
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){if(i>=N){throw SErr("vector: bad
        index");}return v[i];}
16  const T &operator[](size_t i)const{if(i>=N){throw
        SErr("vector: bad index");}return v[i];}
17  Vector operator+(const Vector&b)const{Vector r; for(int
        i=0;i<N;i++){r[i]=v[i]+b.v[i];} return r;}
18  };
19  template<class T>ostream &operator<<(ostream&cout,const
        Vector<T> &v)
20  {cout<<"(";for(int i=0;i<N;i++){cout<<v[i]<<"
        ";}cout<<")";return cout;}
21  //--
22  int main(void)
23  {Vector<int> v,w; int m[N]={0,1,2,3,4,5,6,7,8,9};
24   try{
25   for(int i=0;i<=N;i++)v[i]=m[i];
26   }catch(Vector<int>::SErr err){cout<<err.s<<endl;}
27   cout<<"v="<<v<<endl;
28   w=v;
29   cout<<"w="<<w<<endl;
30   w=v+w;
31   cout<<"v+w="<<w<<endl;
32  return 0;
33  }
```

Здесь реализован вектор целых чисел, в котором тип *int* является параметром шаблона. Вместо класса описывает 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>*, что 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 задать ту же функцию, задающую оператор << для данного типа, по разному в двух местах программы, что приведет к непредсказуемым послед-

ствиям). Не забываем также, что в цикле в функции *main()* происходит выход за границы массива.

Здесь также стоит отметить, что кусок кода между комментариями //-- хорошо бы было поместить в соответствующий include-файл, а в данном коде просто вставлять данный файл в виде include-файла. Данный include-файл можно будет вставлять в разные исходные файлы программы, что не приведет к неприятностям, но надо будет следить за тем, что макроопределение N задается 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 явного их освобождения. Чаще всего под отведением/очисткой ресурсов подразумевается отведение/очистка памяти, но здесь возможны и другие виды ресурсов: сокеты, каналы/потоки работы с файлами, семафоры и т.д. Самая очевидная проблема, не позволяющая работать со сложными классами также как и с простыми, заключается в том, побайтовое присваивание или копирование объектов такого типа создает две побайтовые копии объектов, владеющие одними и теми же ресурсами. Очистка ресурсов в одном объекте делает работу со второй копией объекта некорректной. В свою очередь, вторая копия объекта в создавшейся ситуации может ничего не знать о своей нелигитимности, что приведет к некорректной работе программы. отсюда вытекает основное требование к сложным классам: они должны обеспечивать корректное присваивание и копирование объектов друг другу. Кроме этого, сложные классы должны обеспечивать автоматическую очистку всех захваченных ресурсов при смерти объекта (реализуется с помощью деструктора). Но очистка ресурсов требует корректного состояния всех переменных, идентифицирующих данные ресурсы, а это, в свою очередь, требует наличия обязательной правильной инициализации 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 которой объект a уже существует перед произведением данной операцией. Это требует очистки ресурсов в объекте a перед началом, собственно, копирования. *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 здесь предварительно не следует (банально потому, что все переменные объекта в начале копирования неинициализированы 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)наличие конструктора по умолчанию (и/или других необходимых конструкторов, создающих объект), 2)конструктора копирования, 3)деструктора, 4)оператора присваивания. Правило трех является требованием наличия трех последних пунктов в этом списке, поскольку эти пункты однозначны. Но надо понимать, что правило трех подразумевает 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), но бывают специфические объекты, для которых конструктор по умолчанию просто запрещен (это легко реализовать, создав приватный конструктор по умолчанию; тогда любая попытка создать объект конструктором без параметров приведет к синтаксической ошибке). В этом случае 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 параметров. 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 уже 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 далее 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)* используется на случай присваивания объекта самом себе (a=a). В это случае очистка объекта привела бы к дальнейшей невозможности работы с ним. Конструктор по умолчанию также задается универсальным образом:

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

Однако нам его удобно будет объединить 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 он приводит к созданию вектора заданного размера. Большим недостатком

данной конструкции является то, что она неприменима для достаточно 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];for(size_t i=0;i<n;i++)v[i]=b.v[i];}else
    SetZero();}
```

Разберемся с уже упомянутой проблемой, возникающей при конструировании шаблонов. Для нас было бы весьма удобным стандартно инициализировать все элементы создаваемого вектора (например, для вектора целых или вещественных чисел переменные хорошо бы было инициализировать нулевым значением). Если тип T представляет собой правильно созданный класс, то мы не будем иметь проблем, т.к. 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 - типы) конструктора по умолчанию нет (можно считать, что он есть, но он пустой). Однако, для POD-типов соответствующая конструкция *T()* приводит к созданию объекта соответствующего типа, инициализированного нулем (соответствующего типа). Т.е., с одной стороны, 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 по синтаксису языка C мы таким образом опишем **функцию**, возвращающую целое значение, что является совсем не тем, 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 ее можно использовать как определение 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<stdlib.h>
5  #include<string.h>
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){this->n=n; v=new T[n];
       for(size_t i=0;i<n;i++)v[i]=T();}}
16  ~Vector(){Clean();}
17  Vector(const Vector&b){CopyOnly(b);}
18  Vector &operator=(const Vector&b){if(&b!=this){Clean();
       CopyOnly(b);}return *this;}
19 //--
20  void SetZero(){v=nullptr;n=0;}
21  void Clean(){delete[] v; SetZero();}
22  void CopyOnly(const Vector &b){v=new T[n=b.n];
23   for(size_t i=0;i<n;i++)v[i]=b.v[i];}
24 //--
25  size_t size()const{return n;}
```

```
26  struct SErr{string s; SErr(const char*s){this->s=s;}};
27  T &operator[](size_t i){if(i>=n)
28   {throw SErr("vector: bad index");}return v[i];}
29  const T &operator[](size_t i)const{if(i>=n)
30   {throw SErr("vector: bad index");}return v[i];}
31  Vector operator+(const Vector&b)const{Vector r(n);
32   for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
33 };
34 template<class T>ostream &operator<<(ostream&cout,const
       Vector<T> &v)
35 {cout<<"(";for(size_t i=0;i<v.size();i++)
36  {cout<<v[i]<<" ";}cout<<")";return cout;}
37 int main(void)
38 {Vector<Vector<double>> v(N),w; int
       m[10]={0,1,2,3,4,5,6,7,8,9};
39  try{
40   for(int i=0;i<10;i++)
41   {for(int j=0;j<10;j++)v[i].push_back(m[j]);}
42  }catch(Vector<int>::SErr err){cout<<err.s<<endl;}
43  cout<<"v="<<v<<endl;
44  w=v;
45  cout<<"w="<<w<<endl;
46  w=v+w;
47  cout<<"v+w="<<w<<endl;
48 return 0;
49 }
```

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 мы имеем оператор сложения, а для объектов вектора оператор сложения задается, опять же, тем же самым шаблоном оператора сложения.

Для удобства работы и совместимости со стандартными реализациями векторов в нашем векторе должна присутствовать функция добавления элемента в конец вектора (с увеличением длины вектора на единицу). Принято называть такую функцию *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 легко увидеть, что время работы такой операции будет равно $\Theta(n^2)$. Для того, чтобы время работы данной операции равнялось $\Theta(n)$, модифицируем алгоритм способом, описанным в первом семестре лекций [12]. Введем в классе переменную *size_t nmax;*, в которой будет храниться размер реально созданного массива. А в переменной n все также будем хранить официальную длину вектора. При добавлении элементов к вектору n будет увеличиваться пока не достигнет значения n_max, а при достижении произойдет увеличение n_max вдвое с переотведением памяти. Легко увидеть, 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<stdlib.h>
5  #include<string.h>
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){v=new
        T[nmax=this->n=n];for(size_t i=0;i<n;i++)v[i]=T();}}
15  Vector(const Vector&b){CopyOnly(b);}
16  ~Vector(){Clean();}
17  Vector &operator=(const
        Vector&b){if(&b!=this){Clean();CopyOnly(b);}return
        *this;}
18  //---
19  void SetZero(){v=nullptr;n=0;nmax=0;}
20  void Clean(){delete[] v; SetZero();}
21  void CopyOnly(const Vector&b){if(b.nmax){v=new
        T[nmax=b.nmax];n=b.n; for(size_t
        i=0;i<n;i++)v[i]=b.v[i];}else SetZero();}
22  //---
```

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

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 такие правила называть *стандартным поведением* и *нестандартным поведением* (данные понятия не встречаются в литературе). Данные 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, создаваемы 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(const char*s, const T&x,const
      char*s2){cout<<s<<x<<s2<<endl;}
```

Упростим в предыдущем примере главную функцию до работы 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;
5  cout<<"v+v="<<v<<endl;
6 return 0;
7 }
```

Вставим соответствующий вывод при вызове конструкторов, деструктора и оператора присваивания:

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

а также при вызове оператора сложения:

```
1  Vector operator+(const Vector&b)const{out("Vector
       operator+(const Vector&b)");
2   if(n!=b.n){throw(SErr("n!=b.n"));}Vector r(n);
3   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
```

по умолчанию 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 *-fno-elide-constructors*:

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

мы получаем 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 получается для компилятора C++ Microsoft. Компилятор Microsoft везде далее у автора вызывается командой *cc*:

```
1 cc q.cpp
```

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

На учебных занятиях крайне рекомендуется использовать только стандартное поведение языка C++. Использование нестандартного поведения в некоторых случаях позволяет скрывать ошибки при написании программ, что довольно неприятно при обучении программированию.

3. Преобразование типов. Касты

Простое явное преобразование типов в языке C (при помощи написания требуемого типа в скобках перед преобразуемым выражением) заменено в языке C++ на различные типы преобразований, используемые в различных специфических ситуациях. Преобразование типов в C++ называется *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 выполнять конкретный код для осуществления преобразования между различными типами данных (целыми, вещественными и т.д.). Например, если требуется преобразовать вещественную переменную к целой, то 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. Например, в случае когда мы хотим использовать указатель на один тип как указатель на другой тип. Например, если мы хотим распечатать значения байт целой переменной, то можно взять ее адрес, преобразовать его к указателю на *unsigned char* и рассмотреть получившийся указатель как массив переменных типа *unsigned char*:

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

const_cast используется для изменения константности выражения (если так можно выразиться) и для изменения атрибута *volatile* (о чем пойдет речь ниже). Следует понимать, что в подавляющем числе случаев использование этого преобразование связано с непра-

вильным проектированием программы и может приводить к ошибкам. Приведем корректный пример данного преобразования:

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

А в следующем примере использование данного преобразование синтаксически правильно, но не имеет никакого логического смысла и приведет к непредсказуемым последствиям:

```
void f(const int &x)
{const_cast<int&>(x)=0;}
int main(void)
{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:

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

Отдельного обсуждения требует атрибут *volatile*, который может применяться перед определением/описанием переменной. Чисто формально это атрибут запрещает выполнять какую-либо оптимизацию работы с этой переменной. Приведем пример. Буем для определенности использовать 32-битный компилятор g++ с оптимизацией первого уровня. Т.е. компиляция программы будет происходить с помощью команды

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

Рассмотрим следующий простой пример:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<string.h>
using namespace std;
void f()
{int y=1; cout<<"f:"<<&y<<endl;}
int main(void)
{int x=1; cout<<&x<<endl;
 while(x)
 {
  f(); cout<<">"<<endl; exit(0);
 }
return 0;
}
```

На компьютере автора на экран выдается следующая информация:

```
0x102fe8c
f:0x102fe5c
>
```

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()*:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<string.h>
using namespace std;
void f()
{int y=1; cout<<"f:"<<&y<<endl;
```

```
 (&y)[12]=0;
}
int main(void)
{int x=1; cout<<&x<<endl;
 while(x)
 {
  f(); cout<<">"<<endl; //exit(0);
 }
return 0;
}
```

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, т.к. оно, с точки зрения компилятора, ни на что не влияет.

Однако, мы можем запретить компилятору оптимизировать работы с переменными x и y:

```
#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<string.h>
using namespace std;
void f()
{volatile int y=1; cout<<"f:"<<const_cast<int*>(&y)<<endl;
 (&y)[12]=0;
}
int main(void)
{volatile int x=1; cout<<const_cast<int*>(&x)<<endl;
 while(x)
 {
  f(); cout<<">"<<endl; //exit(0);
 }
return 0;
}
```

И тогда цикл выполнится всего один раз. В данной ситуации преобразование адреса переменных к типу *int** нужно из-за особенностей

компиляторов, согласно которым он отказывается выводить на экран адрес volatile-переменной (оператор вывода определен как пустой?), но согласен выводить адрес обычной переменной.

Для компилятора Microsoft, вызываемого тоже с первым уровнем оптимизации:

```
cc -O1 q.cpp
```

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

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

также заставляет изменить значение переменной x и цикл прекращает свою работу.

4. Сложные классы. Правило пяти. L- R-ссылки

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)приходится создавать временную переменную, куда с помощью отведения памяти помещается локальная переменная, 3)присваивание временной переменной *a* тоже реализуется с использованием отведения памяти.

Использование отведения памяти на втором и третьем шагах, по логике, совершенно не является необходимым, т.к. 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 с предыдущего шага). Для этой цели в языке C++ используются move-операции: move-конструкторы и move-присваивание. На этом этапе можно сформулировать основу 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/копирование 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 *move()* осуществляет содержательную часть действия. В нашем случае эта функция выглядит следующим образом:

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

Заметим, что move-операции у нас определены способом, независимым от решаемой задачи. Специфической является только функция *move()*.

Окончательный код выглядит следующим образом:

```
1  #include<iostream>
2  #include<fstream>
3  #include<string>
4  #include<stdlib.h>
5  #include<string.h>
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,const
       char*s2){cout<<s<<x<<s2<<endl;}
10 //-----------------------------------------
11 template<class T>class Vector
12 {T *v; size_t n,nmax;
13 public:
14  //---
15  Vector(size_t
        n=0){out("Vector(",n,")");SetZero();if(n){v=new
        T[nmax=this->n=n];for(size_t i=0;i<n;i++)v[i]=T();}}
16  Vector(const Vector&b){out("Vector(const
        Vector&b)");CopyOnly(b);}
17  ~Vector(){out("~Vector()");Clean();}
18  Vector &operator=(const Vector&b){out("Vector
        &operator=(const
        Vector&b)");if(&b!=this){Clean();CopyOnly(b);}return
        *this;}
19  Vector(Vector&&b){out("Vector(Vector&&b)");MoveOnly(b);}
20  Vector &operator=(Vector&&b){out("Vector
        &operator=(Vector&&b)"); if(&b!=this){Clean();
        MoveOnly(b);}return *this;}
21  void MoveOnly(Vector&b){if(b.nmax){v=b.v;n=b.n;nmax=b.nmax;
        b.SetZero();}else SetZero();}
22  //---
23  void SetZero(){v=nullptr;n=0;nmax=0;}
24  void Clean(){delete[] v; SetZero();}
```

```
25  void CopyOnly(const Vector&b){if(b.nmax){v=new
        T[nmax=b.nmax];n=b.n; 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*s="error"){this->s=s;}};
29  T &operator[](size_t i){if(i>=n)throw SErr("bad index");
        return v[i];}
30  const T &operator[](size_t i)const{return v[i];}
31  Vector operator+(const Vector&b)const{Vector r(n);
32   for(size_t i=0;i<n;i++){r[i]=v[i]+b.v[i];} return r;}
33  void push_back(const T&x){if(n==nmax){T*w=new T[2*(nmax+1)];
34   size_t i=0; for(;i<n;i++){w[i]=v[i];}
35   for(;i<2*(nmax+1);i++)w[i]=T();
36   nmax=2*(nmax+1); delete[] v; v=w;} v[n++]=x;}
37 };
38 //--
39 template<class T>ostream &operator<<(ostream &cout,
40                                      const Vector<T> &v)
41 {cout<<"("; for(size_t i=0;i<v.size();i++)
42  {cout<<v[i]<<" ";} cout<<")"; return cout;}
43 //--
44 int main(void)
45 {Vector<int> v;int m[5]={0,1,2,3,4};
46   for(int i=0;i<5;i++)v.push_back(m[i]);
47  v=v+v;
48  cout<<"v+v="<<v<<endl;
49 return 0;
50 }
```

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()
```

```
v+v=(0 2 4 6 8 )
~Vector()
```

Нестандартное поведение языка дает, соответственно, более короткую последовательность вызываемых функций:

```
Vector(0)
Vector(5)
Vector &operator=(Vector&&b)
~Vector()
v+v=(0 2 4 6 8 )
~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 C мы смогли бы написать функцию, вообще не отводящую память (и здесь 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 выполнения равно O(1)).

4.2. L-value- и R-value-ссылки

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

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

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 поведение 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 создается три временные переменные, которых хранятся промежуточные результаты трех операций сложения. Если бы мы реализовывали действие *a=b+c+d+e;* на языке C, то мы 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 R-value- и L-value-ссылок. Понятие R-value и L-value присутствовало уже в языке C, но там оно было тривиальным: под *L-value* (left-value) подразумевалось то, что может стоять слева от знака присваивания, а под *R-value* (right-value) — то, что может стоять только справа от знака присваивания. В языке C++ эти понятия несколько изменились. Чисто формально под *R-value* здесь подразумеваются либо литералы (например, 5 или 5.), либо временные переменные (про них мы много упоминали), либо неименованные объекты (объекты вида *T()*). Все остальное считается *L-value*. Например, в вышеопределенном операторе сложения аргумент является L-value (хотя он и имеет модификатор *const*) и сам объект класса также является L-value. Но при обработке выражения *a=b+c+d+e;* результат *b+c* помещается в R-value (во временную переменную), что дает возможность компилятору 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 R-value. Напомним, что операция *b+c* имеет дело с двумя L-value.

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

```
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 const_cast<Vector&&>(*this);}
```

Т.е. в качестве R-value-ссылки здесь выступает **this* (*этот* член класса).

Здесь надо отметить, что (согласно логике использования данного оператора) R-value-ссылка используется как раз для того, чтобы значение по данной ссылке можно было бы изменять (что вступает в противоречие с тем, что R-value, как раз, изменять нельзя!). Т.е. чисто формально R-value-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 L-value (хотя параметр и описан как R-value ссылка). Понятно, что компилятору приходится проявлять изрядную смекалку, чтобы разрешить это противоречие. Например, если в качестве параметра выступает R-value-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 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, кудо надо поместить результат, а второй — именно данной константой.

Последний оператор сложения будет возвращать R-value ссылку на *this (для ее дальнейшего использования), но поскольку *this уже успело превратиться в L-value, в операторе *return* необходим *const_ cast* для преобразования L-value в R-value.

Заметим, что теперь надо подправить старый оператор сложения, чтобы уточнить, что в нем левый аргумент сложения — L-value-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()
```

Мы видим, что мы получили всего одно отведение памяти в первом операторе сложения. Что, опять же, оптимально даже для языка C.

Рассмотрим далее для простоты только стандартное поведение языка C++ в более сложном случае сложения четырех слагаемых:

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

Здесь мы имеем попытку сложения двух R-value, что требует соответствующего оператора сложения. А в случае

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

нам необходимо складывать L-value и R-value. Таким образом нам надо дописать еще два соответствующих оператора сложения (первый — для сложения R-value и R-value, а второй — для сложения L-value и R-value):

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

```
3    return const_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 const_cast<Vector&&>(b);}
```

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

```
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 C 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;
5   cout<<"v+v="<<w<<endl;
6  }
```

Здесь переменная *w* является R-value-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 результата сложения 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 Vector operator+(const Vector&b)&
3 Vector(const Vector&b)
4 Vector operator+(const Vector&b)&&
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 несуществующей переменной (фактически ее место в стеке никто испортить не успел, поэтому мы получили вывод, соответствующий состоянию данной переменной сразу после ее гибели). Подобное поведение 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 не временной переменой переменной v, а значения, возвращаемого вторым оператором сложения, а значение, возвращаемое данным опера-

тором, — R-value ссылка. 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 продляется 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 пересобрать проект в процессе его выполнения (приостанавливаем программу, меняем исходный код, перекомпилирует и пересобираем 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. Но 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 пока не соответствует никаких машинных операций в коде. Остальные команды задают соответствующие машинные команды в исполняемом файле.

Команда *mov* копирует второй аргумент в первый, а команда *sub* — вычитает из первого аргумента второй, *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 появятся дополнительные локальные переменные).

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

Далее двумя командами в регистре *eax* образуется сумма значений созданных переменных, после чего значение данного регистра

записывается по адресу, взятому из регистра *ebp* 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 любая из используемых переменных была бы 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.

Все вышенаписанное фактически является способом реализации полиморфизма при работе с глобальными (статическими) переменными, функциями, шаблонами функций, переменными, созданными при помощи шаблонов классов. Далее разберемся более подробно с алгоритмами создания внутренних имен для отдельных типов объектов.

5.1. Внутренние имена переменных и функций

Разберемся с простым примером, включающем 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
```

На получившийся исполняемый файл *a.exe* напустим команду *nm*, выводящую на стандартный поток вывода (=по умолчанию на экран) список имен глобальных объектов. Стандартный поток вывода перенаправим в файл *nm.txt*, чтобы было проще искать требуемые нам имена глобальных объектов:

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

Поискав в получившемся тексте слово *My__*, мы узнаем, что есть всего два глобальных объекта с именами

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

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

Для компилятора Microsft можно при компиляции попросить параллельно компиляции создавать ассемблерный файл, в котором также будут присутствовать имена глобальных объектов, с помощью указания ключа */FAS*:

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

В файле *q.asm* мы найдем следующие указания н имена глобальных объектов:

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

Таким образом, компилятор Microsoft для языка C создает внутренние имена глобальных объектов, приписывая к исходным именам в

начало имени 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 *gcc* в списке глобальных объектов исполняемого файла мы найдем имена

```
0000000000401520 T _Z7My__Funv
0000000000407030 B My__Struct
```

Т.е. внутренние имена статических переменных образуются тем же способом, что и в языке C (понятие *полиморфизма* для переменных в языке C++ не имеет смысла), 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* найдем имена

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

Здесь все оказывается интереснее: к имени глобальной переменной приписывается имя типа (!). Т.е. 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 C++, нет, но это может служить причиной весьма неожиданных ошибок). Далее мы покажем, что есть возможность иметь разные функции с одинаковыми именами и типами параметров, но различными типами возвращаемых значений (и здесь надо знать о такой возможности, скорее, для понимания возможных ошибок; язык C++, сам по себе, такого не допускает, и если компилятор обнаружит такую экзотику в одном исходном файле, то выдаст ошибку). Рассмотрим пример. Программа состоит из двух исходных файлов. Первый файл *q.cpp*:

```
#include <stdio.h>
int MyVar=0;
void fun1(void);
int fun2(void)
{printf("%d\n",MyVar);
 return 0;
```

```
}
int main(void)
{
 fun1();
 fun2();
return 0;
}
```

Второй файл *q2.cpp*:

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

После их компиляции компилятором Microsoft и сборки командой *cc q.cpp q2.cpp /FAS* найдем в файле *q.asm* имена глобальных объектов:

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

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

```
PUBLIC  ?fun1@@YAXXZ                    ; fun1
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, возвращаемой функцией в данном случае, также включается во внутреннее имя объекта.

Программа успешно скомпилируется, соберется и отработает с выводом на экран

```
1
0
```

Компилятор *gcc* запрещает такие вольности. Тип возвращаемого из функции значения и тип глобальной переменной не добавляются к именам соответствующих объектов, поэтому при попытке собрать данную программу линковщик *gcc* выдаст ошибки по повторяющимся именам глобальных объектов и для переменной, и для функции.

5.2. Шаблоны. Определение функций шаблона в отдельном cpp-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  {
7   cout<<Min(1,2)<<" "<<Max(1,2)<<endl;
8   cout<<Min<int>(1,2)<<" "<<Max<int>(1,2)<<endl;
9  return 0;
10 }
```

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 данные 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 ничего, кроме как смириться с этим, для него не останется. Он будет надеяться, что программист определил шаблон в едином include-файле, поэтому определения данных функций будут везде аналогичны. А на самом 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 является как тип переменной, так 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 запутано. Внутренние имена соответствующей 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 своей сложной системе счисления, которую чисто визуально сложно понять.

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

```
#include<iostream>
using namespace std;
#include "q"
int main(void)
{S<int,10> x; x.fun();
return 0;
}
```

include-файл с именем *q* имеет вид:

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

Программа из одного такого cpp-файла будет замечательно компилироваться, запускаться и выдавать при этом на экран число *10*.

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

Самым простым способом решения данной проблемы является вынесение шаблона функции из шаблона структуры в тот же самый include-файл:

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

Здесь мы не определили никакой функции (только шаблон!), но в файле *q.cpp* стоит вызов данной функции для конкретного экземпляра класса *x*, что заставляет компилятор определить функцию

fun для параметров шаблона *int* и *10*. Данный подход будет успешно работать.

Следующим шагом является попытка полного вынесения определения функции *fun()* из include-файла. Данный подход имеет смысл, когда в include-файле содержится описание многих классов и лишний текст сильно усложняет обзор возможностей этих классов. К сожалению, вынесение определения шаблона функции в отдельный cpp-файл возможно, но обладает определенными неудобствами. Действительно, мы можем определение шаблона функции класса легко вынести в cpp-файл (пусть он имеет имя *q2.cpp*) в том же виде, в котором оно содержится в include-файле. Однако, в таком 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, т.к. шаблон сам по себе не инициирует 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.

В качестве альтернативы последней инструкции 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 требуемой функции в программе, 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)
7  {S<int,10> tmp;
8   tmp.fun();
9  return 0;
10 }
11 static int i_tmp=Def_S_Fun();
```

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

Безусловно, данный подход может приводить к ошибкам. В частности, если определять функции с одинаковыми внутренними именами в разных файлах по-разному, то мы получим различное поведение программы в зависимости от порядка объектных файлов, указываемых 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 два следующих варианта компиляции и сборки программы 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 подходя является существенное увеличение размера объектных файлов и времени компиляции за счет того, что определения всех функций, задаваемых шаблоном, могут появиться во всех объектных файлах программы (если эти функции там вызываются).

6. Умные указатели. Функциональные объекты

6.1. 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 вышеупомянутое локальное действие принято оформлять в виде функции, указатель на которую передается в общую функцию. Стандартным примером данной ситуации является задача сортировки. Сортировку можно выполнять некоторым стандартным способом, а указатель на функцию сравнения сортируемых элементов надо передавать в функцию сортировки. В языке C существует стандартная функция быстрой сортировки *qsort()* (можно упомянуть, что в современном C данная функция имеет весьма эффективную реализацию, что уже упоминалось в первой части конспектов лекций). 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},n=(int)(sizeof(m)/sizeof(*m));
11  qsort(m,n,sizeof(*m),cmp);
12  for(int i=0;i<n;i++){printf("%d ",m[i]);}printf("\n");
13 return 0;
14 }
```

Несмотря на эффективность данного подхода, у него есть два недостатка: во-первых, налицо присутствуют накладные расходы на вызов функции сравнения и при простой функции сравнения они сопоставимы по времени выполнения с, собственно, операцией сравнения. Во-вторых, вместе с указателем на функцию невозможно пере-

дать 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 глобальную переменную, а это является весьма дурным тоном и неприменимо в ряде ситуаций (в частности, в случае, когда сортировки одновременно выполняются в параллельных нитях). Решением данной проблемы (с параллельным устранением вышеперечисленных недостатков) в языке C++ является использование *функциональных объектов*. Напишем собственный шаблон простейшей функции сортировки (сортировка пузырьком):

```
1 #include<iostream>
2 #include<algorithm>
3 #include<stdlib.h>
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* требуется наличие всего одной возможности: способность приписывать справа от этого объекта круглые скобки с передачей в них двух сравниваемых объетов типа T. *lt* должен возвращать *истину* если первый объект меньше второго (в требуемом смысле), иначе — ложь. 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=(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},n=(int)(sizeof(m)/sizeof(*m));
2   sort(m,n,cmp<int>);
3   for(int i=0;i<n;i++){cout<<m[i]<<" ";}cout<<endl;
4   }
```

По сути данный подход ничем не отличается от подхода языка C. 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 C++, к которой можно приписывать круглые скобки. Здесь также присутствуют объекты с определенным оператором *()*, которые также можно использовать для этой цели. Наконец мы непосредственно дошли до понятие *функционального объекта* — объекта созданного пользователем типа, в котором определен оператор *круглые скобки*. В нашем случае в эти скобки надо передавать сравниваемые величины и 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 online-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 это сделали бы. Теперь мы можем использовать все тот же шаблон функции сортировки, передавая ему

в качестве способа сравнения функциональный объект вместо указателя на функцию сортировки:

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

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

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

Безусловно, в библиотеке алгоритмов, описанной в стандартном include-файле *algorithm* уже присутствует стандартный функциональный объект *less*, который можно использовать в тех же целях, самостоятельно не определяя его:

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

Данный код легко привести к виду, неотличимому по написанию от передачи указателя на функцию, следующим образом:

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

Легко понять, что смысл и реализация данного кода принципиально отличаются от кода с передачей указателя на функцию, написанного выше.

Забегая на полгода занятий вперед, следует упомянуть, что в языке C++ подход с использованием функциональных объектов является не единственным подходом к решению подобных задач. Более мощным механизмом, который здесь можно применять является

использование техники *наследования*. Данный подход, безусловно, является более элегантным (и безусловно приветствуемым на собеседованиях), но, по сути, не имеет никаких достоинств по сравнению с описанным подходом.

6.2. Умные указатели. *unique_ptr*

Для простоты изложения (и проверки работоспособности подхода) далее мы будем использовать вышеописанную реализацию шаблона *Vector*, помещенную в include-файл *Vector*. Для вывода требуемой диагностической информации о вызове конструкторов и деструкторов написанные далее cpp-файлы надо компилировать с ключом *-D LOG*, например:

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

Разберем следующий тривиальный пример:

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

По сути, в данном примере видна основная проблема языка C (пусть никого здесь не вводит в заблуждение использование семантики языка C++): при использовании указателей на вручную созданные ресурсы необходимо не забыть эти ресурсы освободить. При соблюдении блочной логики программы это не сложно: можно (как в показанном примере) отвести память (захватить ресурс), потом сразу ее очистить (освободить ресурс), а потом уже вписывать все действия с данным ресурсом между парой написанных конструкций. 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, что наследуется языком C++. Ограничимся, однако, пока только ситуацией, когда захваты-

ваемый 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 именования данной сущности и нам придется называться такие объекты просто как *unique_ptr* (не забывая, что, на самом деле, *unique_ptr* — это не тип класса, а шаблон для типа класса).

unique_ptr является весьма простой конструкцией. Он (умный указатель) является классом, в котором хранится обычный указатель. Конструктор класса задает значение данного обычного указателя. Деструктор класса применяет к данному 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.

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* указанного типа доступны операторы * и ->, эмулирующие работу с обычным указателем. Однако, если имеется потребность извлечения из умного указателя обычного указателя, то можно использовать 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-присваивания и move-копирования) и при их помощи реализован метод данного класса *swap*, позволяющий обменять местами содержимое двух умных указателей. Для тех же целей можно использовать алгоритм swap.

Итак, корректна следующая конструкция:

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

Правила хорошего тона не рекомендуют создавать *unique_ptr* вышеописанным способом. Вместо этого рекомендуется 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 отдельно отводится под, собственно, умный указатель, и отдельно — под объект, которым будет владеть данный умный указатель). Синтаксис такого действия следующий:

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

unique_ptr могут использоваться и для более сложных целей. В первой части лекций мы создавали однонаправленный список на основе реализации с фиктивным элементом. Шаблон для такого класса весьма прост:

```
#pragma once
template<class T> struct SList1
{struct SNode
 {T v; SNode *next; SNode(){next=nullptr;}
  SNode(const T&v){this->v=v; next=nullptr;}};
 SNode b,*cur; int n;
 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;}
       return false;}
14 bool Get(T&v){if(cur==&b){return false;}
15                 v=cur->v;return true;}
16 void insert(const T&v){SNode *n=new SNode(v);
       n->next=cur->next;cur->next=n; this->n++;}
17 bool find(const T&v){T x; for(GoToBegin();GoToNext();)
18  {Get(x);if(x==v)return true;} return false;}
19 bool DelNext()
20 {if(cur->next==nullptr)return false; SNode *n=cur->next;
21  cur->next=cur->next->next;delete n;this->n--;return true;}
22 };
```

Простейший пример использования такого списка строк типа *string* имеет вид:

```
1  #include<iostream>
2  #include<fstream>
3  #include<stdlib.h>
4  #include<string.h>
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, терминированному нулем) не существует. Вместо этого у класса *string* присутствует метод *c_str()*, возвращающий указатель на строку в понимании языка C.

Изменим созданный выше шаблон класса *SList1* так, чтобы в каждом элементе списка хранился бы 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. Т.е. теперь не надо определять деструктор для всего списка. Измененный код будет иметь вид:

```
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()){cur=cur->next.get();
        return true;} return false;}
16  bool Get(T&v){if(cur==&b){return false;}
17                v=cur->v; return true;}
```

```
18  void insert(const T&v){SNode *n=new SNode(v);
        n->next.swap(cur->next);cur->next.reset(n); 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.get()==nullptr)return false; SNode
        *n=cur->next.get(); cur->next.swap(cur->next->next);
        n->next.reset(); this->n--; return true;}
23  };
```

Отметим, что функцию *clear()* можно реализовать аналогично описанному выше способу, просто применив метод *reset()* умного указателя внутри элемента списка *b*. Этот метод очищает память, обычный указатель на которую хранится в указателе, заключенном в данном *unique_ptr*, и замещает данный обычный указатель значением из параметра метода. Если параметра нет, то он по умолчанию считается равным *null_ptr*. Также надо несколько изменить логику методов *insert()* и *DelNext()*. Один из вариантов реализации данных методов приведен выше.

Приведенный метод реализации однонаправленного списка является классическим и обязательным для воспроизведения на много численных собеседованиях. Аналогичная реализация однонаправленного списка приведена на сайте Microsoft. Можно, например, добавить в вышеприведенный 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 данной функции приводит к падению программы:

```
1  #include<iostream>
2  #include<string.h>
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<<(l.empty()?"empty":"not empty")<<endl;
10  }cout<<"done";
11 return 0;
```

```
12 }
```

Текст *done* не будет выведен на экран ни при выполнении программы, скомпилированной обычным способом компилятором *gcc*, ни при выполнении программы, скомпилированной компилятором Microsoft.

Причина неприятностей непосредственно следует из четкого понимания процесса очистки памяти, отведенной в списке, при вызове деструктора. Действительно, легко понять, что 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 уничтожать его надо оператором *delete*. Но в языках C/C++ указатели используются для двух целей: для работы с указателем на объект и для работы с массивом. Создание/очистка массивов в языке C++ производятся другими операторами: *new[]/delete[]*. В раде случаев это не важно для массивов переменных элементарных типов (не всегда!), но попытка освобождения памяти из-под массива, созданного с помощью оператора *new[]*, оператором *delete* всегда приводит к падению программы. Т.е. при владении объектом *unique_ptr* указателя на массив требуется передача в объект информации о способе очситки 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*. Итак, в нашем случае для хранения в умном указателе указателя на массив требуется определить шаблон 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>>());
```

6.3. Умные указатели. *shared_ptr*

shared_ptr является существенно более сложной реализацией умного указателя по сравнению с *unique_ptr*. *shared_ptr* допускает владение несколькими *shared_ptr* обычными указателями на один и тот же объект. Внутри *shared_ptr* хранится счетчик на количество таких 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 структуру, в которой, как раз, и хранятся счетчик и указатель на подвластный объект. Тогда все *shared_ptr*, владеющие указателем на один и тот же объект, будут иметь доступ к одной и то же структуре с описанным содержимым. Безусловно, все эти умные указатели должны создаваться путем присваивания или копирования от одного умного указателя. Т.е. если мы ошибочно создадим два независимых умных указателя, владеющих один и тем же обычным указателем, то ничего хорошего из этого получиться не может (это верно и для *unique_ptr*, и для *shared_ptr*). Поэтому умные указатели очень хорошо помогают для обеспечения очистки памяти из-под созданных объектов, но, все же, 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 }
```

Здесь внутри функции *main()* нет явного отведения памяти под вектор (отведение происходит внутри функции) 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. Т.е. информацию о том, как надо убивать созданный объект, желательно закладывать в программу при создании объекта, когда непосредственно 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_shared<>()* в случае, если отводится память под массив объектов, можно использовать функциональный объект, который указывает, как убивать 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  {
7   void operator()(T*p){delete[] p;}
8  };
9  shared_ptr<Vector<int>> fun()
10 {shared_ptr<Vector<int>> p(new
       Vector<int>[5],Del<Vector<int>>());
11 return p;
12 }
13 int main(void)
14 {shared_ptr<Vector<int>> v=fun();
```

```
15  Vector<int> *w=v.get();
16  //...
17 return 0;
18 }
```

Опять же, 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]);
7  return p;
8  }
9  int main(void)
10 {shared_ptr<Vector<int>[]> v=fun();
11  Vector<int> *w=v.get();
12  v[1].push_back(1); v[1].push_back(2);
13  cout<<v[1]<<endl;
14 return 0;
15 }
```

7. Структуры данных. Вектор. STL

В предыдущей книге [12] 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. При реализации структуры данных (а она может быть различной!) принято каждому предписанию сопоставлять функцию, реализующую соответствующее предписание. При этом для любой структуры данных обычно существует четыре базовых предписания, общих для любой структуры данных:

1. Создать.

2. Уничтожить.

3. Очистить.

4. Пуста ли?

Для структуры данных *вектор* первое предписание логично модифицируетсяв в

1. Создать вектор заданной длины.

Для библиотеки STL, о которой будем говорить позже, стало стандратном включать во все реализации структур данных (контейнеры) предписания

5. Получить количество элементов множества.

Кроме этого 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 в заданную позицию вектора.

10. Уничтожить элемент на заданной позиции вектора.
и т.д.

Разберем далее 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 перменными (целыми или вещественными), либо полноценными классами (удовлетворяющими правилу или трех, или пяти, или нуля), либо простыми классами, то при реализации данного объекта в соответсвующем классе не обязательно определять конструкторы, деструкторы, 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 необходмые 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 бы выполнять все перечисленные операции. Т.е. в обычной жизни желательно создавать классы исключительно на основе правила нуля, но, тем не менее, понимать и создавать классы на основе правила пяти тоже нужно уметь, т.к. в основе всего лежат именно они. Например, спустя какое-то время мы будем разбирать стандартную реализацию вектра в языке C++. Эта реализации покроет существенную часть наших потребностей при работе с векторами. Однако для стандартных векторов, например, нет операции сложения (что весьма нужно в работе), поэтому в ряде случаев приходится писать свою реализацию этой структуры данных. Т.е. 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<stdlib.h>
6  #include<string.h>
7  using namespace std;
8  //--
9  #ifndef LOG
10 #define LOG 0
11 #else
12 #undef LOG
13 #define LOG 1
14 #endif
15 inline void out(const char*s){if(LOG)cout<<s<<endl;}
16 template<class T>void out(const char*s,const T&x,const
       char*s2){if(LOG)cout<<s<<x<<s2<<endl;}
17 //--
18 template<class T> class Vector
19 {T *v; size_t n,nmax;
20 public:
21  //--
22  Vector(size_t n=0){out("Vector(",n,")");if(n){v=new
       T[nmax=this->n=n];for(size_t i=0;i<n;i++)v[i]=T();}
       else SetZero();}
23  Vector(const Vector&b){out("Vector(const
       Vector&b)");CopyOnly(b);}
24  ~Vector(){out("~Vector(",n,")");Clean();}
25  Vector &operator=(const Vector&b){out("Vector
       &operator=(const Vector&b)");
26           if(&b!=this){Clean();CopyOnly(b);}return *this;}
27  Vector(Vector&&b){out("Vector(Vector&&b)");MoveOnly(b);}
28  Vector &operator=(Vector&&b){out("Vector
       &operator=(Vector&&b)");
29           if(&b!=this){Clean();MoveOnly(b);}return *this;}
30  //--
```

```
31 void SetZero(){v=nullptr; n=0; nmax=0;}
32 void Clean(){delete[] v; SetZero();}
33 void CopyOnly(const Vector& b){if(b.nmax){v=new
       T[nmax=b.nmax];
34  n=b.n; for(size_t i=0;i<nmax;i++)v[i]=b.v[i];}else
        SetZero();}
35 void MoveOnly(Vector&b){v=b.v;n=b.n;nmax=b.nmax;
       b.SetZero();}
36 //--
37 struct SErr{string s; SErr(const
       char*s="error"){this->s=s;}};
38 T &operator[](size_t i){if(i>=n)throw SErr("bad index");
       return v[i];}
39 const T &operator[](size_t i)const{return v[i];}
40 void push_back(const T&x){if(n==nmax){size_t
       nmax2=2*nmax+1; T*w=new T[nmax2];
41  size_t i=0; for(;i<n;i++){w[i]=v[i];}
        for(;i<nmax2;i++){w[i]=T();} delete[]v;
        v=w;nmax=nmax2;} v[n++]=x;}
42 Vector operator+(const Vector&b)const&{out("Vector
       operator+(const Vector&b)&");
43       if(n!=b.n){throw SErr("n!=b.n");}Vector r(*this);
44       for(size_t i=0;i<n;i++){r[i]=r[i]+b[i];} return r;}
45 Vector &&operator+(const Vector&b)&&{out("Vector
       operator+(const Vector&b)&&");
46       if(n!=b.n){throw SErr("n!=b.n");}
47       for(size_t i=0;i<n;i++){v[i]=v[i]+b[i];} return
            const_cast<Vector&&>(*this);}
48 Vector &&operator+(Vector&&b)const&{out("Vector
       operator+(Vector&&b)&");
49       if(n!=b.n){throw SErr("n!=b.n");}
50       for(size_t i=0;i<n;i++){b.v[i]=v[i]+b[i];} return
            const_cast<Vector&&>(b);}
51 Vector &&operator+(Vector&&b)&&{out("Vector
       operator+(Vector&&b)&&");
52       if(n!=b.n){throw SErr("n!=b.n");}
53       for(size_t i=0;i<n;i++){v[i]=v[i]+b[i];} return
            const_cast<Vector&&>(*this);}
54 size_t size()const{return n;}
```

```
55  T &back(){return v[n-1];}
56  template<class U> friend ostream &operator<<(ostream
        &cout,const Vector<U> &v);
57  };
58  //--
59  template<class V>
60  ostream &operator<<(ostream &cout,const Vector<V> &v)
61  {cout<<"("; for(size_t i=0;i<v.n;i++){cout<<v[i]<<" ";}
        cout<<")"; return cout;}
62  //--
```

Простейший тест для данной реализации имеет вид:

```
1  #include<iostream>
2  #include<fstream>
3  #include<stdlib.h>
4  #include<string.h>
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;
11  cout<<"v+v="<<w<<endl;
12 return 0;
13 }
```

Попробуем изменить данную реализацию так, чтобы вместо обычного указателя в ней хранился бы 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 нашей целью будет отбрасывание всех конструкторов и операторов присваивания нового класса *Vector* (возможно кроме одного, создающего вектор заданной длины), т.е. мы хотим получить реализацию вектора на основе правила нуля.

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 один из указателей, хранящийся внутри одного из таких *shared_ptr*, то указатели внутри других *shared_ptr* останутся неизменными. Чтобы изменение указателя внутри одного *shared_ptr* приводило бы 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 *shared_ptr*, хранящие указатели на один и тот же указатель на данные.

Предложенный подход 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 гибели данных, на который 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, т.к. после указанных действий создается полноценный умный указатель *shared_ptr*, указывающий на созданный умный указатель *unique_ptr*, хранящий внутри себя указатель, равный *null_ptr*. Тем не менее, полезно создать конструктор, создающий вектор заданной длины:

```
1  Vector(size_t n=0){out("Vector(",n,")");if(n)
```

```
2  {(*v).reset(new T[nmax=this->n=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; T*w=new T[nmax2];
2   size_t i=0; for(;i<n;i++){w[i]=v[i];}
       for(;i<nmax2;i++){w[i]=T();}
       (*v).reset(w);nmax=nmax2;} (*v)[n++]=x;}
```

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

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

Для получившегося шаблона класса *Vector* (с соответствующими заменами *v* на *(*v)*) вышенаписанный тест будет замечательно работать.

Однако, полученная реализация вектора, на самом деле, будет неработоспособной. Проблема заключается в том, что, несмотря на то, что присваивание одного вектора другому создает корректную ссылку на один и тот же массив данных, тем не менее, элементы n и $nmax$ этих двух векторов будут ссылаться на различные переменные. Поэтому если мы к одной из двух таких копий вектора 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 одни и те же 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>
3    n=make_shared<size_t>(0),nmax=make_shared<size_t>(0);
```

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

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

```
18 //--
19 template<class T> class Vector
20 {
21  shared_ptr<unique_ptr<T[]>>v=make_shared<unique_ptr<T[]>>();
22  shared_ptr<size_t>
23    n=make_shared<size_t>(0),nmax=make_shared<size_t>(0);
24 public:
25  //--
26  Vector(size_t
        n=0){out("Vector(",n,")");if(n){(*v).reset(new
        T[*nmax=*(this->n)=n]);for(size_t
        i=0;i<n;i++)(*v)[i]=T();}}
27  Vector copy()const{Vector r(*n); for(size_t
        i=0;i<*n;i++){r[i]=(*v)[i];} return r;}
28  //--
29  struct SErr{string s; SErr(const
        char*s="error"){this->s=s;}};
30  T &operator[](size_t i){if(i>=*n)throw SErr("bad index");
        return (*v)[i];}
31  const T &operator[](size_t i)const{return (*v)[i];}
32  void push_back(const T&x){if(*n==*nmax){size_t
        nmax2=2*(*nmax)+1; T*w=new T[nmax2];
33   size_t i=0; for(;i<*n;i++){w[i]=(*v)[i];}
         for(;i<nmax2;i++){w[i]=T();}
         (*v).reset(w);*nmax=nmax2;} (*v)[(*n)++]=x;}
34  Vector operator+(const Vector&b)const&{out("Vector
        operator+(const Vector&b)&");
35        if(*n!=*(b.n)){throw SErr("n!=b.n");}Vector r;
36        for(size_t i=0;i<*n;i++){r.push_back((*v)[i]+b[i]);}
              return r;}
37  Vector &&operator+(const Vector&b)&&{out("Vector
        operator+(const Vector&b)&&");
38        if(*n!=*(b.n)){throw SErr("n!=b.n");}
39        for(size_t i=0;i<*n;i++){(*v)[i]=(*v)[i]+b[i];} return
              const_cast<Vector&&>(*this);}
40  Vector &&operator+(Vector&&b)const&{out("Vector
        operator+(Vector&&b)&");
41        if(*n!=*(b.n)){throw SErr("n!=b.n");}
```

```
42      for(size_t i=0;i<*n;i++){(*(b.v))[i]=(*v)[i]+b[i];}
            return const_cast<Vector&&>(b);}
43  Vector &&operator+(Vector&&b)&&{out("Vector
        operator+(Vector&&b)&&");
44      if(*n!=*(b.n)){throw SErr("n!=b.n");}
45      for(size_t i=0;i<*n;i++){(*v)[i]=(*v)[i]+b[i];} return
            const_cast<Vector&&>(*this);}
46  size_t size()const{return *n;}
47  T &back(){return (*v)[(*n)-1];}
48  template<class U> friend ostream &operator<<(ostream
        &cout,const Vector<U> &v);
49 };
50 //--
51 template<class V>
52 ostream &operator<<(ostream &cout,const Vector<V> &v)
53 {cout<<"("; for(size_t i=0;i<*(v.n);i++){cout<<v[i]<<" ";}
       cout<<")"; return cout;}
54 //--
```

Следующий тест показывает возможность созданного вектора передаваться по значению в функцию, возвращаться по значению из функции и копироваться так, как это все происходило бы по ссылке. 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;
13  cout<<"v="<<v<<endl;
14  Vector<double> w=v+v;
15  cout<<"v+v="<<w<<endl;
16  v=w;
```

```
17  cout<<"v="<<v<<endl;
18  q=v;
19  fun2(q);
20  cout<<"q="<<q<<endl;
21  cout<<"v="<<v<<endl;
22 return 0;
23 }
```

Следует иметь в виду, что при максимальном уровне замечаний существующему на данный момент компилятору *gcc* не удается скомпилировать приведенный вариант программы. Ему не удается 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 реализации вектора (на основе правила пяти). Для такой реализации существует стандартная задача языка C++: *требуется реализовать вектор (формально) неограниченной длины*. Здесь вместо реализации метода *push_ back()* или задания размера вектора в конструктуре, 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, пердаваемом в качестве параметра функции), то гарантируется неизменение размера вектора (отсутствие переотведения памяти). Т.е. если мы пытаемся **получить значение** элемента вектора, расположенного в рамках отведенной памяти, то используется соответствующий элемент вектора. Если же 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 здесь при необходимости память должна переотводиться. Т.е. при присваивании значения элемнту вектора по индексу, расположенному в рамках отведенной памяти, должно осуществляться обычное присваивание. Если же индекс выходит за границы отведенной памяти, то перед присваиванием должно проис-

ходить автоматическое переотведение памяти под массив объектов вектора. В сухом остатке мы хотим, получить возможность выполнения различных действий при получении знаения элемента вектора и при присваивании элементу вектора некоторого значения.

Практика показывает, что попытки манипуляций с наличием/остутствием модификатора *const* в соответствующем операторе *квадратные скобки* здесь, к сожалению, не помогают. Решение данной задачи кроется в создании 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 двух операторов: преобразованя типа нового типа к типу T и присвавания новому типу элемента типа T. Первый из этих операторов будет применяться 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 сообразительность).

Итак, создадим внутри шаблона класса *Vector* новый тип

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

Здесь *CMP* — 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,

соответствующий оператор преобразования типа и оператор присвивания.

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

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

Здесь оператор присваивания 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.

Здесь надо отметить, 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 классов данного типа), мы просто описываем глобальную переменную, область видимости которой закрыта 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 cpp-файлов (ровно в одном!) должно было бы присутствовать ее оперделение в виде

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

Если бы это определение присутствовало бы более, чем в одном файле, то при сборки 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 файов 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 это наблюдали на примере определения функций. То же самое относится и к определению переменных. Для определения статической переменной шаблона достаточно определить эту перменную 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<fstream>
4  #include<string>
5  #include<stdlib.h>
6  #include<string.h>
7  using namespace std;
8  //--
9  #ifndef LOG
10 #define LOG 0
11 #else
12 #undef LOG
13 #define LOG 1
14 #endif
15 inline void out(const char*s){if(LOG)cout<<s<<endl;}
16 template<class T>void out(const char*s,const T&x,const
       char*s2){if(LOG)cout<<s<<x<<s2<<endl;}
17 //--
18 template<class T> class Vector
19 {T *v; size_t n,nmax; static T TMP;
```

```
20 public:
21  //--
22  struct SV
23  {
24   SV(Vector *v,size_t i){this->v=v;this->i=i;}
25   Vector *v; size_t i;
26   operator T()const{if(i>=v->n)return Vector::TMP; return
         v->v[i];}
27   Vector &operator=(const T&x){if(i>=v->n)v->resize(i+1);
         v->v[i]=x;return *v;}
28  };
29  struct SVC
30  {
31   SVC(const Vector *v,size_t i){this->v=v;this->i=i;}
32   const Vector *v; size_t i;
33   operator T()const{if(i>=v->n)return Vector::TMP; return
         v->v[i];}
34  };
35  //--
36  Vector(size_t n=0){out("Vector(",n,")");if(n){v=new
       T[nmax=this->n=n];for(size_t i=0;i<n;i++)v[i]=T();}
       else SetZero();}
37  Vector(const Vector&b){out("Vector(const
       Vector&b)");CopyOnly(b);}
38  ~Vector(){out("~Vector(",n,")");Clean();}
39  Vector &operator=(const Vector&b){out("Vector
       &operator=(const Vector&b)");
40            if(&b!=this){Clean();CopyOnly(b);}return *this;}
41  Vector(Vector&&b){out("Vector(Vector&&b)");MoveOnly(b);}
42  Vector &operator=(Vector&&b){out("Vector
       &operator=(Vector&&b)");
43            if(&b!=this){Clean();MoveOnly(b);}return *this;}
44  //--
45  void SetZero(){v=nullptr; n=0; nmax=0;}
46  void Clean(){delete[] v; SetZero();}
47  void CopyOnly(const Vector& b){if(b.nmax){v=new
       T[nmax=b.nmax];
48   n=b.n; for(size_t i=0;i<nmax;i++)v[i]=b.v[i];}else
       SetZero();}
```

```
49  void MoveOnly(Vector&b){v=b.v;n=b.n;nmax=b.nmax;
        b.SetZero();}
50  //--
51  struct SErr{string s; SErr(const
        char*s="error"){this->s=s;}};
52  SV operator[](size_t i){return SV(this,i);}
53  SVC operator[](size_t i)const{return SVC(this,i);}
54  void push_back(const T&x){if(n==nmax){size_t
        nmax2=2*nmax+1; T*w=new T[nmax2];
55   size_t i=0; for(;i<n;i++){w[i]=v[i];}
         for(;i<nmax2;i++){w[i]=T();} delete[]v;
         v=w;nmax=nmax2;} v[n++]=x;}
56  void resize(size_t m){if(m>nmax){size_t nmax2=2*m+1;
       T*w=new T[nmax2];
57   size_t i=0; for(;i<n;i++){w[i]=v[i];}
         for(;i<nmax2;i++){w[i]=T();} delete[]v;
         v=w;nmax=nmax2;} n=m;}
58  Vector operator+(const Vector&b)const&{out("Vector
        operator+(const Vector&b)&");
59        if(n!=b.n){throw SErr("n!=b.n");}Vector r(*this);
60        for(size_t i=0;i<n;i++){r[i]=r[i]+b[i];} return r;}
61  Vector &&operator+(const Vector&b)&&{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];} return
            const_cast<Vector&&>(*this);}
64  Vector &&operator+(Vector&&b)const&{out("Vector
        operator+(Vector&&b)&");
65        if(n!=b.n){throw SErr("n!=b.n");}
66        for(size_t i=0;i<n;i++){b.v[i]=v[i]+b[i];} return
            const_cast<Vector&&>(b);}
67  Vector &&operator+(Vector&&b)&&{out("Vector
        operator+(Vector&&b)&&");
68        if(n!=b.n){throw SErr("n!=b.n");}
69        for(size_t i=0;i<n;i++){v[i]=v[i]+b[i];} return
            const_cast<Vector&&>(*this);}
70  size_t size()const{return n;}
71  T &back(){return v[n-1];}
```

```
72  template<class U> friend ostream &operator<<(ostream
        &cout,const Vector<U> &v);
73  };
74  //--
75  template<class V>
76  ostream &operator<<(ostream &cout,const Vector<V> &v)
77  {cout<<"("; for(size_t i=0;i<v.n;i++){cout<<v[i]<<" ";}
        cout<<")"; return cout;}
78  //--
79  template<class T> T Vector<T>::TMP;
```

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

```
1   #include<iostream>
2   #include<memory>
3   using namespace std;
4   #include"Vector"
5   //--------
6   int main(void)
7   {Vector<double> v;  int m[]={0,1,2,3,4,5,6,7,8,9};
8    for(int i=0;i<10;i++)v[i]=(float)m[i];
9    cout<<"v="<<v<<endl;
10   for(int i=0;i<10;i++){float x=v[i]; cout<<x<<"
         ";}cout<<endl;
11   v=v+v;
12   cout<<"v+v="<<v<<endl;
13   for(size_t i=0;i<v.size();i++)v[i]=v[i]+v[i];
14   cout<<"v+v+v+v="<<v<<endl;
15  return 0;
16  }
```

Здесь следует отметить отменную сообразительность компилятора при выполнении совершенно нетривиальных процедур: *v[i]=(float)m[i]* и *v[i]=v[i]+v[i];* .

7.3. C++. Итераторы

Как уже упоминалось в начале главы, для каждой структуры дынных желательно иметь некоторый стандартизированный меха-

низм перебора элементов текущего множества. Данный 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. Эта возможность обладает существенным недостатком: 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*. С одной стороны, итераторы представляют собой просто некоторый унифицированный синтаксис определения необходимых функций и перемнных для задания возможности перебора элементов множества. Т.е. изначально данной понятие не было завязано на, собственно, синтаксис языка C++, а представляло собой некоторый набор договоренностей. Но развитие языка привело к внедрению данного понятия непосредственно в синтаксис языка, о чем пойдет речь позднее.

Договоренность об использовании обычных итераторов заключается в синтаксисе простого цикла для перебора элементов множества. Например, для нашей реализации вектора должна быть предусмотрена следующая конструкция перебора элементов вектора (конкретно в приведенном примере: вывод 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 используемого синтаксиса работы с итератором видно, что для должны быть определены, как минимум, операторы ‘*’, ‘++’, ‘!=’, а внутри обслуживаемого класса должны быть определены функ-

ции, возвращающие соответствующие итераторы, *begin()* и *end()*. Для трех остальных типов итераторов соответствующие функции должны иметь имена *rbegin() / rend()*, *cbegin() / cend()*, *crbegin() / crend()*.

Когда все вышесказанное задано, для нашего класса *Vector* остает весьма небольшая работа по созданию кода, обслуживающего обычный итератор (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;}
6   iterator operator++(int tmp){tmp=tmp;
7    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, соответственно. 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 инкрементирвоания возвращает значение инкрементируемой переменной, которой было до, собственно, выполнения операции инкрементирвоания. Поэтому для возвращаемого значения приходится создавать временную переменную, сохраняющую исходное значение переменной. В префиксном операторе можно просто возвращать **this*. Эту тонкость всегда надо иметь в виду для грамотного использования описываемой операции.

Соответственно, код, обеспечивающий работу трех оставшихся итераторов, имеет вид:

```
1  struct const_iterator{ const Vector *v; size_t i;
2   const_iterator(const Vector*v=nullptr,size_t
        i=0){this->v=v;this->i=i;}
3   const T &operator*(){return v->v[i];}
```

```
4   bool operator!=(const const_iterator &b)const{return
        i!=b.i;}
5   const_iterator operator++(){i++; return *this;}
6   const_iterator operator++(int tmp)
7   {tmp=tmp; const_iterator x=*this; i++; return x;}
8  };
9  const_iterator cbegin()const{return const_iterator(this,0);}
10 const_iterator cend()const{return const_iterator(this,n);}
11 //--
12 struct reverse_iterator{ Vector *v; size_t i;
13  reverse_iterator(Vector*v=nullptr,size_t
        i=0){this->v=v;this->i=i;}
14  T &operator*(){return v->v[i];}
15  bool operator!=(const reverse_iterator &b)const{return
        i!=b.i;}
16  reverse_iterator operator++(){i--; return *this;}
17  reverse_iterator operator++(int tmp)
18  {tmp=tmp; reverse_iterator x=*this; i--; return x;}
19 };
20 reverse_iterator rbegin(){return
       reverse_iterator(this,n-1);}
21 reverse_iterator rend(){return reverse_iterator(this,-1);}
22 //--
23 struct const_reverse_iterator{ const Vector *v; size_t i;
24  const_reverse_iterator(const Vector*v=nullptr,size_t
        i=0){this->v=v;this->i=i;}
25  const T &operator*(){return v->v[i];}
26  bool operator!=(const const_reverse_iterator
        &b)const{return i!=b.i;}
27  const_reverse_iterator operator++(){i--; return *this;}
28  const_reverse_iterator operator++(int tmp)
29  {tmp=tmp; const_reverse_iterator x=*this; i--; return x;}
30 };
31 const_reverse_iterator crbegin()const{return
       const_reverse_iterator(this,n-1);}
32 const_reverse_iterator crend()const{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, т.к. обслуживаемый вектор имеет модификатор *const*, что запрещает изменение его содержимого.

Список использованных источников

1. Агибалов Г.П. *Теория псевдослучайных генераторов.* Томск. Издательский Дом Томского государственного университета. 2019. 68 с. ISBN 978-5-94621-801-6.

2. Антонов А.С. *Параллельное программирование с использованием технологии MPI.* Москва. Издательство Московского Университета. 2004. 71 с. ISBN 5-211-04907-1.

3. Богачёв К.Ю. *Основы параллельного программирования.* Москва. Лаборатория знаний. 2020. 345 с. ISBN 978-5-00101-758-5.

4. Василевский Ю.В., Данилов А.А., Липников К.Н., Чугунов В.Н. *Автоматизированные технологии построения неструктурированных расчетных сеток. Том 4.* Москва. Физматлит, 2016 г. 216 с. ISBN 978-5-9221-1730-2.

5. Брайан Керниган, Деннис Ритчи. *Язык программирования C.* Москва: Вильямс, 2015. 304 с. ISBN 978-5-8459-1975-5.

6. Боресков А.В., Харламов А.А. *Основы работы с технологией CUDA.* ДМК Пресс. 2010. 233 с. ISBN 978-5-94074-578-5.

7. Валединский В.Д., Корнев А.А. *Методы программирования в задачах и примерах на C/C++.* Москва. Издательство Московского университета. 2023. 413 с. ISBN 978-5-19-011927-5.

8. Кнут Д. *Искусство программирования для ЭВМ.* Т. 1–3. Москва. Мир. 1996–1998.

9. Кормен Т., Лейзерсон Ч., Ривест Р. *Алгоритмы. Построение и анализ.* Москва. МЦНМО. 1999.

10. Кронрод М.А. *Оптимальный алгоритм упорядочения без рабочего поля.* Докл. АН СССР, 1969, том 186, номер 6, 1256–1258.

11. Плещинский Н.Б., Плещинский И.Н. *Многопроцессорные вычислительные комплексы. Технологии параллельного программирования.* Казань. Казанский федеральный университет. 2018. 80 с.

12. Староверов В.М., Шевелева А.В. *Лекции по курсу «Работа на ЭВМ и программирование». Часть 1.* Учебное пособие. Москва. Издательство Московского университета. 2024. 317 с. ISBN 978-5-19-012046-2.

13. Столмен Ричард, Пеш Роланд, Шебс Стан и др. *Отладка с помощью GDB. Отладчик для уровня исходного кода. Восьмая Редакция, для GDB версии 5.0.* Free Software Foundation. 59 Temple Place - Suite 330, Boston, MA 02111-1307 USA. Март 2000. Перевод 2000 Дмитрий Сиваченко. ISBN 1-882114-77-9

14. Таненбаум Эндрю, Бос Херберт. *Современные операционные системы. 4-е изд.* Издательство Питер. 2022.

15. Хуанг Т. *Обработка изображений и цифровая фильтрация.* Москва. Мир. 1979.

16. IEEE Computer Society (2019-07-22). *IEEE Standard for Floating-Point Arithmetic.* IEEE STD 754-2019. IEEE. pp. 1–84. doi:10.1109/IEEESTD.2019.8766229. ISBN 978-1-5044-5924-2. IEEE Std 754-2019.

17. Препарата Ф., Шеймос М. *Вычислительная геометрия: Введение.* Перевод с английского. С. А. Вичеса, М. М. Комарова / Под ред. М. Баяковского. Москва. Мир. 1989.

18. Starustrup Bjarne. *The C++ Programming Language (4th Edition)* Addison-Wesley. May 2013. ISBN 978-0321563842.

19. Timothy G. Mattson, Yun (Helen) He, and Alice E. Koniges. *The OpenMP Common Core.* The MIT Press. 2019. 320 pp. ISBN 978-02-625-3886-2.