Министерство образования и науки
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Я. М. ДЕМЯНЕНКО, М. И. ЧЕРДЫНЦЕВА

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В C++

Учебное пособие

Ростов-на-Дону
Издательство Южного федерального университета
2018

УДК 004.43

ББК 32.973.2
Р 88

Рецензенты:
к.ф.-м.н., доцент ЮФУ **М. Э. Абрамян**,
д.т.н., профессор РГУПС **М. А. Бутакова**

Демяненко Я. М.

Р 88 Объектно-ориентированное программирование в C++/ Я. М. Демяненко, М. И. Чердынцева. – Ростов н/Д : Изд-во ЮФУ, 2018. – 142 с.

ISBN

В данном учебном пособии внимание уделяется использованию объектно-ориентированного подхода в языке C++. Изложение материала основано на рассмотрении конкретных примеров. Книга состоит из трёх глав. Учебное пособие адресовано студентам, обучающимся по бакалаврской программе по направлению «Прикладная математика и информатика». Может быть использовано для самостоятельного изучения объектно-ориентированного подхода средствами языка C++.

ISBN

УДК 004.43
ББК 32.973.2

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ ... 3
ПРЕДИСЛОВИЕ ... 4
ВВЕДЕНИЕ ... 5
ГЛАВА 1. Классы и объекты ... 7
1.1.Основы создания классов ... 7
1.2.Конструкторы и деструкторы ... 12
1.3.Дружественные функции ... 17
1.4.Перегрузка операций ... 27
1.5.Перегрузка префиксной и постфиксной операций инкремента ... 36
1.6.Реализация преобразования типов ... 41
1.7.Обработка исключений ... 49
1.8.Статические члены класса ... 55
1.9.Члены класса создаваемые автоматически ... 57
1.10.Семантика перемещения: move-конструктор и move-operator= ... 60
ГЛАВА 2. Отношения между классами ... 66
2.1.Наследование классов ... 66
2.2.Открытое наследование ... 67
2.3.Отношение включения ... 82
2.4.Позднее связывание и виртуальные функции ... 85
2.5.Абстрактные классы ... 91
2.6.Цена виртуальности и система RTTI ... 96
2.7.Отношение подобия ... 98
2.8.Коллекции и итераторы ... 103
2.9.Классы для рекурсивных типов данных ... 113
ГЛАВА 3. Обобщенный подход ... 119
3.1.Шаблоны классов ... 119
3.2.Коллекции ... 127
ЛИТЕРАТУРА ... 140

ПРЕДИСЛОВИЕ

Эта книга продолжает серию учебных пособий по курсу «Языки программирования». Учебное пособие адресовано студентам, обучающимся по бакалаврской программе по направлению «Прикладная математика и информатика». Может быть использовано для самостоятельного изучения объектно-ориентированного подхода средствами языка C++.

В данном издании мы стремились достичь двух целей: во-первых, познакомить студентов с основами объектно-ориентированного программирования; во-вторых, продемонстрировать приёмы решения задач с учетом особенностей языка C++.

Выражаем благодарность доценту ЮФУ М. Э. Абрамяну, высказавшему замечания, которые помогли улучшить качество и ценность книги. Мы признательны доценту ЮФУ С. С. Михалковичу за возможность воспользоваться некоторыми интересными примерами, а также многим нашим коллегам, преподавателями студентам, с которыми нас объединяют общие интересы.

ВВЕДЕНИЕ

В данной книге содержится материал, важный для понимания языка программирования. Материал не ориентирован на определенную версию компилятора, внимание уделяется именно языку C++ и использованию объектно-ориентированного подхода, а не особенностям конкретной реализации.

В главе 1 вводятся основные понятия и определения объектно-ориентированного программирования с особенностями их реализации в языке C++. Изложение материала сопровождается разбором примеров решенных задач.

В главе 2 рассматриваются отношения между классами. Уделяется внимание различным видам наследования. Рассматривается механизм виртуальности и принцип полиморфизма. Изложение материала иллюстрируется подробным рассмотрением примеров решенных задач. Такой подход позволит студентам научиться применять теоретические знания при решении конкретных задач.

Глава 3 посвящена базовым понятиям обобщённого программирования и особенностям их реализации в языке C++. Изучение материала построено на основе разбора примеров программ.

Представленный в учебном пособии материал используется в курсе «Языки программирования» бакалаврской программы по направлению «Прикладная математика и информатика». Книга может быть использована для самостоятельного изучения объектно-ориентированного подхода средствами языка C++.

Для облегчения работы с текстом в книге приняты следующие соглашения:

- ✓ Коды программ, фрагменты примеров, операторы, классы, объекты, методы обозначены специальным шрифтом (`Courier`), что позволяет легко найти их в тексте.
- ✓ Для указания имен файлов и каталогов используется шрифт Arial.
- ✓ Важные термины, встречающиеся впервые, выделены *курсивом*.
- ✓ Определения, термины, объяснения для запоминания предваряются специальным символом 📖.
- ✓ Более подробные объяснения отмечены специальным символом 📚.
- ✓ Знак ➢ означает, что проводятся различные сравнения.
- ✓ Специальный символ ☺ означает рекомендации по написанию программ.
- ✓ Предостережения от ошибок начинаются со знака ☠.
- ✓ Упражнения, которые необходимо выполнить по ходу изучения, обозначены специальным символом 🔔.

ГЛАВА 1. КЛАССЫ И ОБЪЕКТЫ

1.1. Основы создания классов

Попытаемся ответить на вопрос: в чем же состоит принципиальное отличие языка C++ от языка C? Сошлемся при этом на одного из специалистов по языку C++ Брюса Эккеля: «Включение функций в структуры составляет подлинную суть того, что язык C++ добавил в C». При внесении функций в структуры к характеристикам (как у структур в C) добавляется поведение. Так возникает концепция *класса*. В результате чего его поля (характеристики) и функции (поведение) рассматриваются как единое целое. Такое объединение получило название *инкапсуляция* (encapsulation). При этом существует возможность регламентировать доступ к членам класса (полям и функциям).

Существует три спецификатора доступа:

`private` (закрытый) – доступен только для членов класса;

`protected` (защищенный) – доступен для членов класса и их наследников;

`public` (открытый) – доступен для всех.

Эти спецификаторы определяют уровень доступа для всех объявлений, следующих за ними. После любого спецификатора должно стоять двоеточие. Если спецификатор при объявлении опущен, то используется спецификатор по умолчанию.

Для объявления класса можно использовать либо ключевое слово `struct`, либо ключевое слово `class`. Рекомендуется использовать слово `class`. Различие проявляется в спецификаторах доступа по умолчанию (табл. 1).

Таблица 1

Различие в доступе по умолчанию

Описание	`class <имя класса> { <поля и функции> };`	`struct <имя класса> { <поля и функции> };`
Спецификатор доступа по умолчанию	`private`	`public`

Следующие два объявления эквивалентны:

```
struct A{
  int f();
  void g();
private:
  int i, j, k;
};

int A::f() {
  return i+j+k;
}

void A::g() {
  i=j=k=0;
}
```

```
class B{
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i+j+k;
}

void B::g() {
  i=j=k=0;
}
```

```
int main(){
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
}
```

☺ Рекомендуется всегда явно задавать спецификатор доступа к членам класса.

Приняты два варианта стиля объявлений класса:

- ✓ сначала располагаются поля, потом – функции;
- ✓ сначала располагаются функции, потом – поля.

☺ Рекомендуется в одной программе придерживаться одного из вариантов стилей объявлений.

Класс определяет новый тип данных. Переменная данного типа называется *объектом*. Объект также называют *экземпляром класса*.

Пример 1. Реализовать класс для описания геометрической фигуры «круг». Определить конструкторы, функции-члены класса для вычисления площади, функции доступа к радиусу. Реализовать сравнение объектов на равенство.

```
//circle.h
#ifndef CIRCLE_H
#define CIRCLE_H
class circle {
private:
  double x,y,r;
public:
  circle();
  circle(double x1, double y1, double r1);
  double s();
  void set_r(double r1);
  double get_r();
  bool equal(circle a);
  bool operator ==(circle a);
};
#endif

//circle.cpp
#include <cmath>
#include "circle.h"
circle::circle(){
  x=10;
  y=10;
  r=10;
}
circle::circle(double x1, double y1, double r1){
  x=x1;
  y=y1;
  r=r1;
}
double circle::s(){
  return 3.14*r*r;
}
void circle::set_r(double r1){
  if (r1<0)
    r=0;
  else
```

```
    r=r1;
}
double circle::get_r(){
  return r;
}
bool circle::equal(circle a){
  const double eps=0.0001;
  if (abs(x-a.x)<eps && abs(y-a.y)<eps && abs(r-a.r)<eps)
    return true;
  else
    return false;
}
bool circle::operator==(circle a){
  return equal(a);
}

//главный файл-main.cpp
#include "circle.h"
#include <iostream>
using namespace std;

int main(){
  circle a,b(0,0,1);
  cout<<a.s()<<endl<<b.s()<<endl;
  cout<<a.equal(b)<<endl;
  cout<<a.operator==(b)<<endl;
  cout<<(a==b)<<endl;
  return 0;
}
```

Описание класса размещено в заголовочном файле. В описание класса включены поля и заголовки функций.

☺ В C++ рекомендуется реализацию функций располагать в большинстве случаев вне класса, но допустимо и внутри описания класса. Описание класса рекомендуется располагать в заголовочном файле.

Поля класса `x`, `y`, `r` объявлены закрытыми. Все функции этого класса – открытые. Среди функций присутствуют:

- ✓ два конструктора

```
  circle();
  circle(double x1, double y1, double r1);
```

- ✓ функция вычисления площади

```
double s();
```

- ✓ пара функций для установки доступа к закрытому полю класса (их называют сеттеры и геттеры соответственно)

```
void set_r(double r1);
double get_r();
```

- ✓ две функции для сравнения двух объектов на равенство – один реализован как функция, другой как перегруженная операция

```
bool equal(circle a);
bool operator ==(circle a);
```

В C++ для пользовательских типов данных возможна собственная реализация стандартных операций, которая называется *перегрузкой операций*. Перегруженная операция – это функция, имя которой начинается со слова `operator`, за которым следует символ операции. Использовать перегруженную операцию можно двумя способами:

- ✓ традиционное использование операции

```
cout<<(a==b)<<endl;
```

- ✓ вызов функции

```
cout<<a.operator==(b)<<endl;
```

☺ В основном поля рекомендуется делать закрытыми, интерфейсные функции – открытыми, а служебные функции – закрытыми.

Реализация функций расположена в отдельном cpp-файле. При их реализации вне описания классов имена функций должны быть уточненными именем класса. Для этого используется операция разрешения области видимости :: :

```
double circle::s(){
  return 3.14*r*r;
}
```

Текст файла main.cpp начинается с создания объектов a и b. Для создания объекта a вызывается конструктор по умолчанию, для объекта b – конструктор с параметрами:

```
circle a,b(0,0,1);
```

Все функции класса circle являются функциями объектов (экземпляров класса). Они вызываются через точку a.s():

```
cout<<a.s()<<endl<<b.s()<<endl;
```

1.2. Конструкторы и деструкторы

Чтобы обеспечить должную инициализацию каждого объекта, разработчик класса должен включить в него специальную функцию, которая называется *конструктором*. Имя конструктора должно совпадать с именем класса. Конструктор предназначен для инициализации полей объекта начальными значениями, он вызывается в момент создания объекта.

В примере 1 реализованы два конструктора – без параметров и с параметрами:

```
circle();
//без параметров – конструктор по умолчанию

circle(double x1, double y1, double r1);
//конструктор с параметрами
```

Конструктор без параметров называется *конструктором по умолчанию*.

☺ Если в дальнейшем предполагается размещать объекты класса в массиве или в любом другом контейнере, необходимо объявлять конструктор по умолчанию.

В случае если не описан явно ни один конструктор, компилятор автоматически (неявно) сгенерирует конструктор по умолчанию. Поведение

неявно созданного конструктора будет таким же, как если бы он был объявлен явно без списка параметров и с пустым телом.

Когда программа достигает точки выполнения, в которой определяется объект

```
circle a,b(0,0,1);
```

происходит *автоматический вызов конструктора.*

Синтаксис *деструктора* в целом схож с синтаксисом конструктора: имя функции тоже определяется именем класса. Но чтобы деструктор отличался от конструктора, его имя начинается с префикса ~ (тильда). Кроме того, деструктор всегда один и у него нет аргументов. В случае отсутствия явного задания деструктора, компилятор неявно создаёт деструктор с пустым телом.

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости или при уничтожении объекта, размещённого в динамической памяти. При этом очищается память, занимаемая объектом.

Если перед уничтожением объекта необходимо освободить используемые ресурсы (например, динамическую память для размещения полей объекта, открытые файлы и т.д.), то необходимо явно определить деструктор, выполняющий эти действия.

 Конструкторы и деструкторы обладают одной уникальной особенностью: они не имеют возвращаемого значения. В этом они принципиально отличаются от функций, возвращающих пустое значение (значение типа `void`).

Чтобы лучше понять механизмы вызовов конструкторов и деструкторов, можно включать в их код операторы вывода.

 Добавьте в конструкторы класса `circle` операторы вывода информации о том, какой конструктор сработал. Добавьте деструктор, выводящий информацию о том, что он сработал. Попробуйте объяснить, почему количество сообщений от декструктора превышает количество сообщений от конструкторов.

Пример 2. Реализовать два класса для описания геометрических фигур «круг» и «прямоугольник». Описать функцию, позволяющую совместить центры круга и прямоугольника (переместить круг), если круг можно поместить в прямоугольник.

```
//shape.h
#ifndef SHAPE_H
#define SHAPE_H

class rectangle;

class circle {
private:
  double x, y, r;
public:
  circle();
  circle(double x1, double y1, double r1);
  void print();
  friend bool tofit(circle &a, rectangle b);

};

class rectangle {
private:
  double x1, y1, x2, y2;
public:
  rectangle();
  rectangle(double x1, double y1, double x2, double y2);
  double width();
  double height();
  void print();
  friend bool tofit(circle &a, rectangle b);
};

#endif
```

```
//shape.cpp
#include <cmath>
#include <iostream>
#include "shape.h"
using namespace std;

circle::circle(){
  x = 10;
  y = 10;
  r = 10;
}
circle::circle(double x1, double y1, double r1){
  x = x1;
  y = y1;
  r = r1;
}

void circle::print(){
  cout << x << ' ' << y << ' ' << r << endl;
}

rectangle::rectangle(){
  x1 = 0;
  y1 = 0;
  x2 = 10;
  y2 = 10;
}
rectangle::rectangle(double x1,double y1,double x2,double y2 ){
  this->x1 = x1;
  this->y1 = y1;
  this->x2 = x2;
  this->y2 = y2;
}

double rectangle::width(){
  return abs(x2 - x1);
}

double rectangle::height(){
  return abs(y2 - y1);
}

void rectangle::print(){
  cout << x1 << ' ' << y1 << ' ' << x2 << ' ' << y2 << endl;
}

bool tofit(circle & a, rectangle b){
  if (b.width() >= 2*a.r && b.height() >= 2*a.r){
```

```
    a.x = (b.x1 + b.x2) / 2;
    a.y = (b.y1 + b.y2) / 2;
    return true;
  }
  return false;
}

//главный файл – main.cpp
#include "shape.h"
#include <iostream>

using namespace std;

int main(){
  setlocale(0, "Russian");
  circle a(0, 0, 5);
  rectangle b(10,10,20,20);
  a.print();
  b.print();
  if (tofit(a, b)) {
    cout << "перемещение выполнено" << endl;
    a.print();
    b.print();
  }
  else
    cout << "круг не помещается в прямоугольник" << endl;
  return 0;
}
```

В заголовочном файле shape.h размещены описания двух классов `circle` и `rectangle`, а их определения в файле shape.cpp.

В конструкторе с параметрами для класса `rectangle` возникает ситуация, когда имена параметров совпадают с именами переменных-членов класса. Для разрешения коллизии имён используется ключевое слово `this`.

```
rectangle::rectangle(double x1,double y1,double x2,double y2 ){
  this->x1 = x1;
  this->y1 = y1;
  this->x2 = x2;
  this->y2 = y2;
}
```

Ключевым словом this обозначается указатель на объект в функциях-членах класса. Если коллизии имён не возникают, то при обращении к членам класса его обычно опускают. Указатель на объект передаётся как неявный параметр во все функции, которые могут вызываться только для экземпляров классов (в том числе, конструкторы и деструкторы).

1.3. Дружественные функции

В файле shape.cpp расположена функция tofit.

```
bool tofit(circle & a, rectangle b){
  if (b.width() >= 2*a.r && b.height() >= 2*a.r){
    a.x = (b.x1 + b.x2) / 2;
    a.y = (b.y1 + b.y2) / 2;
    return true;
  }
  return false;
}
```

В функции tofit используются не только функции этих классов, но и их поля. При этом поля описаны как private, т.е. защищены от внешнего доступа. Чтобы разрешить внешней функции доступ к защищённым полям объектов, необходимо её сделать дружественной для этих классов. Для этого описание функции со спецификатором friend помещается в интерфейс каждого из классов:

```
friend bool tofit(circle &a, rectangle b);
```

В соответствии с принципом инкапсуляции работа с защищёнными членами класса должна быть организована через открытые функции. В рассматриваемом примере можно было бы организовать геттеры и сеттеры для доступа к закрытым членам классов. Если больше ни для каких других целей он не нужны, то вместо них можно использовать механизм дружественности. Важно, что при этом сокращается количество вызовов функций.

Объявление функции дружественной классу разрешает доступ к защищённым полям класса только для этой функции.

Дружественными по отношению к рассматриваемому классу могут быть не только внешние функции, но и функции другого класса, или даже другой класс (все его функции).

На функции, объявленные дружественными, не распространяются действия спецификаторов (`public`, `private`, `protected`).

☺ Объявление функций дружественными рекомендуется располагать либо в самом начале, либо в самом конце описания класса.

Параметрами функции `tofit` являются объекты обоих классов. Поскольку объявление дружественности для `tofit` расположено в интерфейсах обоих классов, возникает проблема очерёдности их описания. Решением является использование предварительного описания одного из классов.

```
class rectangle;
```

Пример 3. Используя классы, описанные в примере 2, создать объект «прямоугольник» и массив объектов класса «круг». Посчитать количество кругов, которые можно поместить внутри прямоугольника. Для таких кругов совместить их центры с центром прямоугольника.

```
#include "shape.h"
#include <iostream>
#include <cstdlib>  //содержит srand() и rand()
#include <ctime>    //содержит time()
using namespace std;

int main(){
  setlocale(0, "Russian");
  rectangle b(10, 10, 40, 40);
  circle* c[5];
  srand((unsigned)time(0));
  for (int i = 0; i < 5; ++i) {
    c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);
    c[i]->print();
  }
  int count = 0;
  for (int i = 0; i < 5; ++i)
  if (tofit(*c[i], b))
```

```
    count++;
  cout << "количество перемещённых = " << count << endl;
  for (int i = 0; i < 5; ++i)
    c[i]->print();
  for (int i = 0; i < 5; ++i)
    delete c[i];
}
```

В условии задачи требуется создать массив объектов класса `circle`. Если бы описание массива выглядело таким образом

```
circle c[5];
```

то все объекты были бы созданы с помощью конструктора по умолчанию, т.е. с одинаковыми значениями переменных-членов класса. Чтобы иметь возможность для каждого элемента массива вызывать конструктор с параметрами, необходимо разместить их в динамической памяти, т.е. при объявлении использовать массив указателей.

```
circle* c[5];
```

Затем для каждого элемента массива создать объект класса `circle`.

```
for (int i = 0; i < 5; ++i) {
  c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);
  c[i]->print();
}
```

В конце программы динамическая память освобождается.

```
for (int i = 0; i < 5; ++i)
  delete c[i];
```

Пример 4. Реализовать класс – динамический массив целых чисел. Перегрузить операции: + для двух массивов одинакового размера; + для массива и целого числа; += для двух массивов одинакового размера; присваивания; обращение к элементу по индексу; вывода в поток.

```
//classArray.h
#ifndef CLASSARRAY_H
#define CLASSARRAY_H
#include <iostream>
using namespace std;
class Array {
private:
  int n;
```

```
  int *A;
public:
  Array();     //конструктор по умолчанию
  Array(int _n, int x = 0);
  //конструктор с параметром по умолчанию
  Array(const Array &B);      //конструктор копии
  int length() const; //функция для нахождения размера массива
  void resize(int nsize);     //изменение размера массива
  Array operator + (const Array &B);
  Array &operator += (const Array &B);
  Array operator + (const int x);
  Array &operator = (const Array &B);
  int& operator [] (int i);
  ~Array();
  friend ostream & operator << (ostream &out, const Array &B);
};
#endif

//classArray.cpp
#include "classArray.h"
Array::Array() {
  n = 10;
  A = new int[n];
  for (int i = 0; i<n; i++)
    A[i] = 0;
}

Array::Array(int _n, int x) {
  n = _n;
  A = new int[n];
  for (int i = 0; i<n; i++)
    A[i] = x;
}

Array::Array(const Array &B) {
  n = B.n;
  A = new int[n];
  for (int i = 0; i<n; i++)
    A[i] = B.A[i];
}

int Array::length() const{
  return n;
}

void Array::resize(int nsize) {
  int * ndata = new int[nsize];
  int sz = (n < nsize) ? n : nsize;
```

```
  for (int i=0;i<sz;++i)
    ndata[i] = A[i];
  delete[] A;
  A = ndata;
  n = nsize;
}

Array Array::operator + (const Array &B) {
  if (n != B.n)
    throw (1);
  Array C(n);  //можно: Array C(n,0);
  for (int i = 0; i<n; i++)
    C.A[i] = A[i] + B.A[i];
  return C;
}

Array &Array::operator += (const Array &B) {
  if (n != B.n)
    throw (1);
  for (int i = 0; i<n; i++)
    A[i] = A[i] + B.A[i];
  return *this;
}

Array Array::operator + (const int x) {
  Array C(n);  //можно: Array C(n,0);
  for (int i = 0; i<n; i++)
    C.A[i] = A[i] + x;
  return C;
}

Array &Array::operator = (const Array &B) {
  if (this != &B){
    delete[] A;
    n = B.n;
    A = new int[n];
    for (int i = 0; i<n; i++)
      A[i] = B.A[i];
  }
  return *this;
}

int& Array::operator [] (int i) {
  return A[i];
}
Array::~Array(){
  delete[] A;
}
```

```
ostream & operator << (ostream & out, const Array &B) {
  for (int i = 0; i<B.n; i++)
    out << B.A[i] << ' ';
  out << endl;
  return out;
}

//main.cpp
#include "classArray.h"
int main(){
  setlocale(0, "Russian");
  Array Q(10, 5);
  Array P(10);
  Array W, E;
  Array R(Q);
  cout << "Q " << Q;
  cout << "P " << P;
  cout << "W " << W;
  cout << "E " << E;
  cout << "R " << R;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;
  for (int i = 0; i < Q.length(); ++i)
    Q[i] = Q.length() - 1 - i;
  cout << "изменённый Q " << Q;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;
  for (int i = 0; i < E.length(); ++i)
    E[i] = i;
  cout << "изменённый E " << E;
  W = Q + 15;
  cout << "изменённый W " << W;
  P = W + R;
  cout << "изменённый P " << P;
  E += P;
  cout << "изменённый E " << E;
  cout << "срединные элементы массива P "
       << P[P.length() / 2 - 1] << ' '
       << P[P.length() / 2] << endl;
  Array Z(15);
  try {
    Z += Q;
    cout << "изменённый Z " << Z;
  }
  catch (int) {
```

```
    cout << "Массивы имеют разную длину" << endl;
  }
  return 0;
}
```

Когда классы имеют сложную структуру, удобно использовать UML диаграммы классов, которые наглядно показывают их внутреннюю организацию. В частности, на диаграммах представляются члены-данные (поля) и члены-функции (методы) с указанием в виде пиктограмм областей видимости. UML диаграмма для класса `Array` представлен на рисунке 1.1.

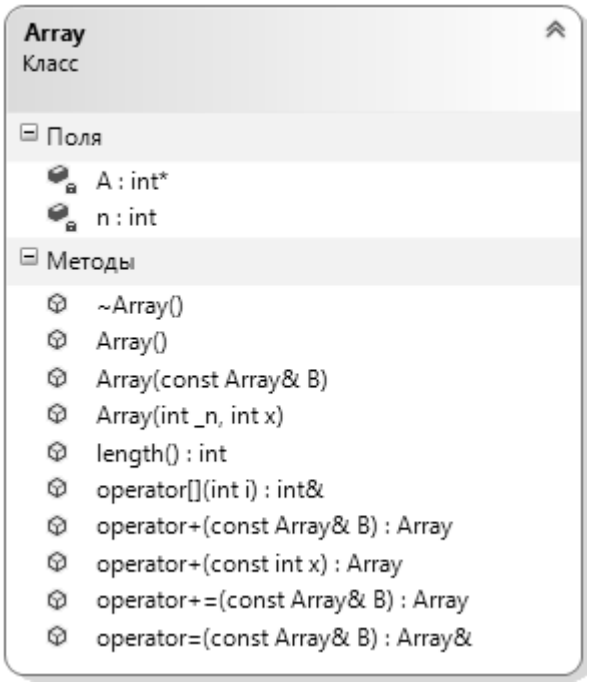


Рис.1.1. UML диаграмма класса `Array`

На диаграмме не отображается перегруженная операция вывода в поток, реализованная с помощью дружественной функции, поскольку она не является членом класса.

Обычно разработку классов начинают с построения UML диаграмм. Помимо этого, UML диаграммы можно использовать при описании требований к задаче.

Для класса `Array` размер динамического массива хранится в приватной переменной-члене класса `n`. Открытая функция-член класса `length()`

предназначена для получения длины динамического массива. Функция `length()` объявлена константной, так как она не изменяет вызывавший её объект.

```
int Array::length() const{
  return n;
}
```

Динамический массив подразумевает, что у него может меняться размер в ходе выполнения программы. Для это в классе предусмотрена функция `resize(int)`.

```
void Array::resize(int nsize) {
  int * ndata = new int[nsize];
  int sz = (n < nsize) ? n : nsize;
  for (int i=0;i<sz;++i)
    ndata[i] = A[i];
  delete[] A;
  A = ndata;
  n = nsize;
}
```

Для создания объектов класса `Array` используются разные определения. Каждому виду определения соответствует свой конструктор.

```
Array Q(10,5); //конструктор с параметром по умолчанию.
//Второй параметр задан явно
Array P(10); //конструктор с параметром по умолчанию.
//Второй параметр задан неявно
Array W,E; //конструктор по умолчанию
Array R(Q); //конструктор копии
```

Конструктор копии, или *копирующий конструктор*, создаёт копию объекта, передаваемого в конструктор в качестве параметра. Кроме явного вызова конструктора копии при создании объекта, он неявно вызывается каждый раз, когда объект передаётся в функцию в качестве параметра по значению или функция возвращает объект.

Поясним эти три случая, когда вызывается конструктор копии.

1. Явное создание нового объекта как копии существующего:

```
Array myv1 (Q);
Array myv2 = myv1;
```

2. Вызов функции с передачей параметра по значению:

```
void f(Array arr) {/* … */}
```

3. Возврат объекта из функции по значению:

```
Array g() {/* … */}
```

Копирующие конструкторы настолько важны, что компилятор автоматически генерирует копирующий конструктор, если этого не делает программист. Такой автоматически создаваемый конструктор копии является конструктором с побитовым копированием переменных-членов класса. Такое копирование называется поверхностным.

В классе `Array` для размещения массива используется динамическая память, адрес которой хранится в переменной `A`. В случае использования конструктора копии, создаваемого автоматически, вместо копии массива будет создана копия ссылки на него, т. е. ссылки двух разных объектов будут указывать на одно и то же место в памяти. Для классов, размещающих данные в динамической памяти, необходим конструктор копии.

☠ Для классов, использующих размещение данных в динамической памяти, отсутствие конструктора копии является грубой ошибкой.

Реализация конструктора копии для класса `Array`:

```
Array::Array (const Array &B){
  n=B.n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=B.A[i];
}
```

Завершая обсуждение конструктора копии, рассмотрим следующую ситуацию.

```
class Array {
  ...
public:
  ...
};
```

```
Array g() {
  return Array();
}
int main()
{
  Array myv1 = g();
}
```

Возникает вопрос – сколько раз будет вызван конструктор копии? Здесь присутствуют случаи 1 и 3 вызова конструктора копии, а значит должны создаваться две копии. Чтобы проверить, так ли это, можно добавить в конструктор копии вывод сообщения о выполнении конструктора.

На самом деле, запуск данного примера покажет, что во время выполнения программы конструктор копии не будет вызван ни разу. Это объясняется работой оптимизирующего компилятора. Заметим, что такая оптимизация может существенно повлиять на поведение программы в случае, когда конструктор копии содержит побочные эффекты (вывод сообщения). Эта оптимизация носит название *Return Value Optimization (RVO)* и выполняется по умолчанию подавляющим большинством современных компиляторов. Требование выполнения RVO по умолчанию явно оговорено в стандарте языка.

Если специальными ключами компиляции запретить RVO, то вызов конструктора копии будет выполнен ровно два, как и ожидалось. Однако в реальных программах просто стараются не помещать в конструктор копии дополнительный код для реализации побочного эффекта.

Если в классе используется выделение памяти, то её необходимо освободить средствами этого же класса. Это должен делать деструктор.

```
Array::~Array( ){
  delete[] A;
}
```

Напомним, что деструктор вызывается компилятором автоматически для каждого созданного объекта.

☺ В случае использования динамической памяти в реализации класса – настоятельно рекомендуется реализовать конструкторы и деструктор. Среди конструкторов обязательно должен присутствовать конструктор копии.

Остальные функции-члены класса `Array` реализуют перегрузку операций.

1.4. Перегрузка операций

Перегружать операции можно только для определённых пользователем типов, т. е. для классов. Операции для стандартных типов перегружать нельзя. Перегруженная операция должна иметь хотя бы один операнд, тип которого определён пользователем. Для определения перегруженных операций используется специальная функция с именем `operator@`, где `@` – идентификатор перегружаемой операции.

В примере 1 для класса `circle` была перегружена операция сравнения на равенство (==):

```
bool circle::operator==(circle a){
  return equal(a);
}
```

Ключевое слово `operator` служит для использования операций в функциональном стиле.

Вызов перегруженной операции можно выполнять двумя способами:

- ✓ используя функциональный стиль `a.operator==(b)`, например,

```
cout<<a.operator==(b)<<endl;
```

- ✓ используя синтаксис операции `a==b`, например,

```
cout<<(a==b)<<endl;
```

Попробовать убрать скобки в примере 1 в операторе вывода в поток

```
cout<<(a==b)<<endl;
```

Объяснить, почему они необходимы.

Хотя C++ позволяет перегружать почти все операции, доступные в C++, возможности перегрузки ограничены. В частности, отсутствует возможность изменения приоритета или количества аргументов у операций. Кроме того рекомендуется не изменять семантику операций.

Количество аргументов в списке перегруженной операции зависит от двух факторов:

- ✓ категории операции – унарная или бинарная;
- ✓ способа определения операции – в виде глобальной функции (один аргумент для унарной, два – для бинарных операций) или функции класса (для унарных операций аргументы отсутствуют, для бинарных операций – один аргумент).

При определении функции-члена класса объект, для которого она будет вызываться, всегда будет её левым операндом.

Рассмотрим реализацию перегрузки бинарной операций + для класса `Array` в примере 4. Эта операция может быть реализована функцией-членом класса с одним аргументом. Напомним, что для всех функций-членов класса первым параметром всегда неявно передаётся указатель на объект, для которого вызывается функция. Именно поэтому у бинарной операции всего один аргумент. Операция сложения перегружается для двух списков параметров.

Если параметром является целое число, то операция предназначена для увеличения всех элементов массива на заданное значение.

```
Array Array::operator + (const int x) {
  Array C(n);  //можно: Array C(n,0);
  for (int i = 0; i<n; i++)
    C.A[i] = A[i] + x;
  return C;
}
```

Данная перегруженная операция сложения является некоммутативной, поскольку её левый операнд — это объект класса `Array`, а правый — целое число.

Операция сложения двух объектов класса `Array` выглядит следующим образом:

```
Array Array::operator + (const Array &B){
  if (n != B.n)
    throw (1);
  Array C(n);  //можно: Array C(n,0);
  for (int i=0; i<n; i++)
    C.A[i]=A[i]+B.A[i];
  return C;
}
```

Хотя предполагается, что эта операция будет использована только для массивов одинаковой размерности, функция начинается с проверки корректности вызова и в случае несоответствия выбрасывается исключение. Чтобы отслеживать эту ошибку, операцию следует использовать внутри блока `try catch`.

```
  Array Z(15);
  try {
    Z += Q;
    cout << "изменённый Z " << Z;
  }
  catch (int) {
    cout << "Массивы имеют разную длину" << endl;
  }
```

Рассмотрим перегрузку операции присваивания. Она занимает важное место при реализации классов, использующих динамическую память. Присваивание, так же, как и конструктор копии, по умолчанию реализуется побайтным копированием, которое подразумевает только копирование значений полей объекта. Поэтому при использовании динамической памяти в классе произойдет копирование ссылки на нее, а сама область динамической памяти станет разделяемой между двумя объектами. Такое разделение доступа к памяти в C++ при работе с указателями не является корректным.

В частности, это приведёт к ошибке двойного освобождения памяти.

```
{
  Array myv1;
  Array myv2;
  myv2 = myv1;
...
}
//вызываются деструктор для объекта myv1 и деструктор для myv2
```

Здесь произойдёт ещё и утечка памяти, так как старое значение указателя `A` в объекте `myv2` потеряется.

☺ В случае использования динамической памяти в реализации класса настоятельно рекомендуется перегружать операцию присваивания.

Функция `operator=` должна быть обязательно функцией класса. В определении функции `operator=` необходимо скопировать всю необходимую информацию из правостороннего объекта в левосторонний.

```
Array &Array::operator = (const Array &B) {
  if (this != &B) {
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

Сначала следует проверить, не происходит ли самоприсваивание объектов (`A=A`). Это позволяет избежать выполнения бесполезных операций.

```
if (this != &B) {
...
}
```

Поскольку размерности полей, размещённых в динамической памяти, у левостороннего и правостороннего операторов могут не совпадать, рекомендуется очистить динамическую память левостороннего операнда и заново выделить память нужного размера.

```
delete[] A;
n=B.n;
A=new int[n];
```

☠ Игнорирование проверки самоприсваивания в случае использования динамической памяти в определении класса может привести к потере данных.

Напомним, что каждая операция в C++ возвращает значение. Операция присваивания возвращает значение, совпадающее со значением левого операнда после выполнения операции. Поскольку рекомендуется не изменять семантику при перегрузке операции присваивания необходимо вернуть знеачение левого операнда. Левый операнд – это объект, на который указывает неявный параметр `this`.

```
return *this;
```

В данной реализации операции могут появиться проблемы при возникновении исключения в операции `new` (такое исключение это не такая уж редкая ситуация). Поскольку к этому моменту уже выполнена операция `delete[] A`, объект после исключения в `new` останется в «полуразрушенном» состоянии. В C++ выделяют три уровня гарантий безопасности кода при возникновении исключений:

- ✓ базовый – при возникновении исключения не возникает утечек ресурсов, однако объекты могут находиться в непредсказуемом состоянии;
- ✓ строгий – если во время операции произошло исключение, то объект будет находиться в том же состоянии, что до начала операции;
- ✓ без исключений – в данном коде не может возникнуть исключений.

Приведённая реализация `operator=` даёт лишь базовую гарантию. Достаточно несложно изменить её на строгую. Заведём вспомогательную

переменную newdata для результата выделения памяти, а операцию удаления A перенесём в конец функции.

```
Array & Array::operator = (const Array &B) {
  if (this != &B) {
    n=B.n;
    int * newdata = new int[n];
    for (int i=0; i<n; i++)
      newdata[i]=B.A[i];
    delete[] A;
    A = newdata;
  }
return *this;
}
```

Заметим, что обе версии реализации operator= выполняют действия, которые используются в других функциях, а именно в деструкторе и в конструкторе копии.

Идиома *copy-and-swap* опирается на это наблюдение и предполагает реализацию операции копирующего присваивания с использованием конструктора копий. При этом требуется вначале создать вспомогательную функцию-член swap(Array & other), для обмена содержимого текущего объекта с объектом other.

```
void Array::swap(Array & other) {
  swap(n, other.n);
  swap(A, other.A);
}

Array& Array::operator=(Array other){//вызов конструктора копии
  this -> swap(other);
  return *this;
}
```

☺ Идиома copy-and-swap позволяет разрабатывать устойчивые к исключениям операторы присваивания и сокращает количество кода в них ценой определения полезной вспомогательной функции swap.

В случае, если для класса перегружены операции + и =, то реализация перегруженной операции += может быть выполнена с их использованием.

Например, реализация перегруженной операции += для класса `Array` из примера 4

```
Array &Array::operator += (const Array &B){
  for (int i=0; i<n; i++)
    A[i]=A[i]+B.A[i];
  return *this;
}
```

может быть изменена следующим образом:

```
Array &Array::operator += (const Array &B){
  *this=(*this)+B;
  return *this;
}
```

 Выполнить эти изменения в примере. Проверить работоспособность программы.

Перегрузка операции индексирования также обладает некоторыми особенностями. Её нужно реализовывать только как функцию-член класса. Важно, чтобы перегруженная операция доступа к элементу массива по индексу [ ] возвращала ссылку на элемент массива. Это обусловлено требованиями к соблюдению семантики.

```
int& Array::operator [] (int i){
  return A[i];
}
```

Перегруженную операцию доступа к элементу массива по индексу [ ] можно использовать как для получения значения элемента массива, так и для его изменения.

```
  for (int i = 0; i < Q.length(); ++i)
    Q[i] = Q.length() - 1 - i;
  cout << "изменённый Q " << Q;
  cout << "срединные элементы массива Q "
       << Q[Q.length() / 2 - 1] << ' '
       << Q[Q.length() / 2] << endl;
  for (int i = 0; i < E.length(); ++i)
    E[i] = i;
```

Возможно, операцию индексирования потребуется использовать в функциях, в которые объект класса `Array` передаётся как константный па-

раметр по ссылке. Например, функция printAndSum, которая выведет на экран содержимое нашего вектора и вычислит сумму его элементов.

```
int printAndSum (const Array &v) {
  int sum=0;
  for (int i = 0; i < v.length(); i++){
    cout << v[i] << ' ';
    sum+=v[i];
  }
  return sum;
}
```

При компиляции этой функции возникнет ошибка «IntelliSense: отсутствует оператор "[]", соответствующий этим операндам: const Array[int]».

Для корректной компиляции в данном случае необходимо определить вторую функцию перегрузки операции индексирования.

```
int Array::operator [] (int i) const {
  return A[i];
}
```

Несмотря на то, что списки параметров совпадают у обеих операций индексации, перегрузка возможна, поскольку модификатор const включается в сигнатуру функции.

Обратите внимание, что аналогичная ошибка возникала бы, если бы функция length не являлась константной. Это связано с тем, что для константных объектов можно вызывать только константные функции.

Иногда бывает необходимо, чтобы левосторонний операнд был объектом другого класса или примитивного типа. Например, для часто перегружаемых операций потокового ввода-вывода левосторонним операндом должен быть объект-поток. Следовательно, перегрузку такой операции нужно организовывать в виде внешней функции. А внешним функциям запрещен доступ напрямую к полям класса, объявленным как private. Способ решения проблемы – объявить такую операцию дружественной для класса.

В примере 4 для класса `Array` реализована перегрузка операции << вывода в поток:

```
ostream & operator << (ostream & out, const Array &B) {
  for (int i=0; i<B.n; i++)
    out << B.A[i] << ' ';
  out << endl;
  return out;
}
```

После выполнения всех действий с потоком ввода или вывода операция возвращает ссылку на поток, что позволяет использовать результат в более сложных выражениях.

Дружественность функции `operator<<` по отношению к классу `Array` дает право функции напрямую обращаться к закрытым членам класса.

 Реализовать для класса `Array` операцию ввода из потока.

Всякий раз, когда нужно перегрузить бинарную операцию для операндов двух разных типов с сохранением коммутативности, часто используют дружественные функции.

 Перегрузить операцию сложения, у которой левой операнд – целое число, а правый – объект класса `Array`.

Большинство операций можно перегружать и как внутренние функции, и как внешние, используя дружественность. Однако некоторые операции можно перегружать только одним из способов. Перегрузку только в виде внешней функции требуют все бинарные операции, у которых левый операнд является объектом другого класса или примитивного типа. Перегрузку только в виде внутренней функции требуют следующие операции: присваивания =, индексирования [], преобразования типов `type`, вызова функции () и доступ к элементу через указатель –>.

1.5. Перегрузка префиксной и постфиксной операций инкремента

При перегрузке операций инкремента и декремента нужно учитывать, что они имеют две формы префиксную и постфиксную.

Пример 5. Реализовать класс – `Time` («Время»). Для него перегрузить операцию инкремента в двух формах (увеличение времени на одну секунду). Требования к классу представлены на UML диаграмме (рис.1.2).

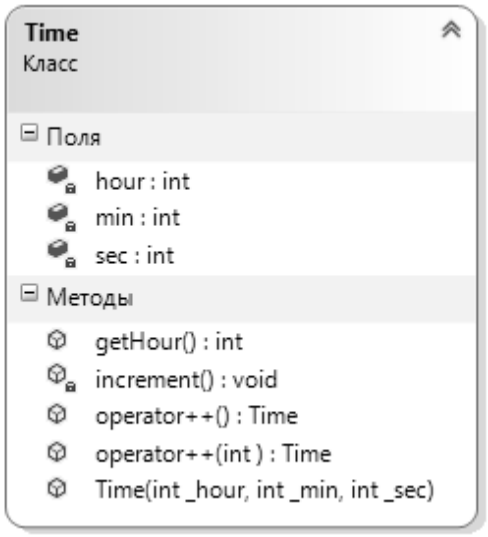


Рис. 1.2. UML диаграмма класса `Time`

```
//classTime.h
#ifndef CLASSTIME_H
#define CLASSTIME_H
class Time{
  private:
  int sec,min,hour;
  void increment(); // Функция-утилита
  public:
  Time(int _hour=0,int _min=0,int _sec=0);
  // Конструктор с параметрами по умолчанию
  int getHour() const;
  Time operator++();
  //Префиксная форма инкремента
  Time operator++(int);
  //Постфиксная форма инкремента
};
#endif
```

Реализация функций класса `Time` в файле classTime.cpp:

```
//classTime.cpp
#include "classTime.h"
Time::Time(int _hour, int _min, int _sec) :hour(_hour),
min(_min), sec(_sec){}
int Time::getHour() const{
  return hour;
}
Time Time::operator++() {
  increment();
  return *this;
}
// фиктивный целый параметр не имеет имени
Time Time::operator++(int) {
  Time temp(*this);
  increment();
  return temp;
}
// Функция-утилита увеличения времени на 1 секунду
void Time::increment() {
  sec++;
  if (sec == 60) {
    sec = 0;
    min++;
  }
  if (min == 60) {
    min = 0;
    hour++;
  }
}
```

В данном классе используется конструктор с параметрами по умолчанию.

☠ Нельзя в классе объявлять одновременно конструктор по умолчанию и конструктор со всеми параметрами по умолчанию. Это приведет к ошибке. Компилятор не сможет определить, какой из конструкторов нужно вызвать.

Реализация конструктора использует инициализаторы элементов. В связи с этим тело конструктора пусто.

```
Time::Time(int _hour,int _min,int _sec):
           hour(_hour),min(_min),sec(_sec){}
```

В общем случае такой способ не является обязательным, но в некоторых ситуациях без него не обойтись. Например, константные элементы класса должны получать начальные значения с помощью инициализаторов элементов. Присваивания в теле конструктора в этом случае недопустимы.

Член-функцию класса можно объявить константной, если она не должна изменять значение полей объекта. Например,

```
int Time::getHour() const{
  return hour;
}
```

☺ Рекомендуется все функции для доступа к полям на чтение (`getXX`-функции) объявлять константными.

Это становится важным, если мы хотим объявить объект-константу для данного класса. Такой объект может использовать только константные член-функции.

Например, можно объявить объект-константу для некоторого эталона времени

```
const Time X(12,0,0);
```

Для объекта `X` можно использовать только функцию `getHour()`.

 Уберите квалификатор `const` из функции `getHour()` и попробуйте использовать его для константного объекта.

При перегрузке операции инкремента (декремента) для получения возможности использования и префиксной, и постфиксной форм, каждая из этих двух перегруженных функций-операций должна иметь разную сигнатуру. Это даст возможность компилятору определить, какая версия инкремента имеется в виду в каждом конкретном случае.

Допустим, мы объявили объект `t` типа `Time`:

```
Time T;
```

По соглашению, принятому в C++, когда компилятор встречает выражение с префиксным инкрементом ++t, генерируется вызов функции-элемента t.operator++() , объявление которой должно иметь вид:

```
Time operator++();
```

Когда компилятор встречает выражение постфиксной формы инкремента t++, он генерирует вызов функции t.operator++(0), объявлением которой является:

```
Time operator++(int);
```

Нуль (0) в генерируемом вызове функции является чисто формальным значением, введенным для того, чтобы сделать список аргументов функции operator++, используемой для постфиксной формы инкремента, отличным от списка аргументов функции operator++, используемой для префиксной формы инкремента. Заметьте, что формальный параметр является фиктивным, и поэтому не имеет имени.

Обе формы перегрузки операции ++ используют private функцию increment. Это связано с нетривиальным алгоритмом увеличения времени на одну секунду.

```
// Функция-утилита увеличения времени на 1 секунду
void Time:: increment() {
  sec++;
  if (sec==60) {
    sec=0;   min++;
  }
  if (min==60) {
    min=0; hour++;
  }
}
```

Перегруженная операция префиксного инкремента возвращает копию текущего объекта с измененным временем. Это происходит потому, что текущий объект *this возвращается как объект класса Time, что активизирует конструктор копии.

```
Time Time::operator++() {
  increment();
  return *this;
}
```

Чтобы эмулировать действие постфиксного инкремента, необходимо вернуть неизмененную копию объекта `Time`. При входе в `operator++` текущий объект `*this` сохраняется во временном объекте `temp`. Затем вызывается `increment()`, чтобы инкрементировать объект. В итоге, возвращается неизмененная копия объекта `temp`.

```
// фиктивный целый параметр не имеет имени
Time Time::operator++(int) {
  Time temp(*this);
  increment();
  return temp;
}
```

Отметим, что эта функция не может вернуть ссылку на объект класса `Time`, потому что значение, которое надо вернуть, сохраняется в локальной переменной в определении функции. Локальные переменные уничтожаются, когда функция, в которой они объявлены, завершена. Таким образом, объявление типа, возвращаемого функцией, как `Time&`, привело бы к ссылке на объект, который после возвращения больше не существует.

☠Возвращение ссылки на локальную переменную является типичной ошибкой, которую трудно найти.

Пример функции `main` для класса `Time`:

```
#include <iostream>
#include "classTime.h"
using namespace std;
void main () {
  Time t1; Time t2(11,59,59);
  cout<<t1.getHour()<<endl;
  t1++;
  cout<<t1.getHour()<<endl;
  cout<<t2.getHour()<<endl;
  t2++;
  cout<<t2.getHour()<<endl;
}
```

1.6. Реализация преобразования типов

В языке C++ определены операции преобразования между встроенными типами данных. А преобразования между встроенными типами и типами, определенными пользователями, или только между типами, определенными пользователями, требуют определения. Для этого используются или конструкторы преобразований, или операции преобразований. Выбор механизма реализации преобразования зависит от типов данных, между которыми производится преобразование.

Конструктор преобразования (конструктор с единственным аргументом) может быть использован для преобразования объектов разных типов (включая встроенные типы) в объекты данного класса.

Операция преобразования (называемая также *операцией приведения*) может быть использована для преобразования объекта одного класса в объект другого класса или в объект встроенного типа. Такая операция преобразования должна быть нестатической функцией-членом. Операция преобразования этого вида не может быть дружественной функцией.

Пример 6. Реализовать класс «Строка». Предусмотреть конструктор для преобразования строк в стиле C `char*` в объекты класса «Строка» и операцию приведения для преобразования класса «Строка» в строку в стиле C `char*`. Требования к классу представлены на UML диаграмме (рис. 1.3).

```
//STR.h
#ifndef STR_H
#define STR_H
#include <iostream>
using namespace std;
class Str{
  friend ostream & operator << (ostream &, const Str &);
  friend istream & operator >> (istream &, Str &);
public:
  Str(const char * = "");     //конструктор преобразования
```

```
  Str(const Str &);           //конструктор копии
  ~Str();                     //деструктор
  Str & operator= (const Str &);   //присваивание
  Str & operator+= (const Str &); //сцепление (конкатенация)
  bool operator!() const;          //проверка строки на пустоту
  bool operator== (const Str &) const;//проверка на равенство
  char & operator[] (int);    //получение ссылки на символ
  char operator[] (int) const; //получение символа по номеру
  Str operator() (int, int);  //получение подстроки
  int getLength() const;      //определение длины строки
  operator char*() const;     //операция приведения к char*
private:
  char * sPtr;      //указатель на начало строки
  int length;       //длина строки
};
#endif
```

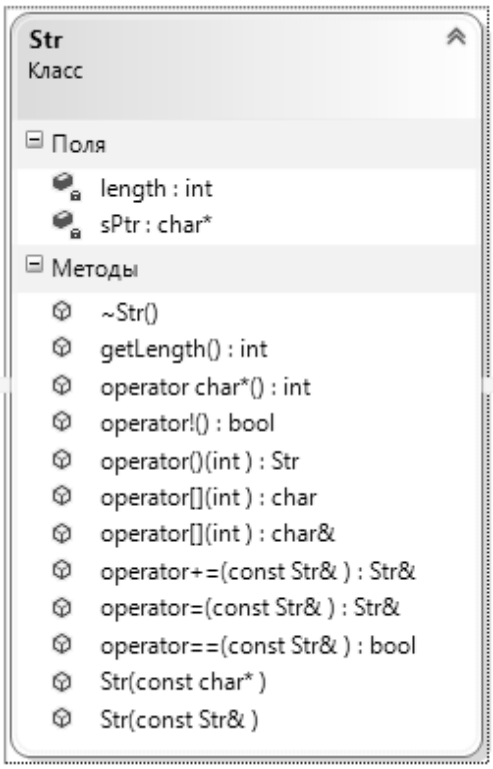


Рис.1.3. UML диаграмма класса `Str`

```
//STR.cpp
// Определение некоторых функций элементов класса Str
#include <iostream>
#include <cstring>
#include <cassert>
#include <iomanip>
#include "str.h"
```

```
// конструктор преобразования: преобразовывает char* в Str
Str::Str(const char* s) {
  length=(int)strlen(s);
  sPtr=new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  strcpy(sPtr,s);
}

// конструктор копии
Str::Str(const Str & copy) {
  length=copy.length;
  sPtr=new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  strcpy(sPtr,copy.sPtr);
}

// деструктор
Str::~Str() { delete [] sPtr; }

//операция присваивания
Str & Str::operator =(const Str & right) {
  if (&right != this) {
    delete [] sPtr;
    length = right.length;
    sPtr = new char[length + 1];
    assert(sPtr != 0);
    strcpy(sPtr, right.sPtr);
  }
  return *this;
}

//сцепление (конкатенация)
Str & Str::operator +=(const Str & right) {
  char * tempPtr = sPtr;
  length += right.length;
  sPtr = new char [length+1];
  assert(sPtr!=0);
  strcpy(sPtr, tempPtr);
  strcat(sPtr, right.sPtr);
  delete [] tempPtr;
  return *this;
}

//проверка строки на пустоту
bool Str::operator !() const {
  return length ==0;
}
```

```
//проверка двух строк на равенство
bool Str::operator == (const Str & right) const {
  if (length!=right.length)
    return 0;
  return strcmp(sPtr, right.sPtr) == 0;
}

//получение ссылки на символ
char & Str::operator[] (int ind) {
  //проверка, не находится ли индекс вне диапазона
  assert(ind >= 0 && ind < length);
  return sPtr[ind];      //создание L-величины
}

//получение символа
char Str::operator[] (int ind) const{
  //проверка, не находится ли индекс вне диапазона
  assert(ind >= 0 && ind < length);
  return sPtr[ind];      //создание R-величины
}

//получение подстроки заданной длины, начинающейся с заданного
//индекса
Str Str::operator() (int ind, int sublength) {
  //проверка, что индекс в диапазоне и длина подстроки >=0
  assert(ind >= 0 && ind < length && sublength>=0);
  Str sub;
  //определение длины подстроки
  if ((sublength==0) || (ind+sublength>length))
    sub.length=length-ind+1;
  else
    sub.length=sublength+1;
  //выделение памяти для подстроки
  sub.sPtr=new char[sub.length];
  assert(sub.sPtr!=0);
  //копирование подстроки в новый Str
  strncpy(sub.sPtr,&sPtr[ind],sub.length);
  sub.sPtr[sub.length]='\0';  //завершение новой строки sPtr
  return sub;
}
//определение длины строки
int Str::getLength() const {
  return length;
}

//операция приведения к char*
Str::operator char*() const {
  char *c=new char[length];
```

```
  assert(sPtr != 0);
  strcpy(c,sPtr);
  return c;
}

//перегруженная операция вывода данных
ostream &operator << (ostream & output, const Str & s) {
  output<<s.sPtr;
  //возврат ссылки на поток для возможности многократного
  //последовательного использования операции <<
  return output;
}

//перегруженная операция ввода данных
istream &operator >> (istream & input, Str & s) {
  char temp[100];   //буфер для хранения входных данных;
  input>>setw(100)>>temp;
  s=temp;
  return input;
}

// String.cpp
#include "Str.h"
int main() {
  char *s=new char[100];
  cin>>s;
  Str s1("Hello"),s2(s), s3;
  cin>>s3;
  cout<<s1<<endl;
  cout<<s2<<endl;
  cout<<s3<<endl;
  s=s3;
  cout<<s<<endl;
  cin>>s;
  return 0;
}
```

Объявление перегруженной функции-операции приведения типа из определенного пользователем типа `Str` в тип `char*`:

```
operator char*() const;
```

Перегруженная функция-операция приведения не указывает тип возвращаемой величины, потому что им является тип, к которому преобразован объект.

Если s – объект класса Str, то когда компилятор встречает выражение (char*)s, он порождает вызов s.operator char*(). Для этого вызова операнд s – это объект класса Str, для которого была активизирована функция-элемент operator char*.

Конструктор преобразования получает аргумент char* и создает объект Str

```
Str(const char * = "");
```

Конструктор с одним параметром char* преобразует соответствующую строку в объект Str, который затем присваивается создаваемому объекту Str.

 Любой конструктор с единственным аргументом можно рассматривать как конструктор преобразования.

Одной из особенностей операций приведения и конструкторов преобразований является то, что при необходимости компилятор может вызывать эти функции автоматически для создания временных объектов.

Если в программе в том месте, где ожидается char*, появился объект s определенного пользователем типа Str, то в этом случае компилятор для преобразования объекта в char* вызывает перегруженную функцию-операцию приведения operator char* и использует в выражении результирующий char*. Например:

```
cout << s;
```

Поэтому операция приведения для класса Str позволяет не перегружать операцию << (поместить в поток), предназначенную для вывода Str с использованием cout.

 Уберите из примера 5 перегрузку операции вывода в поток и проверьте работоспособность программы.

Наличие конструктора преобразования означает, что нет необходимости применять перегруженную операцию специально для присваивания строк символов объектам `Str`. Компилятор автоматически активизирует конструктор преобразования для создания временного объекта `Str`, содержащего строку символов. Затем активизируется перегруженная операция присваивания, чтобы присвоить временный объект другому объекту `Str`.

 Некоторые преобразования типов могут быть источниками ошибок. Например, при преобразовании от вещественного типа к целому может быть получен некорректный результат, хотя преобразование из целого типа в вещественный всегда гарантирует правильный результат. Поэтому преобразование из целого типа в вещественный является безопасным, а обратное — небезопасным. Для выполнения небезопасных преобразований типов используется операция явного преобразования [2, раздел 4.3].

Поскольку любой конструктор с одним параметром будет использоваться всегда по умолчанию для неявного преобразования типа, то для запрета неявного преобразования необходимо дополнить такой конструктор модификатором `explicit`.

В классе `Str` конструктор с параметром `char *` использовался специально для демонстрации возможности неявного преобразования. Поэтому он был описан без модификатора `explicit`.

Чтобы пояснить ситуацию, когда необходимо использовать модификатор `explicit`, рассмотрим следующий пример. Допустим, необходимо к строке, представляемой классом `Str`, дописать константную строку, содержащую число, например, номер курса. Сделаем это, используя перегруженную операцию +=.

```
Str s1("Hello");
s1+="2";      // s1 содержит Hello2
```

Предположим, при наборе кода программы сделана ошибка — оказались пропущены кавычки.

```
s1+=2;       // сообщение об ошибке компиляции
```

В нашем примере произойдёт ошибка компиляции, т.к. компилятор не найдет перегруженной операции += с правым операндом целого типа и не определено приведение целого типа к типу `Str`.

Если бы в классе `Str` был определён конструктор преобразования из типа `int`, такая операция была бы разрешена и не ошибка компиляции не возникла бы. Предположим, что конструктор с одним параметром типа `int` существует и предназначен для формирования строки с заданным количеством пробелов.

```
Str(int n);    //конструктор неявного преобразования
Str::Str(int n) {
  length = n;
  sPtr = new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  for (int i = 0; i < length; ++i)
    sPtr[i] = ' ';
  sPtr[length] = 0;
}
```

По правилам языка C++ он является конструктором преобразования типа, но его назначением является формирование пробельной строки заданной длины. В этом случае результат операции с пропущенными кавычками окажется неожиданным

```
s1+=2;       // s1 содержит Hello и еще два пробела
```

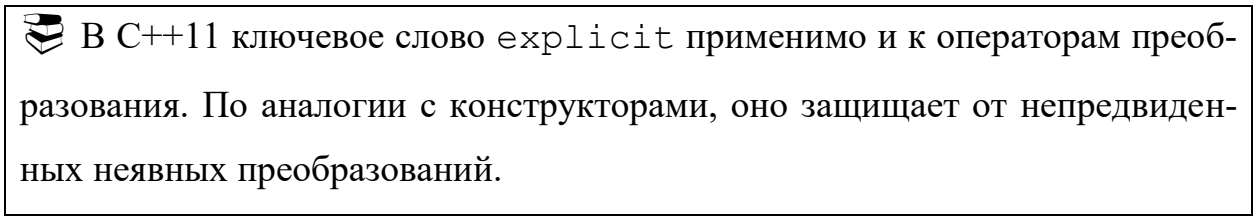
Именно поэтому следовало указать компилятору, что мы не хотим рассматривать такой конструктор как конструктор неявного преобразования. В этом случае конструктор нужно объявить с модификатором `explicit`.

```
explicit Str(int n);
//конструктор преобразования, вызываемого явно
Str::Str(int n) {
  length = n;
```

```
  sPtr = new char[length + 1];
  assert(sPtr != 0);     //завершение, если память не выделена
  for (int i = 0; i < length; ++i)
    sPtr[i] = ' ';
  sPtr[length] = 0;
}
```

При использовании `explicit` конструктора будут получены следующие результаты:

```
s1+=2;       // сообщение об ошибке компиляции
s1+=Str(2);       //s1 содержит Hello и еще два пробела
```



В C++11 ключевое слово `explicit` применимо и к операторам преобразования. По аналогии с конструкторами, оно защищает от непредвиденных неявных преобразований.

1.7. Обработка исключений

Из-за большого количества недиагностируемых ошибок времени выполнения при работе со строками `char*` в примере 6 используется макрос `assert`.

```
void assert(int выражение)
```

Он позволяет включить в программу диагностические сообщения о таких ошибках. Например,

```
assert(sPtr != 0);  //завершение, если память не выделена
```

или

```
assert(ind >= 0 && ind < length && sublength>=0);
//проверка, что индекс находится в диапазоне и длина подстроки >=0
```

Если значение `выражения` есть нуль, то `assert(выражение)` напечатает сообщение следующего вида:

```
Assertion failed: выражение, file имя файла, line nnn
```

После чего будет вызвана функция `abort`, которая завершит вычисления.

Основное применение макроса `assert` — диагностика ошибок на этапе отладки приложения. В частности, такой механизм широко применяется при создании тестов, в том числе и unit-тестов.

Но в некоторых случаях ошибки времени выполнения могут возникать и в отлаженной программе. Например, ошибки при выделении динамической памяти или несоответствия типов входных данных. Для обработки таких ситуаций используется механизм исключений.

Когда во время выполнения программы происходит ошибка, генерируется так называемое *исключение (исключительная ситуация)*, которое можно *перехватить* и *обработать*. Если исключение не обработать, то программа завершится с ошибкой.

Если в программе возникла исключительная ситуация и в текущем контексте не хватает информации для принятия решения о том, как действовать дальше, информацию об ошибке можно передать во внешний, более общий контекст. Для этого в программе создается объект с информацией об исключении, который затем «запускается» из текущего контекста (говорят, что в программе *запускается исключение*).

Для работы с исключениями можно использовать как встроенные типы [2, раздел 1.15], так и создаваемые пользователями классы исключений.

В простейшем случае класс исключения может быть пустым. Этого достаточно, если нам нужно только просигнализировать о возникновении ситуации и никакие данные передавать обработчику не нужно.

Например, создадим класс, описывающий ошибку, возникающую при невозможности выделить динамическую память.

```
class OutOfMemoryException{};
```

Чтобы воспользоваться классом исключения нужно `assert` заменить запуском исключения.

```
//assert(sPtr!=0);
if (sPtr==0) throw OutOfMemoryException();
```

В этом случае определение перегруженной операции конкатенации будет выглядеть следующим образом:

```
//сцепление (конкатенация)
Str & Str::operator +=(const Str & right) {
  char * tempPtr = sPtr;
  length += right.length;
  sPtr = new char [length+1];
  if (sPtr==0)
    throw OutOfMemoryException();
  strcpy(sPtr, tempPtr);
  strcat(sPtr, right.sPtr);
  delete [] tempPtr;
  return *this;
}
```

Оператор генерации исключения `throw` производит целый ряд действий. Сначала создаётся копия запускаемого объекта, которая возвращается из функции, содержащей `throw`, даже если тип объекта исключения не соответствует типу, который положено возвращать этой функции. При этом управление передается в специальную часть программы, называемую *обработчиком исключения*; она может находиться далеко от того места, где было запущено исключение. Также уничтожаются все локальные объекты, созданные к моменту запуска исключения. Автоматическое уничтожение локальных объектов называется «раскруткой стека».

 Создайте класс исключения `BadIndexException` для контроля выхода индекса за границы. Замените все использования `assert` генерацией исключений соответствующих типов.

Поскольку ситуация невозможности выделить память встречается крайне редко, рассмотрим ошибку, связанную с индексацией. После проведённых изменений в программе неправильное значение индекса и в случае использования макроса `assert`, и в случае оператора `throw` приводит к

аварийному завершению программы. Различаются только сообщения, описывающие причину завершения программы. Но в отличие от использования `assert` сгенерированные исключения можно перехватывать и обрабатывать в программе без обязательного аварийного завершения.

В том случае, когда команда `throw` не должна приводить к аварийному завершению, следует создать специальный блок, который должен реагировать на исключение. Этот блок, называемый блоком `try` , представляет область видимости, перед которой ставится ключевое слово `try`:

```
try {
  // Программный код, который может генерировать исключения
}
```

При использовании обработки исключений выполняемый код помещается в блок `try`, а обработка исключений производится после блока `try`. Это существенно упрощает написание и чтение программы, поскольку основной код не смешивается с кодом обработки ошибок. Часто не сам код, который может генерировать исключения, а вызов функции, содержащей этот код, помещают в блок `try`.

Программа должна где-то среагировать на запущенное исключение. Это место называется *обработчиком исключения*. В программу необходимо включить обработчик исключения для каждого типа перехватываемого исключения. Обработчик может перехватывать как определенный тип исключения, так и исключения классов, производных от этого типа.

Обработчики исключений следуют сразу же за блоком `try` и обозначаются ключевым словом `catch`:

```
try {
  // Программный код, который может генерировать исключения
}
catch (type1 id1){
  // Обработка исключений типа type1
}
...
```

```
catch (typeN idN){
  // Обработка исключений типа typeN
}
//Здесь продолжается нормальное выполнение программы...
```

Если id1, ..., idN не нужны, их можно опускать.

Если в программе запускается исключение, механизм обработки исключений начинает искать первый обработчик с аргументом, соответствующим типу исключения. Управление передается в найденную секцию catch, и исключение считается обработанным (т. е. дальнейший поиск обработчиков прекращается). Выполняется только нужная секция catch, а работа программы продолжается, начиная с позиции, следующей за последним обработчиком для данного блока try.

Иногда требуется написать обработчик для перехвата любых типов исключений. Для этой цели используется специальный список аргументов в виде многоточия (...):

```
catch (...){
  cout<<"an exception was thrown"<<endl;
}
```

Поскольку такой обработчик перехватывает все исключения, он размещается в конце списка обработчиков.

Рассмотрим использование в операции индексирования исключения BadIndexException для контроля выхода индекса за границы строки.

```
//получение ссылки на символ
char & Str::operator[] (int ind) {
  //проверка, не находится ли индекс вне диапазона
  if (ind < 0 || ind >= length)
    throw BadIndexException();
  return sPtr[ind];      //создание L-величины
}
```

Тип исключения обычно предоставляет достаточно информации для определения характера ошибки, например:

```
Str s="New string for tests";
for (int i=0; i<10; ++i){
  cout<<"Input number of symbol"<<endl;
```

```
  cin>> i;
  try {
    cout<<s[i];
  }
  catch (BadIndexException){
    //обработка исключения
  .. cout<<"illegal index"<<endl;
  }
}
```

Рассмотренный цикл при любых входных данных будет выполнен 10 раз, но в случаях выхода за границы вместо символа строки будет выведено сообщение: illegal index.

Иногда кроме типа исключения обработчику желательно передавать дополнительную информацию, например, значение индекса, вызвавшего исключение. Тогда для исключений нужно использовать классы, имеющие конструктор с параметрами, поля и методы.

Расширим класс `BadIndexException`, чтобы можно было передавать значение индекса в обработчик.

```
class BadIndexException {
  private:
    int ind;
  public:
    BadIndexException(int i): ind(i) {}
    int getInd() const {
      return ind;
    }
};
```

Теперь при выбрасывании исключения нужно указать значение некорректного индекса.

```
throw BadIndexException(ind);
```

Тогда обработчик будет выглядеть следующим образом:

```
  catch (BadIndexException e){
    //обработка исключения
  .. cout<<"illegal index"<<e.getInd()<<endl;
  }
```

Механизм исключений в первую очередь предназначен для разделения места, где возникает ошибка, и места, где она обрабатывается. Поэтому

чаще всего исключения, выбрасываемые в функциях, обрабатываются там, где эти функции вызываются. При этом такие функции могут входить в состав библиотек, поставляемых сторонними разработчиками. При этом могут быть известны только заголовки функций. Чтобы сообщить, какие исключения может выбрасывать функция, в её объявление можно добавить список возможных исключений.

```
char & operator[] (int ind) throw (BadIndexException);
```

В объявлении функции после списка аргументов указана необязательная *спецификация исключений*. Спецификация исключений состоит из ключевого слова `throw`, за которым в круглых скобках перечисляются типы всех потенциальных исключений, которые могут запускаться данной функцией.

☺ Вообще говоря, вы не обязаны сообщать пользователям вашей функции, какие исключения она может запускать. Однако такое поведение считается нецивилизованным – оно означает, что пользователи не будут знать, как написать код перехвата потенциальных исключений.

1.8. Статические члены класса

Классы могут содержать статические поля и статические функции. Если данные-члены объявлены с квалификатором `static`, то для всех объектов класса поддерживается только одна копия таких данных. Статический член используется совместно всеми объектами данного класса. Для того чтобы существовал статический член, не обязательно, чтобы существовали объекты такого класса.

Пример 7. Реализовать класс, позволяющий подсчитывать количество существующих объектов данного класса.

```
class Counter {
private:
```

```
  static int count;
public:
  static int getCount() {
    return count;
  }
  Counter() {
    ++count;
  }
  ~Counter() {
    --count;
  }
};

int Counter::count = 0;

int main() {
  cout << "before " << Counter::getCount() << endl;
  Counter first;
  cout << "after first " << first.getCount() << endl;
  Counter * second = new Counter();
  cout << "after second " << second->getCount() << endl;
  delete second;
  cout << "after delete second " << Counter::getCount() << endl;
  return 0;
}
```

Внутри класса возможно только объявление статического члена, но не его определение.

```
private:
  static int count;
```

Статические данные можно использовать только после определения. Это делается путем нового объявления статической переменной, причем используется оператор разрешения области видимости для того, чтобы идентифицировать тот класс, к которому принадлежит переменная.

```
int Counter::count = 0;
```

Статические члены-данные подчиняются правилам доступа к членам класса. Однако определение статических членов-данных выполняется вне зависимости от спецификатора доступа.

☺ Определение статических членов-данных класса рекомендуется располагать вместе с определением самого класса в заголовочном файле.

К статическим членам класса можно обращать как через класс, так и через объект или указатель/ссылку на объект.

```
cout << "before " << Counter::getCount() << endl;
Counter first;
cout << "after first " << first.getCount() << endl;
Counter * second = new Counter();
cout << "after second " << second->getCount() << endl;
```

Статические функции-члены не могут обращаться к нестатическим данным или вызывать нестатические функции этого же класса. Это связано с тем, что в статические функции не передается указатель `this` на объект, для которого вызвана функция.

Реализация статической функции записывается так же, как и реализация любой другой функции-члена класса. При этом если реализация вне класса, то ключевое слово `static` не указывается.

Например, определение статической функции `getCount` вне класса могло бы выглядеть следующим образом

```
int Counter::getCount() {
    return count;
}
```

1.9. Члены класса создаваемые автоматически

Поскольку некоторые члены класса создаются компилятором автоматически, следует понимать принципы их создания и знать ситуации, в которых они создаются. Это позволит в одних случаях избежать ошибок, а в других – эффективно использовать действия по умолчанию.

Рассмотрим следующий «не очень полезный» класс.

```
class Empty {};
```

На самом деле, такое описание класса эквивалентно следующему:

```
class Empty{
public:
  Empty() {}
  Empty(Empty const &) {}
  Empty & operator = (Empty const &) {}
  ~Empty() {}
};
```

Здесь присутствуют 4 функции:

`Empty()` — конструктор без параметров,

`Empty(Empty const &)` — конструктор копий,

`Empty & operator = (Empty const &)` — операция копирующего присваивания,

`~Empty()` — деструктор.

Если класс не имеет членов-данных, то эти функции не выполняют никаких действий.

Напомним, что класс для описания исключения часто можно оставить пустым, используя принцип определения по умолчанию. И тогда его описание будет выглядеть именно таким образом.

Добавим член класса `value` в определение «не очень полезного» пустого класса.

```
class NotQuiteEmpty{
public:
  int value;
};
```

У классов, имеющих члены-данные, конструктор без параметров и деструктор, создаваемые по умолчанию, остаются пустыми. Конструктор копии, создаваемый по умолчанию, выполняет побитовое копирование членов-данных. Операция присваивания, создаваемая по умолчанию, выполняет побитовое присваивание.

Принцип создания функций-членов класса по умолчанию состоит в следующем: если любая из перечисленных функций, кроме конструктора без параметров, явно не задана, она создаётся неявно по умолчанию. Прин-

цип создания функций-членов класса по умолчанию для конструктора без параметров имеет особенность. Такой конструктор создается неявно, только если не задан явно ни один конструктор. Если в классе будет явно описан хотя бы один конструктор, неявного создания конструктора без параметров не произойдёт. А это в дальнейшем при использовании класса может привести к ошибкам.

На практике, иногда, создаваемые неявно функции могут привнести нежелательную функциональность, от которой нужно избавиться. Например, нужно создать класс, для которого должен существовать только один экземпляр. Такие классы называются синглетонами. Для них рекомендуется запретить операцию присваивания и конструктор копирования. С другой стороны, рекомендуется наряду с другими конструкторами всегда иметь конструктор без параметров, даже если его тело является пустым.

В C++ предусмотрен механизм управления генерацией стандартных конструкторов и функций. Рассмотрим его на примере класса A:

```
class A {
private:
    int x;
public:
    A(int i) {…}

    // Данная запись указывает на необходимость сгенерировать
    // конструктор по умолчанию
    A() = default;

    // Запретить генерацию конструктора копии по умолчанию
    A(const A&) = delete;

    // А так можно запретить генерацию operator=
    A& operator = (const A&) = delete;
}
```

1.10. Семантика перемещения: move-конструктор и move-operator=

Семантика перемещения появилась в стандарте C++11. Она нацелена на уменьшение количества создаваемых копий объектов при выполнении конструктора копии и операции присваивания, которые вызываются для rvalue выражений. Любое выражение в C++ является или левосторонним (lvalue), или правосторонним (rvalue). Выражение lvalue — это объект, который имеет имя. Все переменные являются lvalue. А выражение rvalue — это временный безымянный объект, не существующий за пределами того выражения, которое его создало. Более подробно rvalue и lvalue выражения в C++ рассматривались в [2, раздел 1.13].

Для иллюстрации проблемы лишних копий рассмотрим упрощённую реализацию класса для динамического массива.

Пример 8. Реализовать move-семантику на примере упрощённого класса для динамического массива.

Для этого достаточно описать конструктор с параметрами по умолчанию и конструктор копии, деструктор и перегрузить операцию сложения двух массивов одинакового размера и операцию присваивания. Чтобы отслеживать процесс создания и удаления объектов, в классе добавлены два поля:

- ✓ поле `name` — имя, отражающее способ создания объекта;
- ✓ статическое поле `f` для подсчёта существующих в текущий момент объектов. Значение поля `f` увеличивается каждый раз при выполнении операции `new` и уменьшается при выполнении операции `delete`.

```
#include <iostream>
#include <string>
using namespace std;
class myvector {
private:
```

```
  static int f;
  string name;
  int size;
  int * vect;
public:
  myvector(int s = 1, string nm ="noname"): size(s), name(nm){
    f++;
    cout << "constructor > " << name<<" "<<f<< endl;
    vect = new int[s];
  }
  myvector(const myvector & v): size(v.size){
    f++;
    name = "Copy ( "+v.name+" )";
    cout << "copy > " << name << " " <<  f << endl;
    vect = new int[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }
  ~myvector(){
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
  myvector& operator= (const myvector &v){
    cout << name <<" operator= > " << v.name <<" "<< f << endl;
    if (&v != this) {
      delete[] vect;
      vect = new int[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }
  myvector operator+(const myvector& v){
    if (size == v.size) {
      myvector v1(size, name + " + "+v.name);
      for (int i = 0; i < size; ++i)
        v1.vect[i] = vect[i] + v.vect[i];
      return v1;
    }
    return *this;//лучше использовать исключение
  }
};
```

В следующей программе выражениями rvalue являются `a + b` внутри вызова конструктора копии для объекта `cc` и `a + D` в операторе при-

сваивания. Именно для них будут создаваться дополнительные временные объекты, которые после использования в конструкторе копии и операции присваивания тут же будут удалены.

```
#include "vect.h"
int myvector::f = 0;
int main() {
  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```

В окне вывода можно проследить последовательность вызовов конструкторов и деструкторов для данной программы (рис. 1.4).

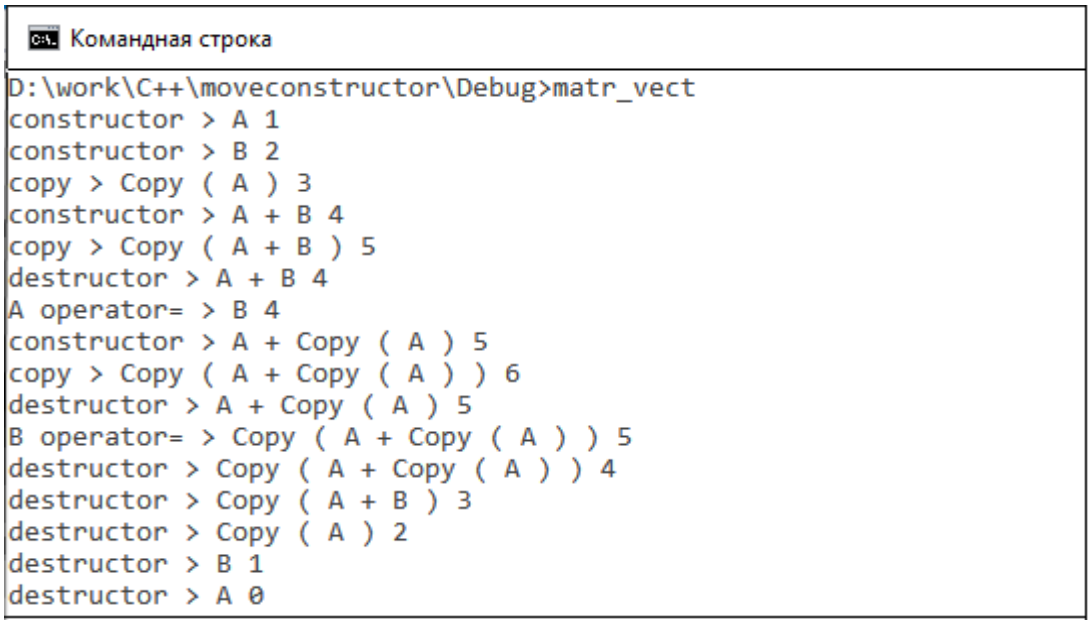
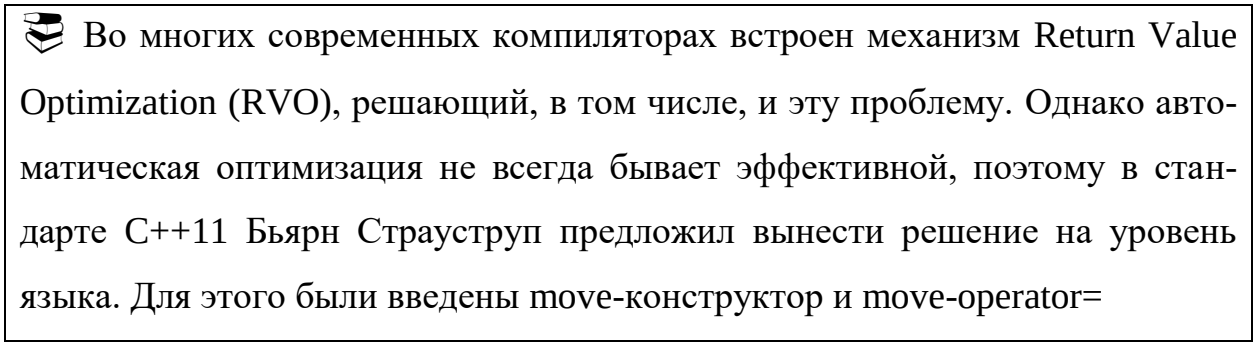


Рис.1.4. Последовательность вызовов конструкторов и деструкторов

Каждая строка содержит информацию о вызываемой функции (конструктор, деструктор, операция присваивания), имени объекта и количестве объектов, существующих в текущий момент. При этом можно заметить создание и уничтожение временных объектов, а также накладные расходы на копирование этих объектов.

Например, при создании объекта `myvector cc (a + b)` сначала создаётся временный объект при вычислении значения `a + b`. Затем вызывается конструктор копии `myvector(const myvector & v)`, в который в качестве значения передаётся, созданный ранее, временный объект с именем `A + B`. После копирования временный объект удаляется. В результате происходит лишнее выделение памяти и копирование данных одного и того же объекта.


Во многих современных компиляторах встроен механизм Return Value Optimization (RVO), решающий, в том числе, и эту проблему. Однако автоматическая оптимизация не всегда бывает эффективной, поэтому в стандарте C++11 Бьярн Страуструп предложил вынести решение на уровень языка. Для этого были введены move-конструктор и move-operator=

Идея move-семантики состоит в том, чтобы не удалять временный объект и не выделять память для полей в новом объекте, а инициализировать поля в создаваемом объекте ссылками на поля временного объекта.

Деструкторы для временных переменных вызываются в тот момент, когда эти переменные уже не используются для вычислений. Однако при использовании move-семантики деструктор не должен освобождать память временного объекта, поскольку ссылкой на нее инициализируется поле в другом объекте. Для этого в move-конструкторе и move-операции присваивания поле указатель временного объекта меняет значение на `nullptr`, а в деструкторе выделенная память освобождается только, если поле указатель не равен `nullptr`.

➢ Конструктор копирования выделяет новую область памяти для хранения данных, вызывая *оператор new*, а перемещающий конструктор — забирает данные у переданного ему временного объекта.

Добавим в класс move-конструктор, move-операцию присваивания и внесём изменения в деструктор.

```
//move-конструктор
myvector(myvector&& v){
  name = "Move_Copy ( " + v.name+" )";
  cout << "move > " << name << " " << f << endl;
  size = v.size;
  vect= v.vect;
  // Не позволит сразу удалить временный объект
  v.vect = nullptr;
}

//измененный деструктор
~myvector(){
  if (vect != nullptr) {
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
}

//move-операция присваивания
myvector& operator=(myvector&& v){
  cout << name <<" operator-move= > "<< v.name <<" "<< f <<endl;
  if (vect != nullptr)
    delete[] vect;
  size = v.size;
  vect = v.vect;
  v.vect = nullptr;
  return *this;
}
```

Чтобы отличать функции с перемещающей семантикой в стандарте C++11 введены rvalue ссылки — `myvector && v`. При этом компилятор будет использовать функции с перемещающей семантикой только в случае, если параметром является rvalue выражение (временный объект).

Теперь, при выполнении той же самой программы, для строки `myvector cc(a+b);` компилятор выберет move-конструктор вместо конструктора копии. А для строки b = a + D; компилятор выберет move-операцию присваивания. Результат выполнения приведен на рисунке 1.5.

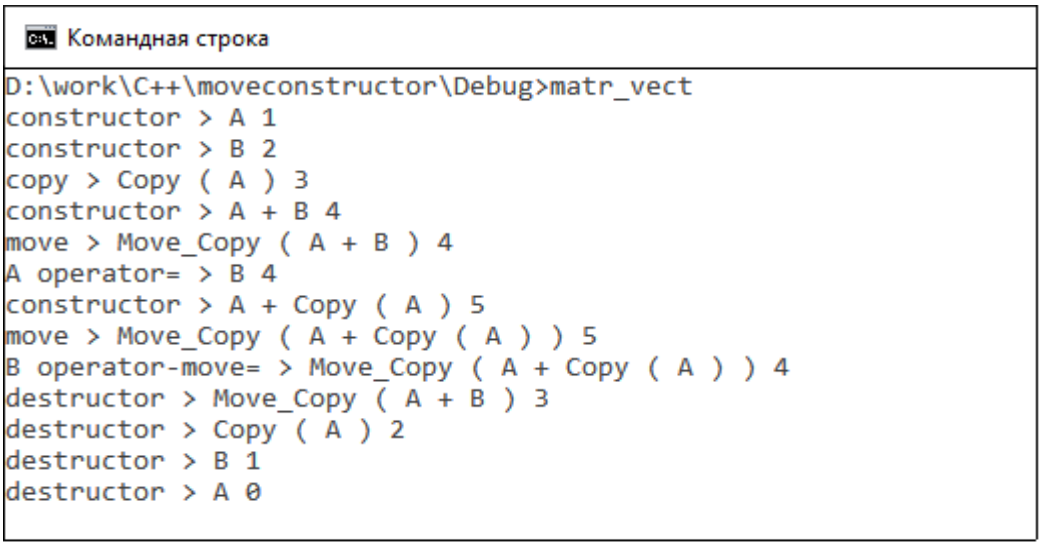


Рис.1.5. Последовательность вызовов для move-семантики

☺ Вследствие наличия большого количества стандартных классов использовать move-конструкторы и move-операции присваивания приходится редко, однако знание такого механизма необходимо.

ГЛАВА 2. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

2.1. Наследование классов

В языке C++ абстракция данных осуществляется с помощью классов, инкапсулирующих типы данных и операции над ними. Однако объектно-ориентированный подход выходит за рамки простой инкапсуляции. Применяя наследование и полиморфизм, в языке C++ можно на основе существующих классов создавать новые.

> 🕮 *Наследование* (inheritance) – это способность класса приобретать свойства ранее определенного класса. Один класс может наследовать структуру и поведение другого класса.

В языке C++ производный класс наследует все члены базового класса, за исключением конструкторов, деструктора, перегруженной операции присваивания и определения друзей класса. Таким образом, производный класс содержит в себе все данные-члены и функции-члены базового класса, добавляя к ним новые члены, определенные в нём самом. Кроме того, производный класс может изменять (переопределять) любую наследуемую функцию-член.

При использовании механизма наследования в описании класса появляется новый раздел – защищённый (protected). Члены, помещённые в защищённый раздел, доступны из функций классов-наследников, но скрыты вне определения класса. Общие правила доступа относительно разделов класса в языке C++ представлены в табл. 7.

Определение производного класса начинается с указания типа наследования – `public`, `protected` или `private` и имени базового класса:

```
class <имя производного класса>:
                {public|protected|private} <имя базового класса>
```

Таблица 7

Правила доступа

Раздел базового класса	Открытый (public)	Защищенный (protected)	Закрытый (private)
Доступность из функций базового класса, функций дружественных классов и из дружественных функций	Да	Да	Да
Доступность из функций классов-наследников базового	Да	Да	Нет
Доступность из других классов и функций	Да	Нет	Нет

Открытое наследование. Открытые и защищенные члены базового класса остаются, соответственно, открытыми и защищенными членами производного класса.

Защищенное наследование. Открытые и защищенные члены базового класса становятся защищенными членами производного класса.

Закрытое наследование. Открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

Среди указанных видов наследования открытое наследование является наиболее важным и часто используемым.

2.2. Открытое наследование

Открытое наследование устанавливает между классами отношение «является», или в английской нотации «is-a».

При открытом наследовании все, что характеризует объекты класса-предка, является справедливым и для объектов класса-наследника. Это свойство называется *совместимостью типов* объектов. Благодаря этому объект производного класса можно применять вместо объекта базового класса, но не наоборот.

Например, объекту базового типа можно присвоить объект производного типа, указателю на объект базового типа – указатель на объект произ-

водного. Объект производного типа также может быть передан в качестве параметра в функцию, вместо объекта базового типа.

Пример 9. Реализовать иерархию классов `Sphere` («сфера») – `Ball` («мяч»), которая представлена на UML диаграмме (рис. 2.1).

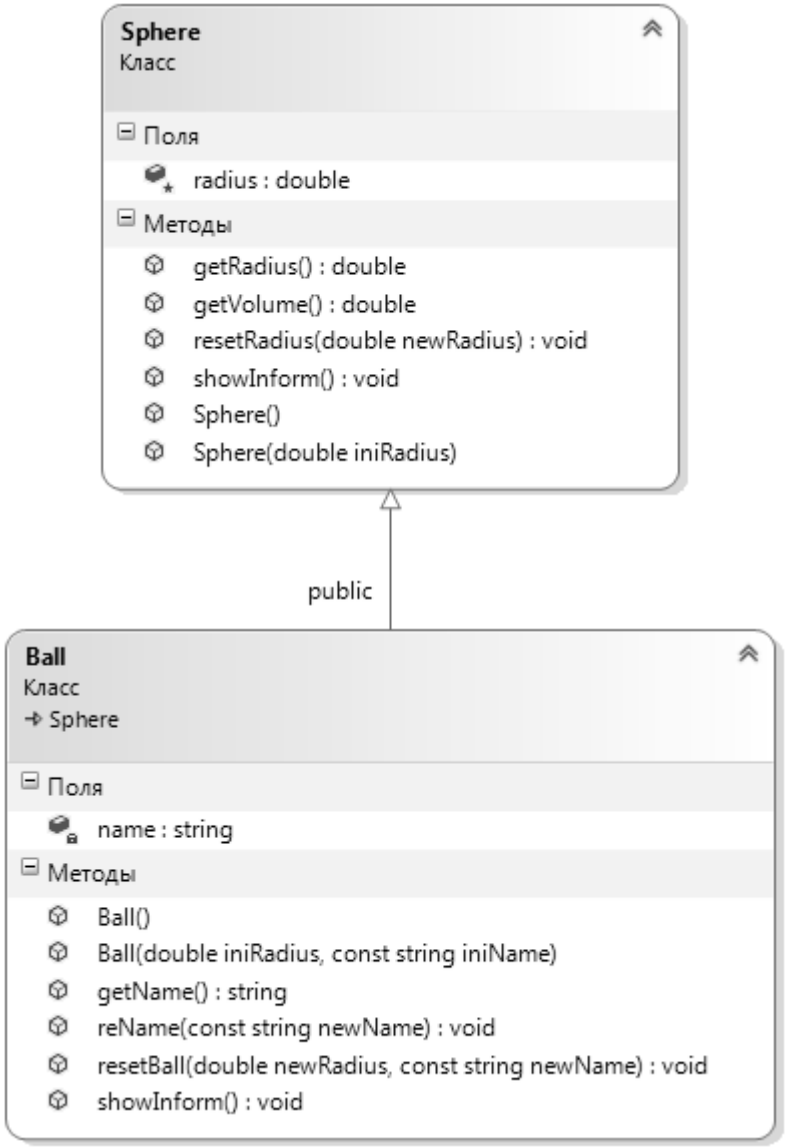


Рис. 2.1. UML диаграмма иерархии классов `Sphere` – `Ball`

Определение класса «сфера» – файл sphere.h.

```
#ifndef SPHERE_H
#define SPHERE_H
class Sphere{
  public:
    //Конструкторы
    Sphere();
```

```
    Sphere(double iniRadius);
    // Операции
    void resetRadius(double newRadius);
    double getRadius();
    double getVolume();
    void showInform();
  protected:
    double radius; //радиус сферы
};
#endif
```

Реализация класса «сфера» – файл sphere.cpp

```
#include "sphere.h"
#include <iostream>
#include <cmath>
using namespace std;

Sphere::Sphere(): radius(1.0){ }
Sphere::Sphere(double iniRadius){
  if (iniRadius>0)
    radius=iniRadius;
  else
    radius=1.0;
}
void Sphere::resetRadius(double newRadius){
  if (newRadius>0)
    radius=newRadius;
  else
    radius=1.0;
}
double Sphere::getRadius(){
  return radius;
}

double Sphere::getVolume(){
  double rad3=radius*radius*radius;
  return (4.0*3.14*rad3)/3.0;
}

void Sphere::showInform(){
  cout<<"\n Radius = "<<radius
      <<"\n Volume = "<<getVolume()<<endl;
}
```

Определение класса «мяч» – файл ball.h

```
#ifndef BALL_H
#define BALL_H
#include "sphere.h"
```

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

class Ball: public Sphere{
  public:
    //конструкторы
    Ball();
    Ball(double iniRadius, const string iniName);
    //дополнительные или измененные операции
    string getName();
    void reName (const string newName);
    void resetBall(double newRadius, const string newName);
    void showInform();
  private:
    string name;
};
#endif
```

Реализация класса «мяч» – файл ball.cpp

```
#include "ball.h"
#include <iostream>
using namespace std;

Ball::Ball(): Sphere(){
  setName("NoName");
}

Ball::Ball(double iniRadius,const string iniName):
                  Sphere(iniRadius), name(iniName){}

void Ball:: reName(const string newName){
  name = newName;
}

string Ball::getName(){
  return name;
}

void Ball::resetBall(double newRadius, const string newName) {
  radius = newRadius;
  name = newName;
}

void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform();
}
```

📖 Конструктор производного класса выполняется после конструктора базового класса. Это правило действует и в цепочках наследования любой длины.

Например, конструктор класса `Ball` выполняется после конструктора класса `Sphere`.

```
Ball::Ball(): Sphere(){setName("NoName");}
Ball::Ball(double iniRadius, const string iniName):
                  Sphere(iniRadius), name(iniName){}
```

Конструкторы класса `Ball` вызывают соответствующие конструкторы класса `Sphere`. Для указания, какой именно из конструкторов базового типа следует вызвать в каждом конкретном конструкторе класса-потомка, используется синтаксис списка инициализаторов. Если конструктор базового класса отсутствует в списке инициализации, используется конструктор базового класса по умолчанию. В этом случае конструктор без параметров для класса `Ball` может выглядеть следующим образом:

```
Ball::Ball(){
  setName("NoName");
}
```

Если в классе-потомке не определен ни один конструктор, то будет создан конструктор по умолчанию, который вызовет конструкторы по умолчанию всех предков.

📖 Важно, чтобы в иерархии классов всегда были определены конструкторы по умолчанию.

Конструктор производного класса отвечает за инициализацию всех элементов данных, добавленных к унаследованным данным из базового класса. Конструктор базового класса выполняет инициализацию унаследованных элементов данных.

📖 Деструктор производного класса выполняется перед деструктором базового класса. В цепочках наследования произвольной длины деструкторы вызываются в порядке обратном порядку вызова конструкторов.

Соответственно, в нашем примере вначале выполнится деструктор класса `Ball`, затем — деструктор класса `Sphere`. Поскольку они явно не определены, то используются автоматические деструкторы.

Наследование не открывает доступ к закрытым членам. Если бы переменная `radius` была объявлена как `private`, она была бы недоступна в классе `Ball`. Тогда внутри класса `Ball` для доступа к ней пришлось бы использовать открытые функции класса `Sphere` — `resetRadius` и `getRadius`. Но всегда следует помнить, что каждый вызов функции приводит к увеличению накладных расходов. Поэтому если наследникам нужен прямой доступ к полям предка, рекомендуют использовать спецификатор доступа `protected`.

```
class Ball{
...
protected:
double radius; //радиус сферы
};
...
void Ball::resetBall(double newRadius, const string newName) {
  radius = newRadius; name = newName;
}
```

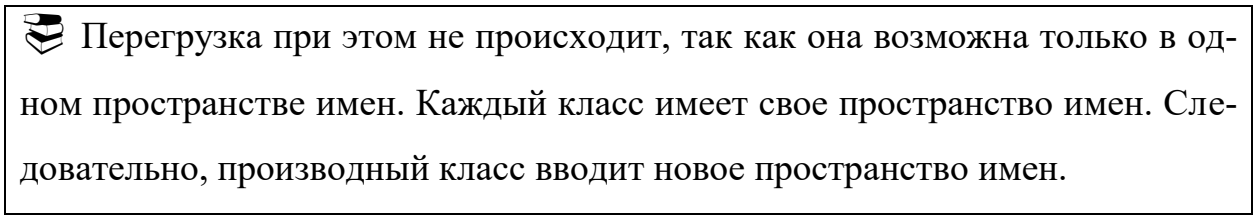
В реализации класса `Ball` можно вызывать функции, наследуемые от класса `Sphere`. Если бы поле `radius` было объявлено как `private`, функция `resetBall` должна была бы вызывать наследуемую функцию `resetRadius`.

```
void Ball::resetBall(double newRadius, const string newName) {
  resetRadius(newRadius);
  name = newName;
}
```

Функция `showInform` переопределяется в классе `Ball`. При этом она вызывает унаследованную версию функции класса предка `Sphere::showInform`. Для разрешения коллизии имён используется операция разрешения области видимости `::`.

```
void Ball::showInform(){
  cout<<" Ball type - "<<name<<". Specifications:";
  Sphere::showInform();
}
```

При переопределении функции базового класса в производном классе списки параметров могут не совпадать. При этом *замещающая функция* переопределяет исходную функцию, но с другим списком параметров.



Перегрузка при этом не происходит, так как она возможна только в одном пространстве имен. Каждый класс имеет свое пространство имен. Следовательно, производный класс вводит новое пространство имен.

Объекты производного класса могут вызывать открытые функции-члены базового класса. В этом выражается связь между производным и базовым классом, например:

```
Ball myBall(5.0, "Volleyball");
cout<<myBall.getVolume();
```

Две другие важные связи заключаются в том, что указатель базового класса может указывать на производный объект без явного преобразования типов. Ссылка на базовый класс тоже может иметь значением адрес объекта производного класса.

```
Ball myBall(5.0, "Volleyball");
...
Sphere &pb = myBall; Sphere *pt = &myBall;
Sphere *p = new Ball(9.0, "basketball");
pb.showInform();
cout << pb.getRadius() << endl;
pt->showInform();
cout << pt->getVolume() << endl;
p->showInform();
cout << p->getVolume() << endl;
```

Однако указатель или ссылка базового типа позволяет вызывать только функции базового класса, поэтому воспользоваться `pb`, `pt` или `p` для вызова функции `getName()` производного класса нельзя.

Следует обратить внимание и на то, что вызов функции `showInform()` через указатели `p`, `pt` или ссылку `pb` на базовый класс обратится к реализации этой функции в классе `Sphere`, игнорируя ее переопределение в классе `Ball`.

Пример 10. Описать иерархию классов `Person` – `Student`. Класс `Student` не должен иметь доступ к личным данным, содержащимся в классе `Person`. Методы класса заданы на UML-диаграмме (рис. 2.2).

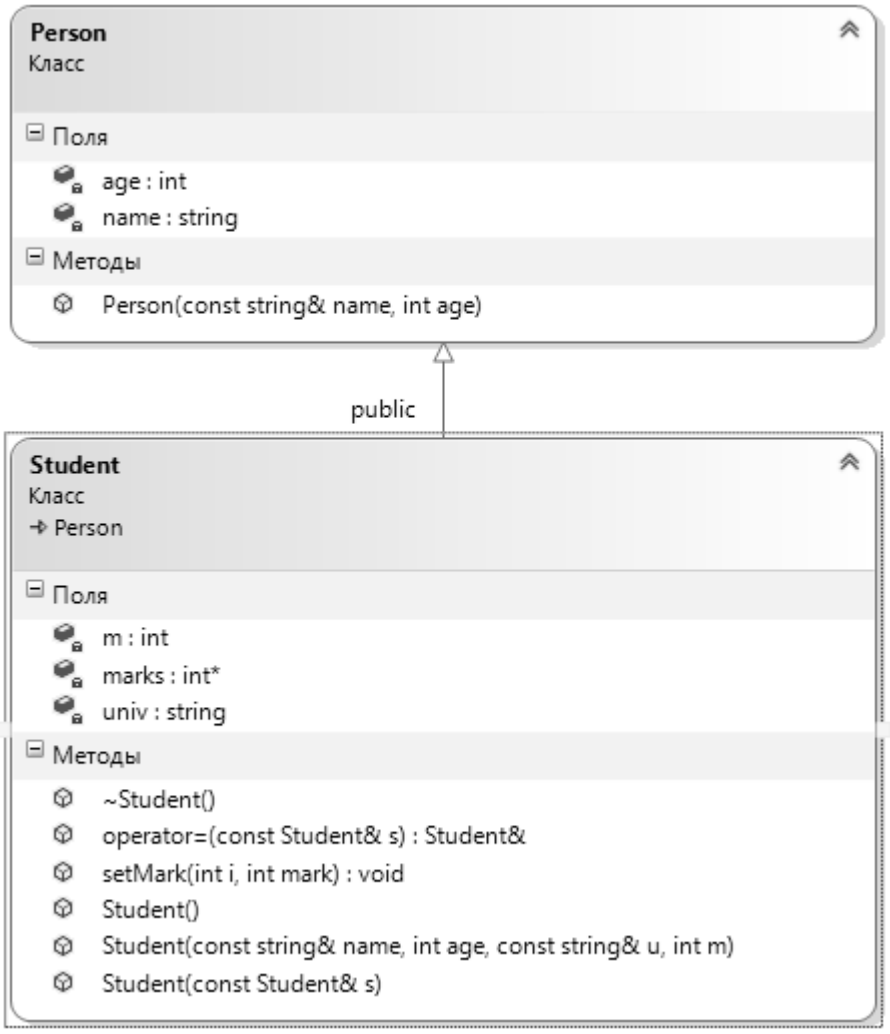


Рис. 2.2. UML диаграмма иерархии классов `Person` – `Student`

```
#pragma once
#include <string>
using namespace std;

class Person {
private:
  string name;
  int age;
public:
  Person(const string& name = "noname", int age = 18) :
name(name), age(age) {}
  friend ostream & operator<< (ostream & os, const Person & p)
  {
    os << p.name << " " << p.age << endl;
    return os;
  }
};

class Student: public Person
{
private:
  string univ;    // Университет
  int m;
  int* marks;     // Оценки
public:
  Student() : univ("МГУ"), m(3)
  {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }

  Student(const string& name, int age, const string& u, int m):
    Person(name, age), univ(u), m(m)
  {
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = 0;
  }

  Student(const Student& s): Person(s), univ(s.univ), m(s.m)
  {
    marks = new int[s.m];
    for (int i =0; i<m; ++i)
      marks[i] = s.marks[i];
  }

  ~Student() { delete[] marks; }
```

```
  Student& operator=(const Student& s)
  {
    if (&s != this)
    {
      delete[] marks;
      Person::operator=(s);
      univ = s.univ;
      m = s.m;
      marks = new int[m];
      for (int i = 0; i<m; ++i)
        marks[i] = s.marks[i];
    }
    return *this;
  };

  void setMark(int i, int mark)
  {
    if (i < 0 || i >= m)
      throw - 1;
    marks[i] = mark;
  }

  friend ostream & operator<< (ostream & os, const Student & s)
  {
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.m; ++i)
      os << s.marks[i] << " ";
    os << endl;
    return  os;
  }
};

#include <string>
#include <iostream>
#include "PS.h"
using namespace std;

int main() {
  Student s("Pinokkio", 18, "ЮФУ", 3);
  s.setMark(0, 5);
  s.setMark(1, 4);
  cout << s;
  Student s1(s);
  cout << s1;
  Student s2;
  cout << s2;
  s2 = s1;
```

```
  cout << s2;
  system("PAUSE");
  return 0;
}
```

Наличие динамически выделяемой памяти `int* marks` в классе `Student` создаёт дополнительные проблемы. В этом случае нельзя использовать создаваемые по умолчанию функции: конструктор без параметров, конструктор копии, деструктор и операцию присваивания. Необходимо предусмотреть их реализацию в классе.

Конструктор без параметров использует список инициализаторов для членов-данных `univ`, `m`. Выделение памяти для массива оценок `marks` и его инициализация происходит в теле конструктора.

```
Student(): univ("МГУ"), m(3){
  marks = new int[m];
  for (int i = 0; i<m; ++i)
    marks[i] = 0;
}
```

Инициализация членов-данных, унаследованных от класса `Person`, может выполниться только внутри конструктора класса `Person`. Вызов конструктора предка возможен только в списке инициализаторов конструктора наследника. В данном примере его нет, поэтому будет вызван конструктор по умолчанию класса предка. Если у класса предка нет конструктора по умолчанию, то произойдёт ошибка.

☺ Если предполагается иерархия наследования, рекомендуется для каждого класса в иерархии определять конструктор по умолчанию.

Конструктор с параметрами требует явного указания конструктора предка в списке инициализаторов.

```
Student(const string& name, int age, const string& u, int m):
  Person(name, age), univ(u), m(m){
  marks = new int[m];
  for (int i = 0; i<m; ++i)
    marks[i] = 0;
}
```

Здесь `Person(name, age)` — это вызов конструктора предка. Если не написать вызов `Person(name, age)`, то произойдет вызов конструктора без параметров, который проинициализирует поля `name` и `age` значениями по умолчанию.

В деструкторе необходимо только освободить память, занимаемую массивом `marks` , а память, выделенная для `univ` и `Person`, будет освобождена автоматически при вызове соответствующих деструкторов в эпилоге деструктора `~Student()`.

```
~Student() {
  delete[] marks;
  // эпилог
}
```

Рассмотрим конструктор копии класса `Student`.

```
Student(const Student& s): Person(s), univ(s.univ), m(s.m) {
  marks = new int[s.m];
  for (int i =0; i<m; ++i)
    marks[i] = s.marks[i];
}
```

Заметим, что `Person(s)` будет работать корректно благодаря тому, что ссылке на объект класса предка можно присвоить ссылку на объект класса потомка. При этом происходит приведение типа-наследника к базовому типу. Такое приведение называется *upcast*.

Рассмотрим преобразование типов в конструкторе копии класса `Student`, которое возникает при вызове конструктора копии `Person(s)` в списке инициализации. Мы фактически параметру типа `const Person &` присваиваем переменную типа `const Student &`. Аналогичные преобразования также могут выполняться как для указателей, так и для самих объектов. Преобразования типов для параметров при вызове функций встречаются достаточно часто, в том числе и для параметра `this`. Например, если вызывается для потомка унаследованная функция предка.

Поскольку передача параметров сводится к выполнению операции присваивания, то рассмотрим преобразование в иерархии «предок-потомок» на примере операции присваивания.

Для преобразования типов в иерархии «предок-потомок» работает правило**:** переменной типа предок можно присвоить переменную типа потомок, но не наоборот.

```
Person p("Иванов", 20);
Student s("Петров", 19, "ЮФУ", 3);
p = s;
```

При присваивании объекта производного класса переменной базового класса происходит копирование только полей базового класса, остальная часть информации объекта производного класса будет утеряна.

Попытка присвоить объекту класса наследника объект класса предка приведёт к ошибке компиляции.

```
s = p; // ошибка компиляции
```

При работе с указателями и ссылками на объекты предка и наследника действует аналогичное правило преобразования типов. Указателю или ссылке на объект базового класс можно присвоить адрес объекта или ссылку на объект, соответственно, производного класса, но не наоборот.

```
Person* pp = &p;
Student* ss = &s;
pp = ss;
ss = pp; // ошибка компиляции

Person& rp = s;
Student& rs = p; // ошибка компиляции
```

Таким образом, если мы напишем `Person& rp = s;`, тогда `rp` будет давать доступ только к двум полям объекта `s`, унаследованным от `Person`. Именно поэтому в конструкторе копии класса `Student` не возникало проблем с вызовом конструктора копии `Person(s)` в списке инициализации.

Операция присваивания будет реализована несколько сложнее:

```
Student& operator=(const Student& s){
  if (&s != this){
    delete[] marks;
    Person::operator=(s);
    univ = s.univ;
    m = s.m;
    marks = new int[m];
    for (int i = 0; i<m; ++i)
      marks[i] = s.marks[i];
  }
  return *this;
};
```

Операция присваивания не наследуется. Поэтому, чтобы выполнить присваивание для `private` полей, унаследованных от базового класса `Person`, необходимо выполнить операцию присваивания, определённую в классе `Person`.

```
Person::operator=(s);
```

Операция вывода в поток демонстрирует ещё одну особенность обращения к функции, определённой для предка.

```
friend ostream & operator<< (ostream & os, const Student & s) {
  os<<(Person)s;
  os << s.univ << endl;
  for (int i = 0; i < s.m; ++i)
    os << s.marks[i] << " ";
  os << endl;
  return  os;
}
```

Поскольку операция вывода в поток является внешней, то к ней невозможно применить разрешение области видимости. Поэтому используется явное приведение типа.

```
os<<(Person)s;
```

Обратное действие, когда происходит приведение базового типа к типу-наследнику, называется *downcast*. В этом случае необходимо использовать явное приведение типов с помощью шаблонной функции `static_cast<тип_наследника>`.

Добавим в класс Student функцию get_univ(), которая возвращает название учебного заведения, где учится студент. Такой функции нет, и не может быть в Person.

```
class Student : public Person{


public:

  string get_univ() const{
    return univ;
  }
};
```

Теперь рассмотрим ситуацию, когда нам может понадобиться приведение базового типа к типу наследника:

```
Person *p = new Student("Петров", 19, "ЮФУ", 3);
p->get_univ(); // ошибка компиляции
```

При вызове p->get_univ() произойдёт ошибка компиляции, так как в Person не определена функция get_univ(). Для корректного вызова указатель p нужно привести к типу *Student.

```
// Совеременный стиль:
static_cast<Student*>(pp)->get_univ();

// Старый стиль:
((Student*)pp)->get_univ();
```

Аналогичная ситуация возникает и при использовании ссылок:

```
Person & rp = *new Student("Петров", 20, "ЮФУ", 3);
static_cast<Student &>(rp).get_univ();
delete &rp;
```

> 🕮 Преобразование downcast в C++ возможно только для указателей или ссылок на объекты.

Корректное преобразование downcast возможно только как обратное преобразование к upcast.

2.3. Отношение включения

Каждый ресурс, под который выделяется память в конструкторе, обычно стремятся обернуть объектом класса, контролирующим этот ресурс, что упрощает код.

В этом случае между классами возникает не отношение наследование, а отношение включения («has-a»). Оно означает, что один класс содержит в качестве члена объект другого/других классов. При этом он использует возможности включённых объектов. Можно сказать, что он реализован посредством классов включённых объектов.

Пример 11. Модифицировать класс `Student` из иерархии `Person-Student`, используя для хранения оценок разработанный ранее класс `Array` в примере 4.

```
#include <string>
#include "classarray.h"
using namespace std;
class Student: public Person{
private:
  string univ;    // Университет
  Array marks;     // Оценки
public:
  Student() : marks(3), univ("МГУ") {   }
  Student(const string& name, int age, const string& u, int m):
    Person(name, age), univ(u), marks(m) { }

  void setMark(int i, int mark) {
    if (i < 0 || i >= marks.length())
    throw - 1;
    marks[i] = mark;
  }
  friend ostream & operator<< (ostream & os, const Student &s){
    os<<(Person)s;
    os << s.univ << endl;
    for (int i = 0; i < s.marks.length(); ++i)
      os << s.marks[i] << " ";
    os << endl;
    return os;
  }
};
```

Поскольку теперь для хранения оценок используется объект `marks` класса `Array`, он берёт на себя всю работу с динамической памятью. Благодаря этому код класса `Student` становится проще. Это отражено на UML диаграмме (рис.2.3).

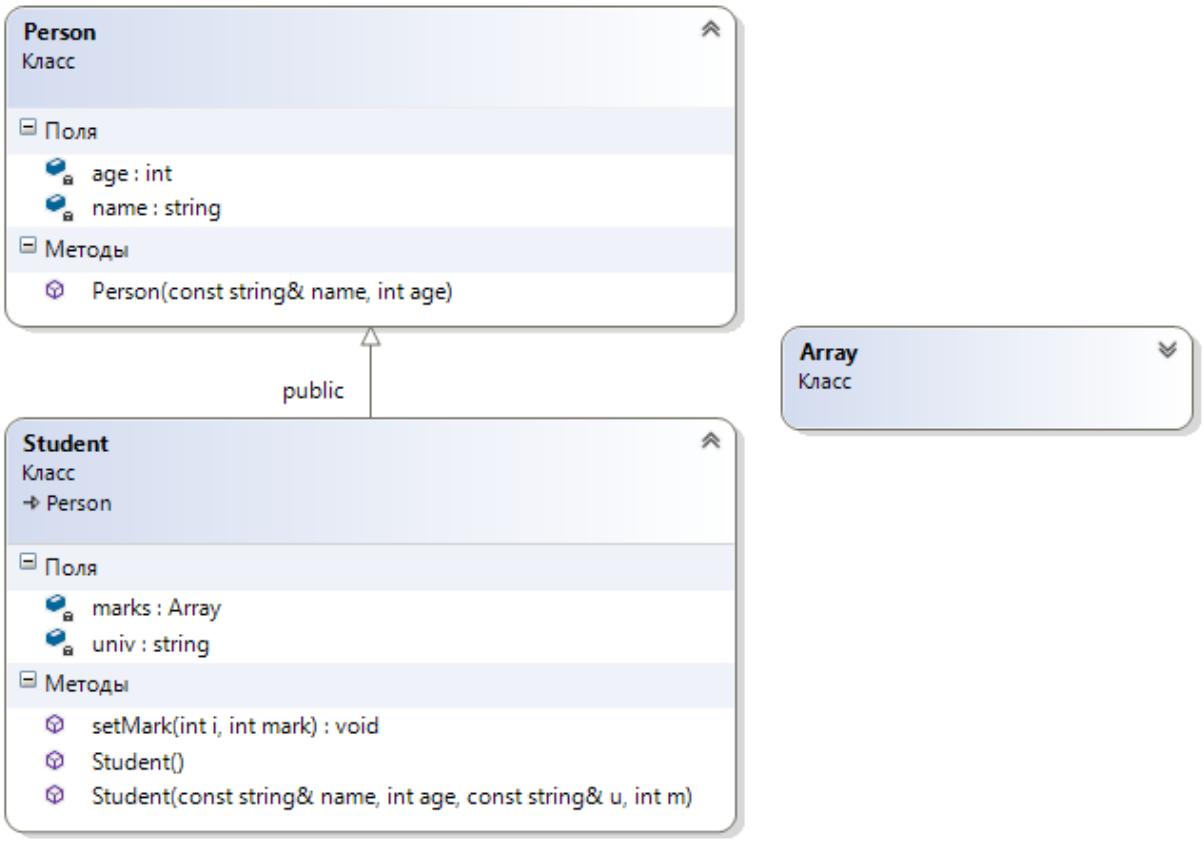


Рис.2.3. UML диаграмма к примеру 9

Теперь класс `Student` не нуждается в переопределении конструктора копии, деструктора и операции присваивания. Работают их версии по умолчанию, в которых происходит вызов соответствующих функций классов `Array` и `Person`.

Например, конструктор копии, создаваемый по умолчанию, выглядит следующим образом:

```
Student(const Student& s):
                    Person(s),univ(s.univ),marks(s.marks){ }
```

Поэтому создавать собственную реализацию нет никакого смысла. Она ничем не будет отличаться от версии по умолчанию.

☺ Старайтесь все динамически выделяемые ресурсы оборачивать в отдельные классы.

Если использовать класс `Array` из примера 4, то возникают ошибки компиляции в операции вывода в поток. Это связано с тем, что для константного объекта `s` класса `Student` используется неконстантные функция для операции индексации класса `Array`.

```
friend ostream & operator<< (ostream & os, const Student & s) {
  os<<(Person)s;
  os << s.univ << endl;
  for (int i = 0; i < s.marks.length(); ++i)
    os << s.marks[i] << " ";
  os << endl;
  return  os;
}
```

Данная проблема описывалась при обсуждении примера 4, для её решения предлагалось добавить реализацию константной операции индексирования. Следующий фрагмент кода демонстрирует описанные изменения.

```
class Array {
…
public:
…
  int& operator [] (int i);
  int operator [] (int i) const;
…
};

int& Array::operator [] (int i) {
  return A[i];
}

int Array::operator [] (int i) const {
  return A[i];
}
```

☺ Если в классе предполагается использование операции индексирования, то рекомендуется всегда реализовывать двумя способами.

В случае разработки сложных классов, использующих механизмы наследования и включения объектов других классов, необходимо хорошо понимать порядок вызова конструкторов и деструкторов. Для конструкторов он выглядит следующим образом:

1. вызов конструктора базового класса;
2. вызов конструкторов для полей включённых объектов;
3. вызов конструктора основного объекта.

Деструкторы вызываются в обратном порядке:

1. вызов деструктора основного объекта;
2. вызов деструкторов полей;
3. вызов деструктора предка.

Порядок вызовов конструкторов для полей включённых объектов зависит от порядка их следования в объявлении класса и не зависит от порядка в списке инициализаторов.

2.4. Позднее связывание и виртуальные функции

Если базовый класс имеет несколько классов-наследников, то единственным способом объединить в одной коллекции объекты различных наследников базового класса является создание массива (или другого контейнера), содержащего указатели (или ссылки) на объекты базового класса.

Предположим, что у нас кроме класса `Student` есть ещё один наследник класса `Person` – класс `Professor`. Создадим массив указателей на объекты класса `Person`.

```
Person *p [5];
```

Инициализируем массив p указателями на объекты классов наследников.

```
p[0] = new Student("Петров", 19, "МГУ", 3);
p[1] = new Student("Кораблина", 18, "ЮФУ", 3);
p[2] = new Professor ("Тьюринг", 32, "ИВЭ");
p[3] = new Professor ("Страуструп", 32, "ПМП");
p[4] = new Student("Гончаров", 20, " ЮФУ ", 3);
```

Предположим, в классе Person есть функция showInform, которая переопределена в классах-наследниках Student и Professor. При этом в случае следующего кода:

```
for ( int i=0; i<5; ++i)
  p[i]->showInform();
```

мы увидим результат выполнения функции showInform, определённой в классе Person. Это обусловлено тем, что определение, какую из переопределённых функций вызвать, происходит в момент компиляции по типу переменной-объекта или переменной-ссылки (указателя).

📖 Определение на этапе компиляции вызываемого варианта переопределённой функции называется *статическим,* или *ранним, связыванием*.

Однако в классах Student и Professor функция showInform переопределена и хотелось бы при вызове p[0]->showInform() увидеть информацию о студенте Петрове, а при вызове p[2]->showInform() – о профессоре Тьюринге.

Для того чтобы обеспечить возможность определения, какая из переопределённых функций должна быть вызвана, не на основе типа переменной-ссылки или указателя, а на основе типа объекта, на который они ссылаются, нужен другой механизм – *динамическое,* или *позднее, связывание*.

📖 Определение на этапе выполнения вызываемого варианта переопределённой функции называется динамическим, или поздним, связыванием.

Позднее связывание реализует принцип *полиморфизма*. Полиморфизм позволяет выбирать вариант вызываемой функции в ходе выполнения программы.

📖 Позднее связывание реализуется с помощью *виртуальных функций*.

Для того чтобы сделать функцию виртуальной, нужно перед её объявлением в базовом классе указать спецификатор `virtual`.

Например, объявление класса `Person` должно быть изменено следующим образом:

```
class Person{
  public:
    //все, как было описано выше, кроме функции showInform
    ...
    virtual void showInform();
  private:
    //все, как было описано выше
};
```

Реализация функции `showInform` как для класса `Person`, так и для классов-наследников остаётся без изменений.

📖 Если объявление функции в базовом классе начинается с ключевого слова `virtual`, то это делает функцию виртуальной для базового класса и всех классов, производных от базового класса, — «виртуальная функция всегда виртуальна».

Поэтому добавлять спецификатор `virtual` в объявление функции `showInform` для классов-наследников не обязательно, но его использование улучшает читаемость программы.

☺ Рекомендуется всегда использовать спецификатор `virtual` в объявлении виртуальных функций, независимо от их расположения в иерархии наследования.

Пример 12. Создать базовый класс `shape`, для хранения параметров произвольной геометрической фигуры на плоскости и вычисления её площади по этим параметрам. Создать два класса-потомка `rectangle` (параметры: две стороны) и `triangle` (параметры: сторона и высота к ней), представляющие, соответственно, прямоугольник и треугольник.

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

class shape {
  protected:
    double x,y;
  public:
    //конструкторы не создаём, так как
    //используется автоматический конструктор по умолчанию
    void set_param(double x0, double y0) {
      x=x0;
      y=y0;
    }
    virtual void show_area(){
      cout<<"Area calculation for this class is undefined";
      cout <<endl;
    }
};

class triangle: public shape {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"triangle with height "<<x<<" and base "<<y;
      cout<<endl<<"has an area "<<x*0.5 *y<<endl;
    }
};

class rectangle : public shape {
  public:
    //используется автоматический конструктор по умолчанию
    //вызывающий конструктор по умолчанию базового класса
    void show_area() {
      cout<<endl<<"rectangle with sides "<<x<<" * "<<y;
      cout<<endl<<"has an area "<<x*y<<endl;
    }
};
```

```
void tellMeAboutYourself(shape *s) {
  s-> show_area();
}

int main() {
  shape *p[2];
  triangle t;
  rectangle r;

  p[0]=&t;
  p[0]->set_param(10.0,5.0);
  p[0]->show_area();

  p[1]=&r;
  p[1]->set_ param (10.0,5.0);
  p[1]->show_area();

  for (auto x: p)
    tellMeAboutYourself(x);
  return 0;
}
```

В этом примере наследование не имеет целью расширение базового класса (рис. 2.4). Каждый из классов-наследников реализует свое собственное поведение на основе виртуальной функции, объявленной в базовом классе.

Функция `tellMeAboutYourself` ожидает от любого наследника класса `shape` собственную реализацию функции `show_area`. Принято говорить, что между функцией `tellMeAboutYourself` и наследниками класса `shape` устанавливается соглашение по возможностям взаимодействия. Такое соглашение называется *интерфейсом*.

В C++11 добавили новый тип цикла — foreach, который предоставляет более простой и безопасный способ итерации по массиву или любой другой структуре типа списка. Синтаксис цикла foreach для случая массива:

```
for (переменная: массив)
  statement;
```

Выполняется итерация по каждому элементу массива, присваивая значение текущего элемента массива переменной. В целях лучшей произ-

водительности объявляемый элемент должен быть того же типа, что и элементы массива, иначе произойдёт неявное преобразование типа. Чтобы не задумываться о типах, в заголовке цикла `for (auto x: p)` используется автоматическое определение типа.

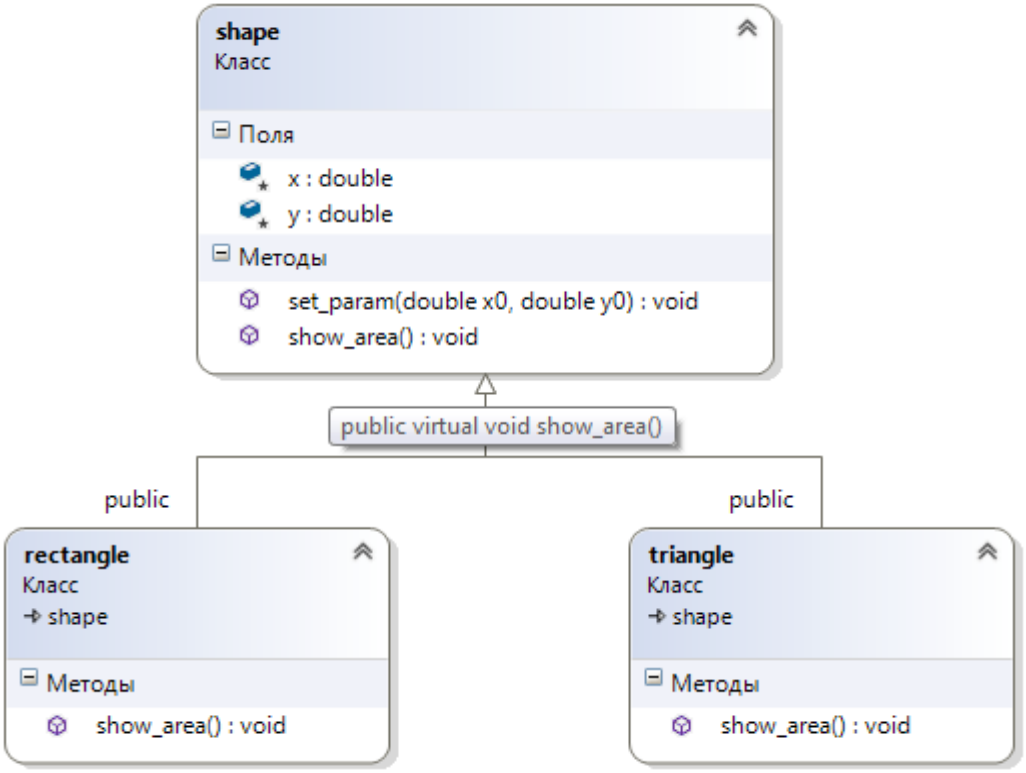


Рис.2.4. UML диаграмма классов примера 10

☺ Для пользовательских типов данных, используемых в качестве параметра цикла, рекомендуется применять `auto`.

До C++ 11 ключевое слово `auto` использовалось для явного указания, что переменная имеет автоматический класс памяти (время существования ограничено блоком, в котором она определена). Однако, поскольку все переменные в современном C++ по умолчанию имеют автоматический класс памяти, ключевое слово `auto` стало излишним и, следовательно, устаревшим.

Начиная с C++ 11, ключевое слово `auto` при инициализации переменной может использоваться вместо типа переменной, чтобы сообщить компилятору, что он должен определить тип переменной исходя из инициализируемого значения. Это называется *выводом типа* (или *автоматическим определением типов*).

2.5. Абстрактные классы

Во многих случаях вообще нет смысла давать определение виртуальной функции в базовом классе. Например, в классе `shape` определение функции `show_area()` — лишь способ корректно сообщить об ошибке использования. Объект класса `shape` без конкретизации типа фигуры не может иметь площади. Для того чтобы не прибегать к такому искусственному приёму, имеется возможность определять виртуальную функцию без реализации.

> 🕮 Виртуальная функция без реализации называется *чисто виртуальной*.

Синтаксис объявления чисто виртуальной функции:

```
virtual <тип> <имя> (<список параметров>) = 0;
```

> 🕮 Если класс имеет хотя бы одну чисто виртуальную функцию, его называют *абстрактным классом*.

Для абстрактного класса не могут быть созданы объекты. Такой класс может служить только в качестве базового в системе наследования и для создания указателей и ссылок, которые будут использованы при реализации полиморфизма.

Если в базовом классе имеется чисто виртуальная функция, производный класс должен иметь определение ее собственной реализации. Если

реализация хотя бы одной из чисто виртуальных функций не будет выполнена, производный класс, в свою очередь, останется абстрактным.

Абстрактные классы посредством чисто виртуальных функций описывают интерфейс, который можно использовать в других классах или функциях, ожидая, что наследники реализуют эти функции.

Рассмотрим еще одну иерархию наследования для геометрических фигур на плоскости, на базе абстрактного класса `shape`.

Пример 13. Создать абстрактный класс `shapeC` и его наследников: классы `circle` и `filled_circle`.

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

class shapeC {
  public:
    //используется автоматический конструктор по умолчанию
    virtual ~shapeC() {}
    virtual double square() const =0;
    virtual shapeC* clone() const =0;
    virtual void debug(ostream &out) const=0;
};
class circle: public shapeC {
  public:
    circle(double r=0): radius(r) { }
    ~circle() {}
    double square() const {
      return 3.14*radius*radius;
    }
    circle* clone() const {
      return new circle(radius);
    }
    void debug (ostream & out) const {
      out<<" radius = "<<radius <<endl;
    }
  protected:
    double radius;
};

class filled_circle: public circle {
  public:
    filled_circle (double r, int c): circle(r), color(c) { }
```

```
    ~filled_circle() {}
    filled_circle* clone() const {
      return new filled_circle(radius, color);
    }
    void debug (ostream & out) const {
      circle::debug(out);
      out<<" color = "<<color <<endl;
    }
  private:
    int color;
};

int main() {
shapeC *p[4];
  circle t(5);
  filled_circle r(10, 255);
  p[0] = &t;
  p[1] = &r;
  p[2] = p[0]->clone();
  p[3] = p[1]->clone();
  for (auto x : p) {
    x->debug(cout);
  }
  return 0;
}
```

Соответствующая UML диаграмма представлена на рисунке 2.5.

Массив `p` указателей на `shapeC`, представляет собой полиморфный контейнер, поскольку он может содержать указатели на `circle` и `filled_circle`.

Имея объекты наследников класса `shapeC`, можно их адреса присвоить элементам массива `p`.

```
circle t(5);
filled_circle r(10, 255)
p[0] = &t;
p[1] = &r;
```

Но если потребуется создать копии этих двух элементов, операция присваивания не поможет, поскольку будет выполнено присваивание адресов.

```
p[2] = p[0];
p[3] = p[1];
```

В этом случае `p[2]` и `p[0]` будут ссылаться на один и тот же объект `t`, а `p[3]` и `p[1]` — на объект `r`.

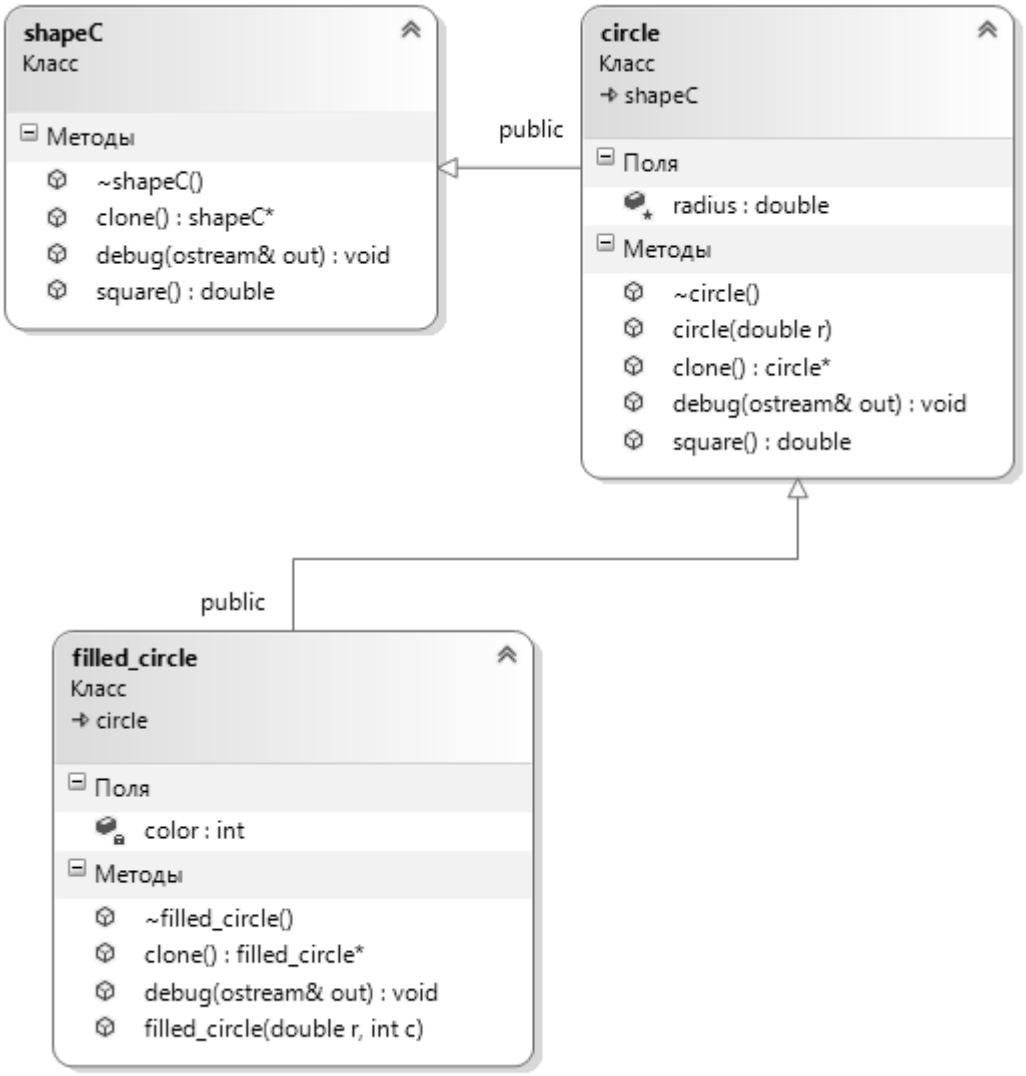


Рис. 2.5. UML диаграмма классов к примеру 11

Решением данной проблемы могло бы быть создание виртуального конструктора, но конструкторы в C++ не могут быть виртуальными. Поэтому для решения этой проблемы следует использовать функцию `clone()`.

```
virtual shapeC* clone() const =0;

class circle: public shapeC {
  public:
...
```

```
  circle* clone() const {
    return new circle(radius);
...
  }
}
class filled_circle: public shapeC {
  public:
...
  filled_circle* clone() const {
    return new filled_circle(radius, color);
  }
...
}
```

Эта функция является виртуальной и для каждого наследника создаёт и возвращает объект соответствующего класса. Клонирование является полиморфным, так как объект должен клонировать себя, а не объект базового типа. То есть `circle` клонирует `circle`, `filled_circle` клонирует `filled_circle`.

```
p[2] = p[0]->clone();
p[3] = p[1]->clone();
```

Следует обратить внимание на то, что деструктор базового класса тоже объявлен виртуальным.

```
virtual ~shapeC() {}
```

Важно, чтобы деструктор был виртуальным, если класс будет использоваться в качестве базового класса.

☠ Если базовый класс не имеет явного деструктора, а у потомков класса появятся явные деструкторы, например, освобождающие динамическую память, то возможна ошибка утечки памяти.

Важно гарантировать, чтобы при уничтожении объекта был вызван деструктор именно того класса-наследника, к которому он относится. Если базовый класс не требует выполнения явного деструктора, не следует полагаться на деструктор по умолчанию. Вместо этого необходимо описать виртуальный деструктор с пустой реализацией.

Конструкторы не могут быть виртуальными. Производный класс не наследует конструкторы базового класса, поэтому бессмысленно делать их виртуальными.

Статические функции также не могут быть виртуальными или объявляться с модификаторами `const` или `volatile`.

2.6. Цена виртуальности и система RTTI

Полиморфизм в C++ реализуется с помощью таблиц виртуальных функций — Virtual Methods Table **(**VMT**)**.

Для каждого класса, содержащего хотя бы одну виртуальную функцию, создаётся VMT, которая содержит адреса всех виртуальных функций, как этого класса, так и всех его предков. В каждом объекте такого класса появляется дополнительный указатель `vptr` на таблицу виртуальных функций. Если в классе и его предках нет виртуальных функций, то в его объекте поле vptr отсутствует (реализуется принцип: не платим за то, что не используем).

> ➢ В C++ нет общего главного класса-предка, как например, `Object` в PascalABC.Net, C#, Java**.** Это делается в целях повышения эффективности. Так как общий предок предоставляет набор виртуальных функций (методов), что приводит к созданию VMT для каждого класса.

Пусть в классе `filled_circle` есть функция `set_color()`.

```
void set_color(int c) {color = c;}
```

Даже, если мы точно знаем, что в элементе массива `p[1]` находится указатель на объект класса `filled_circle`, вызвать функцию `set_color()` через этот указатель нельзя.

```
p[1]->set_color(10); // ошибка компиляции !!!
```

Нужно выполнить явное приведение типа с помощью операции dynamic_cast <тип> для преобразования указателя на shapeC к указателю на filled_circle. Оператор dynamic_cast может быть применён к указателям или ссылкам.

```
dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

> ➢ В отличие от обычного приведения типа в стиле Си, проверка корректности приведения типов в стиле C++ для классов с виртуальными функциями производится во время выполнения программы.

Если в классе имеются виртуальные методы, то на этапе выполнения операция dynamic_cast пытается выполнить преобразование к указанному типу данных. В случае преобразования указателя к типу данных, который не является фактическим типом объекта, в результате будет получен нулевой указатель. При работе со ссылками, если преобразование невозможно, будет сгенерировано исключение std::bad_cast. Для его обработки операцию dynamic_cast следует поместить в блок try/catch.

Если виртуальных методов в классе и его предках нет, то dynamic_cast будет работать как static_cast.

Операция dynamic_cast использует механизм динамической идентификации типа данных RTTI (Runtime Type Identification). RTTI основана на способности системы сообщать о динамическом типе объекта и предоставлять информацию об этом типе во время выполнения (в отличие от времени компиляции). RTTI доступен только для классов, которые являются полиморфными, т.е. у них есть хотя бы одна виртуальная функция.

Таким образом, операция dynamic_cast позволяет идентифицировать динамический тип переменной во время выполнения. Операция typeid() используется для определения типа переменной во время вы-

полнения. Он возвращает ссылку на объект класса `std::type_info`, который содержит поля, позволяющие получить информацию о типе. Этот класс содержит перегруженные операции == и !=, а также функцию `name()`.

Следующие проверки идентичны по результатам.

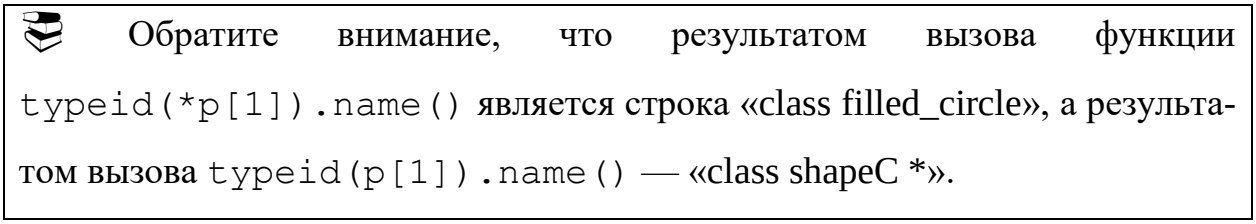
Первый вариант:

```
filled_circle *q = dynamic_cast<filled_circle *>(p[1]);
if (q!=nullptr)
  q ->set_color(10);
```

Второй вариант:

```
#include <typeinfo> //требуется для использования typeid()
...
if (typeid(*p[1]).name() == typeid(filled_circle).name())
  dynamic_cast<filled_circle *>(p[1])->set_color(10);
```

Операция `typeid()` для полиморфных типов работает полиморфным образом — возвращает динамический тип объекта.



Обратите внимание, что результатом вызова функции `typeid(*p[1]).name()` является строка «class filled_circle», а результатом вызова `typeid(p[1]).name()` — «class shapeC *».

2.7. Отношение подобия

Используя закрытое наследование, можно реализовать отношение «подобен», или «as-a». В этом случае потомок может воспользоваться при своей реализации средствами предка, не позволяя, однако, ни объектам, ни собственным потомкам их использовать.

Напомним, что C++ рассматривает открытое наследование как отношение типа «является». В частности, компиляторы, столкнувшись с иерархией открытого наследования, неявно преобразуют указатель или ссылку на объект класса потомка в указатель или ссылку на объект предка, если

это необходимо для вызова функций. Например, в рассмотренных примерах класс `Student` открыто наследует классу `Person`. При этом в случае необходимости, компилятор неявно преобразует указатель или ссылку на объект класса `Student` в указатель или ссылку на объект класса `Person`.

В противоположность открытому наследованию компиляторы в случае закрытого наследования не преобразуют указатели или ссылки на объекты производного класса в указатели или ссылки на объекты базового класса. Кроме того, члены, наследуемые от закрытого базового класса, становятся закрытыми, даже если в базовом классе они были объявлены как защищенные или открытые. Поэтому для объектов наследника мы не можем вызывать функции предка.

Закрытое наследование означает «реализовано посредством…». Делая класс `Derived` закрытым наследником класса `Base`, мы заинтересованы в использовании уже написанного для `Base` кода.


Можно сказать, что закрытое наследование означает наследование одной только реализации, без интерфейса. Закрытое наследование ничего не означает в ходе проектирования программного обеспечения и обретает смысл только на этапе реализации.

Таким образом, закрытое наследование — это исключительно прием реализации, который является альтернативой включению (композиции). Например, класс `Derived` может не наследовать, а содержать объект класса `Base`.

Если класс `Base` имеет защищённые (`protected`) члены, то в случае включения в класс `Derived` они недоступны для использования во включающем классе. Если же эти члены необходимы классу `Derived`, то следует применять закрытое наследование.

Пример 14. Предположим, что имеется реализация класса `Point` «точка на прямой». Использовать этот класс при реализации класса `GreenHopper` «исполнитель Кузнечик из задач ЕГЭ по информатике». С его помощью решить следующую задачу. Кузнечик может перемещаться по числовой оси с помощью команды `jump(N)`, где `N` — любое целое число. Кроме того, он может сообщать о своём положении на числовой оси с помощью команды `WhereAreYou()`. Кузнечик выполнил программу из 50 пар команд: `jump(5)`, `jump(-3)`. На какую одну команду можно заменить эту программу, чтобы Кузнечик оказался в той же точке, что и после выполнения программы.

```
class Point {
private:
  int x;
public:
  Point(int x = 0) : x(x){}
protected:
  void setx(int newx) {
    x = newx;
  }
  int getx() {
    return x;
  }
};
```

По условию задачи в основу решения должен быть положен предложенный класс `Point`. Использование открытого наследования невозможно вследствие требования, что кузнечик должен понимать только две команды `jump(N)` и `WhereAreYou()`. Использование включения не позволяет обращаться к функциям `setx(N)` и `getx()`, поскольку они объявлены как `protected`. Единственным вариантом решения является закрытое наследование (рис.2.6).

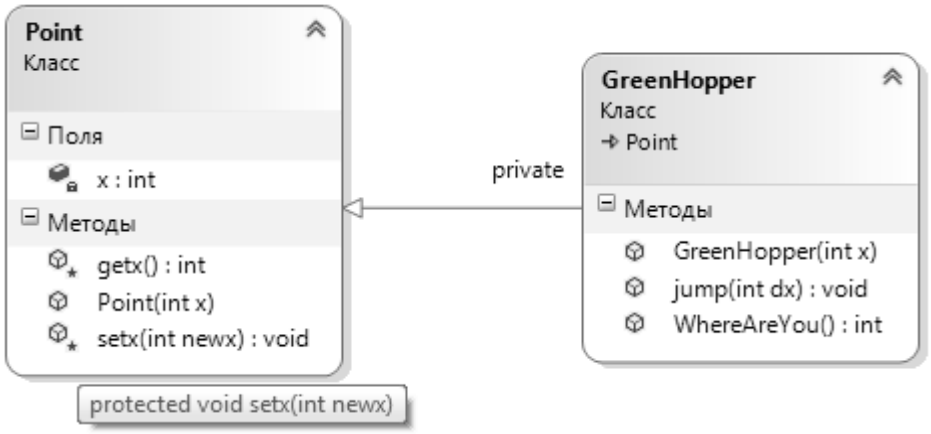


Рис. 2.6. UML диаграмма классов к примеру 12

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

class GreenHopper : private Point {
public:
  GreenHopper(int x=0) :Point(x) {}
  void jump(int dx) {
    setx(getx()+dx);
  }
  int WhereAreYou() {
    return getx();
  }
};

int main() {
  GreenHopper G;
  //cout<<G.getx();// ошибка доступа
  int t = G.WhereAreYou();
  cout <<  t << endl;
  for (int i = 0; i < 50; ++i) {
    G.jump(5);
    G.jump(-3);
  }
  cout << "jump(" << G.WhereAreYou() - t << ")"<<endl;
  return 0;
}
```

Пример 15. Используя готовую реализацию класса «список целых чисел», создать класс «стек целых чисел».

```
class error{};

class list{
private:
  // внутренняя организация класса может быть разной
```

```
public:
  list();
  ~list();
  bool isEmpty() const;
  void inHead(int val);
  void inTail(int val);
  int getFirst()const throw (error);
  int getLast()const throw (error);
  void delFirst()throw (error);
  void delLast()throw (error);
  //в полной реализации могут быть еще функции
};
```

Теперь реализуем класс «стек» (рис. 2.7).

```
class Stack:private list{
  public:
    Stack():list(){}
    bool isEmpty(){
      return list::isEmpty();
  }
  void push(int i){
    inHead(i);
  }
  int top() const throw (error);{
    return getFirst();
  }
  int pop throw (error); (){
    int res=getFirst();
    delFirst();
    return res;
  }
};
```

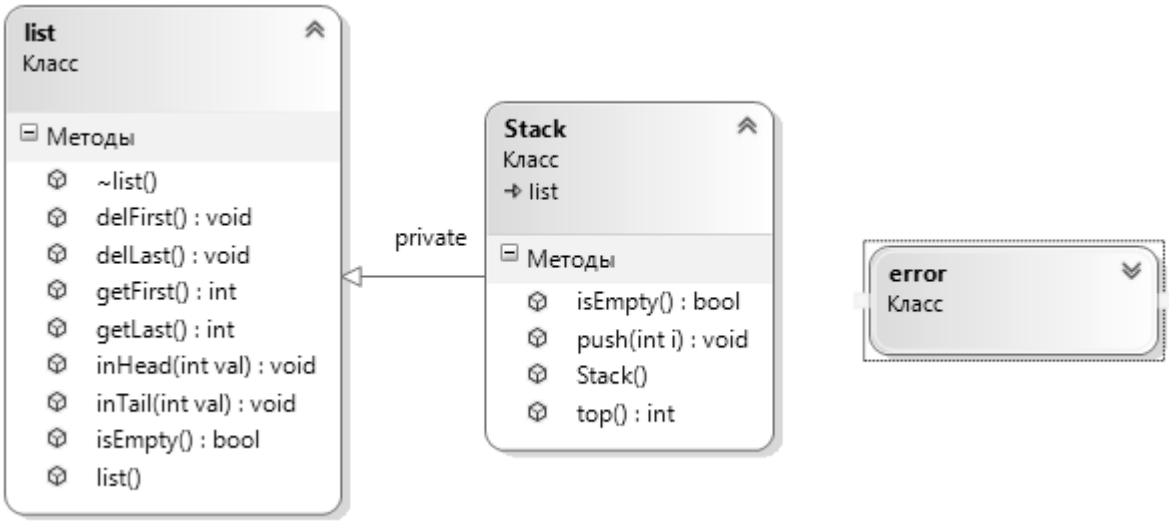


Рис. 2.7. UML диаграмма классов к примеру 13

Использование закрытого наследования позволило в этом примере скрыть возможности базового класса и тем самым защитить стек от некорректного использования. Например, нельзя попытаться вынуть элемент «снизу» из стека.

Приведем текст основной программы.

```
int main() {
  Stack p;
  for (int i=0;i<10;i++)
    p.push(i);
  while (!p.isEmpty())
    cout<<p.pop()<<' ';
  return 0;
}
```

 Для проверки работоспособности основной программы необходимо иметь реализацию класса «список целых чисел». Создать реализацию класса на базе массива.

 Реализовать класс стек, используя принцип включения, а не закрытого наследования.

2.8. Коллекции и итераторы

Коллекция или *контейнер* — объект программы, содержащий в себе набор значений одного или различных типов и позволяющий обращаться к этим значениям.

 Термин «объект» в данной книге используется в двух смыслах: программный объект и экземпляр класса. С точки зрения коллекции — это программный объект.

Назначение коллекции — служить хранилищем программных объектов и обеспечивать доступ к ним. Обычно коллекции используются для

хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные функции, в зависимости от ее логической организации. Реализация может допускать выполнение отдельных операций над коллекциями в целом. Наличие операций над коллекциями во многих случаях может существенно упростить программирование.

Примерами коллекций являются классы для реализации массива, строки, списка.

Итератор — объект, позволяющий программисту перебирать все элементы коллекции без учета особенностей ее реализации.

В простейшем случае итератором в низкоуровневых языках является указатель. Операцию индексирования можно считать примитивной формой итератора. Необходимо отметить, что счётчик цикла иногда называют итератором цикла. Тем не менее, счётчик цикла обеспечивает только перебор элементов, но не доступ к элементу.

Использование итераторов в обобщённом программировании позволяет реализовать универсальные алгоритмы работы с контейнерами. Примерами коллекций (контейнеров), по которым может перемещаться итератор, могут быть: список, очередь, множество, ассоциативный массив и др.

Итератор похож на указатель своими основными операциями: указание одного отдельного элемента в коллекции объектов (доступ к элементу) и изменение своего значения так, чтобы указывать на следующий элемент (перебор элементов). Также должен быть определён способ создания итератора, указывающего на первый элемент коллекции, и способ узнать, достигнут ли конец коллекции. В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Проектирование класса итератора тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет функции создания итераторов.

Существует множество разновидностей итераторов, различающихся своим поведением, включая: однонаправленные, обратные (реверсные) и двунаправленные итераторы; итераторы ввода и вывода; константные итераторы (защищающие контейнер или его элементы от изменения).

В тех случаях, когда коллекция представляет собой структуру, основанную на указателях, итератор может быть реализован в виде дополнительного поля — текущего указателя. В этом случае следует заботиться о его состоянии при любых модификациях коллекции, даже если его не придётся использовать. Кроме того, такой встроенный итератор обычно только один, а могут возникнуть задачи, в которых потребуется наличие двух, трёх или даже большего количества итераторов.

Итераторы-объекты имеют преимущество по сравнению со встроенным текущим указателем, так как позволяют создавать для одной коллекции любое количество итераторов, действующих независимо. Таким образом, итератор позволяет вынести рабочий указатель за пределы коллекции, но при этом дает возможность перемещаться по объектам, содержащимся в коллекции, аналогично тому, как текущий указатель позволяет перемещаться внутри коллекции.

Как правило, итератор имеет операцию доступа к элементу коллекции по ссылке. Такая операция может быть реализована путём перегрузки операции разыменования *. Для итераторов предусматриваются также операции перемещения вперёд и назад по коллекции объектов. Чаще всего для этого перегружаются операции ++ и --. Кроме того, операции == и != обычно перегружаются для проверки равенства итераторов. В классе коллекции для использования итератора обычно создаются две функции:

`iterator begin()` – инициализация итератора ссылкой на первый элемент коллекции;

`iterator end()` – значение, сообщающее, что итератор достиг конца коллекции.

Через итератор могут быть реализованы другие операции над коллекцией, например, вставка элемента после позиции итератора или удаление элемента в позиции итератора. При реализации таких операций очень важно оговорить правила поведения итератора после выполнения операции.

Пример 16. Реализовать линейный двусвязный список и итератор для него с возможностью перемещения в двух направлениях.

Класс список `List` содержит элементы, описываемые структурой `node`:

```
struct node{
  int data;
  node* next;
  node* prev;
  node(int data, node* next, node* prev) {
    this->data = data;
    this->next = next;
    this->prev = prev;
  }
};
```

Для упрощения в дальнейшем вывода элементов списка перегружена операция вывода в поток. Для структуры не требуется объявлять её дружественной.

```
ostream & operator<<(ostream & out, const node& X) {
  out << X.data;
  return out;
}
```

Чтобы в определении класса `List` операции `begin()` и `end()` могли возвращать объекты типа итератор списка, нужно сделать предварительное объявление класса итератора.

```
class listIterator;
```

Это связано с рекурсией в определении классов коллекции `List` и итератора `listIterator`.

Определение класса `List`:

```
class List{
public:
  class Error {
  public:
    void what(){
      cout << "List is empty" << endl;
    }
  };

  List() {
    head = 0; tail = 0;
  }

  List(const List& l);
  ~List();
  bool isEmpty() const;
  void inHead(int val);
  void inTail(int val);
  int getFirst()const;
  int getLast()const;
  void delFirst();
  void delLast();
  listIterator begin() const;
  listIterator end() const;
  friend ostream& operator<<(ostream & os, const List& l);
  //в полной реализации могут быть еще методы
```

```
private:
  node* head;
  node* tail;
  Error err;
};
```

Определение класса итератора для списка:

```
class listIterator {
public:
  class Error {
  public:
    void what(){
      cout << "Iterator error" << endl;
    }
  };
private:
  const List *collection;
  node *cur;
public:
  listIterator(const List *s, node *e) :collection(s), cur(e){}
  const int operator *(){
    return cur->data;
  }
  listIterator operator++(); //префиксный ++
  listIterator operator--(); //префиксный --
  int operator == (const listIterator &ri) const;
  int operator != (const listIterator &ri) const;
};
```

Реализация класса `list` для двусвязного списка:

```
List::List(const List& l) {
  head = 0; tail = 0;
  node* q = l.head;
  while (q){
    inTail(q->data);
    q = q->next;
  }
}
List::~List(){
  while (head){
    node* cur = head;
    head = head->next;
    delete cur;
  }
  tail = head = nullptr;
}
bool List::isEmpty() const {
  return (head == nullptr);
}
```

```
void List::inHead(int val){
  node* t = new node(val,head,nullptr);
  if (!head)
    tail = t;
  else
    head->prev = t;
  head = t;
}
void List::inTail(int val){
  node* t = new node(val, nullptr,tail);
  if (head){
    tail->next = t;
    tail = t;
  }
  else{
    head = tail = t;
  }
}

int List::getFirst() const{
  if (head)
    return head->data;
  else
    throw err;
}
int List::getLast()const{
  if (head)
    return tail->data;
  else
    throw err;
}
void List::delFirst(){
  if (head){
    node *t = head;
    head = head->next;
    head->prev = nullptr;
    delete t;
  }
  else
    throw err;
}

void List::delLast(){
  if (head){
    if (head == tail) { delete tail; head = nullptr; }
    else{
      node *t = head;
      while (t->next != tail)
```

```
        t = t->next;
      delete tail;
      tail = t;
      t->next = nullptr;
    }
  }
  else
    throw err;
}

ostream& operator<<(ostream & os, const List& l){
  node *p = l.head;
  while (p) { os << *p << " "; p = p->next; }
  os << endl;
  return os;
}
```

Реализация класса `listIterator` для итератора двусвязного списка:

```
listIterator listIterator:: operator++() {
  if (cur) {
    cur = cur->next;
    return *this;
  }
  else throw Error();
}

listIterator listIterator:: operator--() {
  if (cur) {
    cur = cur->prev;
    return *this;
  }
  else throw Error();
}

int listIterator:: operator==(const listIterator &ri) const {
  return ((collection == ri.collection) && (cur == ri.cur));
}
int listIterator:: operator!=(const listIterator &ri) const {
  return !(*this == ri);
}
listIterator List::begin() const {
  return listIterator(this, head);
}
listIterator List::end() const {
  listIterator iter(this, nullptr);
  return iter;
}
```

В закрытых полях класса хранится указатель на коллекцию, с которой работает итератор и указатель на текущее положение итератора в коллекции. Конструктор с параметрами позволяет инициализировать эти поля.

Структуры являются открытыми классами с открытым доступом, поэтому возможно обращение к полям структуры, представляющей элемент списка напрямую. Если бы элемент списка был представлен классом с закрытыми полями, для доступа к ним пришлось бы использовать соответствующие функции или класс итератора нужно было бы объявить дружественным классу элемента списка.

Чтобы использовать итератор для работы с классом-коллекцией, необходимо, чтобы класс содержал функции инициализации итератора:

```
listIterator begin() const;
listIterator end() const;
```

Инициализация итератора ссылкой на первый элемент списка осуществляется следующей функцией:

```
listIterator list::begin() const {
  return listIterator (this,head);
}
```

В качестве параметров конструктору итератора передаются указатель на объект-список, для которого будет создан итератор, и указатель на голову списка.

Функция `listIterator list::end() const` возвращает значение, которое является признаком того, что итератор достиг конца списка:

```
listIterator list::end() const{
  listIterator iter(this,nullptr);
  return iter;
}
```

Функцию `end()` нельзя заменить простым сравнением указателя с `nullptr`, так как в операциях `==` и `!=` для итератора прежде всего проверяется, относятся ли два итератора к одному и тому же списку.

Рассмотрим использование итератора для линейного двусвязного списка на примере нескольких функций.

Нахождение суммы элементов списка, расположенных между двумя итераторами.

```
int sum(listIterator b, listIterator e) {
  int sum = 0;
  while (b != e){
    sum+= *b;
    ++b;
  }
  return sum;
}
```

Нахождение итератора, ссылающегося на максимальный элемент списка.

```
listIterator max(listIterator b, listIterator e) {
  listIterator maxx = b;
  while (b != e){
    if (*b > *maxx)
      maxx = b;
    ++b;
  }
  return maxx;
}
```

Вывод в обратном порядке элементов списка, расположенных между двумя итераторами.

```
void reverseprint(listIterator b, listIterator be){
  while (b!=be) {
    cout << *(--be) << ' ';
  }
  cout << endl;
}
```

Ниже приведён код для тестирования созданных функций.

```
int main(){
  List l1;
  for (int i = 0; i < 5;++i)
    l1.inHead(i);
  for (int i = 5; i < 10; ++i)
    l1.inTail(i);
  List l2(l1);
  cout << " list \n" << l2 << endl;
  cout << sum(l2.begin(), l2.end())<<endl;
```

```
  listIterator mt = max(l2.begin(), l2.end());
  cout << *mt << endl;
  reverseprint(mt,l2.begin());
  reverseprint(mt, l2.end());
  return 0;
}
```

2.9. Классы для рекурсивных типов данных

Рекурсивное определение данных возникает, когда структура данных ссылается на объект такой же структуры. При этом в случае одной ссылки чаще всего в алгоритмах обработки эффективней использовать итерацию. Если же ссылок больше одной (как в деревьях), то в большинстве случаев применяются рекурсивные реализации. Рекурсия помогает разрабатывать изящные и эффективные структуры данных и алгоритмы в тех случаях, когда решение без использования рекурсии оказывается сложным и неочевидным.

Рекурсивное определение бинарного дерева можно реализовать как класс, содержащий внутри себя указатель на узел дерева (корень) и рекурсивное описание узла дерева. При этом открытые функции-члены класса не будут рекурсивными. Второй способ определения класса бинарного дерева состоит в том, чтобы определить в нём указатели на два поддерева и открытые рекурсивные функции для работы с деревом.

Для демонстрации этих двух подходов определим классы с минимально необходимым количеством функций-членов.

Пример 17. Реализовать класс бинарного дерева поиска, определив в нём указатели на два поддерева и открытые рекурсивные функции для работы с деревом: конструктор, конструктор копии, деструктор, добавление элемента в дерево и вывод всех элементов дерева в поток.

```
class TreeR{
private:
  int data;
  TreeR *lt, *rt;
```

```
public:
  TreeR (int val=0, TreeR *l = nullptr, TreeR *r = nullptr):
         data(val), lt(l), rt(r) {}

  TreeR(const TreeR * t){
    if (t->lt) lt = new TreeR(t->lt);
      data = t->data;
    if (t->rt) rt = new TreeR(t->rt);
  }
  ~TreeR(){
    if (lt != nullptr) delete lt;
    if (rt != nullptr) delete rt;
  }

  void add(int a){
    if (data > a){
      if (lt) lt->add(a);
      else lt = new TreeR(a);
    }
    else{
      if (rt) rt->add(a);
      else rt = new TreeR(a);
    }
  }
  friend ostream& operator<<(ostream& os, const TreeR *t){
    if (t->lt) os << t->lt;
    os <<  t->data << "  ";
    if (t->rt) os<< t->rt;
    return os;
  }
  void printRKL(){
    if (rt) rt->printRKL();
    cout << "  " << data << " ";
    if (lt) lt->printRKL();
  }
};
```

В данном примере вывод в поток реализован двумя способами: перегрузкой операции выдачи в поток << и функцией `printRKL`.

В перегруженную операцию << указатель на дерево передаётся параметром, посредством которого реализуются рекурсивные вызовы.

```
friend ostream& operator<<(ostream& os, const TreeR *t)
```

Аналогично реализуется рекурсия и в функции `printRKL`, в которую указатель на дерево передаётся через неявный параметр `this`.

```
if (rt) rt->printRKL();
//эквивалентно
if (this->rt) this->rt->printRKL();
```

При использовании класса `TreeR` приходится объявлять переменную указатель на объект и создавать объект динамически. При этом конструктор без параметров создаёт не пустое дерево, а дерево с одной вершиной, имеющей значение по умолчанию.

```
TreeR *t= new TreeR();// это непустое дерево
t->add(5);
t->add(-7);
t->add(3);
t->add(-4);
cout <<"T "<< t << endl;
t->printRKL();
TreeR *t1 = new TreeR(t);
cout << "T1 "<<t1 << endl;
```

Для класса `TreeR` пустота дерева определяется не на уровне класса, а на уровне указателя на объект класса.

```
TreeR* t0= nullptr; // это пустое дерево
```

Поэтому нельзя реализовать функцию-член класса для проверки на пустоту. Для безопасной работы перед вызовом функций-членов класса следует выполнять проверку на пустоту следующим образом:

```
if (t0!=nullptr) t0->add(3);
if (t0!=nullptr) t0->RKL();
if (t0!=nullptr) cout <<"T "<< t0 << endl;
```

Для динамических объектов при выполнении операции `delete` вызывается деструктор, но указатель на объект не обнуляется. Если предполагается дальнейшее использование указателя, ему необходимо явно присвоить значение `nullptr`.

```
if (t!=nullptr){
  delete t;
  t=nullptr;
}
```

При использовании операции удаления отдельных узлов из дерева возникает проблема при удалении последней вершины, которая не может быть решена в самой операции.

Пример 18. Реализовать класс бинарного дерева поиска, содержащий внутри себя указатель на узел дерева (корень) и рекурсивное описание узла дерева. Определить открытые функции-члены класса: конструктор, конструктор копии, деструктор, добавление элемента в дерево и вывод всех элементов в поток. При их реализации использовать рекурсивные функции в закрытой части описания класса.

```
class Tree{
private:
  struct TNode;
  typedef TNode* node_ptr;

  struct TNode{
    int data;
    node_ptr lt, rt;
    TNode (int val, node_ptr l=nullptr, node_ptr r=nullptr):
           data(val), lt(l), rt(r){}
  };

  node_ptr root;

  void delTree(node_ptr t){
    if (t!=nullptr){
      delTree(t->lt);
      delTree(t->rt);
      delete t;
    }
  }

  void add(node_ptr& t, int a){
    if (t == nullptr) t = new TNode(a);
    else if (t->data > a) add(t->lt, a);
    else add(t->rt, a);
  }

  void printLKR(node_ptr t, ostream& os) const{
    if (t){
      printLKR(t->lt, os);
      os << t->data << " ";
```

```
      printLKR(t->rt, os);
    }
  }

  void copy(node_ptr t, node_ptr &newT) const {
    if (t != nullptr){
      newT = new TNode(t->data, 0, 0);
      copy(t->lt, newT->lt);
      copy(t->rt, newT->rt);
    }
    else newT = nullptr;
  }

public:
  Tree(): root(nullptr) {}
  Tree(const Tree& t){
    copy( t.root, root);
  }

  ~Tree(){
    delTree (root);
  }

  void addNode(int a){
    add(root, a);
  }

  friend ostream& operator<<(ostream& os, const Tree &t){
    t.printLKR(t.root,os);
    return os;
  }
};
```

В примере типы `TNode` для узла дерева и `node_ptr` для указателя на узел определены внутри класса, что подчёркивает инкапсуляцию внутренней организации класса `Tree`. Именно в определении типа `TNode` присутствует рекурсия. Поэтому рекурсивные функции оперируют с указателями на узел дерева и должны быть объявлены как `private`. Для доступа к ним используются открытые функции класса, которые вызывают рекурсивные функции, передавая в качестве параметра указатель на корень дерева.

Такое описание класса допускает существование пустого дерева. Конструктор без параметров создаёт пустое дерево. Если предусмотреть операцию удаления узла дерева, то можно получить в процессе её выполнения пустое дерево. Для безопасной работы с таким деревом рекомендуется иметь операцию проверки дерева на пустоту.

```
Tree t;
t.addNode(5);
t.addNode(7);
t.addNode(3);
t.addNode(4);
cout <<"T "<< t << endl;
Tree t1(t);
cout << "T1 "<<t1 << endl;
```

Очевидно, что реализация класса бинарного дерева в примере 18 лишена недостатков, перечисленных в примере 17. Именно такой способ реализации рекомендуется использовать при создании классов для рекурсивных структур.

ГЛАВА 3. ОБОБЩЕННЫЙ ПОДХОД

Одной из важнейших целей объектно-ориентированного программирования является поддержка повторного использования кода. Один из механизмов для достижения этой цели — шаблоны C++. Шаблоны функций, которые были рассмотрены в книге Демяненко Я.М., Чердынцевой М.И.[2], позволяют избавиться от множественной перегрузки функций. При этом обобщается один и тот же алгоритм для разных типов данных. Этот подход относится к обобщённому программированию.

Обобщённое программирование (generic programming) заключается в описании структур данных и алгоритмов в терминах типов, подставляемых в качестве параметров в эти алгоритмы и структуры. Обобщённое программирование является одним из проявлений полиморфизма. Обобщённый подход в языке C++ основывается на шаблонах функций и шаблонах классов.

3.1. Шаблоны классов

Шаблоны классов, так же как и шаблоны функций, являются трафаретами, по которым компилятор создает *шаблонные классы* и шаблонные функции.

Шаблоны классов часто называют *параметризованными типами*, так как они имеют один или большее количество параметров типа, определяющих настройку шаблона класса на специфический тип данных при создании объекта класса.

Например, шаблон класса `Stack` может служить основой для создания многочисленных классов `Stack`: «`Stack` для данных типа `int`», «`Stack` для данных типа `double`», «`Stack` для данных типа `Time`» и т. д.

Пример 19. Создать шаблон класса `Stack` с реализацией на основе массива.

```
//Шаблон класса Stack         файл stack.h
#ifndef TSTACK_H
#define TSTACK_H
#include <iostream>

class stackEmpty {
};

class stackFull {
};

template <typename T>
class Stack {
private:
  T * start;   //указатель на стек
  int head;    //положение вершины стека
  int size;    //размер стека
public:
  //конструктор с параметром по умолчанию для размера стека
  Stack(int = 10);
  ~Stack() { delete[] start; }          //деструктор
  void push(const T&);   //добавление элемента в стек
  T pop();     //извлечение элемента из стека
  T top();     //просмотр элемента на вершине стека
  bool isEmpty() const {  //true, если стек пустой
    return head == -1;
  }
  bool isFull() const {  //true, если стек полон
    return head == size - 1;
  }
};

//конструктор
template <typename T>
Stack<T>::Stack(int n) {
  //определение «разумного» размера стека
  size = n>0 && n<1000 ? n : 10;
  start = new T[size];
  head = -1;
}

// добавление объекта в стек
//в случае переполнения стека выбрасывается исключение
template <typename T>
void Stack<T>::push(const T& x) {
```

```
  if (isFull()) throw stackFull();
    start[++head] = x;
}
//извлечение элемента из стека
template <typename T>
T Stack<T>::pop() {
  if (isEmpty()) throw stackEmpty();
  T x = start[head--];
  return x;
}

// просмотр элемента на вершине стека
template <typename T>
T Stack<T>::top() {
  if (isEmpty()) throw stackEmpty();
  T x = start[head];
  return x;
}
#endif
```

Определение шаблона класса `Stack` похоже на определение класса, только впереди добавляется конструкция:

```
template <typename T>
```

Добавленная к заголовку конструкция `template <typename T>` указывает на то, что описывается шаблон класса `Stack` с параметром типа `T`. Идентификатор `T` определяет тип данных-элементов, хранящихся в стеке, и может использоваться при описании типов полей класса и в функциях-членах класса.

Компилятор рассматривает все функции-члены класса как шаблоны функций, параметры которых совпадают с параметрами шаблона класса. Поэтому каждое определение функции-члена класса вне шаблона класса начинается с конструкции:

```
template <typename T>
```

Внешнее определение каждой функции-члена шаблона класса использует операцию разрешения области видимости с именем шаблона класса `Stack<T>`.

☺ Хорошей идеей является написать и протестировать конкретный класс, а затем преобразовать его в шаблон. Таким образом, можно решить многие проблемы проектирования и обнаружить большую часть ошибок кода в тексте конкретного класса.

Аналогично шаблонам функций, все шаблоны классов и структур должны быть помещены в заголовочные файлы. Для шаблонов классов в отличие от просто классов выносить реализации их членов-функций в отдельный *.cpp файл нельзя. Это связано с тем, что шаблоны в языке C++ не компилируются, потому что шаблон представляет собой не фрагмент программного кода, а лишь инструкции для построения кода.

Для шаблонов классов и структур код генерируется в момент определения объекта. Причём генерируются только те члены-функции класса, которые используются.

Подстановка конкретного типа в шаблон приводит к созданию шаблонного класса, т.е. к *инстанцированию* шаблона (template instantiation). Аналогично функция инстанцируется (генерируется, конкретизируется) из шаблона функции и аргумента шаблона. Использование различных терминов для одного понятия связано с терминологическими проблемами, возникающими при переводе. О них мы уже говорили в книгах Демяненко Я.М., Чердынцевой М.И.[2] и Русанова Я.М., Чердынцева М.И.[6].

Количество инстанций зависит от количества используемых типов. Версия шаблона для конкретного аргумента шаблона называется специализацией. Только специализации шаблонов содержат настоящий код. Генерация версий шаблона для набора аргументов шаблона является задачей компилятора, а не программиста.

По умолчанию, шаблон предоставляет единственное определение, которое должно использоваться для всех аргументов шаблона. Явная спе-

циализация позволяет обеспечить альтернативные реализации шаблона. Выделяют специализации, определяемые пользователями, или просто пользовательские специализации. Например, если для специфического типа данных нужен класс, который не соответствует общему шаблону класса, можно явно определить его, отменив тем самым действие шаблона для этого типа.

Так шаблон класса `Stack` может использоваться практически для любого типа. Однако может возникнуть необходимость создать специфический класс `Stack` для некоторого типа, например, `Specific`. Для этого нужно просто создать новый класс с именем `Stack<Specific>`.

```
Stack<Specific>{
...//определение специфического стека
}
```

- В C++ разрешаются все действия с типом T. Поэтому если при инстанцировании будет использован тип, для которого не определены какие-нибудь операции, используемые в шаблоне, то возникнет ошибка компиляции шаблонного класса или шаблонной функции. А в .NET и Java существуют возможность введения ограничения на тип, с которым можно инстанцировать шаблон. Тем самым повышается надёжность использования обобщённых типов.
- В C++ в результате компиляции шаблона получается исполняемый код инстанцированных функций, структур и классов. В .NET в результате компиляции обобщения создается исполняемый код самого обобщения, т.е в .NET можно создать dll с обобщенным классом.

Рассмотрим текст программы, в которой используется шаблон класса `Stack`.

```
#include <iostream>
#include "stack.h"
```

```
using namespace std;
int main() {
  Stack<int> intStack(5);
  for (int i = 0; i<5; i++)
    intStack.push(10 - i);
cout << intStack.top() << endl;
  while (!intStack.isEmpty())
    cout << intStack.pop() << ' ';
  cout << endl;
  return 0;
}
```

Объект `intStack` объявляется как экземпляр класса `Stack<int>` (произносится как «`Stack` типа `int`»). При генерации исходного кода для класса `Stack` типа `int` компилятор заменит параметр `T` на тип `int`.

Когда создается объект `intStack` типа `Stack<int>`, конструктор класса `Stack` создает массив элементов типа `int`, представляющий элементы данных стека. Компилятор замещает оператор

```
start=new T[size];
```

в описании шаблона класса `Stack` оператором

```
start=new int[size];
```

в шаблонном классе `Stack<int>`.

Шаблон класса `Stack` использовал только один параметр типа в заголовке шаблона. Но в шаблонах имеется возможность использования любого количества параметров. Кроме параметров типа, могут использоваться и *нетиповые параметры*. Например, заголовок можно модифицировать, указав в нем нетиповой параметр `int elements` для задания размера стека:

```
//заголовок для шаблона класса StackParam
//аналога шаблона класса Stack
template <typename T, int elements>
class StackParam {
private:
  T start [elements];    // массив для размещения данных стека
  int head;    //положение вершины стека
  int size;    //размер стека
```

```
public:
  StackParam ():size(elements),head(-1){}    //конструктор
  ~ StackParam () {}                         //деструктор
  void push(const T&);        //добавление элемента в стек
  T pop();                    //извлечение элемента из стека
  T top();          //просмотр элемента на вершине стека
  bool isEmpty() const {  //true, если стек пустой
    return head == -1;
  }
  bool isFull() const {  //true, если стек полон
    return head == size - 1;
  }
};
```

Тогда объявление типа

```
StackParam <int, 100> intStack1;
```

приведет к созданию во время компиляции шаблонного класса `StackParam` с именем `intStack1`, состоящего из 100 элементов данных типа `int`. Этот шаблонный класс будет иметь тип `StackParam<int,100>`.

В описании класса в разделе закрытых данных-членов помещено следующее объявление массива:

```
T start [elements]; //массив для размещения данных стека
```

Поэтому необходимость выделение динамической памяти под массив в конструкторе и освобождение её в деструкторе отпадает.

Если размер класса контейнера, например, массива или стека, может быть определён во время компиляции (например, при помощи нетипового параметра шаблона, указывающего размер), это устранит расходы при динамическом выделении памяти во время выполнения программы.

 Определение размера класса контейнера во время компиляции (например, через нетиповой параметр шаблона) исключает возможность возникновения потенциально неисправимой ошибки во время выполнения программы, если оператору `new` не удастся получить необходимое количество памяти.

Поскольку у шаблона класса изменился список параметров, необходимо внести изменения в определение шаблонов членов-функций класса.

```
// добавление объекта в стек
//в случае успеха возвращается true, в противном случае false
template <typename T, int elements>
void StackParam<T,elements>::push(const T& a) {
  if (isFull()) throw stackFull();
  start[++head] = a;    //элемент помещается в стек
}

//выталкивание элемента из стека
template <typename T, int elements>
T StackParam<T, elements>::pop() {
  if (isEmpty()) throw stackEmpty();
  T a = start[head--];  //элемент выталкивается из стека
  return a;
}

//просмотр элемента из вершины стека
template <typename T, int elements>
T StackParam<T, elements>::top() {
  if (isEmpty()) throw stackEmpty();
  T y = start[head];  //элемент выталкивается из стека
  return y;
}
```

Пример 20. Дана символьная строка, содержащая латинские буквы, цифры и четыре вида скобок: ( ),[ ],{ }, < >. Проверить правильность расстановки скобок — каждой открывающей соответствует закрывающая скобка того же вида. Для решения задачи необходимо использовать стек.

Воспользуемся шаблоном класса `StackParam`.

```
#include <iostream>
#include <string>
#include "paramstack.h"
using namespace std;

class unpair {};

class missRight{};

int main() {
  locale::global(locale(""));
  string str;
  cin >> str;
```

```
  string left = "([{<";
  string right = ")]}>";
  StackParam<char,1024> St;
  char q;
  int pr;
  try {
    for (char c : str) {
      if (left.find(c) != string::npos)
        St.push(c);
      else {
        pr = right.find(c);
        if (pr != string::npos) {
          q = St.pop();
          if (pr != left.find(q))
            throw unpair();
        }
      }
    }
    if (!St.isEmpty())
      throw missRight();
    cout <<"скобки расставлены верно" << endl;
  }
  catch (stackEmpty) {
    cout << "не хватает левой скобки" << endl;
  }
  catch (unpair) {
    cout << "непарные скобки" << endl;
  }
  catch (missRight) {
    cout << "не хватает правой скобки" << endl;
  }
  return 0;
}
```

Принцип проверки основан на том, что каждая открывающаяся скобки заносится в стек, а каждая закрывающаяся выталкивает из стека элемент — открывающуюся скобку. Полученная пара проверяется на соответствие.

При этом возможны три ошибочные ситуации, которые обрабатываются с помощью исключений.

3.2. Коллекции

Пример 21. Реализовать шаблон класса «Линейный односвязный список». Для списка использовать ссылочную организацию.

Воспользуемся для создания шаблона линейного списка шаблоном структуры node:

```
template<typename T>
struct node{
  T data;
  node<T>* next;
  node(T data, node<T>* next) {
    this->data = data;
    this->next = next;
  }
};
template<typename Q>
ostream & operator << (ostream & out, const node<Q> & x) {
  out << x.data;
  return out;
}
```

Шаблон конструктора node содержит два параметра для заполнения информационной и ссылочной частей.

```
node(T data, node<T>* next) {
  this->data = data;
  this->next = next;
}
```

Такая форма конструктора позволяет легко добавлять новый узел в любое место списка. Например:

```
//создание единственного элемента pn1
node<int>* pn1 = new node<int>(5, nullptr);
//добавление элемента перед pn1
node<int>* pn2 = new node<int>(5, pn1);
//добавление элемента за pn1
pn1->next = new node<int>(7, nullptr);
```

Шаблон перегруженной операции вывода в поток для шаблона структуры node в поток не требуется объявлять дружественным, поскольку по умолчанию в структуре все поля открыты. При этом имя параметра типа в шаблоне не обязано совпадать с именем параметра в шаблоне структуры. В нашем примере используются Q и T.

```
template<typename Q>
ostream & operator << (ostream & out, const node<Q> & x) {
  out << x.data; return out;
}
```

Теперь, используя шаблон структуры `node`, описываем шаблон класса `list` для реализации линейного односвязного списка.

```
template<typename T>
class list{
private:
  node<T>* first;
  node<T>* last;
  error err;
public:
  list(): first(nullptr), last(nullptr) {   }
  ~list();
  bool isEmpty() const;
  void addFirst(T x);
  void addLast(T x);
  T getFirst()const;
  T getLast()const;
  T delFirst();
  T delLast();
  //Обработка с предикатом
  int kol(bool(*f) (T));
  void for_each(void(*action)(T&));
  template<typename Q>
  friend ostream & operator<< (ostream &out, const list<Q> &y);
  //в полной реализации могут быть еще функции
};
```

Поскольку в процессе работы возможно возникновение исключительных ситуаций, создаём пустой класс `error`.

```
class error{};
```

Реализация деструктора и функции проверки на пустоту не содержат никаких особенностей.

```
template<typename T>
list<T>::~list(){
  while (first){
    node<T>* cur = first;
    first = first->next;
    delete cur;
  }
  last = first = nullptr;
}
template<typename T>
bool list<T>::isEmpty() const {
  return (first == nullptr);
}
```

Обратим внимание на объявление шаблона дружественной функции. Имя параметра шаблона дружественной функции может быть любым. Чтобы это подчеркнуть мы использовали имя Q, хотя могли оставить T.

```
template<typename Q>
friend ostream & operator<< (ostream &out, const list<Q> &y);
```

При этом в описании реализации шаблона дружественной функции не обязательно использовать то же самое имя параметра шаблона.

```
template<typename T>
ostream & operator << (ostream & out, const list<T> & y) {
  node<T>* p = y.first;
  while (p) {
    out << *p<<" ";
    p = p->next;
  }
  return out;
}
```

В шаблоне функции добавления элемента в начало списка используем шаблон конструктора структуры `node`. В качестве значения второго параметра используем указатель на начало списка `first`. При этом если список пуст, выполняется инициализация не только указателя `first`, но и указателя `last`.

```
template<typename T>
void list<T>::addFirst(T x){
  first = new node<T>(x, first);
  if (!last)
    last = first;
}
```

Обратите внимание, что добавление в конец имеет более сложный алгоритм.

```
template<typename T>
void list<T>::addLast(T x){
  if (first){
      last->next = new node<T>(x, nullptr);
      last = last->next;
  }
  else
    first = last = new node<T>(x, nullptr);
  }
```

В шаблонах функций чтения первого и последнего элементов списка возможны ошибки доступа в случае, если список пуст. При этом выбрасывается исключение. Выбрасываемый объект `err` класса `error` объявлен в закрытой части шаблона класса `list`.

```
template<typename T>
T list<T>::getFirst() const{
  if (first)
    return first->data;
  else
    throw err;
}

template<typename T>
T list<T>::getLast()const{
  if (first)
    return last->data;
  else
    throw err;
}
```

Функции удаления первого и последнего элементов списка в качестве результата возвращают значение удалённого элемента. В случае пустого списка они выбрасывают исключение.

```
template<typename T>
T list<T>::delFirst(){
  T x;
  if (first){
    x = first->data;
    node<T>*t = first;
    first = first->next;
    delete t;
    return x;
  }
  else
    throw err;
}
template<typename T>
T list<T>::delLast(){
  T x;
  if (first){
    x = last->data;
    node<T>*t = first;
    while (t->next != last)
      t = t->next;
```

```
    delete last;
    last = t;
    last->next = nullptr;
    return x;
  }
  else
    throw err;
}
```

Шаблон функции `int kol(bool(*f)(T))` реализует подсчёт количества элементов, для которых выполняется условие, задаваемое одноместным предикатом. Предикат задаётся указателем на функцию, соответствующую типу `bool(*f)(T)`.

```
//Обработка с предикатом
template<typename T>
// f — переменная типа "указатель на функцию"
int list<T>::kol(bool(*f) (T)) {
  int k = 0;
  for (node<T>* p = first; p; p = p->next)
    if (f(p->data))
      k++;
  return k;
}
```

В качестве примера такой функции для специализации шаблона списка типом `int` использована функция проверки на нечётность.

```
bool odd(int x){
  return x % 2 != 0;
}
```

Шаблон функции `void for_each(void(*action)(T&))` реализует применение функции `action` к информационному полю каждого элемента списка. Параметр `action` является указателем на функцию, соответствующую типу `void(*action)(T&)`.

```
template <typename T>
// action — переменная типа "указатель на функцию"
void list<T>::for_each(void(*action)(T&)) {
  node<T>* p = first;
  while (p) {
    action(p->data);
    p = p->next;
  }
}
```

В качестве примера такой функции для специализации шаблона списка типом int использована функция удвоения значения.

```
void mult2(int & x) {
  x *= 2;
}
```

В функции main использованы специализации шаблона списка типами int и char.

```
int main(void){
  list<int> l1;
  l1.addFirst(3);
  l1.addLast(4);
  l1.addLast(99);
  cout << l1<<endl;
  cout << l1.kol(odd)<<endl;
  l1.for_each(mult2);
  cout << l1 << endl;
  cout << l1.kol(odd) << endl;

  list<char> *l2 = new list<char>;
  l2->addFirst('1');
  l2->addLast('q');
  l2->addLast('a');
  cout << *l2 << endl;
  cout << *l2 << endl;
  cout << l2->delFirst() << endl;
  cout << *l2 << endl;
  cout << l2->delLast() << endl;
  cout << *l2 << endl;

  return 0;
}
```

Пример 22. Реализовать шаблон итератора для шаблона класса «Линейный односвязный список».

Предположим, что уже имеется шаблон класса линейный односвязный список из примера 21. Необходимо только добавить в него методы, возвращающие итераторы на начало и конец списка.

```
template<typename T>
class listIterator;
template<typename T>
class list{
```

```
private:
  node<T>* first;
  node<T>* last;
  error err;
public:
...
  listIterator<T> list<T>::begin() const;
  listIterator<T> list<T>::end() const;
};
```

В шаблоне итератора для шаблона линейного односвязного списка определены: операция доступа к элементу, префиксная операция инкремента и две операции сравнения итераторов. Реализация шаблона итератора аналогична реализации итератора из примера 16.

```
template <typename T>
class listIterator {
public:
  class Error {
  public:
    void what(){
      cout << "Iterator error" << endl;
    }
  };
private:
  const list<T> *collection;
  node<T> *cur;
public:
  listIterator(const list<T> *s, node<T> *e) :
              collection(s), cur(e){}
  T operator *(){
  if (cur)
      return cur->data;
    else
      throw Error();
  }
  listIterator operator++(); //префиксный ++
  bool operator == (const listIterator<T> &ri) const;
  bool operator != (const listIterator<T> &ri) const;
};

template <typename T>
listIterator<T> listIterator<T>:: operator++() {
  if (cur) {
    cur = cur->next;
    return *this;
  }
```

```
  else throw Error();
}

template <typename T>
bool listIterator<T>:: operator==(const listIterator<T> &ri)
const {
  return ((collection == ri.collection) && (cur == ri.cur));
}

template <typename T>
bool listIterator<T>:: operator!=(const listIterator<T> &ri)
const {
  return !(*this == ri);
}

template <typename T>
listIterator<T> list<T>::begin() const {
  return listIterator<T>(this, first);
}

template <typename T>
listIterator<T> list<T>::end() const {
  listIterator<T> iter(this, nullptr);
  return iter;
}
```

Пример 23. Создать шаблон класса `matrix` для представления матрицы как вектора, элементами которого являются векторы.

На основе класса `myvector` из примера 8 создадим шаблон класса `myvector<T>`.

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
  myvector(int s = 1): size(s){
    vect = new T[s];
  }

  myvector(const myvector<T> & v): size(v.size){
    vect = new T[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }
```

```
  myvector(myvector<T>&& v){
    size = v.size;
    vect= v.vect;
    v.vect = nullptr;
  }

  ~myvector(){
    if (vect != nullptr)
    delete[] vect;
  }

  T& operator [] (int i){
    return vect[i];
  }
  T operator [] (int i) const {
    return vect[i];
  }

  myvector<T>& operator= (const myvector<T> &v){
    if (&v != this) {
      delete[] vect;
      vect = new T[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }

  myvector<T>& operator=(myvector<T>&& v){
    if (vect != nullptr)
      delete[] vect;
    size = v.size;
    vect = v.vect;
    v.vect = nullptr;
    return *this;
  }

  myvector<T> operator+(const myvector<T>& v){
    if (size == v.size) {
      myvector<T> v1(size);
      for (int i = 0; i < size; ++i)
        v1[i] = vect[i] + v[i];
      return v1;
    }
    return *this;//лучше исключение использовать
  }
};
```

Попробуем объявить матрицу как вектор векторов.

```
myvector<myvector<int>> m(3);
```

Обратим внимание, что можно задать только одну размерность — количество строк, поскольку негде указать количество столбцов. Это связано с тем, что `myvector<int>` — это параметр типа для шаблонного класса `myvector<myvector<int>>`. При этом для каждого элемента `m[i]` будет вызван конструктор без параметров, в нашей реализации для него задано значение размера массива по умолчанию 1.

Чтобы изменить количество столбцов в матрице, необходимо для каждой строки выполнить функцию изменения размера `resize`. Добавим её в шаблон класса `myvector<T>`.

```
template <typename T>
class myvector {
private:
  int size;
  T * vect;
public:
...
  void resize(int n){
    T* newVect = new T[n];
    int sz = (n < size) ? n: size;
    for (int i = 0; i<sz; ++i)
      newVect[i] = vect[i];
    delete[] vect;
    vect = newVect;
    size = n;
  }
};
```

Используем эту функцию в конструкторе шаблона класса матрицы на базе шаблона класса вектора.

```
template<typename T>
class matrix{
  int rows;
  myvector<myvector<T>> mdata;

public:
  matrix(int m, int n): rows(n), mdata(m){
    for (int i = 0; i < m; i++)
```

```
        mdata[i].resize(n);
    }
};
```

Обратим внимание на то, что вызывать конструктор `mdata(m)` в теле конструктора `matrix` уже поздно (уже отработает конструктор по умолчанию), а при объявлении еще рано, поэтому конструктор необходимо вызывать в списке инициализации.

Деструктор для данного класса писать не надо, т.к. достаточно сгенерированного деструктора по умолчанию `~matrix(){}`. Поскольку деструктор по умолчанию вызывает деструкторы для всех полей класса, то будет вызван деструктор шаблона класса `myvector<myvector<T>>`.

Конструктор копии и `operator=` для шаблона класса `matrix` также сгенерируются автоматически и будут работать правильно.

 Если в классе есть подобъект, который берёт на себя функции по выделению и освобождению динамической памяти, то определять конструктор копии и `operator=` не нужно. Они необходимы, только если непосредственно в конструкторе данного класса выделяется динамическая память, а в деструкторе возвращается.

Автоматически сгенерированный конструктор копии вызывает конструкторы копий для всех своих полей. Автоматически сгенерированная операция присваивания вызывает операции присваивания для всех своих полей. Например, сгенерированный конструктор копии будет выглядеть так:

```
matrix(const matrix<T> & mm): mdata(mm.mdata) {}
```

Доступ к элементу матрицы по индексам можно реализовать двумя способами.

Первый способ предполагает перегрузку операции индексации для матрицы с возвращением вектора (по номеру строки). Элемент из этой строки возвращается перегруженной для вектора операцией индексации.

```
myvector<T>& operator [] (int i) {
  return mdata[i];
}

myvector<T> operator [] (int i)const {
  return mdata[i];
}
```

Второй способ предполагает перегрузку операции вызова функции `()` с двумя параметрами.

```
T& operator()(int i, int j){
  return mdata[i][j];
}

 T operator() (int i, int j) const {
  return mdata[i][j];
}
```

ЛИТЕРАТУРА

1. *Дейтел Х. М., Дейтел П. Дж.* Как программировать на C++. – М.: ЗАО «Издательство БИНОМ», 2000. – 1024 с.
2. *Демяненко Я.М., Чердынцева М.И.* Методы процедурного программирования в C++. – Ростов-на-Дону: Издательство Южного федерального университета, 2014. – 207 с.
3. *Кениг Эндрю, Му Барбара.* Эффективное программирование на C++. Практическое программирование на примерах. – М.: Издательский дом «Вильямс», 2002. – 384 с.
4. *Керниган Б., Ритчи Д.* Язык программирования Си. – М.: Финансы и статистика, 1992. – 250 с.
5. *Лишнер Рэй.* C++. Справочник. – СПб.: Питер, 2005. – 907 с.
6. *Русанова Я.М., Чердынцева М.И.* C++ как второй язык обучения приёмам и технологиям программирования. – Ростов-на-Дону: Издательство Южного федерального университета, 2010. – 200 с.
7. *СкоттМайерс*. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2010 – 302 с
8. *Страуструп Бьерн.* Язык программирования C++. Специальное издание. – М.: ООО «Бином-Пресс», 2004. – 1104 с.
9. *Эккель Брюс.* Философия C++. Введение в стандартный C++. – СПб.: Питер, 2004. – 572 с.
10. *Эккель Брюс, Эллисон Чак.* Философия C++. Практическое программирование. – СПб.: Питер, 2004. – 608 с.

11. *П. Дж. Плаугер, Александр Степанов, Менг Ли, Дэвид Р. Мюссер*. STL – стандартная библиотека шаблонов C++. – СПб.: БХВ-Петербург, 2004. – 656 с.
12. *Дэвид Р. Мюссер, Жилмер Дж. Дердж, Атул Сейни*. C++ и STL. Справочное руководство. – М.: Вильямс, 2010. – 432 с.

Учебное издание

Демяненко Яна Михайловна
Чердынцева Марина Игорьевна

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В C++

Учебное пособие

Компьютерная верстка *Я. М. Демяненко, М. И. Чердынцева*

Формат 60x841/16. Бумага офсетная. Гарнитура Times.
Печать офсетная. Усл. печ. л. . Уч.-изд. л. 6,5.
Тираж 50 экз. Заказ № от 2018

Издательство Южного федерального университета

Отпечатано в типографии ЮФУ.
344090, г. Ростов-на-Дону, пр. Стачки, 200/1. Тел. 247-80-51.