Министерство науки и высшего образования Российской Федерации
ФГБОУ ВО «Удмуртский государственный университет»
Институт математики, информационных технологий и физики
Кафедра вычислительной механики

А.В. Селезнева

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык высокого уровня C++

Часть II

Учебное пособие

Ижевск
2023

УДК 004.438(075.8)
ББК 32.973.22я73
С29

Рекомендовано к изданию Учебно-методическим советом УдГУ

Рецензенты: канд. физ.-мат. наук, доцент, факультет Математика и естественные науки, каф. Прикладная математика и информационные технологии ИжГТУ К.И. Дизендорф, канд. физ.-мат. наук, доцент, каф. дифференциальных уравнений ИМИТиФ УдГУ Т.С. Быкова.

Селезнева А.В.

С29 Основы программирования. Язык высокого уровня С++. Ч. II : учеб. пособие – Ижевск : Удмуртский университет, 2023. –151 с.

ISBN 978-5-4312-1105-8

Учебное пособие состоит из двух частей, каждая из которых содержит теоретический материал, примеры задач с их реализацией, ряд лабораторных работ. Первая часть данного пособия знакомит читателя с основными понятиями программирования, его базовыми конструкциями, средами разработки. Во второй части представлены основы объектно-ориентированного программирования и его базовые принципы. Учебное пособие направлено на приобретение студентами общих знаний и навыков программирования на высокоуровневом языке С++.

Пособие предназначено для студентов, обучающихся по направлению 02.00.00 «Компьютерные и информационные науки».

УДК 004.438(075.8)
ББК 32.973.22я73

ISBN 978-5-4312-1105-8

Оглавление

Введение

Учебное пособие представляет собой введение в алгоритмизацию и программирование. Основной его целью является изложение базовых сведений о программировании и основ программирования на языке C++. Пособие предназначено для студентов, обучающихся по направлению «Компьютерные и информационные науки», а также для студентов, обучающихся по другим направлениям и изучающих программирование и алгоритмизацию.

Данное учебное пособие состоит из двух частей. В первой части пособия излагаются основы синтаксиса и семантики языка программирования C++, алгоритмические конструкции, элементарная работа в интегрированных средах разработки. Рассматриваются, с подробными пояснениями, простые для понимания примеры задач с их реализацией [1]. Во второй части представлены основы объектно-ориентированного программирования и его базовые принципы.

Теоретический материал изложен в последовательности от простого к сложному, что позволяет поэтапно изучать материал, не испытывая трудностей в понимании. Практическая часть пособия содержит лабораторный практикум по четырем темам в 25 вариантах.

Учебное пособие ориентировано как на читателя, не имеющего знаний и опыта программирования, так и на тех, кто имеет первоначальные знания и небольшой опыт в алгоритмизации и программировании. Основной задачей обучающегося по данному пособию является приобретение навыков программирования базового уровня.

1 Работа с функциями

1.1 Перегрузка функций

До этого момента при написании программы предполагалось, что все функции в программе должны иметь уникальные имена. Однако, язык программирования C++ позволяет создавать функции с одинаковыми идентификаторами, при условии, что такие функции будут отличатся набором параметров и/или их типом. Такую возможность языка принято называть **перегрузкой функций**.

Вызов перегруженных функций производится аналогично вызову обычной функции, т.е. указывается имя функции и список передаваемых параметров. Компилятор автоматически определяет (анализируя количество аргументов и их типы данных), какая именно функция вызывается.

Если при одинаковом списке параметров функции будут различаться только типом возвращаемого значения, компилятор сообщит об ошибке неоднозначности [2].

Примечание: хорошим стилем считается, когда перегруженные функции решают схожие задачи. Благодаря такому подходу программный код становится более читаемым.

Рассмотрим реализацию перегрузки функций на примере решения задачи 1.1.

Задача 1.1. Написать функцию сложения двух целых чисел. Перегрузить данную функцию для целочисленного и вещественного аргументов. Результат вычислений вывести в консоль.

Реализация задачи 1.1 приведена в листинге 1.

Листинг 1. Перегрузка пользовательской функции

```
1 #include <iostream>
2
3 using namespace std;
4 // функция сложения двух целых чисел
5 int summ(int num1, int num2){
6     return num1+num2;
```

```
7 }
8 // функция сложения двух чисел – целого и вещественного
9 float summ(int num1, float num3){
10     return num1+num3;
11 }
12
13 int main(){
14     setlocale(LC_CTYPE, "");
15     int num1, num2;
16     float num3;
17     cout << "Введите целое число A: "; cin >> num1;
18     cout << "Введите целое число B: "; cin >> num2;
19     cout << "Введите вещественное число C: "; cin >> num3;
20     cout << "A + B = " << summ(num1, num2) << endl;
21     cout << "A + C = " << summ(num1, num3) << endl;
22     system("pause");
23     return 0;
24 }
```

В листинге 1 приведен пример перегрузки функции `summ`, программный код содержит два варианта ее реализации. В первом случае функция складывает два целых числа (строки 5–7), во втором случае функция складывает целое и вещественные числа (строки 9–11). Обе функции называются одинаково, но имеют различные типы возвращаемого значения и различные типы второго аргумента. Обратим внимание, что компилятор сам «понимает», какую из функций необходимо использовать.

Результат работы программного кода из листинга 1 представлен на рисунке 1.

```
Введите целое число A: 4
Введите целое число B: 3
Введите вещественное число C: 1.3
A + B = 7
A + C = 5.3
```

Рис. 1. Результат работы программы из листинга 1

Примечание: функции не могут быть перегружены [3], если

набор их параметров отличается только использованием ссылки или модификатором `const`.

1.2 Шаблонные функции

При разработке программного обеспечения существует очень важный принцип «Don't repeat yourself» или DRY, который гласит «Не повторяйся». Данный принцип был сформулирован Энди Хантом и Дейвом Томасом. Принцип говорит о том, что нужно максимально уменьшать повторение участков программного кода во избежание избыточности. В результате код становится короче и чище. Рассмотрим проблему избыточности на примере решения простых задач.

Задача 1.2. Пусть имеются две переменные, значение которых пользователь вводит в консоли. Требуется написать функцию, проверяющую, равны ли данные переменные. Если значения переменных совпадают, вывести сообщение «Переменные равны», в противном случае «Переменные НЕ равны».

На первый взгляд задача очень простая. Для работы программы с целыми числами программный код будет небольшим и простым (листинг 2).

Листинг 2. Решение задачи 1.2

```
1 #include <iostream>
2
3 using namespace std;
4
5 int equality(int num1, int num2){
6         setlocale(LC_ALL,"rus");
7         if(num1==num2) cout<<"Переменные равны";
8         else cout<<"Переменные НЕ равны";
9         return 0;
10 }
11
12 int main(){
13         setlocale(LC_ALL,"rus");
14         int num1,num2;
15         cout<<"Введите первое число: "; cin>>num1;
```

```
16          cout<<"Введите второе число: "; cin>>num2;
17          equality(num1,num2);
18          return 0;
19 }
```

Программный код из листинга 2 работает только при условии, что пользователь будет работать с целыми числами, таким образом задача решена только для частного случая. В условиях задачи не сказано, что пользователь будет вводить целые числа, следовательно, необходимо расширить исходную программу. Можно воспользоваться перегрузкой и добавить функцию для проверки дробных чисел на их равенство (листинг 3).

Листинг 3. Решение задачи 1.2 с помощью перегруженной функции

```
1 #include <iostream>
2
3 using namespace std;
4
5 int equality(int num1, int num2){
6          setlocale(LC_ALL,"rus");
7          if(num1==num2) cout<<"Переменные равны";
8          else cout<<"Переменные НЕ равны";
9          return 0;
10 }
11 float equality(float num1, float num2){
12         setlocale(LC_ALL,"rus");
13         if(num1==num2) cout<<"Переменные равны";
14         else cout<<"Переменные НЕ равны";
15         return 0;
16 }
17
18 int main(){
19         setlocale(LC_ALL,"rus");
20         int num1,num2;
21         float num3, num4;
22         cout<<"Введите первое число: "; cin>>num1;
23         cout<<"Введите второе число: "; cin>>num2;
24         cout<<"Введите первое дробное число: "; cin>>num3;
25         cout<<"Введите второе дробное число: "; cin>>num4;
26         equality(num1,num2);
```

```
27          return 0;
28 }
```

Заметим, что логика второй функции не изменилась, изменился лишь тип данных аргументов в самой функции. Возникает вопрос, а как быть в случае, если необходимо сравнивать переменные любого допустимого типа данных? Можно воспользоваться перегрузкой функций, но в результате получится очень большой программный код с одинаковыми функциями, различие которых будет только в типах данных. Для того, чтобы избежать подобной избыточности были предложены шаблонные функции или шаблоны.

Шаблонная функция (template function) — это функция, полностью контролирующая соответствие типов данных, которые задаются ей как параметры [2].

Объявление шаблонной функции начинается с ключевого слова `template`, после чего в угловых скобках $< >$ пишется ключевое слово `typename`, а далее следует имя типа, которое будет являться формальным аргументом для обобщенного типа данных. После угловых скобок указывается тип возвращаемого значения и имя пользовательской функции с набором аргументов и их типом данных.

Общий синтаксис объявления шаблонной функции:

```
template <typename аргумент_шаблона>
тип_возвращаемого_значения имя_функции(аргументы)
{
//тело функции
}
```

Пример объявления шаблонной функции:

```
template <typename A> void summ(A num1, A num2)
{
//тело функции
}
```

При объявлении шаблонной функции в угловых скобках вместо ключевого слова `typename` можно использовать ключевое слово `class`. Переменная, следующая за словом `typename` (или `class`, например, X и Y в листинге 4), называется **аргументом шаблона** [4]. Аргумент шаблона может иметь любое допустимое в языке программирования C++ имя.

Вызов шаблонной функции полностью совпадает с вызовом обычной пользовательской функции, т.е. пишется имя функции и набор передаваемых аргументов. Рассмотрим на примере решения задачи 1.2 реализацию шаблонной функции (листинг 4).

Листинг 4. Пример шаблонной функции с двумя аргументами

```
1 #include <iostream>
2 using namespace std;
3
4 //шаблонная функция
5 template <class X, class Y> void equality(X num1, Y num2){
6     setlocale(LC_CTYPE, "");
7     if(num1==num2) cout<<"Переменные равны";
8         else cout<<"Переменные НЕ равны";
9 }
10 int main(){
11     setlocale(LC_ALL,"rus");
12     int num1 = 3;
13     int num2 = 3;
14     float num3 = 1.4;
15     float num4 = 5.8;
16     cout<<"Программа сравнения двух чисел"<<endl;
17     cout<<"Результат сравнения чисел num1=3 и num2=3"<<endl;
18     equality(num1, num2); cout<<endl;
19     cout<<"Результат сравнения чисел num3=1.4 и num2=5.8"<<
       endl;
20     equality(num3, num4); cout<<endl;
21     return 0;
22 }
```

Шаблонная (обобщенная) функция сравнения двух чисел реализована в строках 5–9. В качестве типа возвращаемого значения используется `void`, так как функция ничего не возвращает,

а лишь выводит сообщение о результате сравнения двух чисел. В строках 18 и 20 производится вызов функции `equality`. Обратим внимание, что вызов функции в обоих случаях идентичен, несмотря на то, что тип аргументов различен. При вызове функции определяется, какие типы данных необходимо использовать вместо обозначенных X и Y. Так, при вызове функции в строке 18 в качестве типов данных X и Y используется тип `int`, а в строке 20 – `float`.

Ознакомиться с результатом работы программного кода из листинга 4 можно на рисунке 2.

```
Программа сравнения двух чисел
Результат сравнения чисел num1=3 и num2=3
Переменные равны
Результат сравнения чисел num3=1.4 и num2=5.8
Переменные НЕ равны
```

Рис. 2. Результат работы программного кода из листинга 4

2 Классы и объекты

В этой главе мы переходим к изучению обобщенного программирования [5] (или объектно-ориентированного программирования). Объектно-ориентированное программирование (сокращено ООП) – это методология программирования, в основе которой заложена работа с совокупностью объектов, имеющих свои свойства и методы. Использование объектно-ориентированного программирования позволяет очень просто представлять сложные структуры данных, объединяя в единое целое данные и действия, производимые над этими данными. Идея объектно-ориентированного программирования строится на трех базовых принципах: *инкапсуляции, наследовании и полиморфизме.*

2.1 Основные понятия и определения

Одним из основных понятий объектно-ориентированного программирования является понятие класса. Класс в C++ является абстрактным типом данных, построенный на основе других типов данных и функций (с этой стороны он схож со структурами, о которых писалось в первой части данного пособия). Он помогает описывать объекты с их свойствами, а также правила их взаимодействия.

Класс является определяемым пользователем типом данных. Он состоит из встроенных типов, других пользовательских типов и функций [6]. Тип данных, созданный пользователем, обладает, практически, теми же свойствами, что и стандартные типы данных [7]. Однако, у классов имеется существенное отличие — возможность скрывать и защищать данные от пользователя за интерфейсом.

Классы в C++ делятся на локальные и глобальные. **Локальные классы** объявляются внутри блока (например, внутри другого класса или функции). **Глобальные классы** объявляются вне любого блока [7].

Разберем основные термины, которые неразрывно связа-

ны с понятием класса в языке программирования C++: объекты класса, методы класса, поля класса.

Объектом класса (экземпляром класса) называется переменная типа класс. **Методы класса** – это функции, которые содержатся в классе, и в их функционале используются поля класса. **Поля класса** – это обычные переменные, которые содержатся в классе и отражают свойства объекта. Класс может содержать любое количество полей, но может и не содержать ни одного поля. Методы и поля класса называются его *элементами*.

В качестве примера класса можно рассмотреть книгу. Объектом класса «книга» будет является некоторая конкретная книга. У этой книги имеются автор, год издания, количество страниц, количество экземпляров книги, жанр и т.д. – данные, которые являются полями (свойствами) класса. В качестве методов могут выступать изменение количества книг, вычисление стоимости (цена×количество) и др.

Объявление класса начинается с ключевого слова `class`, после чего следует идентификатор класса и фигурные скобки, в которых содержится тело класса. В конце класса, по аналогии со структурами, ставится точка с запятой.

Общий синтаксис объявления класса:

```
class Имя_класса {
//закрытые члены класса и методы
public:
//открытые члены класса и методы
}список объектов класса;
```

или

```
class Имя_класса {
private:
//закрытые члены класса и методы
public:
//открытые члены класса и методы
}объекты класса;
```

Объекты класса указывать после его объявления не обязательно. Из синтаксиса объявления класса видно, что его поля могут быть открытыми и закрытыми, подробно об этом будет написано в следующем параграфе. Сначала принято определять закрытые поля и методы класса, а после – открытые, хотя нарушение этого порядка не приведет к синтаксической ошибке. Идентификатор класса, по аналогии со структурами, принято писать с прописной буквы. Рассмотрим реализацию класса на примере задачи 2.1, решение которой представлено в листинге 5.

Задача 2.1. Написать программу, реализующую класс «Треугольник». Поля класса – длины сторон треугольника. Вывести длины сторон треугольника в консоль.

Листинг 5. Пример реализации класса «Треугольник»

```
1 #include <iostream>
2 using namespace std;
3
4 //объявление класса Треугольник
5 class Triang{
6     //открытые поля класса
7     public:
8     int a, b, c;
9 };
10
11 int main(){
12     //локализация
13     setlocale(LC_CTYPE, "");
14     //объявление объекта класса
15     Triang my_triang;
16     //задаем полям объекта класса значения
17     my_triang.a = 3;
18     my_triang.b = 4;
19     my_triang.c = 5;
20     //вывод результата в консоль
21     cout << "Длина стороны треугольника A: " << my_triang.a
      << endl;
22     cout << "Длина стороны треугольника B: " << my_triang.b
      << endl;
23     cout << "Длина стороны треугольника C: " << my_triang.c
```

```
    << endl;
24      system("pause");
25      return 0;
26  }
```

Объявление класса «Треугольник» (`Triang`) начинается со строки 5, данный класс содержит три поля: переменные целого типа `a, b, c`. Поля класса объявляются точно также, как и обычные переменные (строка 8). На данном этапе класс не содержит ни одного метода, а все его поля являются открытыми (строка 7). Для работы с классом в строке 15 был создан объект `my_triang` класса `Triange`, имя класса выступает в качестве типа (пользовательского) данных. Несмотря на то, что объект класса создан, значения его полям еще не были заданы. В строках 17–19 полям объекта `my_triang` присваиваются значения. Для обращения к полю объекта используется оператор «.» (точка). В строках 21–23 реализован вывод информации в консоль.

Результат работы программного кода из листинга 5 представлен на рисунке 3.

```
Длина стороны треугольника A: 3
Длина стороны треугольника B: 4
Длина стороны треугольника C: 5
```

Рис. 3. Результат работы программы из листинга 5

2.2 Модификаторы доступа

Одним из трех базовых принципов ООП, как говорилось ранее, является инкапсуляция. **Инкапсуляция** – это механизм языка C++, ограничивающий доступ к составляющим компонентам (методам и атрибутам) объекта. Данный механизм позволяет сделать поля и методы защищенными или приватными, то есть доступными только внутри объекта.

Инкапсуляция реализуется с помощью модификаторов доступа. Существует три метки, задающих область видимости: `public, private и protected`, которые называют **модификаторами (спецификаторами) доступа**.

По умолчанию все поля и методы класса являются закрытыми, т.е. если в теле класса не прописать спецификатор доступа, то к полям и методам класса нельзя будет обратиться вне этого класса. Как правило, поля класса оставляют закрытыми, а пользователю дают возможность взаимодействовать с этими полями с помощью открытых методов.

Модификатор private

Поля, относящиеся к модификатору `private` являются *закрытыми*. Доступ к ним возможен лишь внутри класса или с помощью друзей класса (об этом будет сказано далее), имеющим доступ к этим закрытым данным. Закрытыми члены класса делают для ограничения несанкционированного внешнего доступа к этим членам, а также для упрощения процесса обработки объектов за счет формального уменьшения количества доступных атрибутов [8].

Модификатор public

Поля, относящиеся к модификатору `public` являются *открытыми*. Доступ к ним возможен из любого места программы любым функциям. Стоит отметить, что не рекомендуется делать открытыми все поля и методы класса, это приводит к уязвимости программы.

Модификатор protected

Поля, относящиеся к модификатору `protected` являются *защищенными*. Доступ к ним возможен внутри класса, производным классам и друзьям класса [2].

2.3 Методы. Перегрузка методов

Функции, принадлежащие классу, помогающие взаимодействовать с полями класса, другими объектами класса или объектами других классов называются **методами**. Метод класса можно вызвать только в том случае, если создан объект этого класса. Вызов методов осуществляется с помощью оператора «.» точка (по аналогии со структурами).

Общий синтаксис вызова метода класса для определенного объекта:

```
имя_объекта.имя_метода(аргументы);
```

Объявление метода класса аналогично объявлению обычной функции: сначала указывается тип возвращаемого значения, далее – имя метода и список аргументов.

Общий синтаксис объявления метода:

```
тип_результата имя_метода (агрументы){
тело метода
}
```

Для более читаемого кода описание метода можно вынести за пределы класса, тогда внутри самого класса необходимо указать прототип.

Общий синтаксис описания метода за пределами класса:

```
тип_результата имя_класса::имя_функции(аргументы){
тело функции
}
```

Рассмотрим реализацию методов класса на примере решения задачи 2.2.

Задача 2.2. Написать программу, реализующую класс «Треугольник». Поля класса – длины сторон треугольника. Методы класса – ввод длин сторон треугольника, вывод информа-

ции о треугольнике в консоль (если треугольник с заданными сторонами существует).

Листинг 6. Пример использования методов класса

```
#include <iostream>
using namespace std;
//Определение класса Треугольник
class Triang{
    //закрытые поля класса
    int a, b, c;
    //открытые методы класса
    public:
    void set_info(int x, int y, int z);
    void print_info();
};

int main(){
    //локализация
    setlocale(LC_CTYPE, "");
    //создание объекта класса
    Triang my_triang;
    //объявление переменных
    int x, y, z;
    cout<<"Введите длину стороны A: "; cin>>x;
    cout<<"Введите длину стороны B: "; cin>>y;
    cout<<"Введите длину стороны C: "; cin>>z;
    //вызов методов класса
    my_triang.set_info(x, y, z);
    my_triang.print_info();
    return 0;
}
//описание методов класса
void Triang::set_info(int x, int y, int z){
    a = x;
    b = y;
    c = z;
};

void Triang::print_info(){
    if (a+b>c && a+c>b && b+c>a){
        cout<<"Длина стороны A: "<<a<<endl;
        cout<<"Длина стороны B: "<<b<<endl;
```

```
39          cout<<"Длина стороны C: "<<c<<endl;}
40      else
41          cout<<"Трегольник с данными сторонами не существует"<<
        endl;
42  };
```

Класс «Треугольник» определен в строках 4–11, он имеет закрытые поля – длины сторон треугольника (строка 6) и открытые методы – метод задания значений полям объекта класса (строка 9) и метод вывода информации об объекте в консоль (строка 10). Отметим, что в строках 9 и 10 содержатся прототипы методов, а их описание вынесено за пределы класса (строки 29–42). Для работы с классом в строке 17 создается его объект с именем `my_triang`. В строках 24–25 осуществляется вызов методов для данного объекта. С результатом работы программного кода из листинга 6 можно ознакомится на рисунках 4–5. На рисунке 4 представлен результат работы программы при корректно введенных данных (треугольник существует), а на рисунке 5 отражен результат, если не выполнено необходимое и достаточное условие существования треугольника.

```
Введите длину стороны A: 3
Введите длину стороны B: 4
Введите длину стороны C: 5
Длина стороны A: 3
Длина стороны B: 4
Длина стороны C: 5
```

Рис. 4. Результат работы программы из листинга 6

```
Введите длину стороны A: 1
Введите длину стороны B: 1
Введите длину стороны C: 2
Трегольник с данными сторонами
не существует
```

Рис. 5. Результат работы программы из листинга 6

Методы класса, как и обычные функции, можно перегружать. По аналогии с функциями, создается несколько вариантов одного и того же метода с разным набором аргументов и/или типом возвращаемого значения. Рассмотрим перегрузку метода класса на примере решения задачи 2.3, ее решение представлено в листинге 7.

Задача 2.3. Написать программу, реализующую класс

«Точка». Поля класса – координаты точки x и y. Метод класса – инициализация полей объектов класса. Перегрузить метод для дробных чисел.

Листинг 7. Пример перегрузки метода

```
#include <iostream>
using namespace std;
//описание класса Point
class Point{
    public:
    //поля класса
    int x_i, y_i;
    double x_f, y_f;
    //прототипы методов класса
    void set_point(int x, int y);
    void set_point(double x, double y);
} point_1, point_2;
int main(){
    setlocale(LC_CTYPE, "");
    //вызов метода set_point
    point_1.set_point(3, 7);
    point_2.set_point(0.1, 10.2);
    //вывод результатов
    cout<<"Координаты первой точки: \n";
    cout<<"X = "<<point_1.x_i<<"\t Y = " << point_1.y_i<<
    endl;
    cout<<"Координаты второй точки: \n";
    cout<<"X = "<<point_2.x_f<<"\t Y = "<< point_2.y_f<<
    endl;
    cout<<endl;
    return 0;
}
//описание методов
void Point::set_point(double x, double y){
    x_f = x;
    y_f = y;
}
void Point::set_point(int x, int y){
    x_i = x;
    y_i = y;
}
```

Рассмотрим подробнее листинг 7. Его программный код содержит один класс – `Point`, который имеет четыре поля и перегруженный метод `set_point()`. Данный метод имеет две реализации – для целых и дробных чисел. В строке 12 создаются два объекта класса – `point_1` и `point_2`. Для данных объектов в строках 17–18 вызывается метод `set_point()`, но в зависимости от передаваемых аргументов для одного объекта вызывается метод, работающий с целыми числами (объект `point_1`), а для другого (объект `point_2`) – с дробными числами. Обратим внимание, что тип возвращаемого значения и количество аргументов остались неизменными, но изменился тип передаваемых параметров, и, в зависимости от того, какие параметры передаются, вызывается та или иная реализация метода. Результат работы программы представлен на рисунке 6.

```
Координаты первой точки:
X = 3   Y = 7
Координаты второй точки:
X = 0.1  Y = 10.2
```

Рис. 6. Результат работы программы из листинга 7

2.4 Работа с объектами

2.4.1 Функции и объекты

Как и в случае с переменными, объекты можно изменять и передавать функциям и методам. Передача аргументами объектов – подход достаточно эффективный, поскольку позволяет обрабатывать целиком объекты, а не передавать каждое поле отдельно [8]. Синтаксис передачи объекта в качестве аргумента функции аналогичен обычной базовой переменной, т.е. указывается тип объекта (имя класса, которому принадлежит объект) и имя самого объекта.

Общий синтаксис передачи объекта функции:
`имя_функции (тип_объекта имя_объекта)`

Задача 2.4. Написать программу, реализующую класс «Точка». Поля класса – координаты точки x и y. Программа должна содержать функцию, выводящую информацию об объекте (координаты точки) в консоль (в функцию в качестве аргумента передается объект).

Решение задачи 2.4 представлено в листинге 8, с результатом работы программного кода из данного листинга можно ознакомиться на рисунке 7.

Листинг 8. Передача объекта функции

```
#include <iostream>
using namespace std;
//описание класса Point
class Point{
    public:
    int x, y;
};
//функция вывода информации об объекте
void print_info(Point obj){
    cout<<"x = "<<obj.x<<"\t y = "<<obj.y<<endl;
}

int main()
{
    setlocale(LC_CTYPE, "");
    //создание объекта класса
    Point my_point;
    //инициализация полей объекта класса
    my_point.x = 0;
    my_point.y = 0;
    cout << "информация об объекте my_point:" << endl;
    //вызов функции
    print_info(my_point);
    return 0;
}
```

Программный код из листинга 8 содержит один класс – «Point» (описание класса производится в строках 4–7), который содержит два открытых целочисленных поля (строка 6). В строках 9–11 описана внешняя функция вывода информации об объекте в консоль, в качестве аргумента функция принимает объект класса `Point`. Вызов функции производится в строке 23, в функцию передается аргумент `my_point` – объект класса `Point`.

```
информация об объекте my_point:
x = 0    y = 0
```

Рис. 7. Результат работы программы из листинга 8

Кроме того, что функция может принимать в качестве параметра объект, она также может и возвращать его. Функция, возвращающая объект, должна иметь в качестве типа возвращаемого значения имя класса соответствующего объекта, и кроме того, она должна содержать локальный объект данного класса.

Рассмотрим пример программы с функцией, возвращающей объект класса (листинг 9). Результат работы данной программы представлен на рисунке 8.

Задача 2.5. Реализовать класс «Комплексное число». Поля класса – действительная и мнимая части комплексного числа. Метод класса – инициализация полей объекта. Внешние функции – сложение двух комплексных чисел, вывод информации об объекте в консоль.

Листинг 9. Возвращение объекта функцией

```
1 #include <iostream>
2
3 using namespace std;
4 //описание класса MyComplex
5 class MyComplex{
6     public:
7     float im, re;
8     void my_init(float x, float y);
```

```
9  };
10 //прототипы внешних функций
11 MyComplex Summ(MyComplex obj1, MyComplex obj2);
12 void print_info(MyComplex obj);
13
14 int main()
15 {
16     setlocale(LC_CTYPE, "");
17     //создание объектов класса
18     MyComplex number1, number2, number3;
19     //вызов метода my_init
20     number1.my_init(1,3);
21     number2.my_init(4,6);
22     //вызов функции Summ
23     number3 = Summ(number1, number2);
24     //вызов функции print_info
25     cout<<"Первое комплексное число: ";
26     print_info(number1);
27     cout<<"Второе комплексное число: ";
28     print_info(number2);
29     cout<<"Сумма комплексных  чисел: ";
30     print_info(number3);
31     return 0;
32 }
33 //метод инициализации объекта класса MyComplex
34 void MyComplex::my_init(float x, float y){
35     re = x;
36     im = y;
37 }
38 //функция сложения двух объектов класса MyComplex
39 MyComplex Summ(MyComplex obj1, MyComplex obj2){
40     //создание локального объекта sum_obj
41     MyComplex sum_obj;
42     sum_obj.re = obj1.re + obj2.re;
43     sum_obj.im = obj1.im + obj2.im;
44     return sum_obj;
45 }
46 //функция вывода информации об объекте
47 void print_info(MyComplex obj){
48     cout<<obj.re<<" + "<<obj.im<<" im"<<endl;
49 }
```

Программный код из листинга 9 содержит класс `MyComplex`, в котором имеются два поля (действительная и мнимая части комплексного числа) и прототип метода (строка 8), описание данного метода произведено в строках 34–36. Строки 11 и 12 содержат прототипы двух внешних функций – функция вывода информации об объекте в консоль и функция сложения двух комплексных чисел. Описание функции сложения двух комплексных чисел находится в строках 39–45. В качестве параметров данная функция принимает два объекта `obj1` и `nobj2` класса `MyComplex`. Для того, чтобы функция возвращала объект внутри нее был создан локальный объект `sum_obj` класса `MyComplex` в 41 строке программного кода. В строках 42–43 реализовано сложение действительных и мнимых частей комплексных чисел. В результате работы функция `Summ` возвращает объект `sum_obj` в 44 строке программного кода.

Функция `main` содержит три объекта класса `MyComplex` (создание объектов произведено в строке 18). Для инициализации двух объектов `number1` и `number2` используется метод `my_init`, который вызывается в строках 20–21 данной программы. Объект `number3` содержит информацию о сложении объектов `number1` и `number2` (строка 23, вызывается функция `Summ`, которая в качестве параметров получает объекты `number1` и `number2`). В строках 26–30 вызывается функция `print_info` для всех трех объектов класса.

```
Первое комплексное число: 1 + 3 im
Второе комплексное число: 4 + 6 im
Сумма комплексных  чисел: 5 + 9 im
```

Рис. 8. Результат работы программы из листинга 9

2.4.2 Указатели и объекты

В языке программирования C++ можно создавать указатели на объекты, значением такого указателя будет являться адрес первой ячейки памяти, выделенной под объект. Синтаксис указателя на объект аналогичен синтаксису указателя на переменную обычного типа данных, т.е. сначала указывается тип данных (в данном случае имя класса), далее ставится звездочка, после чего указывается имя переменной-указателя.

Общий синтаксис объявления указателя на объект:
```
имя_класса * имя_указателя;
```

Пример объявления указателя на объект класса:
```
MyClass * p;
```

В данном примере `MyClass` – имя класса, которому принадлежит объект, на который будет ссылаться указатель, а `p` – имя самого указателя.

Синтаксис инициализации указателя на объект аналогичен синтаксису указателя на переменную базового типа:
```
имя_указателя = &имя_объекта;
```

Под `именем_указателя` записывается его идентификатор, под `именем_объекта` записывается идентификатор объекта класса, на который ссылается указатель. Перед именем ставится & – операция взятия адреса.

Пример инициализации указателя на объект:
```
MyClass obj, *p;
p = &obj;
```

В примере выше указателю `p` присваивается адрес первой ячейки памяти объекта `obj` класса `MyClass`.

Примечание: указатель и объект, на который он будет ссылаться должны быть одного типа данных.

С помощью указателя можно получить доступ не только к первой ячейке памяти объекта, но и к полям и методам объекта класса, для этого используется оператор `->` (стрелка).

Общий синтаксис доступа к полям и методам объекта класса выглядит следующим образом:

```
имя_указателя->имя_поля = значение;
```

Рассмотрим работу с указателями на примере программного кода из листинга 10.

Листинг 10. Пример использования указателя на объект

```
1 #include <iostream>
2 using namespace std;
3
4 //класс Point
5 class Point{
6     public:
7     float x, y;
8     void print_info();
9 };
10
11 int main(){
12     setlocale(LC_CTYPE, "");
13     //создание объекта и указателя
14     Point obj, *k;
15     //значение указателя - ссылка на объект класса
16     k = &obj;
17     //доступ к полям через указатель
18     k->x = 0;
19     k->y = 0;
20     //доступ к методу через указатель
21     k->print_info();
22     return 0;
23 }
24
25 //метод вывода информации об объекте
26 void Point::print_info(){
```

```
27      cout<<"Координаты точки: \n";
28      cout<<"X = "<<x<<"\t Y = "<<y<<endl;
29  }
```

Программный код листинга 10 содержит класс `Point` с открытыми полями `x` и `y` (координаты точки) и методом вывода информации об объекте в консоль `print_info()`. В главной функции `main()` создается объект класса `obj` и указатель на данный объект `k` (строка 13). Указателю задается значение в строке 14, инструкция `k = &obj`. В строках 17 и 18 с помощью указателя полям объекта задаются значения, равные нулю. В строке 19 с помощью указателя вызывается метод `print_info()`. Результат работы данного программного кода представлен на рисунке 9.

```
Координаты точки:
X = 0    Y = 0
```

Рис. 9. Результат работы программы из листинга 10

Кроме оператора `->` (стрелка) доступ к полям и методам объекта класса можно получить с помощью конструкции:

```
(*имя_указателя).имя_поля = значение;
```

В результате использования данной конструкции результат будет аналогичен использованию оператора `->` (стрелка).

Кроме указателей, созданных пользователем, для каждого объекта класса существует свой особый указатель – `this`. Данный указатель представляет собой неявно определенный указатель на сам объект [2]. Слово `this` является ключевым (зарезервированным) словом. Нельзя явно определить или описать данный указатель. Кроме того, указатель `this` является константным, т.е. его нельзя изменить. В каждом методе класса данный указатель ссылается на тот объект, для которого был вызван данный метод.

Рассмотрим работу с указателем `this` на примере решения задачи 2.6, которое представлено в листинге 11.

Задача 2.6. Написать программу, реализующую класс «Линия». Класс должен содержать:

1. Поля – координаты начала и конца отрезка.

2. Методы – метод задания значений полям объекта, метод вычисления длины отрезка, метод вывода информации об объекте в консоль (координаты начала и конца отрезка, длина отрезка).

Листинг 11. Пример использования указателя `this`

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
5 class Line{
6     float nx, ny, kx, ky;
7     public:
8     void set_point(float nx, float ny, float kx,float ky);
9     float length_line();
10    void print_info();
11 };
12
13 int main(){
14    setlocale(LC_CTYPE, "");
15    //создание объекта класса
16    Line obj;
17    //вызов методов класса
18    obj.set_point(0,0,1,1);
19    obj.print_info();
20    return 0;
21 }
22 //задание значений полям объекта класса
23 void Line::set_point(float nx, float ny, float kx, float
       ky){
24    this->nx = nx;
25    this->ny = ny;
26    this->kx = kx;
27    this->ky = ky;
28 }
29 //вычисление длины отрезка
30 float Line::length_line(){
31    float len_line;
```

```
32      len_line = sqrt(pow(kx-nx,2)+pow(ky-ny,2));
33      return len_line;
34  }
35  //вывод информации об отрезке
36  void Line::print_info(){
37      cout<<"Координаты начала отрезка: \n";
38      cout<<"x = "<<nx<<"\t y = "<<ny<<endl;
39      cout<<"Координаты конца отрезка: \n";
40      cout<<"x = "<<kx<<"\t y = "<<ky<<endl;
41      cout<<"Длина отрезка: ";
42      cout<<length_line();
43  }
```

Класс `Line` содержит четыре закрытых поля `nx, ny, kx, ky` и три открытых метода `set_point(), length_line()` и `print_info()`. Остановимся подробнее на каждом из методов. Метод `set_point()` содержит четыре аргумента (координаты начала и конца отрезка), он необходим для задания значений полям объекта класса. В теле метода присутствует указатель `this`, он помогает компилятору различать поля класса и одноименные аргументы метода, в следствие чего не возникает ошибки неоднозначности. Выражение в строке 24 `this->nx = nx` буквально читается как «Полю объекта `nx` (инструкция this->nx) присвоить значение аргумента `nx`(правая часть от оператора присваивания)».

Метод `length_line()` необходим для вычисления длины отрезка по заданным координатам, в результате работы метода возвращается значение его длины (`len_line`).

Уже знакомый нам метод `print_info()` выводит в консоль информацию об объекте класса. В самом методе вызывается метод `length_line()`, т.е. в теле главной функции `main()`при вызове метода `print_info()` сразу будут выведены и координаты начала и конца отрезка, и его длина. Таким образом нет необходимости в отдельном вызове метода `length_line()`.

В теле главной функции `main()` создается один объект `obj` класса `Line` (строка 16). Для данного объекта вызываются два метода: `set_point()`(задание значения полям данного объекта)

и `print_info()`(вывод информации в консоль) (строки 18–19).

Результат работы программного кода, разобранного выше, представлен на рисунке 10.

```
Координаты начала отрезка:
x = 0    y = 0
Координаты конца отрезка:
x = 1    y = 1
Длина отрезка: 1.41421
```

Рис. 10. Результат работы программы из листинга 11

Любой метод, описанный в классе, может в качестве возвращаемого значения использовать конструкцию `return *this`, т.к. `this` – это указатель на объект, тогда `*this` – это сам объект класса, т.е. метод будет возвращать в качестве результата сам объект класса, из которого он вызван.

2.5 Конструкторы

В языке программирования C++ существуют специальные методы класса, позволяющие при создании объекта автоматически задавать значения его полям. Такие методы получили название конструктор. **Конструкторы** – это специальные методы (функции) класса, которые автоматически вызываются при создании объекта класса [8]. Конструкторы в классах используются, как правило, для автоматической инициализации полей объектов класса. Несмотря на то, что конструктор – это метод класса, он не возвращает никакое значение, даже `void`, и, кроме того, на него нельзя получить указатель.

Правильное написание и использование конструкторов значительно повышает гибкость программного кода.

2.5.1 Конструктор по умолчанию

При создании класса автоматически создается неявно заданный конструктор по умолчанию (default constructor). Если в классе явно не описано ни одного конструктора, этот конструктор будет вызываться при каждом создании объекта класса. Рассмотрим работу неявно заданного конструктора по умолчанию на примере решения задачи 2.7. Решение данной задачи представлено в листинге 12.

Задача 2.7. Написать программу, реализующую класс «Точка». Класс должен содержать:

1. Поля класса – координаты точки x и y.

2. Методы класса – вывод информации о координатах точки в консоль.

Листинг 12. Решение задачи 2.7

```
#include <iostream>
using namespace std;
//описание класса Точка
class Point{
    //закрытые поля класса
    int x, y;
    //открытые методы класса
    public:
    //прототип метода вывода информации
    void print_info();
};
int main(){
    //локализация
    setlocale(LC_CTYPE, "");
    //создание объекта класса
    Point obj;
    //вызов метода print_info
    obj.print_info();
    return 0;
}
//описание метода вывода информации в консоль
void Point::print_info(){
    cout<<"x = "<<x<<"\t y =  "<<y<<endl;
};
```

Из результата работы программы (представлен на рисунке 11) видно, что поля объекта `obj` были инициализированы, несмотря на то, что явно не был написан метод для задания значений полям объекта. В данном случае значения полей объекта `obj` содержат «мусор».

```
x = -8256        y =  32767
```

Рис. 11. Результат работы программы из листинга 12

Кроме неявно заданного конструктора, в языке C++ возможно использование явно заданных конструкторов. При наличии хотя бы одного явно заданного конструктора неявно заданный конструктор создаваться не будет. Рассмотрим создание явно заданного **конструктора по умолчанию**.

Создание конструктора похоже на создание обычного метода за некоторым исключением: имя конструктора полностью совпадает с именем класса и при объявлении конструктора не указывается тип возвращаемого значения.

Общий синтаксис создания конструктора по умолчанию:

```
имя_класса(){
 //тело конструктора
}
```

Рассмотрим реализацию конструктора по умолчанию на следующем примере (задача 2.8).

Задача 2.8. Написать программу, реализующую класс «Точка». Класс должен содержать:

1. Поля – координаты точки x и y.
2. Конструктор по умолчанию.
3. Метод вывода информации о координатах точки в консольное окно.

С решением задачи 2.8 можно ознакомиться в листинге 13.

Листинг 13. Пример конструктора по умолчанию

```
1 #include <iostream>
2 using namespace std;
3 //описание класса Точка
4 class Point{
5     //закрытые поля класса
6     int x, y;
7     //открытые методы класса
8     public:
9     //прототип конструктора по умолчанию
10    Point();
11    //прототип метода вывода информации
12    void print_info();
13 };
14 int main(){
15    setlocale(LC_CTYPE, "");
16    //создание объекта класса
17    Point obj;
18    //вызов метода print_info
19    obj.print_info();
20    return 0;
21 }
22 //описание конструктора по умолчанию
23 Point::Point(){
24    x = 0;
25    y = 0;
26 };
27 //описание метода вывода информации в консоль
28 void Point::print_info(){
29    cout<<"x = "<<x<<"\t y =  "<<y<<endl;
30 };
31
```

В условии задачи было сказано, что класс должен содержать конструктор по умолчанию, он реализован в строках 23–26, а в описании класса содержится его прототип. По умолчанию поля объектов класса получают значения равные нулю. В строке 17 создается объект `obj` класса `Point`. В строке 19 для объекта `obj` вызывается метод `print_info`, выводящий информацию об

объекте в консольное окно. С результатами работы программы из листинга 13 можно ознакомиться на рисунке 12.

```
x = 0    y =  0
```

Рис. 12. Результат работы программы из листинга 13

2.5.2 Конструктор с параметрами

Как и в случае с методами, конструкторы можно перегружать, т.е. изменять тип и/или количество аргументов. В одном классе можно создавать любое количество конструкторов. В примере, представленном выше, при создании объекта вызывался конструктор по умолчанию, и сколько бы объектов класса не было создано, поля каждого объекта имели бы значения, полученные с помощью конструктора по умолчанию (т.е. равные нулю). Однако данный подход не всегда является практичным.

Рассмотрим следующий вид конструктора – конструктор с параметрами. В отличие от конструктора по умолчанию он содержит аргументы. **Конструктор с параметрами** используется для явного задания значений полям объекта. Таким образом можно легко и быстро инициализировать поля объекта класса какими-либо значениями. С реализацией конструктора с параметрами на примере решения задачи 2.9 можно ознакомиться в листинге 14.

Общий синтаксис конструктора с параметрами:

```
имя_класса(аргументы с их типами){
 //тело конструктора
}
```

Задача 2.9. Написать программу, реализующую класс «Точка». Класс должен содержать:

1. Поля – координаты точки x и y.

2. Конструктор по умолчанию, конструктор с параметрами.

3. Метод вывода информации о координатах точки x и y на экран.

Листинг 14. Пример конструктора с параметрами

```
#include <iostream>
using namespace std;
//описание класса Точка
class Point{
    int x, y;
    public:
    //прототип конструктора по умолчанию
    Point();
    //прототип конструктора с параметрами
    Point(int a, int b);
    //прототип метода вывода информации
    void print_info();
};
int main(){
    setlocale(LC_CTYPE, "");
    //создание объектов класса
    Point obj_1;
    Point obj_2(2, 3);
    //вызов метода print_info
    cout<<"Координаты первой точки: "<<endl;
    obj_1.print_info();
    cout<<"Координаты второй точки: "<<endl;
    obj_2.print_info();
    return 0;
}
//описание конструктора по умолчанию
Point::Point(){
    x = 0;
    y = 0;
};
//описание конструктора с параметрами
Point::Point(int a, int b){
    x = a;
    y = b;
};
```

```
36 //описание метода вывода информации в консоль
37 void Point::print_info(){
38     cout<<"x = "<<x<<"\t y =  "<<y<<endl;
39 };
```

Рассмотрим программный код из листинга 14. В нем реализовано два конструктора: конструктор по умолчанию (описание в строках 27–30, в теле класса содержится прототип данного конструктора) и конструктор с параметрами (описание в строках 32–35, в теле класса содержится прототип данного конструктора). В строках 17 и 28 создается два объекта класса. Для объекта `obj_1` вызывается конструктор по умолчанию, его поля принимают значения, равные нулю. Для объекта `obj_2` вызывается конструктор с параметрами, его поля принимают значения, равные двум и трем соответственно (значения, указанные в качестве аргументов). В качестве аргументов возможно указывать не только числа, но и переменные, значение которых пользователь может вводить в консоли. С результатом работы программы можно ознакомиться на рисунке 13.

```
Координаты первой точки:
x = 0    y =  0
Координаты второй точки:
x = 2    y =  3
```

Рис. 13. Результат работы программы из листинга 14

2.5.3 Конструктор копирования

Рассмотрим следующий вид конструктора – конструктор копирования. **Конструктор копирования** (copy constructor) предназначен для создания копии объекта с помощью другого объекта того же класса [2]. В качестве аргумента данный конструктор получает ссылку на объект, который необходимо копировать.

Общий синтаксис описания конструктора копирования:

```
имя_класса (const имя_класса &имя_объекта){
 //тело конструктора
}
```

Конструктор копирования, как правило, используется в следующих случаях:

1. При создании нового объекта, который должен стать копией уже имеющегося;

2. При передаче объекта функции по значению;

3. При возврате объекта из функции [7].

Помимо этого, используется *операция присваивания* в случаях, когда один объект необходимо скопировать в другой, при условии, что оба были созданы ранее.

Если в программе явно не задано ни одного конструктора копирования, компилятор сгенерирует его автоматически, и именно этот конструктор будет использоваться в программе. Такой конструктор называется конструктором копирования по умолчанию. Данный конструктор копирования реализует побитовое копирование для получения копии объекта.

Целесообразно использовать конструктор копирования в классах, где осуществляется динамическое выделение памяти, в противном случае могут возникать ошибки, связанные с использованием указателей.

Рассмотрим программный код из листинга 15 который демонстрирует использование конструктора копирования по умолчанию без выделения динамической памяти.

Листинг 15. Конструктор копирования по умолчанию и операция присваивания

```
1 #include <iostream>
2 using namespace std;
3
4 class Point{
5     int x, y;
6     public:
```

```
 7      Point(){x = 0; y = 0;}
 8      Point(int x, int y);
 9      void print_info();
10 };
11
12 int main(){
13      setlocale(LC_CTYPE, "");
14      Point obj1(7,1);
15      cout<<"Точка 1: "; obj1.print_info();
16      Point obj2(obj1);
17      cout<<"Точка 2: "; obj2.print_info();
18      Point obj3 = obj2;
19      cout<<"Точка 3: "; obj3.print_info();
20      Point obj4;
21      cout<<"Точка 4: "; obj4.print_info();
22      obj1 = obj4;
23      cout<<"Точка 1: "; obj1.print_info();
24      cout<<endl;
25      return 0;
26 }
27 //конструктор с параметрами
28 Point::Point(int x, int y){
29      this->x = x;
30      this->y = y;
31 }
32 //метод вывода информации об объекте в консоль
33 void Point::print_info(){
34      cout<<"X = "<<x<<" Y = "<<y<<endl;
35 }
```

В классе `Point` содержится два поля (координаты точки), два конструктора (конструктор по умолчанию и конструктор с параметрами) и один метод (вывод информации о точке в консольное окно).

В главной функции программы создается четыре объекта данного класса. Для первого объекта вызывается конструктор с параметрами (строка 14), его поля получают значения `obj.x=7, obj.y=1`. Для второго объекта `obj2` вызывается конструктор копирования по умолчанию, данный объект инициализируется значением объекта `obj1`. Для объекта `obj3` также вызывается кон-

структор копирования, он инициализируется значением объекта `obj2` (строка 18).

Для объекта `obj4` вызывается конструктор по умолчанию, его поля получают значения, равные нулю. В строке 22 в объект `obj1` копируется `obj4`, в данном случае используется операция присваивания (*НЕ конструктор копирования!*), так как оба объекта уже были созданы ранее. Операция присваивания предопределена для абстрактных типов данных по умолчанию, т.е. ее можно использовать без каких-либо дополнительных действий.

Результат работы программного кода из листинга 15 представлен на рисунке 14.

```
Точка 1: X = 7 Y = 1
Точка 2: X = 7 Y = 1
Точка 3: X = 7 Y = 1
Точка 4: X = 0 Y = 0
Точка 1: X = 0 Y = 0
```

Рис. 14. Результат работы программы из листинга 15

Рассмотрим следующий пример, в котором демонстрируется использование конструктора копирования, описанного в явном виде (листинг 16).

Листинг 16. Определение конструктора копирования в явном виде без выделения динамической памяти

```
1  #include <iostream>
2  using namespace std;
3
4  class Point{
5      int x, y;
6      public:
7      Point();
8      Point(int x, int y);
9      //прототип конструктора копирования
10     Point(const Point &obj);
11     void print_info();
```

```
12 };
13
14 int main(){
15     setlocale(LC_CTYPE,"");
16     Point p1;
17     cout<<"p1: "; p1.print_info();
18     Point p2(1, 1);
19     cout<<"p2: "; p2.print_info();
20     Point p3(p1);
21     cout<<"p3: "; p3.print_info();
22     Point p4 = p1;
23     cout<<"p4: "; p4.print_info();
24     return 0;
25 }
26 //конструктор по умолчанию
27 Point::Point(){
28     x = 0;
29     y = 0;
30     cout<<"Конструктор по умолчанию "<<this<<endl;
31 }
32 //конструктор с параметрами
33 Point::Point(int x, int y){
34     this->x = x;
35     this->y = y;
36     cout<<"Конструктор с параметрами "<<this<<endl;
37 }
38 //конструктор копирования
39 Point::Point(const Point &obj){
40     this->x = obj.x;
41     this->y = obj.y;
42     cout<<"Конструктор копирования "<<this<<endl;
43 }
44
45 void Point::print_info(){
46     cout<<"X = "<<x<<" Y = "<<y<<endl;
47 }
```

В примере выше класс `Point` содержит два целочисленных поля (координаты точки), один метод (вывод информации об объекте в консоль) и три конструктора (конструктор по умолчанию, конструктор с параметрами и конструктор копирования).

Конструктор копирования в данной программе явно определен в строках 39–43, в качестве аргументов он принимает некоторый объект класса `Point`.

В функции `main` создается четыре объекта класса `Point`. Для первого объекта `p1` вызывается конструктор по умолчанию и его поля получают значения, равные нулю. Для второго объекта `p2` вызывается конструктор с параметрами и его поля получают значения `p2.x=1, p2.y=1`. Объекты `p3` и `p4` инициализируются значением объекта `p1`. Для более детального анализа программного кода в теле каждого конструктора содержится строка, выводящая информацию о том, какой конструктор вызван для объекта, а также ячейка памяти данного объекта.

С результатом работы данной программы можно ознакомиться на рисунке 15.

```
Конструктор по умолчанию 0x7fffffffdea8
p1: X = 0 Y = 0
Конструктор с параметрами 0x7fffffffdeb0
p2: X = 1 Y = 1
Конструктор копирования 0x7fffffffdeb8
p3: X = 0 Y = 0
Конструктор копирования 0x7fffffffdec0
p4: X = 0 Y = 0
```

Рис. 15. Результат работы программы из листинга 16

Данный пример является лишь ознакомительным, так как программа не осуществляет выделение динамической памяти, и, следовательно, не требует явного определения копирующего конструктора.

Рассмотрим следующий пример – листинг 17, в котором иллюстрируется использование копирующего конструктора с выделением динамической памяти.

Листинг 17. Определение конструктора с выделением динамической памяти

```
1 #include <iostream>
2 using namespace std;
3
4 class Number{
5     private:
6     //поля класса
7     int x, *y;
8     public:
9     //прототипы методов
10    Number();
11    Number(int x1, int y1);
12    Number(const Number &obj);
13    void printinfo();
14    ~Number();
15 };
16
17 int main(){
18    setlocale(LC_CTYPE, "");
19    cout<<"Объект o1: "; Number o1(2,2);
20    o1.printinfo();
21    cout<<"Объект o2: "; Number o2 = o1;
22    o2.printinfo();
23    cout<<"Объект o3: "; Number o3;
24    o3.printinfo();
25    cout<<"Объект o4: "; Number o4(o1);
26    o4.printinfo();
27    return 0;
28 }
29 //конструктор по умолчанию
30 Number::Number() {
31    y = new int;
32    x = 0;
33    *y = 0;
34    cout<<"Конструктор по умолчанию "<<this<<endl;
35 }
36 //конструктор с параметрами
37 Number::Number(int x1, int y1){
38    y = new int;
39    x = x1;
40    *y = y1;
41    cout<<"Конструктор с параметрами "<<this<<endl;
42 }
```

```
43 //конструктор копирования
44 Number::Number(const Number &obj){
45     y = new int;
46     x = obj.x;
47     *y = *(obj.y);
48     cout<<"Конструктор копирования "<<this<<endl;
49 }
50 //деструктор
51 Number::~Number(){
52     delete y;
53     cout<<"Деструктор "<<this<<endl;
54 }
55 //метод вывода информации об объекте в консоль
56 void Number::printinfo(){
57     cout<<"Адрес Y: "<<y<<endl;
58     cout<<"X = "<<x;
59     cout<<" Y = "<<*y<<endl<<endl;
60
61 }
```

Класс `Number` из программного кода, приведенного выше, содержит два поля (одно из которых является указателем), один метод (вывод информации в консоль) и три конструктора (по умолчанию, с параметрами, копирования). Так как одно из полей класса является указателем, необходимо для каждого конструктора предусмотреть выделение динамической памяти во избежание различного рода ошибок в работе программы.

Обратим внимание на определение конструкторов по умолчанию и с параметрами (строки 30–42). В отличие от предыдущих примеров в этих конструкторах осуществляется выделение динамической памяти, после чего поля получают свои значения. Для того, чтобы в работе программы не возникало ошибок, в программе явно определен конструктор копирования (строки 44–49). В нем также предусматривается выделение динамической памяти, после чего в новые поля копируются значения *(НЕ ссылки)* из объекта, копию которого необходимо получить. Данный процесс создания копии объекта класса называется *глубоким копированием.*

Для детального анализа программного кода в нем предусмотрен вывод номеров ячеек памяти объектов, а также поля `Y` для каждого объекта. На рисунке 16 можно заметить, что адреса для полей объектов являются различными, т.е. создается четыре различных объекта, которые не связаны друг с другом, в следствие чего программный код будет работать корректно при изменении или удалении объектов (т.е. изменение одного объекта не приведет к изменению других). Также программа из листинга 17 содержит деструктор (строки 51–54) о котором пойдет речь в следующем параграфе.

```
Объект o1: Конструктор с параметрами 0x7fffffffde70
Адрес Y: 0x55555556c350
X = 2 Y = 2

Объект o2: Конструктор копирования 0x7fffffffde80
Адрес Y: 0x55555556c370
X = 2 Y = 2

Объект o3: Конструктор по умолчанию 0x7fffffffde90
Адрес Y: 0x55555556c390
X = 0 Y = 0

Объект o4: Конструктор копирования 0x7fffffffdea0
Адрес Y: 0x55555556c3b0
X = 2 Y = 2

Деструктор 0x7fffffffdea0
Деструктор 0x7fffffffde90
Деструктор 0x7fffffffde80
Деструктор 0x7fffffffde70
```

Рис. 16. Результат работы программы из листинга 17

2.6 Деструкторы

Деструктор – это специальный метод класса, используемый для освобождения памяти, занимаемой объектом. Деструктор всегда вызывается автоматически при выходе объекта из об-

ласти видимости или при использовании операции `delete` (для объектов, заданных через указатели). Как и в случае с конструкторами, имя деструктора совпадает с именем класса, которому предшествует символ ~ (тильда). Деструктор не имеет типа возвращаемого значения, даже `void`, не имеет параметров и его нельзя перегрузить.

Общий синтаксис определения деструктора:

```
~имя_класса() {
 //тело деструктора
}
```

Деструктор всегда должен иметь открытый спецификатор доступа. Если деструктор явно неопределен в классе, его автоматически создаст компилятор. Деструктор обязательно описывается явно, если для объекта выделяется динамическая память, в противном случае при удалении объекта память, занимаемая им не будет помечена как свободная.

Рассмотрим работу деструктора на примере решения задачи 2.10, ее решение представлено в листинге 18.

Задача 2.10. Написать программу, реализующую класс книга. Класс должен содержать:

1. Поля – название книги, автор, количество страниц.
2. Конструкторы – по умолчанию, с параметрами.
3. Деструктор.
4. Метод вывода информации об объекте в консоль.

Листинг 18. Использование деструктора

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Book{
6     string name, autor;
7     int page;
8     public:
```

```
9       Book();
10      Book(string name, string autor, int page);
11      ~Book(); //прототип деструктора
12      void print_book();
13  };
14
15  int main(){
16      setlocale(LC_CTYPE, "");
17      Book mybook_1;
18      mybook_1.print_book();
19      Book mybook_2("Хищные вещи века", "Стругацкие", 225);
20      mybook_2.print_book();
21      return 0;
22  }
23  //конструктор по умолчанию
24  Book::Book(){
25      name = "";
26      autor = "";
27      page = 0;
28      cout<<"Конструктор по умолчанию "<<this<<endl;
29  }
30  //конструктор с параметрами
31  Book::Book(string name, string autor, int page){
32      this->name = name;
33      this->autor = autor;
34      this->page = page;
35      cout<<"Конструктор с параметрами "<<this<<endl;
36  }
37  //деструктор
38  Book::~Book(){
39      cout<<"Вызов деструктора "<<this<<endl;
40  }
41  //метод вывода сведений об объекте в консоль
42  void Book::print_book(){
43      cout<<"Название книги: "<<name<<endl;
44      cout<<"Автор книги: "<<autor<<endl;
45      cout<<"Количество страниц: "<<page<<endl;
46  }
```

В программе из листинга 18 определен деструктор в явном виде в строках 38–40. Несмотря на то, что деструктор определен явно, он содержит лишь одну строку – вывод сообщения в кон-

соль о том, что он вызван, а также адрес объекта, для которого он вызван. Это связано с тем, что в программе не осуществляется выделение динамической памяти и, следовательно, тело деструктора может быть пустым.

Рассмотрим рисунок 17 на котором представлен результат работы программы, представленной выше. Для первого объекта вызывается конструктор по умолчанию, после чего выводится информация об объекте, для второго объекта вызывается конструктор с параметрами, после чего также выводится информация об объекте. После того, как объекты `mybook_1` и `mybook_2` выходят из области видимости (завершение работы функции `main`) вызывается деструктор для объекта `mybook_2`, а затем для объекта `mybook_1`. Обратим внимание, вызов деструктора производится в обратном порядке создания объектов.

```
Конструктор по умолчанию 0x7fffffffde20
Название книги:
Автор книги:
Кол-во страниц: 0
Констуктор с параметрами 0x7fffffffde70
Название книги: Хищные вещи века
Автор книги: Стругацкие
Кол-во страниц: 225
Вызов деструктора 0x7fffffffde70
Вызов деструктора 0x7fffffffde20
```

Рис. 17. Результат работы программы из листинга 18

Если в программе реализовано выделение динамической памяти необходимо явно определять деструктор. В таком случае в теле деструктора используется операция `delete` для очищения памяти, выделенной под объект. Такой деструктор реализован в программном коде листинга 17 (строки 51–54).

2.7 Статические члены класса

До этого момента мы работали с полями и методами класса, имеющими свои собственные значения для каждого его объекта. Однако в классах с помощью ключевого слова `static` можно описать статические поля и методы. Статические поля и методы класса являются общими для всех объектов этого класса, т.е. данные, которые они хранят будут общими для всех объектов.

Особенности статических полей:

1. Существуют в единственном экземпляре.

2. Доступны как через имя класса, так и через имя объекта.

3. На статические поля распространяется действие спецификаторов доступа.

4. Статическое поле не исчезает при удалении объекта.

5. Статическое полу существует даже в том случае, если не создано ни одного объекта класса.

Общий синтаксис определения статического поля:

```
static тип_переменной имя_переменной;
```

Статические методы класса предназначены для работы со статическими полями. Такие методы могут обращаться *только* к статическим полям, либо вызывать другие статические методы. Обращение к статическим методам производится аналогично к обращению обычного метода, т.е. через имя класса или имя объекта класса.

Общий синтаксис определения статического метода:

```
static тип_результата имя_метода (аргументы){
 //тело метода;
}
```

Рассмотрим случай использования статического поля и статического метода, приведенного в программном коде листинга 19.

Листинг 19. Пример статического поля и статического метода

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

class Test{
    //статическое поле
    static int k;
    //нестатическое поле
    int m = 0;
    public:
    //прототип статического метода
    static void k_count();
    //прототип нестатического метода
    void print_k();
};
int Test::k;
int main(){
    setlocale(LC_CTYPE, "");
    //создание объектов класса
    Test obj1, obj2;
    cout<<"Объект obj1: ";
    //вызов статического метода для obj1
    obj1.k_count();
    obj1.print_k();
   //вызов статического метода для obj2
    obj2.k_count();
    cout<<"Объект obj1: ";
    obj1.print_k();
    cout<<"Объект obj2: ";
    obj2.print_k();
    return 0;
}
//определение статического метода k_count
void Test::k_count(){
    k++;
}
//определение метода print_k
void Test::print_k(){
    cout<<" k = "<<k<<" m = "<<m<<endl;
}
```

В примере, приведенном выше, класс `Test` содержит поми-

мо нестатических полей и методов одно статическое поле `k` (строка 6) и один статический метод `k_count()` (стока 13). Статическое поле класса является закрытым, доступ к нему производится с помощью методов. Для того, чтобы выделить место под статическое поле вне класса (строка 15) выполняется его объявление: `int Test::k`. В главной функции `main()` создается два объекта `obj1, obj2` класса `Test`. Для объекта `obj1` в 22 строке вызывается статический метод `k_count` (значение статического поля `k` увеличивается на единицу), после чего вызывается метод `print_k()`, выводящий информацию об объекте в консоль. На этом этапе значения полей равны `k=1, m=0` (рисунок 18). Далее снова вызывается статический метод, но уже для объекта `obj2`. Далее два раза вызывается метод `print_k()` для вывода информации о каждом объекте в консоль. На рисунке 18 видно, что после вызова статического метода для второго объекта изменилось значение статического поля `k` для обоих объектов. Можно убедится, что при изменении значения статического поля для одного объекта, оно будет изменено для всех объектов данного класса.

```
Объект obj1:  k = 1 m = 0
Объект obj1:  k = 2 m = 0
Объект obj2:  k = 2 m = 0
```

Рис. 18. Результат работы программы из листинга 19

Несмотря на то, что статические поля используются достаточно редко, в некоторых случаях их использование очень удобно, например, в случаях, когда каждый объект должен хранить информацию о количестве всех объектов данного класса.

2.8 Друзья класса

К закрытым полям и методам класса нельзя получить доступ вне этого класса, но возникают случаи, когда это необходимо сделать. Данное правило не распространяется на дружественные

функции. Функции, объявленные для класса как дружественные, имеют доступ ко всем крытым полям и методом класса, для которого они задекларированы как дружественные.

Для объявления дружественных функций используется ключевое слово `friend`, которое указывается перед прототипом функции в описании класса. В качестве дружественной функции может выступать любая внешняя функция и метод класса, кроме того, одна и та же функция может быть дружественной по отношению к нескольким классам. В качестве параметра таким функциям передается объект класса или ссылка на объект данного класса.

Примечание: на дружественные функции не распространяется действие спецификатора доступа.

Пример использования дружественной функции приведен в программном коде листинга 20.

Листинг 20. Пример использования дружественной функции

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

class ComplexNumb{
    int re, im;
    public:
    ComplexNumb(int re, int im);
    //дружественная функция
    friend void print_complNumb(ComplexNumb obj);
};
//описание дружественной функции
void print_complNumb(ComplexNumb obj){
    cout<<obj.re<<" + "<<obj.im<<"i"<<endl;
}

int main(){
    setlocale(LC_CTYPE, "");
    ComplexNumb number(1, 2);
    cout<<"Комплексное число: ";
    //вызов дружественной функции
    print_complNumb(number);
    return 0;
```

```
23 }
24 //конструктор с параметрами
25 ComplexNumb::ComplexNumb(int re, int im){
26     this->re = re;
27     this->im = im;
28 }
```

Класс `ComplexNumb` имеет два закрытых поля, для заполнения данных полей предусмотрен конструктор с параметрами. Вывод информации об объекте в консоль осуществляется с помощью внешней функции `print_complNumb()` которая объявлена дружественной для данного класса (строка 9), в качестве параметра она принимает объект класса `ComplexNumb`. Вызов функции `print_complNumb()` производится в строке 21, аргументом она принимает объект `number`.

Результат работы программы представлен на рисунке 19.

```
Комплексное число: 1 + 2i
```

Рис. 19. Результат работы программы из листинга 20

2.9 Лабораторная работы №1 «Введение в ООП»

Цель работы: знакомство с ООП, изучение общих понятий о классах и приобретение навыков их реализации.

Вариант 1

Написать программу, реализующую класс «Товар». Класс должен содержать:

1. Поля: название, категория, цена, количество.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления стоимости всех единиц товара.

Вариант 2

Написать программу, реализующую класс «Автомобили». Класс должен содержать:

1. Поля: id, марка, модель, цена, количество.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления автомобиля с наименьшей ценой.

Вариант 3

Написать программу, реализующую класс «Эллипс». Класс должен содержать:

1. Поля: цвет, большая ось эллипса, малая ось эллипса.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления площади эллипса.

Вариант 4

Написать программу, реализующую класс «Круг». Класс должен содержать:

1. Поля: имя, цвет и радиус.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Методы вычисления площади и длины круга.

Вариант 5

Написать программу, реализующую класс «Комплексное число». Класс должен содержать:

1. Поля: мнимая и действительная части компл-го числа.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод сложения комплексных чисел.

Вариант 6

Написать программу, реализующую класс «Обыкновенная дробь». Класс должен содержать:

1. Поля: числитель и знаменатель дроби.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод сложения двух дробей.

Вариант 7

Написать программу, реализующую класс «Обыкновенная дробь». Класс должен содержать:

1. Поля: числитель и знаменатель дроби.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод деления двух дробей.

Вариант 8

Написать программу, реализующую класс «Трапеция». Класс должен содержать:

1. Поля: a, b, c, d – основания трапеции, h – высота трапеции.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления периметра и площади трапеции.

Вариант 9

Написать программу, реализующую класс «Параллелограмм». Класс должен содержать:

1. Поля: длины сторон и высота.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления площади и периметра объектов класса «Параллелограмм».

Вариант 10

Написать программу, реализующую класс «Ромб». Класс должен содержать:

1. Поля: длины стороны и высоты ромба.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления площади и периметра ромба.

Вариант 11

Написать программу, реализующую класс «Шар». Класс должен содержать:

1. Поля: имя и радиус шара.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления площади и периметра шара.

Вариант 12

Написать программу, реализующую класс «Круг». Класс должен содержать:

1. Поля: радиус круга и длина дуги сектора.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления площади сектора круга.

Вариант 13

Написать программу, реализующую класс «Конус». Класс должен содержать:

1. Поля: радиус основания и высота конуса.
2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления площади и объема конуса.

Вариант 14

Написать программу, реализующую класс «Правильная призма». Класс должен содержать:

1. Поля: высота, длина и количество сторон призмы.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления объема правильной призмы.

Вариант 15

Написать программу, реализующую класс «Правильный цилиндр». Класс должен содержать:

1. Поля: радиус основания и высота.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления площади и объема цилиндра.

Вариант 16

Написать программу, реализующую класс «Вектор». Класс должен содержать:

1. Поля: координаты x, y и z.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления суммы и произведения векторов.

Вариант 17

Написать программу, реализующую класс «Прямая». Класс должен содержать:

1. Поля: координаты начала и конца отрезка.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления длины отрезка.

Вариант 18

Написать программу, реализующую класс «Врач». Класс должен содержать:

1. Поля: ФИО, должность, стаж, оклад.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления надбавки за стаж (если стаж работы более 15 лет, надбавка составляет 10% от оклада).

Вариант 19

Написать программу, реализующую класс «Цифровая техника». Класс должен содержать:

1. Поля: id, наименования, категория, цена, количество.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления категории с наибольшим количеством товаров.

Вариант 20

Написать программу, реализующую класс «Книга». Класс должен содержать:

1. Поля: id, автор, название, цена, количество.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления цены книги со скидкой в 10%.

Вариант 21

Написать программу, реализующую класс «Студент ВУЗа». Класс должен содержать:

1. Поля: ФИО, курс, специальность, ср. балл, стипендия.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления средней стипендии на курсе.

Вариант 22

Написать программу, реализующую класс «Зоомагазин». Класс должен содержать:

1. Поля: id, наименование, категория, цена, количество.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления надбавочной стоимости (18% от стоимости товара) и стоимости товара без него.

Вариант 23

Написать программу, реализующую класс «Дата». Класс должен содержать:

1. Поля: число, месяц и год.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).
4. Метод вычисления времени года.

Вариант 24

Написать программу, реализующую класс «Треугольная пирамида» (правильная). Класс должен содержать:

1. Поля: длина стороны основания и высота.
2. Конструкторы: по умолчанию, с параметрами.
3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления объема пирамиды.

Вариант 25

Написать программу, реализующую класс «Четырехугольная пирамида» (правильная). Класс должен содержать:

1. Поля: длина стороны основания и высота.

2. Конструкторы: по умолчанию, с параметрами.

3. Деструктор (с явным выводом сообщения о том, что он был вызван).

4. Метод вычисления объема пирамиды.

3 Перегрузка операций

Язык программирования C++ позволяет перегружать не только пользовательские функции, но и стандартные операции, такие как сложение, вычитание, умножение и т.д. Перегрузка операторов – это возможность языка программирования применять встроенные операторы языка к различным типам данных, в том числе и абстрактным.

Под перегрузкой операторов подразумевается создание новой функции, в названии которой содержится ключевое слово `operator`, после которого следует символ перегружаемого оператора. Возможно переопределить практически все операторы, как унарные, так и бинарные.

Непереопределяемые операторы: .(выбор члена), .*(выбор указателя на член), :: (разрешение области), ?: (тернарный оператор), # (препроцессор), ## (препроцессор), `sizeof`.

3.1 Перегрузка унарных операций

Унарные операции – это операции, производящиеся над одним операндом.

Общий синтаксис перегрузки унарных операций:

```
тип_результата operator знак_оператора(аргумент){
  //программный код
}
```

Рассмотрим перегрузку унарных операций на примере инкремента. Инкремент, как и декремент, имеет две формы: префиксную и постфиксную. Особенность перегрузки данных операций состоит в том, что необходимо перегружать обе формы.

Общий синтаксис для переопределения префиксной формы:

```
тип_результата operator ++ (тип_аргумента аргумент){
```

```
  //программный код
}
```

Общий синтаксис переопределения постфиксной формы:

```
тип_результата operator ++ (тип_аргумента аргумент,
int аргумент){
  //программный код
}
```

При переопределении постфиксной формы функция содержит два аргумента. Второй аргумент (целочисленный) является формальным и при расчетах не учитывается. Он необходим для того, чтобы компилятор различал переопределение префиксной и постфиксной форм оператора.

Рассмотрим перегрузку унарных операций на примере решения задачи 3.1 (листинг 21).

Задача 3.1 Реализовать класс «Комплексное число». Класс должен содержать:

1. Поля – мнимая и действительная части комплексного числа.
2. Метод вывода информации об объекте в консоль.
3. Конструктор по умолчанию, конструктор с параметрами.
4. Деструктор.

Для данного класса перегрузить операции инкремента и сложения объектов.

Листинг 21. Программа с переопределением унарных и бинарных операций

```
1 #include <iostream>
2 using namespace std;
3
4 class NComplex{
5     int Re, Im;
6     public:
7     NComplex();
8     NComplex(int Re, int Im);
9     ~NComplex(){};
```

```
10      //прототипы перегруженных операций
11      friend NComplex operator++(NComplex &obj, int n);
12      friend NComplex operator++(NComplex &obj);
13      friend NComplex operator+ ( NComplex obj1, NComplex
        obj2);
14      friend void print(NComplex obj);
15 };
16
17 int main(){
18      setlocale(LC_CTYPE, "");
19      NComplex n, k(1,2), c;
20      cout<<"Комплексное число n: "; print(n);
21      cout<<"Комплексное число k: "; print(k);
22      cout<<"n++: "; n++; print(n);
23      cout<<"++n: "; ++n; print(n);
24      c = n+k;
25      cout<<"n+k: "; print(c);
26      return 0;
27 }
28 //конструктор по умолчанию
29 NComplex::NComplex(){
30      Re = 0;
31      Im = 0;
32 }
33 //конструктор с параметрами
34 NComplex::NComplex(int Re, int Im){
35      this->Re = Re;
36      this->Im = Im;
37 }
38 //инкремент (префиксный)
39 NComplex operator++(NComplex &obj){
40      obj.Re++;
41      return obj;
42 }
43 //инкремент (постфиксный)
44 NComplex operator++(NComplex &obj, int n){
45      obj.Re++;
46      obj.Im++;
47      return obj;
48 }
49 //сложение комплексных чисел
50 NComplex operator+(NComplex obj1, NComplex obj2){
```

```
51      NComplex mynumb;
52      mynumb.Re = obj1.Re + obj2.Re;
53      mynumb.Im = obj1.Im + obj2.Im;
54      return mynumb;
55  }
56  //вывод числа в консоль
57  void print(NComplex obj){
58      cout<<obj.Re<<" + "<<obj.Im<<"i"<<endl;
59  }
```

Программа одержит переопределение двух форм инкремента: префиксную (строки 39 – 42) и постфиксную (строки 44 – 48). Так как унарные операторы переопределены внешними функциями, то для того, чтобы получить доступ к закрытым членам класса данные функции указаны как дружественные для класса (строки 11 – 14). Обеим унарным функциям в качестве параметра передается ссылка на объект класса, так как при работе оператора изменяется сам объект. В теле операторных функций изменяются значения полей объекта (в префиксной увеличивается на единицу действительная часть, а в постфиксной обе части комплексного числа), а в качестве результата возвращается сам объект класса. Результат работы данного программного кода представлен на рисунке 20.

```
Комплексное число n: 0 + 0i
Комплексное число k: 1 + 2i
n++: 1 + 1i
++n: 2 + 1i
n+k: 3 + 3i
```

Рис. 20. Результат работы программы из листинга 21

3.2 Перегрузка бинарных операций

Бинарные операции – это операции, производящиеся над двумя операндами.

Как и в случае с унарными, бинарные операции могут быть перегружены. Возможны различные ситуации переопределения бинарных операторов: например, оба аргумента являются объектами класса; первый аргумент – объект класса, второй аргумент – операнд стандартного типа данных и наоборот.

Общий синтаксис перегрузки бинарных операций:

```
тип_результата operator знак_оператора(тип_аргумента имя_аргумента, тип_аргумента имя_аргумента){
  //программный код
}
```

В листинге 21 программный код содержит перегрузку операции сложения для объектов класса `NComplex` (строки 50 – 55). Так как при сложении двух комплексных чисел получается также комплексное число, значит в качестве типа возвращаемого значения указывается тип `NComplex`. В самой функции не требуется изменять значения передаваемых параметров, следовательно, аргументы передается по значению (*а не по ссылке!*). В теле функции создается локальный объект `mynumb` типа `NComplex`, который и будет являться суммой двух комплексных чисел и будет возвращается в качестве результата работы функции. С результатом работы перегруженной операции сложения, реализованной в программном коде листинга 21, можно ознакомиться на рисунке 20.

3.3 Перегрузка операций сравнения

При необходимости сравнения двух объектов имеется возможность переопределения операций сравнения. Переопределить операции сравнения можно как внешними функциями, так и методами класса.

Общий синтаксис перегрузки оператора сравнения методами класса:

```
bool operator знак_оператора(const тип& имя_арг-та){
  //программный код
}
```

Разберем перегрузку операторов сравнения на примере решения задачи 3.2, решение данной задачи представлен в листинге 22.

Задача 3.2. Реализовать класс Отрезок. Класс должен содержать:

1. Поля – координаты конечной точки отрезка x и y.

2. Метод определения длины отрезка, началом отрезка является начало системы координат.

3. Конструктор с параметрами.

4. Метод вывода информации об объекте в консоль.

Перегрузить операторы сравнения объектов (по длинам отрезков): ==, !=, <, >.

Листинг 22. Перегрузка операторов сравнения

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

class Line{
    float x, y;
    float len();
    public:
    Line(float x, float y);
    ~Line(){};
    bool operator == (Line & other);
    bool operator != (Line & other);
    friend bool operator < (Line& obj1, Line& obj2);
    friend bool operator > (Line& obj1, Line& obj2);
    void print_line();
};

int main(){
    setlocale(LC_CTYPE, "");
    Line line_1(1,1);
    Line line_2(2,2);
```

```
22      bool result;
23      cout<<"\n Отрезок 1: "; line_1.print_line();
24      cout<<"\n Отрезок 2: "; line_2.print_line();
25      result = line_1==line_2;
26      if(result) cout<<"\n Отрезки равны";
27      else cout<<"\n Отрезки не равны";
28      result = line_1 > line_2;
29      if(result) cout<<"\n Отрезок 1 длиннее";
30      else cout<<"\n Отрезок 2 длиннее";
31      cout<<endl<<endl;
32      return 0;
33  }
34  //конструктор с параметрами
35  Line::Line(float x, float y){
36      this->x = x;
37      this->y = y;
38  }
39  //вычисление длины отрезка
40  float Line::len(){
41      float lenline = sqrt(x*x+y*y);
42      return lenline;
43  }
44  //вывод информации об объекте
45  void Line::print_line(){
46      cout<<"\n Начало отрезка отрезка: X = 0 Y = 0";
47      cout<<"\n Конец отрезка отрезка: X = "<<x<<" Y = "<<y<<
        endl;
48      cout<<" Длина отрезка: "<<this->len();
49  }
50  //перегрузка операторов сравнения
51  bool Line::operator==(Line & other){
52      return this->len()==other.len();
53  }
54
55  bool Line::operator !=(Line & other){
56      return !(this->len()==other.len());
57  }
58  bool operator < (Line& obj1, Line& obj2){
59      return obj1.len() < obj2.len();
60  }
61  bool operator > (Line& obj1, Line& obj2){
62      return obj1.len() > obj2.len();
```

```
63 }
```

Перегруженные операторные функции равенства (==) и неравенства (!=) являются методами класса, следовательно имеют непосредственный доступ к закрытым полям и методам класса. В качестве параметра данные функции принимают всего один параметр (несмотря на то, что оператор является бинарным) – ссылку на объект класса (строки 51 и 55). Это связано с тем, что один аргумент данной функции является объектом класса, из которого эта функция вызывается. Так как при сравнении вызывается метод вычисления длины отрезка, то параметры метода не являются константными. В качестве типа возвращаемого значения указан тип `bool`, функции возвращают значение `true`, если условие истинно, или `false`, если условие ложно (строки 52 и 56).

Перегруженные операторные функции больше (>) и меньше (<) являются глобальными функциями, следовательно, в определении класса они указываются как дружественные (строки 13 и 14). Каждая из функций в качестве параметра принимает две ссылки на объекты класса `Line`. Программный код и логика данных функций проста и аналогична перегруженным операторным функциям равенства и неравенства (строки 59 и 62). Результат работы программы представлен на рисунке 21.

```
Отрезок 1:
Начало отрезка отрезка: X = 0 Y = 0
Конец отрезка отрезка: X = 1 Y = 1
Длина отрезка: 1.41421
Отрезок 2:
Начало отрезка отрезка: X = 0 Y = 0
Конец отрезка отрезка: X = 2 Y = 2
Длина отрезка: 2.82843
Отрезки не равны
Отрезок 2 длиннее
```

Рис. 21. Результат работы программы из листинга 22

3.4 Перегрузка операции присваивания

Операция присваивания (=) по умолчанию определена для любого стандартного типа, а также для любого класса. Данная операция вызывается при присваивании одному объекту значения другого объекта *одного* класса, при этом происходит поэлементное копирование значений.

Перегружать операцию присваивания необходимо для класса, содержащего поля, под которые производится выделение динамической памяти. Переопределение данной операции возможно только методом класса.

Общий синтаксис перегрузки операции присваивания:

```
имя_класса& operator = (имя_класса & имя_аргумента){
  //программный код
}
```

Пример перегрузки операции присваивания приведен в программном коде листинга 23.

Листинг 23. Перегрузка операции присваивания

```
1 #include <iostream>
2 using namespace std;
3
4 class Point{
5     int x, *y;
6 public:
7     Point(int x, int y);
8     void change(int x, int y);
9     void print_info();
10    //прототип перегруженной операции присваивания
11    Point &operator=(Point &obj);
12    ~Point();
13 };
14
15 int main(){
16    setlocale(LC_CTYPE, "");
17    Point a(1,1), b(0,0);
18    cout<<"a: "; a.print_info();
```

```
19      cout<<"b: "; b.print_info();
20      b = a;
21      a.change(2,2);
22      cout<<"a: "; a.print_info();
23      cout<<"b: "; b.print_info();
24      return 0;
25  }
26  //конструктор с параметрами
27  Point::Point(int x, int y){
28      this->y = new int;
29      *this->y = y;
30      this->x = x;
31  }
32  //вывод информации в консоль
33  void Point::print_info(){
34      cout<<" x = "<<x<<" y = "<<*y<<endl;
35  }
36  //изменение объекта
37  void Point::change(int x, int y){
38      this->x = x;
39      *this->y = y;
40  }
41  //перегрузка операции присваивания
42  Point& Point::operator=(Point &obj){
43      this->x = obj.x;
44      *this->y = *(obj.y);
45      return *this;
46  }
47  //деструктор
48  Point::~Point(){
49      delete y;
50  }
```

В программе, представленной выше, реализуется класс Точка (`Point`). Класс содержит: два закрытых поля – координаты точки x и y, последнее из которых является указателем; конструктор с параметрами; деструктор; методы изменения объекта и вывода информации об объекте в консоль; перегруженную операцию присваивания.

Рассмотрим перегрузку операции присваивания. Перегрузка операции производится методом класса (строка 11). В качестве

параметра функция принимает один аргумент — ссылку на объект класса (строка 42). Для сохранения семантики присваивания в качестве возвращаемого значения выступает ссылка на объект, для которого вызывалась функция [7]. Программный код перегруженной операции присваивания достаточно прост и понятен: текущим полям объекта присваиваются значения объекта `obj`. Из результата работы программы (рисунок 22) видно, что объекты между собой не связаны и изменение одного не влечет за собой изменение другого.

```
a:  x = 1 y = 1
b:  x = 0 y = 0
a:  x = 2 y = 2
b:  x = 1 y = 1
```

Рис. 22. Результат работы программы из листинга 23

3.5 Перегрузка операции индексации массива

Операция индексации массива `[]` используется для доступа к элементам массива и, как и многие другие операции, ее можно переопределить. В качестве результата работы функция должна возвращать ссылку на элемент, содержащийся в массиве.

Общий синтаксис перегрузки операции индексации:

```
тип_результата& operator [](тип& имя_аргумента){
  //программный код
}
```

Правила перегрузки операции индексации массива [7]:

1. Оператору запрещено применять «дружественные» функции.

2. Данный перегруженный оператор может быть нестатической функцией-членом.

Перегрузка операции индексации массива позволяет контролировать выход индекса за пределы массива.

Рассмотрим простой пример перегрузки операции индексации массива (программный код листинга 24).

Листинг 24. Пример перегрузки операции индексации без выделения динамической памяти

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

class MyClass{
    int array[5]{1,2,3,4,5};
    public:
    //перегрузка оператора []
    int & operator [](int index){
        return array[index];
    }
    void print(){
        for(int i =0; i < 5; i++){
            cout<<array[i]<<" | ";
        }
    }
};

int main(){
    MyClass a;
    cout<<" Объект до изменения: \n";
    a.print();
    //изменение нулевого элемента
    a[0] = 0;
    cout<<"\n Объект после изменения: \n";
    a.print();
    cout<<endl;
    return 0;
}
```

В программе реализован простой класс `MyClass`, который содержит одно закрытое поле `array` – массив чисел (размер массива заранее известен), и два открытых метода: перегруженный оператор индексации массива и вывод данных объекта (в данном

случае элементов массива) в консоль. Перегруженный оператор принимает в качестве параметра целое число и в результате возвращает ссылку на заданный элемент массива.

Главная функция программы `main()` содержит объект `a` класса `MyClass`. С помощью функции `print()`(строка 21) в консоль выводятся исходные значения элементов массива (данный массив является закрытым полем класса!), после чего с помощью перегруженного оператора индексации происходит обращение к нулевому элементу массива и его изменение (строка 23). После изменений вновь с помощью функции `print()` выводится значения элементов массива. Обратим внимание, что перегруженный оператор индексации позволил изменить значение элемента массива, который является закрытым полем класса. Результат работы данной программы представлен на рисунке 23.

```
Объект до изменения:
1 | 2 | 3 | 4 | 5 |
Объект после изменения:
0 | 2 | 3 | 4 | 5 |
```

Рис. 23. Результат работы программы из листинга 24

Рассмотрим программный код листинга 25 в котором представлена более сложная реализация перегрузки оператора индексации массива.

Листинг 25. Пример перегрузки операции индексации с выделением динамической памяти

```
1 #include <iostream>
2 using namespace std;
3
4 class MyClass{
5     int *n;
6     int mysize;
7 public:
8     MyClass();
9     MyClass(int size);
```

```
10      ~MyClass();
11      void print();
12      //прототип перегруженного оператора []
13      int &operator[](int index);
14  };
15  int main(){
16      int h;
17      cout<<"Введите количество элементов массива: ";
18      cin>>h;
19      MyClass array(h);
20      array.print();
21      //изменение объекта
22      array[1] = 3;
23      array.print();
24      array[14] = 3;
25      cout<<endl;
26      return 0;
27  }
28
29  //методы класса MyClass
30  MyClass::MyClass(){
31      n = 0;
32      mysize = 0;
33  }
34  MyClass::MyClass(int size){
35      this->mysize = size;
36      n = new int[mysize];
37      for(int i = 0; i<mysize; i++){
38          n[i] = rand()%10;
39      }
40  }
41  MyClass::~MyClass(){
42      delete [] n;
43  }
44  void MyClass::print(){
45      for(int i = 0; i < mysize; i++){
46          cout<<n[i]<<" | ";
47      }
48      cout<<endl;
49  }
50  //перегруженный оператор []
51  int& MyClass::operator[](int index){
```

```
52      if(index < 0 || index > mysize){
53          cout<<"Выход за границы массива"<<endl;
54      }
55      return n[index];
56  }
```

Данная программа иллюстрирует перегрузку операции индексации динамического массива. В конструкторе с параметрами стандартным образом для динамического массива выделяется память и производится его заполнение случайными числами (строки 34 – 40). В перегруженном операторе индексации массива производится проверка выхода индекса массива за пределы его допустимого диапазона (строки 51 – 56).

Главная функция программы содержит объект `array` класса `MyClass`. Поля данного объекта инициализируются в конструкторе с параметрами, после чего для данного объекта вызывается метод `print()`, с помощью которого выводятся значения элементов массива (который является полем объекта) в консоль. Далее производится обращение к ячейке массива и изменение ее значения с помощью перегруженного оператора индексации. Строка 24 иллюстрирует обращение к ячейке массива, индекс которой выходит за пределы диапазона, в данном случае выведется сообщение о выходе индекса за пределы массива. Результат работы программы представлен на рисунке 24.

```
Введите количество элементов массива: 10
3 | 6 | 7 | 5 | 3 | 5 | 6 | 2 | 9 | 1 |
3 | 3 | 7 | 5 | 3 | 5 | 6 | 2 | 9 | 1 |
Выход за границы массива
```

Рис. 24. Результат работы программы из листинга 25

3.6 Лабораторная работа №2 «Перегрузка операторов в C++»

Цель работы: знакомство и получения навыков перегрузки операторов в C++.

Вариант 1

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 2

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 3

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 4

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;

3. operator=;
4. operator!=.

Вариант 5

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator+;
3. operator=;
4. operator!=;

Вариант 6

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator+;

Вариант 7

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator/;

Вариант 8

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 9

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 10

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 11

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 12

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 13

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 14

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 15

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 16

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator+.

Вариант 17

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;

4. operator!=.

Вариант 18

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 19

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 20

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 21

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 22

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 23

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 24

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

Вариант 25

Для класса, реализованного в лабораторной работе №1 перегрузить операторы:

1. operator<<;
2. operator<;
3. operator=;
4. operator!=.

4 Наследование

Наследование является одним из фундаментальных понятий объектно-ориентированного программирования. Данный механизм позволяет строить иерархии, в которых одни классы получают свойства других, имея возможность изменять их и дополнять своими собственными. Такой подход позволяет более эффективно управлять большим набором классов, не требуя значительных усилий и многократного повторения участков программного кода.

В данной главе рассматриваются основные принципы наследования и технологии, базирующие на нем.

4.1 Основные понятия

Основными понятиями механизма наследования являются **базовый** и **производный классы**. Базовый класс (родительский класс, суперкласс) – класс, содержащий наиболее общие поля и методы объектов. Производный класс (дочерний класс) – класс, наследующий поля и методы базового класса. Производный класс может иметь свои собственные поля и методы, кроме того, он может выступать в качестве базового класса по отношению к другому классу.

Производный класс создается на основе базового класса. Регулирование того, какие методы и поля будут наследоваться производным классом осуществляется с помощью модификаторов доступа базового класса и механизмом самого наследования. Различают следующие виды наследования: открытое, защищенное и закрытое.

Для понимания того, как работают различные механизмы наследования, рассмотрим синтаксис простого наследования:

```
class Производный_класс: модификатор_доступа Базовый_класс{
  //поля и методы производного класса};
```

Таблица 1

Наследование и доступ

Механизм наследования	Спецификатор в базовом классе	Доступ в производном классе
public	public	public
	protected	protected
	private	нет
protected	public	protected
	protected	protected
	private	нет
private	public	private
	protected	private
	private	нет

В качестве модификатора доступа (механизма наследования) указывают `public, private` или `protected`. В случае, когда не указан модификатор доступа, используется модификатор по умолчанию – `private`.

Модификатор доступа указывает доступность наследуемых полей и методов класса. В Таблице 1, приведенной ниже, отражены возможности использования спецификаторов доступа.

Открытое наследование (public) является одним из самых простых для понимания и наиболее часто используемых. При данном наследовании уровень доступа к полям и методам в производном классе остается неизменным. Исключения составляют private-поля, они недоступны в дочернем классе.

Защищенное наследование (protected) наименее распространено и, как правило, используется в особых случаях. При данном виде наследования открытые и защищенные поля и методы в производном классе станут protected, а закрытые члены базового класса будут недоступны.

При закрытом наследовании (private) открытые и защи-

щенные члены становятся закрытыми, а закрытые остаются недоступными для производного класса.

4.2 Простое наследование

Простым наследованием называется такое наследование, при котором производный класс имеет одного родителя.

Общий синтаксис наследования:

```
class П-класс: модификатор_доступа Б-класс{
  //поля и методы производного класса
};
```

где `П-класс` – имя производного класса, а `Б-класс` – имя базового класса.

Рассмотрим реализацию простого наследования на примере программного кода из листинга 26.

Листинг 26. Пример простого наследования

```
1 #include <iostream>
2 using namespace std;
3 //базовый класс
4 class One{
5     int a;
6 public:
7     int b;
8     void print_a();
9     void get_a(int a);
10 };
11 //производный класс
12 class Two: public One{
13     int c;
14 public:
15     void print();
16     void get(int b, int c);
17 };
18
19 int main(){
20     setlocale(LC_CTYPE, "");
21     //создание объекта класса Two
```

```
22      Two obj;
23      //изменение значений полей
24      obj.get(2, 7);
25      //вывод информации в консоль
26      obj.print();
27      //изменение значения поля A
28      obj.get_a(11);
29      //вывод значения поля A в консоль
30      obj.print_a();
31      cout<<endl;
32      return 0;
33  }
34  //метод вывода информации
35  void Two::print(){
36      cout<<" b = "<<b<<endl;
37      cout<<" c = "<<c<<endl;
38  }
39  //метод изменения значений полей объекта
40  void Two::get(int b, int c){
41      this->b = b;
42      this->c = c;
43  }
44  //метод вывода значения поля A в консоль
45  void One::print_a(){
46      cout<<" a = "<<a<<endl;
47  }
48  //метод изменения поля A
49  void One::get_a(int a){
50      this->a=a;
51  }
```

Программа содержит два класса: базовый класс `One` и производный `Two`. Базовый класс содержит закрытое поле `x` и открытое поле `y`. Производный класс `Two` строится на основе класса `One` и наследует все его поля и методы, используется открытый механизм наследования. Кроме наследуемых полей и методов класс `Two` содержит свои методы `print`, `get` и поле `c`. Для доступа производного класса к закрытому наследуемому полю `a` используются открытые методы класса `One` (`get_a` и `print_a`).

В главной функции `main()` создается объект `obj` класса `Two`.

В строке 24 вызывается метод для изменения значений полей `b, c` объекта `obj`, после этого в 26 строке выводятся значения данных полей в консоль. Для того, чтобы изменить значение поля `a` в 28 строке программы вызывается метод `get_a`, а для вывода его значения в консоль в 30 строке вызывается метод `print_a`. В результате работы программного кода в консоль будут выведены значения полей объекта: `b = 2, c = 7, a = 11`.

4.3 Конструкторы и деструкторы производных классов

Конструктор и наследование

В языке программирования C++ конструкторы НЕ наследуются, производный класс должен иметь свой собственный конструктор. Однако существует ряд правил, которых необходимо придерживаться при работе с производными классами:

1. Если в производном классе не определен ни один конструктор, тогда компилятор сам сгенерирует конструктор по умолчанию.

2. Если базовый класс содержит конструктор по умолчанию, тогда для производного класса конструктор определяется только в том случае, ели необходимо инициализировать поля, введенные в производном классе.

3. Если базовый класс содержит конструкторы с параметрами, тогда в производном классе необходимо задать конструктор с параметрами для их передачи конструктору базового класса.

Деструктор и наследование

Как и в случае с конструкторами, деструкторы НЕ наследуются. Определение деструктора для производного класса требуется только в том случае, если необходимо освобождение динамической памяти, выделенной конструктором производного класса для его полей.

Рассмотрим работу с конструкторами производных классов на примере программы из листинга 27.

Листинг 27. Конструктор и наследование

```
#include <iostream>
#include <string>
using namespace std;
class Shape{
    protected:
    string name;
    string color;
    public:
    Shape(string color);
    ~Shape(){};
};

class Circle: public Shape{
    //радиус круга
    float r;
    public:
    Circle(string color, float r);
    void print_cir();
};

class Square: public Shape{
    //длина стороны
    float l;
    public:
    Square(string color, float l);
    void print_sq();
};

int main(){
    setlocale(LC_CTYPE, "");
    //объект производного класса Circle
    Circle obj1("красный", 1);
    //объект производного класса Square
    Square obj2("желтый", 7);
    //вывов метода print_cir для объекта obj1
    obj1.print_cir();
    //вызов метода print_qs для объекта obj2
    obj2.print_sq();
```

```
39      return 0;
40  }
41  //Базовый класс
42  Shape::Shape(string color){
43      this-> name = "Фигура";
44      this->color = color;
45  };
46
47  //Производный класс Circle
48   void Circle::print_cir(){
49      cout<<name<<"\n Радиус: "<<r<<endl<<"Цвет: "<<color<<
        endl<<endl;
50  }
51  Circle::Circle(string color, float r): Shape(color){
52      name = "Круг";
53      this->r = r;
54  };
55
56  //Производный класс Square
57  Square::Square(string color, float l): Shape(color){
58      name = "Квадрат";
59      this->l = l;
60  }
61  void Square::print_sq(){
62      cout<<name<<"\n Длина стороны:  "<<l<<"\n Цвет: "<<
        color<<endl;
63  }
```

Программный код из листинга 27 содержит один базовый класс – `Shape` (Фигура, строки 4 – 11). В нем описаны два поля: `name` и `color` имеющие модификатор доступа `protected`, конструктор с параметрами и деструктор, которые являются открытыми членами-функциями класса. Кроме базового класса в программе имеется два производных: `Circle` (Круг) и `Square` (Квадрат). Каждый из производных классов классов имеется свое поле (`Circle` – радиус, а `Square` – длину стороны), следовательно, производные классы должны иметь собственные конструкторы для инициализации данных полей. При создании производных классов необходимо учитывать способ передачи аргументов производному классу. В производном классе `Circle` один аргумент переда-

ется конструктору базового класса (`color`), другой аргумент используется для инициализации поля, описанного в производном классе (`r` – радиус)(строки 51 – 54). Те же действия произведены и для второго производного класса (строки 57 – 60).

Данный механизм обусловлен способом создания объектов производного класса. Так, при создании объекта производного класса сначала вызывается конструктор базового класса, а затем конструктор производного класса.

В главной функции программы создаются два объекта производных классов: для класса `Circle` объект `obj1`, а для класса `Square` – объект `obj2` (строки 32 – 34). Для каждого из этих объектов вызывается конструктор с параметрами. Каждый из производных классов имеет собственный метод для вывода информации об объекте в консоль, далее в главной функции и производится вызов данных методов из объектов (строки 36 – 38). Результат работы программы представлен ниже на рисунке 25

```
Круг
Радиус: 1
Цвет: красный

Квадрат
Длина стороны:  7
Цвет: желтый
```

Рис. 25. Результат работы программы из листинга 27

4.4 Множественное наследование

В языке программирования C++ имеется возможность **множественного наследования**, т.е. производный класс может наследовать поля и методы сразу от нескольких базовых классов.

Для доступа к полям и методам такого производного класса используются те же правила, что и для производного класса, порожденного от одного базового класса. Однако при мно-

жественном наследовании могут возникнуть несколько проблем (неоднозначность наследования):

1. В производном классе содержатся поле с таким же именем, как и в одном из базовых классов.

2. В базовых классах имеются поля и/или методы с одинаковыми именами.

В таких случаях необходимо использовать оператор разрешения контекста(`имя_класса::имя_поля`) для уточнения, к какому из полей осуществляется доступ.

Конструкторы для производных классов при множественном наследовании вызываются в том же порядке, в котором они наследуются.

Ознакомимся с реализацией множественного наследования в программном коде листинга 28, который является решением задачи 4.1.

Задача 4.1. Написать программу, содержащую два базовых класса и один производный от них. Каждый из классов должен содержать одно закрытое поле и конструктор.

Листинг 28. Реализация множественного наследования

```
#include <iostream>
using namespace std;
//базовый класс
class A{
    int number_1;
    public:
    A(int n){
        number_1 = n;
        cout<<" "<<number_1;
    }
};
//базовый класс
class B{
    int number_2;
    public:
    B(int n){
        number_2 =  n;
        cout<<" "<<number_2;
```

```
19      }
20  };
21  //производной класс
22  class C: public A, public B{
23      int number_3;
24      public:
25      C(int a, int b, int c): A(a), B(b){
26          number_3 = c;
27          cout<<" "<<number_3<<endl;
28      }
29  };
30
31  int main(){
32      setlocale(LC_CTYPE, "");
33      //объект производного класса
34      C obj3(101, 1022, 3003);
35      cout<<endl;
36      return 0;
37  }
```

Программа из листинга 28 иллюстрирует реализацию множественного наследования. В программе содержится три класса: два базовых (`class A` и `class B`) и один производный (`class C`). Класс `C` является производным сразу для двух классов (`A` и `B`) и наследует их поля и методы. То, что класс C является наследником двух классов отражено в 22 строке программы (после имени производного класса указано два родительских).

Кроме того, производный класс содержит свое собственное поле `number_3`, и, следовательно, для инициализации этого поля необходимо описание собственного конструктора. При определении конструктора производного класса также необходимо указать конструкторы базовых классов в порядке их наследования (строка 25). Параметры `a` и `b` передаются конструкторам базовых классов (строка 25), а параметром `c` инициализируется поле производного класса в конструкторе этого класса (строка 26).

Главная функция программы содержит объект `obj3` производного класса `C`. При создании объекта передается три параметра, которыми инициализируются поля класса.

4.5 Многоуровневое наследование

В C++ можно осуществлять наследование не только от базового класса, но и от производного, который будет выступать базовым для класса, построенного на его основе. Такое наследование принято называть **многоуровневым наследованием**.

Рассмотрим реализацию многоуровневого наследования на примере решения задачи 4.2, представленной в программном коде листинга 29.

Задача 4.2. Написать программу, реализующую иерархию классов точка (в одномерном пространстве, в двухмерном пространстве, в трехмерном пространстве). Поля классов – координаты точки. Методы класса – конструктор, вывод информации об объекте в консоль.

Листинг 29. Многоуровневое наследование

```
#include <iostream>
using namespace std;
//описание классов
class point1D{
    protected:
    int x;
    public:
    point1D(int x);
    void print_point();
};
class point2D: public point1D{
    protected:
    int y;
    public:
    point2D(int x, int y);
    void print_point();
};
class point3D: public point2D{
    protected:
    int z;
    public:
    point3D(int x, int y, int z);
    void print_point();
};
```

```
25
26 int main(){
27     //создание объектов
28     point1D point1(1);
29     point2D point2(2, 4);
30     point3D point3(0, 0, 0);
31     //вызов методов для объектов
32     cout<<"Координаты точки point1: \n"; point1.print_point
       ();
33     cout<<"Координаты точки point2: \n"; point2.print_point
       ();
34     cout<<"Координаты точки point3: \n"; point3.print_point
       ();
35     return 0;
36 }
37 //описание методов классов
38 point1D::point1D(int x){
39     this->x = x;
40 }
41 void point1D::print_point(){
42     cout<<"X = "<<x<<endl;
43 }
44 point2D::point2D(int x, int y):point1D(x){
45     this->y = y;
46 }
47 void point2D::print_point(){
48     cout<<"X = "<<x<<endl<<"Y = "<<y<<endl;
49 }
50 point3D::point3D(int x, int y, int z):point2D(x, y){
51     this->z = z;
52 }
53 void point3D::print_point(){
54     cout<<"X = "<<x<<endl<<"Y = "<<y<<endl<<"Z = "<<z<<
       endl;
55 }
```

Рассмотрим реализацию многоуровневого наследования в программном коде, представленном выше: класс `point1D` является базовым классом для класса `point2D`, который, в свою очередь, является базовым для класса `point3D`. Класс `point1D` содержит одно защищенное поле (`x` – координата точки оси `x`) и

открытые методы: конструктор с параметрами и вывод информации в консоль. Класс `point2D` наследует от базового класса (`point1D`) его поля и методы, а также имеет свое собственное поле (`y` – координата точки оси `y`) и собственный конструктор; метод вывода информации в консоль `print_point()` переопределен в производном классе. Класс `point3D` является производным для класса `point2D`, он наследует его поля и методы, имеет собственное поле и собственный конструктор, метод `print_point()` также переопределен для данного производного класса.

Главная функция программы содержит объекты данных классов. Для каждого из объектов вызывается метод вывода информации об этом объекте в консоль. Результат работы программы представлен на рисунке 26.

```
Координаты точки point1:
X = 1
Координаты точки point2:
X = 2
Y = 4
Координаты точки point3:
X = 0
Y = 0
Z = 0
```

Рис. 26. Результат работы программы из листинга 29

5 Полиморфизм

5.1 Основные понятия

Полиморфизм – это один из базовых принципов объектно-ориентированного программирования, возможность обрабатывать объекты различных типов аналогичными способами. Данное свойство позволяет использовать одно и тоже имя функции для решения двух и более схожих, но технически разных задач; замещать методы объекта-родителя методами объекта-потомка, имеющих то же имя.

В языке программирования C++ различают два вида полиморфизма:

1. Статический полиморфизм.
2. Динамический полиморфизм.

Статический полиморфизм достигается путем использования перегруженных функций (механизм раннего связывания). Динамический полиморфизм достигается при реализации наследования с использованием виртуальных функций (механизм позднего связывания).

5.2 Виртуальные функции

Понятие полиморфизма тесно связано с понятием виртуальных функций. Виртуальная функция – это функция, объявленная в базовом классе и переопределяемая в производном классе. Объявление виртуальной функции производится с помощью ключевого слова `virtual`.

Общий синтаксис объявления виртуальной функции:

```
virtual тип_функции имя_функции (параметры);
```

Виртуальная функция имеет свою собственную реализацию для каждого наследуемого класса, выполняя действия, свойственные только этому классу.

Рассмотрим пример использования виртуальных функции в программном коде листинга 30.

Листинг 30. Использование виртуальной функции

```
1 #include <iostream>
2 using namespace std;
3 //базовый класс
4 class BaseClass{
5     public:
6     //виртуальная функция
7     virtual void print(){
8         cout<<"Базовый класс \n";
9     }
10 };
11 //производный класс
12 class MyClass: public BaseClass{
13     public:
14     //переопределение виртуальной функции
15     virtual void print(){
16         cout<<"Производный класс \n";
17     }
18 };
19 int main(){
20     //создание объекта базового класса
21     BaseClass obj1;
22     //создание объекта производного класса
23     MyClass obj2;
24     //указатель на базовый класс
25     BaseClass *p;
26     //вызов методов для объектов
27     obj1.print();
28     obj2.print();
29     //доступ к виртуальной функции через указатель
30     p = &obj2;
31     p->print();
32     return 0;
33 }
```

Программа содержит два класса: базовый класс `BaseClass` и производный от него `MyClass`. В базовом классе описана виртуальная функция `print()`(строки 7 – 9), которая наследуется

производным классом. В производном класса функция `print()` переопределяется (строки 15 – 17).

Главная функция программы содержит объект `obj1` базового класса, объект `obj2` производного класса и указатель `p` на базовый класс. Для каждого объекта вызывается метод `print()` (строки 27 – 28), который вывод сообщение в консоль. Указателю присваивается ссылка на объект `obj2` (строка 30), после чего по указателю вызывается метод `print()` для производного класса (строка 31). Результат работы программы представлен на рисунке 27.

Несмотря на то, что в базовом классе функция `print()` уже указана как виртуальная, в производном классе также рекомендуется явно указывать функцию как виртуальную.

```
Базовый класс
Производный класс
Производный класс
```

Рис. 27. Результат работы программы из листинга 30

5.3 Абстрактные классы

Перед изучением абстрактных классов познакомимся с понятием чисто виртуальных функций. **Чисто виртуальными функциями** называются функции, при объявлении которых в базовом классе после прототипа указывается оператор присваивания и нуль.

Общий синтаксис объявления чисто виртуальной функции:
`virtual тип_функции имя_функции (параметры) = 0;`

Класс, содержащий хотя бы одну чисто виртуальную функцию называется **абстрактным.**

В производных классах чисто виртуальные функции переопределяются, вызов таких функций из базового класса невозможен, т.к. отсутствует описание функции.

Класс, производный от абстрактного также может быть абстрактным. В этом случае такой класс просто переопределяет функцию базового абстрактного класса.

Рассмотрим программу листинга 31, в которой реализован абстрактный класс и чисто виртуальная функция.

Листинг 31. Абстрактный класс и чисто виртуальная функция

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;
//абстрактный класс
class Figure{
    string name;
    protected:
    Figure(string name);
    //чисто виртуальная функция
    virtual float area() = 0;
};
//производный класс
class Circle:public Figure{
    float R;
    public:
    Circle(string name, float r);
    //прототип переопределяемой функции
    float area();
};
//производный класс
class Square:public Figure{
    float a;
    public:
    Square(string name, float a);
    //прототип переопределяемой функции
    float area();
};
int main(){
    //объект класса Circle
```

```
31      Circle f1("Круг", 2);
32      cout<<"Площадь: "<<f1.area()<<endl<<endl;
33      //объект класса Square
34      Square f2("Квадрат", 4);
35      cout<<"Площадь: "<<f2.area()<<endl;
36      return 0;
37  }
38  //методы классов
39  Figure::Figure(string name){
40      this->name = name;
41      cout<<name<<endl;
42  }
43  Circle::Circle(string name, float r):Figure(name){
44      R = r;
45      cout<<"Длина радиуса: "<<R<<endl;
46  }
47  float Circle::area(){
48      return M_PI*R*R;
49  }
50  Square::Square(string name, float a): Figure(name){
51      this->a = a;
52      cout<<"Длина стороны: "<<a<<endl;
53  }
54  float Square::area(){
55      return a*a;
56  }
```

Данная программа содержит один базовый класс `Figure`, этот класс является абстрактным, т.к. содержит чисто виртуальную функцию `area()` (строка 11). Абстрактный класс `Figure` является базовым для классов `Circle` и `Square`. В производных классах переопределена функция `area()`, реализующая вычисление площади фигур (строки 47 – 49, 54 – 56).

В главной функции программы имеются два объекта: объект `f1` класса `Circle` и объект `f2` класса `Square`. Для каждого из объектов вызывается свой метод `area()`, возвращающий площадь объекта. На рисунке 28 представлен результат работы данного программного кода.

Примечание: создание объекта абстрактного класса *невоз-*

можно, исходя из этого очевидно, что не имеет смысла делать конструктор открытым, поэтому он относится к модификатору доступа `protected`.

```
Круг
Длина радиуса: 2
Площадь: 12.5664

Квадрат
Длина стороны: 4
Площадь: 16
```

Рис. 28. Результат работы программы из листинга 31

5.4 Лабораторная работа №3 «Наследование и полиморфизм»

Цель работы: изучение и приобретение навыков реализации наследования и полиморфизма в языке C++.

Задания 1 – 5. Написать программу на языке программирования C++ согласно своему варианту.

1. Согласно своему варианту спроектировать несколько классов, используя механизм наследования.

2. Предусмотреть у классов наличие полей и методов.

3. Один из методов базового класса должен быть перегружен в классе-наследнике.

4. Один из классов должен содержать виртуальный метод, который переопределяется в производном классе.

5. Продемонстрировать работу всех объявленных методов, а также конструкторов и деструкторов.

Вариант 1

Написать программу, в которой описана иерархия классов: треугольник (равносторонний, разносторонний, равнобедренный,

прямоугольный). Базовый класс должен иметь: поле, хранящее имя треугольника; методы вычисления площади и периметра. Продемонстрировать работу всех методов классов.

Вариант 2

Написать программу, в которой описана иерархия классов: окружность (круг, кольцо, эллипс). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления площади и периметра. Продемонстрировать работу всех методов классов.

Вариант 3

Написать программу, в которой описана следующая иерархия классов: четырехугольник (квадрат, прямоугольник, ромб). Базовый класс должен иметь: поле, хранящее имя четырехугольника; методы вычисления площади и периметра четырехугольника. Продемонстрировать работу всех методов классов.

Вариант 4

Написать программу, в которой описана иерархия классов: тела вращения (тор, конус, цилиндр). Базовый класс должен содержать: поле, хранящее имя объекта; методы вычисления площади и объема тела. Продемонстрировать работу всех методов классов.

Вариант 5

Написать программу, в которой описана иерархия классов: призма (правильная, прямая, наклонная). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления площади и периметра фигур. Продемонстрировать работу всех методов классов.

Вариант 6

Написать программу, в которой описана иерархия классов: обыкновенная дробь (правильная, неправильная). Базовый класс должен иметь: поля числителя и знаменателя; методы сложения

и умножения дробей (перегруженные операторы). Продемонстрировать работу всех методов классов.

Вариант 7

Написать программу, в которой описана иерархия классов: матрица (вектор, квадратная, прямоугольная). Базовый класс должен иметь: поле, хранящее имя объекта; метод вычисления максимальной суммы элементов столбца матрицы. Продемонстрировать работу всех методов классов.

Вариант 8

Написать программу, в которой описана иерархия классов: квадратная матрица (две треугольные матрицы: L и R). Базовый класс должен иметь: поле, хранящее имя объекта; метод заполнения матрицы случайными числами; метод вычисления суммы элементов матрицы. Продемонстрировать работу всех методов классов.

Вариант 9

Написать программу, в которой описана иерархия классов: круг (конус, цилиндр). Базовый класс должен иметь: поле, хранящее имя объекта; поле, хранящее радиус круга; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 10

Написать программу, в которой описана иерархия классов: квадрат (пирамида, прямоугольный параллелепипед). Базовый класс должен иметь: поле, хранящее имя объекта; поля, хранящее длины сторон прямоугольника; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 11

Написать программу, в которой описана иерархия классов: равносторонний треугольник (правильный тетраэдр, пирамида).

Базовый класс должен иметь: поле, хранящее имя объекта; поле, хранящее длину стороны треугольника; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 12

Написать программу, в которой описана иерархия классов: правильный шестиугольник (правильная шестиугольная пирамида, правильный шестиугольный параллелепипед). Базовый класс должен иметь: поле, хранящее имя объекта; поле, хранящее длину стороны шестиугольника; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 13

Написать программу, в которой описана иерархия классов: эллипс (тор, эллипсоид вращения). Базовый класс должен иметь: поле, хранящее имя объекта; поля, хранящие длины большой и малой осей эллипса; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 14

Написать программу, в которой описана иерархия классов: четырехугольник (квадрат, прямоугольник, прямоугольник со скругленными углами). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 15

Написать программу, в которой описана иерархия классов: круг (кольцо, шар). Базовый класс должен иметь: поле, хранящее имя объекта; поле, хранящее радиус; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 16

Написать программу, в которой описана иерархия классов: печатное издание (книга, журнал, газета). Базовый класс должен иметь: поле, хранящее имя объекта; поля, хранящие количество страниц и тираж; метод вычисления стоимости печатного издания (количество страниц * тип издания (книга: *10; журнал: *0.5*тираж+100; *0.1*тираж+10)). Продемонстрировать работу всех методов классов.

Вариант 17

Написать программу, в которой описана иерархия классов: окружность (цилиндр, полый цилиндр). Базовый класс должен иметь: поле, хранящее имя объекта; поля, хранящие радиус окружности и высоту; методы вычисления площади и объема фигуры. Продемонстрировать работу всех методов классов.

Вариант 18

Написать программу, в которой описана иерархия классов: отрезок (в двухмерном пространстве, в трехмерном пространстве). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления и сравнения длины отрезка. Продемонстрировать работу всех методов классов.

Вариант 19

Написать программу, в которой описана иерархия классов: кинокартина (фильм, мультфильм, сериал). Базовый класс должен иметь: поля – название, режиссер, длительность, год выпуска; методы вычисления стоимости кинокартины. Для фильма и мультфильма стоимость вычислить по формуле: длительность (мин) $\times$ 30 $\times$ год/10, если режиссер Стивен Спилберг, то стоимость увеличивается в 3 раза. Стоимость сериала – длительность серии $\times$ 30 $\times$ количество серий. Продемонстрировать работу всех методов классов.

Вариант 20

Написать программу, в которой описана иерархия классов: фигура (треугольник, четырехугольник). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления периметра и площади фигуры. Продемонстрировать работу всех методов классов.

Вариант 21

Написать программу, в которой описана иерархия классов: вагон (пассажирский, грузовой, цистерна). Базовый класс должен иметь: поля, хранящие имя объекта, объем и количество осей; методы вычисления и сравнения грузоподъемности вагона (пассажирский: объем / 10; грузовой: 0.5×объем×количество осей; цистерна: объем×количество осей). Продемонстрировать работу всех методов классов.

Вариант 22

Написать программу, в которой описана иерархия классов: геометрические фигуры (круг, квадрат, треугольник). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления и сравнения площадей фигур. Продемонстрировать работу всех методов классов.

Вариант 23

Написать программу, в которой описана иерархия классов: геометрические фигуры (трапеция, ромб, параллелограмм). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления и сравнения площадей фигур. Продемонстрировать работу всех методов классов.

Вариант 24

Написать программу, в которой описана иерархия классов: геометрические фигуры (куб, параллелепипед, сфера). Базовый класс должен иметь: поле, хранящее имя объекта; методы вычисления площади и объема фигур. Продемонстрировать работу всех методов классов.

Вариант 25

Написать программу, в которой описана иерархия классов: уравнение (линейное, квадратное, кубическое ($ax^3+bx^2+cx=0$)). Базовый класс должен иметь: поле, хранящее имя объекта; метод вычисления корней уравнения. Продемонстрировать работу всех методов классов.

6 Работа с файлами

6.1 Классы файловых потоков

До этого момента мы работали с программами, которые выводили и считывали информацию с экрана. Однако в программировании зачастую приходится работать с файлами, т.е. считывать и записывать информацию из них.

Дадим определение понятию «файл». **Файл** — это именованная область данных на носителе, содержащая некоторую логически объединенную информацию. В программировании файлы делятся на два типа: текстовые и двоичные (или бинарные).

Текстовые файлы – это файлы, данные в которых интерпретируются как последовательность символьных кодов. В отличие от текстовых файлов, **бинарные файлы** могут состоять не обязательно из печатаемых символов со стандартными разделителями между ними. Основной способ работы с ними – чтение и запись набора байт указанного размера. Данное разделение можно считать условным, так как вся информация, вне зависимости от типа файла, представляется в двоичном формате и в целом работа с обоими типами данных аналогична. Однако, все же существуют некоторые нюансы в работе с двоичными файлами.

Для работы с файлами в C++ используется библиотека `<fstream>`. Она хранит реализацию логики работы с потоками ввода и вывода. **Поток** – это некоторая последовательность байтов, которая либо последовательно считывается, либо последовательно записывается в файл.

При работе с файлами используются следующие классы: `ifstream` — для чтения данных из файла (файловый ввод), `ofstream` — для записи информации в файл (файловый вывод) и `fstream` — чтение и запись. Общий синтаксис создания объекта класса для работы с файлом аналогичен объявлению переменной стандартного типа, т.е. сначала указывается тип объекта, а затем его имя.

Рассмотрим этапы работы с файлами. Сначала создаются

необходимые потоки (объекты), в зависимости от предстоящей работы.

Общий синтаксис создания объекта для записи в файл:

```
ofstream имя_объекта;
```

Общий синтаксис создания объекта для чтения файла:

```
ifstream имя_объекта;
```

Данный синтаксис позволяет создавать объекты для работы с файлами, но не связывает эти объекты с файлами. Следующим этапом работы будет являться открытие файла и его связывание с объектом, созданным на предыдущем этапе.

Общий синтаксис открытия файла и связывание его с объектом выглядит следующим образом:

```
имя_объекта.open(«имя_файла»);
```

На данном этапе указанный файл связывается с объектом соответствующего класса, с помощью которого и будет производится дальнейшая работа с файлом. В качестве имени файла указывается имя файла с расширением (если текстовый файл находится в папке с проектом), либо указывается полное имя файла (т.е. полный путь до файла). Функция `open()` необходима для открытия файла.

Данные два этапа можно объединить в следующую конструкцию (для чтения файла):

```
ifstream имя_объекта("имя_файла");
```

Для записи в файл:

```
ofstream имя_объекта("имя_файла");
```

Стоит отметить, что всегда необходимо производить проверку успешного открытия файла, т.к. могут возникать различного рода ошибки, например: файл не существует, нет прав доступа к указанному файлу и т.д.

Таблица 2

Режимы работы с файлами

Режим открытия	Назначение
ios::in	Открыть файл для чтения
ios::out	Открыть файл для записи
ios::ate	После открытия перейти в конец файла
ios::app	Открыть файл для записи в конец файла (если не установлен trunc)
ios::trunc	Удаление всей информации из файла
ios::binary	Открытие файла в двоичном режиме

Для проверки успешного открытия файла используется функция `is_open()`, которая возвращает `true` при успешном открытии файла и `false` в противном случае.

Общий синтаксис использования функции `is_open()`:
`имя_объекта.is_open();`

Возможны несколько режимов работы с файлами. В режиме по умолчанию опускается второй параметр конструкторов `ofstream, ifstream` (метод `open()` также может принимать два параметра) и предполагается работа с текстовыми файлами. В Таблице 2 указаны возможные режимы работы с файлами.

Пример открытия файла в режиме чтения:
```
ifstream ffile("test.txt",ios::in);
```

В примере выше файл `test.txt` связывается с объектом `ffile`, и открывается в режиме чтения, о чем нам указывает параметр `ios::in`.

Для работы с файлом в нескольких режимах используется логический оператор ИЛИ (|).

Таблица 3

Основные методы класса `ifstream`

Метод	Описание
open	Открывает файл для чтения
close	Закрывает файл
get	Читает один или более символов из файла
getline	Читает символьную строку из текстового файла или данные из бинарного файла до определенного ограничителя
read	Считывает заданное число байт из файла
eof	Возвращает `true`, когда указатель потока достигает конца файла
peek	Выдает очередной символ потока, но не выбирает его
seekg	Перемещает указатель позиционирования файла в заданное положение
tellg	Возвращает текущее значение указателя позиционирования файла

Общий синтаксис работы с файлом в нескольких режимах:
`имя_кл имя_об(имя_ф, режим_работы | режим_работы);`,
где `имя_кл` – имя класса, `имя_ф` – имя файла, `имя_об` – имя объекта.

6.2 Чтение из файла

Класс `ifstream` предназначен для чтения файла. Он содержит ряд методов, помогающих в работе с файлами. В Таблице 3 представлены основные методы класса.

Для детального изучения принципов работы с файлами рассмотрим листинги 32 – 33, в которых представлено решение задач 6.1 и 6.2 соответственно.

Задача 6.1 Написать программу, считывающую данные из

файла в двухмерный массив целого типа размером $n \times n$. Размер массива содержится в первой строке файла. В полученном массиве найти наименьший элемент. Вывести в консоль массив, его размерность и минимальный элемент.

Листинг 32. Чтение квадратной матрицы из файла

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

int main(){
    //локализация
    setlocale(LC_CTYPE, "");
    //создание объекта для считывания из файла
    ifstream file;
    //указатель на массив
    int** array;
    int size;
    int min;
    //открытие файла
    file.open("test.txt");
    //если файл не удалось открыть
    if (!file.is_open()){
      cout<<"Ошибка открытия файла!"<<endl;
    }
    else{
      //при успешном открытии файла
      cout<<"Файл успешно открыт!"<<endl;
      //чтение размера матрицы
      file >> size;
      //выделение динамической памяти
      array = new int*[size];
      for (int i = 0; i < size; i++){
          array[i] = new int[size];
          //чтение элементов и их запись в массив
          for (int j = 0; j < size; j++)
              file >> array[i][j];
      }
      //закрытие файла
      file.close();
      min = array[0][0];
```

```
36          cout<<"Размер матрицы: "<<size<<endl;
37          cout<<"Матрица из файла: "<<endl;
38          //вывод массива в консоль и
39          //нахождение минимального элемента массива
40          for (int i=0; i<size; i++){
41              for(int j=0; j<size; j++){
42                  cout<<array[i][j]<<" ";
43                  if(array[i][j]<min){
44                      min=array[i][j];
45                  }
46              }
47              cout<<endl;
48          }
49          cout<<"Минимальный элемент матрицы: "<<min<<endl;
50          //освобождение памяти
51          for(int i=0;i<size;i++){
52              delete [] array[i];
53          }
54          delete [] array;
55      }
56      return 0;
57  }
```

В 9 строке данного программного кода создается объект `file` класса `ifstream`, с помощью которого и будет производится дальнейшая работа с файлом. Так как заранее неизвестен размер массива необходимо создать указатель (строка 11) для дальнейшего выделения динамической памяти под массив. В 12 и 13 строках создаются переменные для хранения размера массива и значения его минимального элемента. В 15 строке файл `test.txt` связывается с объектом `file` и открывается. В связи с тем, что необходимо удостоверится в успешном открытии файла, в строке 17 для этого определено условие, которое выводит информацию об успешном или неуспешном открытии файла. В случае успешного открытия файла считывается его размер (строка 24), после чего выделяется динамическая память под массив и происходит заполнение самого массива данными из файла (строки 26 – 32). Обратим внимание на механизм заполнения массива. В целом он аналогичен функции `cin`, только вместо данной функ-

ции пишется имя объекта, который был ранее связан с файлом (строка 31). После завершения работы с файлом необходимо его закрытие, делается это с помощью функции `close()` (строка 34). Далее производится стандартная работа с массивом, с помощью циклов массив выводится в консоль и параллельно определяется его наименьший элемент (строки 35 – 48), после чего выводится информация в консоль об наименьшем элементе массива. После завершения работы с массивом необходимо освобождение ранее выделенной под него динамической памяти, что реализовано в строках 51 – 55.

Результаты работы программы из листинга 32 представлены на рисунке 29.

```
Файл успешно открыт!
Размер матрицы: 3
Матрица из файла:
45 12 72
31 89 41
-1 56 34
Минимальный элемент матрицы: -1
```

Рис. 29. Результат работы программы из листинга 32

Задача 6.2 Написать программу, считывающую данные из файла в двухмерные массивы целого типа размером $n \times m$. Структура файла следующая: в первой строке содержится размер первой матрицы, затем следуют элементы массива, далее снова указывается размер матрицы, а после этого следуют элементы массива. Найти сумму матриц и вывести в консоль.

С решением задачи 6.2 можно ознакомиться в листинге 33.

Листинг 33. Решение задачи 6.2

```
1 #include <iostream>
2 #include <fstream>
3 #include <iomanip>
4 using namespace std;
```

```
5
6 int main(){
7     //локализация
8     setlocale(LC_CTYPE, "");
9     //создание объекта для считывания из файла
10    ifstream file;
11    //указатели на массивы
12    int** array1;
13    int** array2;
14    //размеры массивов
15    //n1 - количество строк, m1 - количество столбцов первого массива
16    //n2 - количество строк, m2 - количество столбцов второго массива
17    int size_n1, size_m1, size_n2, size_m2;
18    //открытие файла
19    file.open("test.txt");
20    //если не удалось открыть файл
21    if (!file.is_open()){
22        cout<<"Ошибка открытия файла!"<<endl;
23    }
24    //при успешном открытии файла
25    else{
26        cout<<"Файл успешно открыт!"<<endl;
27        //чтение размеров первого массива
28        file >> size_n1;
29        file >> size_m1;
30        //выделение динамической памяти, чтение данных из
31        //файла и заполнение первого массива
32        array1 = new int*[size_n1];
33        for (int i = 0; i < size_n1; i++){
34            array1[i] = new int[size_n1];
35            for (int j = 0; j < size_m1; j++)
36                file >> array1[i][j];
37        }
38        //чтение размеров второго массива
39        file >> size_n2;
40        file >> size_m2;
41        //выделение динамической памяти, чтение данных из
42        //файла и заполнение второго массива
43        array2 = new int*[size_n2];
44        for(int i = 0; i < size_n2; i++){
45            array2[i] = new int[size_n2];
46            for(int j = 0; j < size_m2; j++)
```

```
47              file >> array2[i][j];
48          }
49          //закрытие файла
50          file.close();
51          //вывод первого массива
52          cout<<"Размер матрицы:"<<size_n1<<"*"<<size_m1<<endl;
53          cout<<"Матрица 1 из файла: "<<endl;
54          //вывод первого массива в консоль
55          for (int i=0; i<size_n1; i++){
56              for(int j=0; j<size_m1; j++){
57                  cout<<setw(4)<<array1[i][j]<<" ";
58              }
59              cout<<endl;
60          }
61          //вывод второго массива
62          cout<<endl<<"Размер матрицы: "<<size_n2<<"*"<<size_m2
    <<endl;
63          cout<<"Матрица 2 из файла: "<<endl;
64          //вывод второго массива в консоль
65          for (int i=0; i<size_n2; i++){
66              for(int j=0; j<size_m2; j++){
67                  cout<<setw(4)<<array2[i][j];
68              }
69              cout<<endl;
70          }
71          //сложение матриц при совпадении размеров
72          if (size_n1==size_n2 && size_m1==size_m2){
73              cout<<endl<<"Сумма матриц: "<<endl;
74              for(int i=0; i<size_n1;i++){
75                  for(int j=0; j<size_m1;j++)
76                      cout<<setw(5)<<array1[i][j]+array2[i][j];
77                  cout<<endl;
78              }
79          }
80          //освобождение памяти (1 массив)
81          for(int i=0;i<size_n1;i++){
82              delete [] array1[i];
83          }
84          delete [] array1;
85          //освобождение памяти (2 массив)
86          for(int i=0;i<size_n2;i++){
87              delete [] array2[i];
```

```
88          }
89          delete [] array2;
90      }
91      return 0;
92  }
```

Решение данной задачи аналогично предыдущей за небольшим исключением. По условию задачи даны две прямоугольные матрицы, следовательно, заполнение массивов будет производится в два этапа. На первом этапе в переменные `size_n1` и `size_m1` считывается размер первой матрицы (строки 28 – 29), после чего производится динамическое выделение памяти под массив и его последующее заполнение (строки 32 – 37). После заполнения первого массива, на втором этапе, необходимо считать и записать в переменные размер второго массива, что реализовано в строках 39 – 40. После этого необходимо произвести те же действия, что и для первого массива: выделить динамическую память и заполнить массив (строки 43 – 48).

Перед сложением двух массивов необходимо удостоверится, что размеры этих массивов совпадают и только потом производить их сложение (строка 72). В условии задачи не сказано о том, что необходимо хранить сумму первого и второго массивов, поэтому их сумма просто выводится в консоль без создания дополнительного массива (строки 73 – 79).

Результат программы представлен на рисунке 30.

Задача 6.3 Написать программу, считывающую данные из двух файлов в массивы. Структура первого файла: в первой строке содержится размер вектора, затем следуют элементы вектора (вещественные числа). Структура второго файла: в первой строке содержится размер матрицы, затем следуют элементы массива (вещественные числа). Реализовать функцию, вычисляющую произведение матрицы на вектор и выводящую результат в консоль.

Решение задачи представлено в листинге 34.

```
Файл успешно открыт!
Размер матрицы: 3*4
Матрица 1 из файла:
  45   12   72   14
  31   89   11   19
  71   56   34   88

Размер матрицы: 3*4
Матрица 2 из файла:
  77  18   9   7
  13  14  -3   9
  14   2  10   1

Сумма матриц:
  122   30   81   21
   44  103    8   28
   85   58   44   89
```

Рис. 30. Результат работы программы из листинга 33

Листинг 34. Решение задачи 6.3

```
1 #include <iostream>
2 #include <fstream>
3 #include <iomanip>
4 using namespace std;
5
6 void mult(float**matrix,float*vect, int n, int m){
7     float rez = 0;
8     cout << "Результат умножения матрицы на вектор: \n";
9     for(int i = 0; i < n; i++){
10        for(int j = 0; j < m; j++){
11            rez += matrix[i][j]*vect[j];
12        }
13        cout << rez << "  ";
14        rez = 0;
15    }
16 }
17 int main(){
18    // размер вектора, указатель на массив
19    int n_vect;
20    float *vect;
21    // размер матрицы, указатель на указатель
```

```
22      int n_matrix, m_matrix;
23      float **matrix;
24      //чтение вектора из файла
25      ifstream my_vector("vect.txt");
26      if (!my_vector.is_open()){
27          cout << "Ошибка чтения файла!";
28      }
29      else{
30          my_vector >> n_vect;
31          cout << "Размерность вектора: " << n_vect<< "\n";
32          vect = new float[n_vect];
33          cout << "Исходный вектор: \n";
34          for(int i = 0; i<n_vect; i++){
35              my_vector >> vect[i];
36              cout << vect[i]<<"  ";
37          }
38          cout << endl;
39      }
40      my_vector.close();
41      //чтение матрицы из файла
42      ifstream my_matrix("matrix.txt");
43      if(!my_matrix.is_open()){
44          cout << "Ошибка чтения файла!";
45      }
46      else{
47          my_matrix >> n_matrix;
48          my_matrix >> m_matrix;
49          cout<< "\Размерностьn матрицы: "<<n_matrix<<"*"<<
        m_matrix<<endl;
50          cout << "Исходная матрица:\n";
51          matrix = new float*[n_matrix];
52          for(int i = 0; i<n_matrix; i++)
53              matrix[i] = new float[m_matrix];
54          for(int i = 0; i < n_matrix; i++){
55              for(int j = 0; j<m_matrix; j++){
56                  my_matrix >> matrix[i][j];
57                  cout<<right << setw(7)<<matrix[i][j];
58              }
59              cout<<endl;
60          }
61      }
62      my_matrix.close();
```

```
63      if(m_matrix == n_vect){
64          mult(matrix, vect, n_matrix, m_matrix);
65      }
66      //освобождение динамической памяти
67      delete [] vect;
68      for(int i = 0; i < n_matrix; i++){
69          delete [] matrix[i];
70      }
71      delete [] matrix;
72      cout << endl;
73      return 0;
74  }
```

Для записи данных из файла, содержащего элементы вектора создаются переменная целого типа для хранения размера вектора и одномерный динамический массив типа `float` для хранения элементов вектора. Создание переменных и считывание данных из первого файла производится в строках 19 – 20 и 25 – 40, все действия аналогичны предыдущим задачам.

Подобные действия проделываются и для работы со вторым файлом, однако здесь для работы с матрицей используется двухмерный динамический массив (строки 22 – 23, 42 – 62).

После считывания файлов и заполнения их данными массивов в 64 строке вызывается функция `mult`, осуществляющая умножение матрицы на вектор. Функция принимает четыре параметра: указатель на матрицу, указатель на вектор, количество строк и столбцов матрицы. Описание функции реализовано в строках 6 – 16, т.к. в условиях задачи не сказано, что функция должна возвращать полученный результат – она имеет тип `void`.

После завершения работы с массивами в строках 67 – 71 осуществляется освобождение динамической памяти.

Результат работы программы представлен на рисунке 31.

6.3 Запись в файл

Для записи данных в файл предназначен класс `ofstream`. Как и класс `ifstream` он имеет ряд методов для работы с фай-

```
Размерность вектора: 4
Исходный вектор:
11.1  4.2  5.11  6.9

Размерность матрицы: 3*4
Исходная матрица:
    1.7    4.8      4   8.25
  11.33      1   7.01   0.04
   9.22   7.99   1.99   0.74

Результат умножения матрицы на вектор:
116.395  166.06  151.175
```

Рис. 31. Результат работы программы из листинга 34

лами, основные из них представлены в Таблице 4.

Для изучения принципов работы с файлами, а именно записи данных в файл, рассмотрим решение задачи 6.4, представленное в листинге 35.

Таблица 4

Основные методы класса `ofstream`

Метод	Описание
open	Открывает файл для чтения
close	Закрывает файл
put	Записывает одиночный символ в файл
write	Записывает данное число байт из памяти в файл
seekp	Перемещает указатель позиционирования файла в заданное положение
tellp	Возвращает текущее значение указателя позиционирования файла

Задача 6.4. Написать программу, считывающую данные из файла в двухмерный массив целого типа размером $n \times n$. Размер массива содержится в первой строке файла. В полученном

массиве найти минимальный и максимальный элементы и записать их в файл «minmax.txt».

Листинг 35. Запись данных в файл

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

int main(){
    //локализация
    setlocale(LC_CTYPE, "");
    //указатель на массив
    int** array;
    int size;
    int min, max;
    //Создание объекта и открытие файла
    ifstream file("test.txt");
    //если файл не удалось открыть
    if (!file.is_open()){
      cout<<"Ошибка открытия файла!"<<endl;
    }
    else{
      //при успешном открытии файла
      cout<<"Файл успешно открыт!"<<endl;
      //чтение размера матрицы
      file >> size;
      //выделение динамической памяти
      array = new int*[size];
      for (int i = 0; i < size; i++){
          array[i] = new int[size];
          //чтение элементов и их запись в массив
          for (int j = 0; j < size; j++)
              file >> array[i][j];
      }
      min = array[0][0];
      max = array[0][0];
      cout<<"Размер матрицы: "<<size<<endl;
      cout<<"Матрица из файла: "<<endl;
      //вывод массива в консоль и
      //нахождение минимального элемента массива
      for (int i=0; i<size; i++){
          for(int j=0; j<size; j++){
```

```
39              cout<<array[i][j]<<" ";
40              if(array[i][j]<min){
41                  min=array[i][j];
42              }
43              if(array[i][j]>max){
44                  max = array[i][j];
45              }
46          }
47          cout<<endl;
48      }
49      //Создание объекта и открытие файла
50      ofstream outfile("minmax.txt");
51      //запись информации в файл
52      outfile<<"Минимальный элемент файла: "<<min<<endl
53      <<"Максимальный элемент файла: "<<max<<endl;
54      outfile.close();
55      //освобождение памяти
56      for(int i=0;i<size;i++){
57          delete [] array[i];
58      }
59      delete [] array;
60  }
61  //закрытие файла
62  file.close();
63  return 0;
64 }
```

Программный код, представленный выше, работает с двумя классами: `ifstream` и `ofstream`. В 13 строке создается объект `file` класса `ifstream`, файл связывается с объектом и открывается. Здесь, в отличие от листингов 32 и 33, использована компактная запись создания объекта и открытия файла. Процесс чтения и записи данных в переменные и массивы нам уже известен (строки 21 – 30), остановимся на записи данных в файл.

На первом этапе для записи данных в файл создается объект класса `ofstream`, после чего созданный объект связывается с файлом (если файл с указанным именем не существует, он будет создан автоматически) и файл открывается для записи. Данный этап реализован в 50 строке программы. На втором этапе

найденные минимальный и максимальный элементы, по условию задачи, необходимо записать в файл «`minmax.txt`». Запись осуществляется аналогично выводу информации в консоль, только вместо `cout` при записи данных в файл указывается имя объекта класса `ofstream` (строки 52 – 53). После завершения работы с файлом необходимо его закрытие (строка 57).

В результате работы программы в файл (существующий либо созданный в результате работы программы) `minmax.txt` будет записана информация о максимальном и минимальном элементах массива. Примерное содержание файла (в зависимости от данных исходного файла) представлено ниже:

```
Минимальный элемент файла: 11
Максимальный элемент файла: 89
```

Задача 6.5. Написать программу, записывающую в файл квадратную матрицу и ее размер. Элементы матрицы – случайные числа в диапазоне от 1 до 10 включительно.

Решение задачи 6.5. представлено в листинге 36.

Листинг 36. Запись данных в виде матрицы в файл

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main(){
6     //локализация
7     setlocale(LC_CTYPE, "");
8     //размер матрицы
9     int size;
10    cout<<"Введите размер матрицы: "; cin >> size;
11    //создание и открытие объекта для записи в файл
12    ofstream mymatr("my_matr.txt");
13    //запись размера массива в файд
14    mymatr << size <<endl;
15    //заполнение случайных чисел в файл в виде матрицы
16    for(int i=0; i < size; i++){
17        for(int j=0; j<size; j++){
```

```
18              mymatr << 1 + rand() % 10 <<" ";
19          }
20          mymatr << endl;
21      }
22      //закрытие файла
23      mymatr.close();
24      return 0;
25  }
```

По аналогии со считыванием информации из файла, сначала создается объект класса, который будет связан с файлом для дальнейшей работы. Создание объекта класса `ofstream` и открытие файла реализовано в строке 12. Предварительно уже была создана переменная `size`, в которую записан размер матрицы, указанный пользователем в консоли (строки 9 – 10). Значение переменной `size` записывается в первую строку файла (строка 14 программного кода), после чего в виде матрицы в файл записываются случайные числа в диапазоне от 1 до 10 (строки 16 – 21). Синтаксис аналогичен функции `cout`, только вместо данной функции указывается имя объекта `mymatr` класса `ofstream`.

В результате работы программы в корневой папке проекта создается файл `my_matr.txt`, который в первой строке содержит размер квадратной матрицы, а в последующих – значения ее элементов.

6.4 Бинарные файлы

Процесс работы с бинарными файлами аналогичен работе с обычными текстовыми файлами. Однако в отличие от текстовых файлов в данном режиме работа идет непосредственно с битами, следовательно, возможна запись и считывание специальных символов (управляющих последовательностей). Кроме того, работая с бинарными файлами имеется возможность произвольного доступа к данным, т.е. по указателю можно начинать работу с определенной позиции в файле.

Для того, чтобы начать работу с файлом в двоичном режи-

ме необходимо на этапе открытия файла указать режим работы `binary`.

Общий синтаксис открытия файла в бинарном режиме для записи:

`ofstream имя_о("имя_ф", ios::binary|ios::out);`

где `имя_о` – имя объекта, `имя_ф` – имя файла.

Общий синтаксис открытия файла только для чтения в бинарном режиме:

`ifstream имя_о("имя_ф", ios::binary|ios::ins);`

где `имя_о` – имя объекта, `имя_ф` – имя файла.

Рассмотрим работу с бинарными файлами на примере решения задачи 6.6. Решение данной задачи представлено в листинге 37.

Задача 6.6 Написать программу, создающую бинарный файл и записывающую в него произвольную строку. Считать информацию из созданного файла и вывести ее в консоль.

Листинг 37. Работа с бинарными файлами

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
    string str;
    //открытие файла в двоичном режиме для записи
    ofstream myfile("myfile.txt", ios::binary|ios::out);
    if(!myfile.is_open()){
        cout<<"Не удалось открыть файл!";
    }
    else{
        cout<<"Введите информацию для записи в файл: ";
        getline(cin, str);
        myfile.write((char*)&str, sizeof(str));
    }
```

```
18      myfile.close();
19      //открытие файда в двоичном режиме для чтения
20      ifstream myfilein("myfile.txt", ios::binary|ios::in);
21      if(!myfilein.is_open()){
22          cout<<"Не удалось открыть файл!";
23      }
24      else{
25          cout<<"Информация из файла: \n";
26          myfilein.read((char*)&str, sizeof(str));
27          cout<<str;
28      }
29      myfilein.close();
30
31      return 0;
32  }
```

В данном программном коде создается и связывается объект `myfile` для работы с файлом (запись в файл) в двоичном режиме (строка 9). На то, что работа с файлом будет производится в двоичном режиме указывает параметр `ios::binary`. Далее производится стандартная проверка открытия файла, если открытие файла прошло успешно, будет произведена запись информации, введенной пользователем с консоли, в файл (строки 14 – 16). Для записи всей строки, включая пробелы используется функция `getline`(строка 15), введенная строка записывается в переменную `str`. Для записи данных из переменной `str` в файл используется метод `write`, который принимает два параметра: указатель на тип `char` и количество записываемых байт (для этого используется функция `sizeof`). Таким образом данные, хранящиеся в переменной `str` преобразуются в указатель `char` и побайтово записываются в файл (строка 16).

Для чтения из файла производятся действия, аналогичные действиям записи в файл: сначала создается объект для работы с файлом с указанием двоичного режима работы, после чего производится проверка на успешное открытие файла. При успешном открытии файла с помощью метода `read` данные побайтово считываются и записываются в переменную `str`, после чего вы-

водятся в консоль стандартным образом.

Результат работы программы представлен на рисунке 32.

```
Введите информацию для записи в файл:
Рукописи не горят
Информация из файла:
Рукописи не горят
```

Рис. 32. Результат работы программы из листинга 37

Задача 6.7 Написать программу, реализующую класс «Точка» с полями x, y. Записать в бинарный файл объект класса «Точка». Считать из созданного файла данные и вывести в консоль.

Листинг 38. Работа с файлами и объектами

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4 class Point{
5     double x, y;
6 public:
7     Point();
8     Point(double x, double y);
9     void point_print();
10 };
11
12 int main(){
13     //создание объекта класса
14     Point mypoint(4,4);
15     //открытие файла в двоичном режиме работы для записи
16     ofstream myfileout("test.txt", ios::binary|ios::out);
17     if(!myfileout.is_open()){
18         cout<<"Не удалось открыть файл!";
19     }
20     //запись данных в файл при успешном открытии
21     else{
22         myfileout.write((char*)&mypoint, sizeof(Point));
23     }
24     myfileout.close();
```

```
25      //открытие файла в двоичном режиме работы для чтения
26      ifstream myfilein("test.txt", ios::binary|ios::in);
27      if(!myfilein.is_open()){
28          cout<<"Не удалось открыть файл!";
29      }
30      else{
31      //чтение данных из файла при успешном открытии
32          cout<<"Файл успешно открыт! \n";
33          Point point;
34          myfilein.read((char*)&point, sizeof(Point));
35          point.point_print();
36      }
37      myfilein.close();
38      return 0;
39  }
40  //методы класса Point
41  Point::Point(){
42      x = y = 0;
43  }
44  Point::Point(double x = 0, double y = 0){
45      this->x = x;
46      this->y = y;
47  }
48  void Point::point_print(){
49      cout<<"X = "<<x<<"\t Y = "<<y<<endl;
50  }
```

Запись объекта в бинарный файл практически полностью совпадает с записью данных, хранимых в переменных стандартного типа. Отличие заключается лишь в передаваемых параметрах методу `write`. В данном случае методу передается объект класса, а функции `sizeof` – тип объекта (т.е. имя класса).

Чтение данных выполняется с помощью функции `read`, ее параметры полностью совпадают с параметрами функции `write`. Данные записываются в объект класса `Point`, после чего выводятся в консоль с помощью метода `point_print`.

Примечание: если открыть бинарный файл после записи в него каких-либо данных можно заметить, что файл содержит не обычную текстовую информацию, а набор «непонятных симво-

лов». Однако после правильного чтения файла программой можно удостоверится, что записанные данные совпадают с теми, которые вводились в программе.

Результат работы программы представлен на рисунке 33.

```
Файл успешно открыт!
X = 4    Y = 4
```

Рис. 33. Результат работы программы из листинга 38

6.5 Лабораторная работа №4 «Работа с файлами»

Цель работы: приобретение навыков программирования в работе с файлами.

Задание. Написать программу решения задачи согласно своему варианту на языке программирования C++.

Вариант 1

Создать текстовый файл и заполнить его случайными числами в диапазоне от -17 до 19. Найти суммы положительных и отрицательных чисел. Сохранить полученные результаты в новый текстовый файл.

Вариант 2

Создать текстовый файл и заполнить его произвольной информацией. Найти в данном файле и сохранить в новый файл слова, содержащие букву «к».

Вариант 3

Создать текстовый файл и заполнить его случайными числами. Сохранить в новый файл числа из исходного файла в порядке возрастания.

Вариант 4

Создать текстовый файл и записать в него числовой массив размером $3 * 3$. Вычислить и записать в этот же файл определитель данной матрицы.

Вариант 5

Создать текстовый файл записать в него квадратную матрицу и ее размер (размер матрицы вводится пользователем с клавиатуры). Сохранить в новый файл транспонированную матрицу.

Вариант 6

Создать текстовый файл и записать в него произвольную информацию. В новый файл сохранить только четные слова исходного файла.

Вариант 7

Создать текстовый файл и записать в него произвольную информацию. В новый файл сохранить слова исходного файла, длина которых более четырех символов.

Вариант 8

Создать текстовый файл и записать в него произвольную информацию. В новый файл сохранить информацию о количестве слов исходного файла, содержащих букву «о».

Вариант 9

Создать текстовый файл и записать в него случайные числа. В новый файл сохранить те числа исходного файла, которые являются целыми степенями числа два.

Вариант 10

Создать текстовый файл и записать в него две матрицы и их размер $n \times n$. В новый текстовый файл записать результат произведения данных матриц.

Вариант 11

Создать текстовый файл и записать в него произвольные числа. В новый файл сохранить сумму четных чисел и произведение нечетных, а также их количество.

Вариант 12

Создать текстовый файл и записать в него произвольную информацию. В новый файл из исходного сохранить цитаты.

Вариант 13

Создать текстовый файл и записать в него произвольную информацию. Изменить в данном файле все гласные буквы на символ «%».

Вариант 14

Создать текстовый файл и записать в него произвольную информацию. В новый файл записать информацию о самом длинном слове исходного файла: само слово и его номер.

Вариант 15

Создать текстовый файл и записать в него количество координат векторов, а также сами координаты двух векторов. Вычислить и записать в этот же файл скалярное произведение данных векторов.

Вариант 16

Создать файл и записать в него координаты трех точек (x, y). Вычислить и записать в новый файл расстояние между данными точками.

Вариант 17

Создать файл и записать в него n случайных чисел. В новый файл записать из исходного файла все возрастающие последовательности.

Вариант 18

Создать файл и записать в него размер квадратной матрицы и ее элементы. Посчитать и записать в новый файл сумму элементов матрицы исходного файла, стоящих выше главной диагонали.

Вариант 19

Создать файл и записать в него размер квадратной матрицы и ее элементы. Посчитать и записать в новый файл удвоенную сумму элементов матрицы исходного файла, стоящих ниже главной диагонали.

Вариант 20

Создать файл и записать в него размер квадратной матрицы и ее элементы. Посчитать и записать в новый файл корень суммы элементов матрицы исходного файла, стоящих ниже главной и побочной диагоналей.

Вариант 21

Создать файл и записать в него размер квадратной матрицы и ее элементы. Посчитать и записать в новый файл квадрат суммы элементов матрицы исходного файла, стоящих выше главной и побочной диагоналей.

Вариант 22

Создать файл и записать в него размер квадратной матрицы и ее элементы. Посчитать и записать в новый файл сумму элементов матрицы исходного файла, стоящих левее главной и побочной диагоналей.

Вариант 23

Создать файл и записать в него размер квадратной матрицы и ее элементы. Записать в новый файл элементы матрицы исходного файла, стоящие правее главной и побочной диагоналей в порядке возрастания.

Вариант 24

Создать файл и записать в него размер квадратной матрицы и ее элементы. Записать в новый файл сумму элементов главной диагонали и корень суммы элементов побочной диагонали матрицы исходного файла.

Вариант 25

Создать файл и записать в него размер квадратной матрицы и ее элементы. Записать в новый файл сумму элементов матрицы исходного файла, стоящих выше побочной диагонали.

7 Визуализация данных

7.1 Знакомство с Gnuplot

При работе с данными зачастую необходима их визуализация. Мощным и простым средством визуализации данных является портативная графическая утилита Gnuplot. Данная утилита легко интегрируется в программный код современных языков программирования и позволяет представлять в виде графиков любые текстово-табличные данные.

Работа с Gnuplot возможна в двух режимах: из командной строки и программного кода. Команды в обеих режимах работы совпадают. В рамках данного курса рассмотрим возможности и принципы работы утилиты в режиме работы из программного кода.

Примечание: все примеры программ в листингах реализованы для операционной системы Ubuntu. В зависимости от операционной системы некоторые функции и/или их использование могут различаться (например, открытие канала).

Для работы с gnuplot в C++ будем использовать следующий шаблон:

Листинг 39. Минимальный шаблон для работы с gnuplot на языке C++

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7     // gp - указатель на FILE
8     FILE *gp = popen("gnuplot -persist","w");
9     // при неуспешном открытии канала
10    if (gp == NULL)
11    {
12        printf("Ошибка при открытии канала для Gnuplot.\n");
13        exit(0);
14    }// при успешном открытии канала
```

```
15      ...
16      fprintf(gp, "команды gnuplot");
17      //закрытие канала
18      pclose(gp);
19
20      return 0;
21  }
```

Вся работа с утилитой будет выполняется через файловую переменную – указатель на структуру типа FILE, определённую в стандартной библиотеке. Для работы с gnuplot из программного кода необходимо открытие канала. Каналы – это средство передачи данных между процессами. Для открытия канала используется функция `popen()`, канал перенаправит вывод команд на вход gnuplot (что и указано в качестве аргумента функции). Параметр «`-persist`» необходим для задержки окна gnuplot после окончания построения графика, если опустить данный параметр окно сразу закроется после построения графика. Параметр «`w`» указывает на то, что работа с каналом будет происходить в режиме записи (т.е. в канал будут передаваться какие-либо данные (или команды), но не будут обратного сообщения).

Для закрытия канала используется функция `pclose()`. Вывод графика осуществляется с помощью функции `fprint()`, которая принимает в качестве аргументов два параметра – дескриптор канала и строку, состоящую из команд gnuplot. Команды gnuplot в строке разделяется между собой символом перевода на новую строку (`\n`).

Вычерчивание графика в gnuplot производится командой `plot` или `splot` (двухмерные и трехмерные графики соответственно), после которой указывается функция, график которой необходимо построить, либо файл, содержимое которого требуется визуализировать, после чего следует некоторый набор параметров вывода. Если необходимо построение нескольких графиков, тогда после ключевого слова `plot` перечисляются функции и/или файлы через запятую.

Наиболее часто используемые команды утилиты gnuplot

приведены в Таблице 5.

Таблица 5

Команды для работы с утилитой gnuplot

Команды	Назначение
set grid	отображение координатной сетки
set size square	установка одинакового масштаба по осям
set xrange $[x_n : x_k]$	установка диапазона по оси абсцисс
set yrange $[x_n : x_k]$	установка диапазона по оси ординат
plot 'file_name.dat' with lines	построение графика линией
plot 'file_name.dat' with l	построение графика линией
plot 'file_name.dat' w l	построение графика линией
plot 'file_name.dat' with l lw 2	построение графика линией удвоенной толщины
plot 'file.dat' with linespoint pt 9 ps 1	построение графика линией с распределенными по ней значками (треугольники)
plot 'file.dat' w points pt 9 ps 1	построение графика треугольниками

После ключевого слова `width` указывается стиль, по которому будет строится график:

1. `lines` — линия, после которой указываются опции `lt` (linetype) и `lw` (linewidth) соответственно;

2. `points` — значок, опции `pt` (pointtype) и `ps` (pointsize);

3. `linespoint` — линия, на которой распределены значки, опции `lt, lw, pt и ps`;

4. `dots` — точки;

5. `impulses` — отрезки ординат от оси абсцисс.

7.2 Примеры решения задач

Задача 7.1 Дан файл plot.dat содержащий некоторые данные. Визуализировать данные из предоставленного файла.

Листинг 40. Визуализация данных из файла

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     //открытие канала
6      FILE *gp = popen("gnuplot -persist", "w");
7
8     if (gp == NULL)
9     {
10         printf("Ошибка при открытии канала для Gnuplot.\n");
11         exit(0);
12    } else {
13         fprintf(gp, "set grid \n plot 'plot.dat' with line
      \n");
14         pclose(gp);
15    }
16    return 0;
17 }
```

Рассмотрим подробнее программный код листинга 40. В 6 строке функцией `popen` открывается канал с именем `gp`. Если открытие канала произошло успешно, тогда в 13 строке с помощью функции `fprint`, данные из файла будут визуализированы. Функция `fprint` принимает два параметра: дескриптор канала и строку, содержащую команду, которая передается утилите gnuplot. В данной строке содержится: команда `grid` для отображения сетки, команда `plot` для построения графика, имя файла (из которого необходимо взять данные) и стиль черчения (линия). Обратим внимание, что все команды в 14 строке разделены символом `\n`. В 15 строке функцией `pclose()` закрывается ранее открытый канал.

При возникновении ошибки в открытии канала выведется соответствующее сообщение (строка 10).

Результат работы программы представлен на рисунке 34.

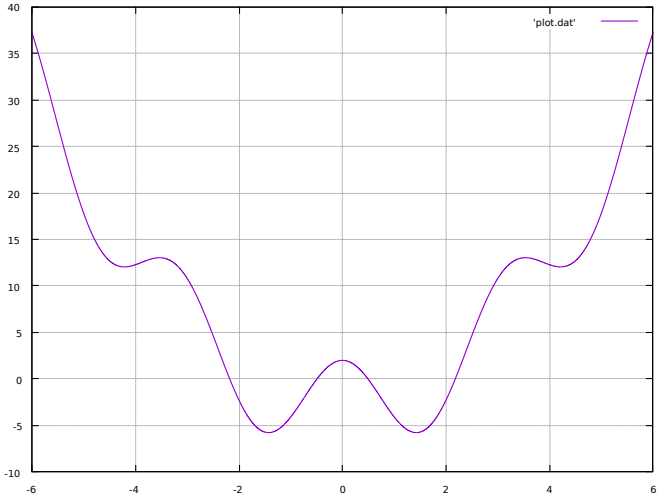

Рис. 34. Результат работы программы из листинга 40

Задача 7.1 Написать программу, записывающую в файлы данные в виде таблицы.

Файл 1:

1. Номер строки

2. Значение функции $y = x^2 + x$.

3. Значение переменной x на отрезке [-10;10] включительно с шагом 0,02.

Файл 2:

1. Номер строки.

2. Значение функции $y = cos(x)$.

3. Значение переменной x на отрезке [-10.0; 12.3] включительно с шагом 0,01.

Визуализировать данные из обоих файлов в следующих диапазонах: $x[-6; 5], y[-20; 25]$.

Листинг 41. Решение задачи 7.1

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

int main(){
    float y, j = 1;
    //создание первого файла и запись информации в него
    ofstream file1("parab.dat");
    if(!file1.is_open()){
        cout<<"Не удалось открыть или создать файл.\n";
    }
    else{
        file1 << "# №\t\t" << "#x\t\t" << "#y(x) = \n";
        for(float x = -10; x <= 10; x+=0.02){
            file1 << j << "\t\t"; j++;
            file1 << setprecision(4) <<x <<"\t\t";
            y = x*x - 3*x;
            file1 << setprecision(4) <<y << "\n";
        }
    }
    file1.close();
    //создание второго файла и запись информации в него
    ofstream file2("sin.dat");
    if(!file2.is_open()){
        cout<<"Не удалось открыть или создать файл.\n";
    }
    else{
        j = 1;
        file2 << "№#\t\t" << "#x\t\t" << "#y(x) = \n";
        for(float x = -10.0; x <= 12.3; x+=0.01){
            file2 << j << "\t\t"; j++;
            file2 << setprecision(4) << x <<"\t\t";
            y = 15*cos(x);
            file2 << setprecision(4) << y << "\n";
        }
    }
    file2.close();
    //визуализация данных
    FILE *gp = popen("gnuplot -persist", "w");
```

```
42      if (gp == NULL)
43      {
44          printf("Ошибка при открытии канала для Gnuplot.\n");
45          exit(0);
46      }
47      fprintf(gp, "set grid \n set size square\n plot
        [-6:5][-20:25] 'parab.dat' using 2:3 with line, \'sin.
        dat' using 2:3 with line \n");
48      pclose(gp);
49      return 0;
50  }
```

Рассмотрим программный код листинга, представленного выше. Для записи данных в первый файл сначала создается объект типа `ofstream`, далее производится открытие файла. При успешном открытии файла в цикле по заданной формуле производится его заполнение полученными значениями (строки 10 – 22). Обратим внимание на строку 15, в которой произведена запись первой строки в файл, все данные вводятся через символ #, данный символ служит комментарием при работе с утилитой gnuplot, поэтому не будет возникать ошибок при визуализации данных (несмотря на присутствие нечисловых данных) из файла. После заполнения файла с помощью функции `close()` он закрывается (строка 23). Аналогичные действия производятся для второго файла (строки 25 – 37). Для округления полученных результатов в обоих случаях используется функция `setprecision` (*Примечание*: стоит учитывать, что функция принимает общее количество цифр, как до, так и после запятой, сама запятая не учитывается).

Для визуализации данных сначала стандартным образом открывается канал (строка 41), при успешном открытии канала в функцию `fprintf()` передаются два параметра: дескриптор канала и строка, содержащая ряд команд, передаваемых утилите gnuplot. Параметр `set grid` необходим для отображения сетки, параметр `set size square` используется для установки одинакового масштаба по осям, далее, после команды `plot` указываются отрезки, на которых необходимо отобразить данные (сначала для

оси абсцисс, затем ординат), после чего следуют названия файлов и номера столбцов (`usinng 2:3`), по данным из которых и строятся графики.

Из результата работы программы (рисунок 34) видно, что построились два графика на заданных отрезках по осям абсцисс и ординат. На рисунке видно, что присутствует легенда, в которой отражено, из какого файла выводился график, и данные из каких столбцов были визуализированы.

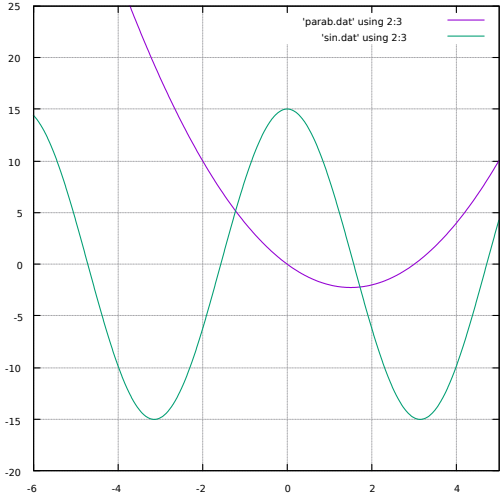

Рис. 35. Результат работы программы из листинга 41

8 Стандартная библиотека шаблонов

В языке C++ существует стандартная библиотека шаблонов (STL), которая представляет различные типобезопасные контейнеры для хранения коллекций связанных объектов.

Контейнеры — это классы библиотеки STL, предназначенные для хранения коллекции однотипных объектов данных. Все контейнеры в данной библиотеке можно разделить на три категории: последовательные контейнеры, ассоциативные контейнеры и контейнеры адаптеры.

Последовательные контейнеры — это контейнеры, которые поддерживают порядок вставляемых элементов, указанный пользователем. **Ассоциативные контейнеры** — это контейнеры в которых элементы выставляются в предварительно определенном порядке, например, с сортировкой по возрастанию. **Контейнеры-адаптеры** являются разновидностью последовательного или ассоциативного контейнера, который ограничивает интерфейс для простоты и ясности.

Для работы с контейнерами используются **итераторы** — интерфейсы, предназначенные для взаимодействия с элементами контейнера.

В данной главе будет рассмотрено четыре STL-контейнера: vector, list, stack и deque.

8.1 Класс-контейнер vector

Класс vector — это структура данных, которая является моделью динамического массива. Размер такого массива по мере необходимости может быть изменен. Доступ к элементам класса осуществляется с помощью квадратных скобок.

Для использования данного класса и его методов необходимо подключение заголовочного файла `<vector>` и пространства имен `std`.

Общий синтаксис объявления вектора (vector):

```
vector <тип_данных> имя_вектора;
```

Вектор может быть заполнен еще на этапе объявления, делается это аналогично заполнению массива на этапе его создания, т.е. в фигурных скобках через запятую указываются значения элементов. Кроме того, на этапе создания вектора возможно указание количества его ячеек в круглых скобках.

Рассмотрим программный код из листинга 42, в котором представлены различные способы создания и заполнения массивов типа `vector`.

Листинг 42. Содание массивов типа vector

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 int main(){
7     //создание пустого вектора
8     vector <int> my_array_1;
9
10    //создание вектора из 5 элементов
11    vector <string> my_array_2(5);
12
13    //создание вектора из 7 элементов
14    vector <int> my_array_3(7, 2);
15
16    //создание вектора на основании другого
17    vector <int> my_array_4(my_array_3);
18
19    //создание вектора с его заполнением
20    vector <float> my_array_5 = {1.1, 2.4, 7.0, 4.11};
21
22    //обращение к элементу вектора
23    cout<<my_array_5[0];
24    return 0;
25 }
```

В строке 8 создается пустой вектор, элементы которого имеют тип `int`, в 11 строке — вектор из 5 элементов типа `string`. Обратим внимание на строку 14, в ней создается вектор из 7 элементов типа `int` и в качестве его значений записывается целое

число два. В 17 строке на основании вектора `my_array_3` создается новый вектор `my_array_4`. В строке 20 создается и заполняется вектор, элементами которого являются вещественные числа. Для обращения к элементу вектора используются квадратные скобки — операция индексирования (по аналогии с обычными массивами), что и отражено в 23 строке программного кода.

Для работы с векторами существует ряд методов, в Таблице 6 представлены основные из них.

Таблица 6

Методы класса vector

Метод	Назначение
size()	Определение размера вектора
empty()	Определить, пустой ли вектор
reserve()	Зарезервировать память для дополнительных элементов массива
resize()	Изменить размер массива
push_back()	Добавление элемента в конец вектора
pop_back()	Удаление последнего элемента вектора
clear()	Удаление всех элементов из массива
swap()	Обмен местами двух векторов
erase()	Удаление одного или нескольких элементов заданного диапазона
insert()	Вставка элемента или группы элементов в вектор
at()	Получение элемента вектора по его позиции
front()	Ссылка на первый элемент вектора
back()	Ссылка на последний элемент вектора

8.2 Класс-контейнер stack

Стек (stack) — это динамическая структура данных, работающая по принципу «первый пришел — последний ушел». В стеке добавление новых элементов и удаление существующих производится с одного конца, который называется **вершиной стека**.

Для использования данного класса и его методов необходимо подключение заголовочного файла `stack`.

Общий синтаксис объявления стека:
`stack <тип_данных> имя_стека;`

Для работы со стеком предусмотрен ряд методов, которые представлены в Таблице 7.

Таблица 7

Методы класса stack

Метод	Назначение
size()	Определение размера стека
empty()	Определить, пустой ли стек
push()	Добавление элемента в стек
pop()	Удаление верхнего элемента стека
top()	Получение верхнего элемента стека

Рассмотрим программный код из листинга 43, в котором продемонстрирована работа с классом-контейнером `stack`.

Листинг 43. Пример создания стека. Использование некоторых методов стека

```
1 #include <iostream>
2 #include <stack>
3 using namespace std;
4
5 int main(){
6     int n, s;
```

```
7       //создание стека
8       stack <int> my_stack;
9       cout<<"Введите предполагаемый размер стека: ";
10      cin>>n;
11      //заполение стека
12      for(int i = 0; i<n; i++){
13          cout<<"Введите значение элемента стека: ";
14          cin>>s;
15          my_stack.push(s);
16      }
17      //определение вершины стека
18      cout<<"Значение элемента вершины стека: "
19     <<my_stack.top()<<endl;
20      //проверка, пустой ли стек
21      cout<<"Стек пустой?: "<<my_stack.empty()<<endl;
22      //удаление верхнего элемента стека
23      my_stack.pop();
24      //добавление элемента стека
25      my_stack.push(4);
26      my_stack.push(7);
27      //определение размера стека
28      cout<<"Размер стека: "<<my_stack.size()<<endl;
29      return 0;
30  }
```

В строке 8 создается стек, элементы которого имеют тип `int`. Первоначальный размер стека определяется пользователем (строка 10), после чего производится его заполнение с помощью цикла и метода `push()` (строки 12 – 16). В строках 18 – 19 с помощью метода `top()` определяется значение верхнего элемента стека. В строке 21 вызывается метод `empty()`, который определяет, является ли стек пустым (возвращает 1, если стек пуст, 0 – в обратном случае). В 23 строке с помощью метода `pop()` удаляется верхний элемент стека, а в строках 25 и 26 с помощью метода `push` добавляются два новых элемента со значениями, указанными в скобках. Для определения размера стека в 28 строке использован метод `size()`. С результатом работы программы можно ознакомиться на рисунке 36.

```
Введите предполагаемый размер стека: 4
Введите значение элемента стека: 4
Введите значение элемента стека: 3
Введите значение элемента стека: 5
Введите значение элемента стека: 6
Значение элемента вершины стека: 6
Стек пустой?: 0
Размер стека: 5
```

Рис. 36. Результат работы программы из листинга 43

8.3 Класс-контейнер deque

Дек (deque) — это структура данных, позволяющая добавлять элементы как в начало, так и в конец контейнера, удалять их, обращаться по индексу и изменять. Дек – это двусторонняя очередь. Контейнер дек очень похож на класс-контейнер вектор (`vector`). Разница заключается в том, что в отличие от вектора доступ к элементам дека открыт с двух сторон.

Для использования данного класса-контейнера и его методов необходимо подключение заголовочного файла `deque`.

Общий синтаксис объявления дека:
`deque <тип_данных> имя_дека;`

Интерфейс дека почти полностью совпадает с интерфейсом класса-контейнера вектор. В Таблице 8 представлены основные методы класса-контейнера deque.

8.4 Класс-контейнер list

Список (list) — это структура данных, которая построена на двусвязных списках. Это означает, что каждый элемент «знает» только о предыдущем и следующем. В классе-контейнере `list`, в отличие от других контейнеров, не определена операция обращения по индексу.

Таблица 8

Методы класса deque

Метод	Назначение
size()	Определение размера дека
empty()	Определить, пустой ли дек
resize()	Изменить размер дека
push_back()	Добавление элемента в конец дека
push_front()	Добавление элемента в начало дека
pop_back()	Удаление последнего элемента дека
pop_front()	Удаление первого элемента дека
clear()	Удаление всех элементов контейнера
swap()	Обмен элементами между двумя объектами
erase()	Удаление одного или нескольких элементов заданного диапазона
insert()	Вставка элемента или группы элементов в дек
at()	Получение элемента дека по его позиции
front()	Ссылка на первый элемент дека
back()	Ссылка на последний элемент дека

Для использования данного класса-контейнера и его методов необходимо подключение заголовочного файла `<list>`.

Общий синтаксис объявления списка:

```
list <тип_данных> имя_списка;
```

Для работы со списками предусмотрен ряд методов, которые представлены в Таблице 9.

Основным преимуществом данного класса-контейнера является быстрое удаление и добавление ячеек, кроме того, доступ к ячейкам открыт с двух сторон списка.

Примечание. Для обращения к элементам, находящимся в середине списка необходимо использование циклов или итерато-

Таблица 9

Методы класса list

Метод	Назначение
size()	Определение количества элементов в списке
empty()	Проверяет, пуст ли список
resize()	Изменение размера списка
remove()	Стирание элементов в списке, которые соответствуют указанному значению
push_back()	Добавление элемента в конец списка
push_front()	Добавление элемента в начало списка
pop_back()	Удаление последнего элемента списка
pop_front()	Удаление первого элемента списка
clear()	Удаление всех элементов списка
swap()	Обмен элементами между двумя объектами
erase()	Удаление одного или нескольких элементов заданного диапазона
insert()	Вставка элемента или группы элементов в список
front()	Ссылка на первый элемент списка
back()	Ссылка на последний элемент списка

ров, что существенно замедляет работу программы. Если необходимо многократное обращение к элементам, оптимальным вариантом будет использование класса-контейнера `vector`.

Литература

[1] *Селезнева А. В.* Основы программирования. Язык выского уровня C++. Часть I: учебное пособие. — Издательский центр «Удмуртский университет»: Ижевск, 2022. — 199 pp. 5

[2] *Довбуш Г. Ф., Хомоненко А. Д.* Visual C++ на примерах. — СПБ.: БХВ-Петербург, 2007. — 528 pp. 6, 10, 17, 29, 38

[3] *Павловская Т. А., Щупак Ю. А.* C/C++. Структурное и объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2011. — 352 pp. 7

[4] *Лафоре Р.* Объектно-ориентированное программирование в C++. — М.[и др.]:Питер, 2003. — 923 pp. 11

[5] *Страуструп Б.* Язык программирования C++. Краткий курс, 2-е изд. — Пер. с англ.–СПб.: ООО «Диалектика», 2019. — 320 pp. 13

[6] *Страуструп Б.* Программирование: принципы и практика с использованием C++. — Пер. с англ. – М. : ООО «И .Д. Вильямс», 2016. — 1328 pp. 13

[7] *Павловская Т. А.* C/C++. Программирование на языке высокого уровня. — СПб.: Издательство Питер, 2003. — 461 pp. 13, 39, 72

[8] *Васильев А. Н.* Самоучитель C++ с примерами и задачами. 4-е издание (переработанное). — СПб.: Наука и Техника, 2016. — 480 pp. 17, 22, 32

Учебное издание

Селезнева Анна Викторовна

ОСНОВЫ ПРОГРАММИРОВАНИЯ
Язык высокого уровня С++

Часть II

Учебное пособие

Авторская редакция

Подписано в печать 02.05.2023. Формат 60x84 1/16.
Усл. печ. л. 8,78. Уч. изд. л.7,96.
Тираж 33 экз. Заказ № 782.

Издательский центр «Удмуртский университет»
426034, Ижевск, ул. Ломоносова, 4Б, каб. 021
Тел.: + 7 (3412) 916-364, E-mail: editorial@udsu.ru

Типография Издательского центра «Удмуртский университет»
426034, Ижевск, ул. Университетская, 1, корп. 2.
Тел. 68-57-18