



**К.И. ЗЫРЯНОВ**

**Н.П. КИСЛЕНКО**

# **ПРОГРАММИРОВАНИЕ НА C++**

НОВОСИБИРСК 2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ  
АРХИТЕКТУРНО-СТРОИТЕЛЬНЫЙ УНИВЕРСИТЕТ (СИБСТРИН)

**К.И. Зырянов, Н.П. Кисленко**

# **ПРОГРАММИРОВАНИЕ НА C++**

Учебное пособие

НОВОСИБИРСК 2017

УДК 004.43  
ББК 32.973.26-018.1  
3-976

***Зырянов, К. И.***

Программирование на C++ : учеб. пособие / К. И. Зырянов, Н. П. Кисленко ; Новосиб. гос. архитектур.-строит. ун-т (Сибстрин). – Новосибирск : НГАСУ (Сибстрин), 2017. – 128 с.

**ISBN 978-5-7795-0817-9**

В учебном пособии рассматриваются основы языка C++ и методы процедурного программирования на этом языке.

Пособие соответствует программе дисциплины "Языки программирования", читаемой во 2–3 семестрах студентам, обучающимся по направлению подготовки 09.03.02 "Информационные системы и технологии". Также оно будет полезным для всех студентов и специалистов, интересующихся современным программированием.

Рецензенты:

- И.А. Бедарев, канд. физ.-мат. наук,  
ст. науч. сотрудник ИТПМ СО РАН;
- А.Ф. Задорожный, канд. техн. наук,  
доцент, зав. кафедрой ИСТ  
НГАСУ (Сибстрин)

**ISBN 978-5-7795-0817-9** © Зырянов К.И., Кисленко Н.П., 2017  
© Новосибирский государственный  
архитектурно-строительный  
университет (Сибстрин), 2017

## Оглавление

|                                                                                                                           |     |
|---------------------------------------------------------------------------------------------------------------------------|-----|
| Введение.....                                                                                                             | 4   |
| Глава 1. Введение в C++. Линейный алгоритм<br>и простейшее приложение.....                                                | 5   |
| Глава 2. Разветвляющийся вычислительный процесс<br>и условный оператор.....                                               | 16  |
| Глава 3. Ввод и вывод в библиотеке <code>stdio.h</code> .<br>Оператор цикла. Циклы с предусловием<br>и постусловием ..... | 26  |
| Глава 4. Массивы. Описание, ввод, вывод<br>и обработка массивов .....                                                     | 36  |
| Глава 5. Функции .....                                                                                                    | 44  |
| Глава 6. Указатели и строки <code>Сi</code> .....                                                                         | 50  |
| Глава 7. Строки класса <code>string</code> .....                                                                          | 60  |
| Глава 8. Работа с текстовыми и бинарными файлами .....                                                                    | 68  |
| Глава 9. Работа с файлами с помощью потоков<br>и классов ввода/вывода .....                                               | 81  |
| Глава 10. Составные типы данных (структуры).....                                                                          | 92  |
| Глава 11. Составные типы данных (перечисления,<br>объединения, структуры структур).....                                   | 100 |
| Глава 12. Динамические структуры данных.....                                                                              | 109 |
| Глава 13. Вопросы к экзамену и примеры<br>экзаменационных задач .....                                                     | 121 |
| Заключение.....                                                                                                           | 127 |
| Библиографический список.....                                                                                             | 128 |

## **Введение**

C++ развивается с 1980-х годов и всё это время остаётся основным языком системного программирования и одним из популярнейших средств разработки прикладных приложений. Более того, многие распространённые сегодня языки программирования, такие как Java, Javascript, PHP, C# и другие, имеют во многом Си-подобные синтаксис и концепцию.

С другой стороны, современный C++ далеко ушёл от своего прямого предшественника – классического C. Развитие языка продолжается, в частности, в стандартах C++11 и C++14, постоянно выходят новые версии популярных сред разработчика на C++, таких как C++ Builder, Visual Studio, QT Creator.

Таким образом, изучение C++ остаётся важным элементом подготовки специалиста, профессиональная деятельность которого связана с программированием.

Главы 1–4 настоящего учебного пособия написаны К.И. Зыряновым, главы 5–13 – Н.П. Кисленко. Материал пособия соответствует первой и второй частям курса "Языки программирования", читаемого студентам, обучающимся по направлению подготовки 09.03.02 "Информационные системы и технологии".

При работе с материалами и примерами пособия предполагается использование IDE Visual Studio версии 2015 или выше либо QT Creator версий 5.X.

# Глава 1. Введение в C++.

## Линейный алгоритм и простейшее приложение

**Создание консольного приложения. Компиляция программного кода, отладка и запуск программы.** В настоящее время для разработки приложений на языках C, C++ и C# в учебных заведениях используются последние версии Microsoft Visual Studio (2013, 2015), поэтому технология создания первого консольного приложения будет изложена применительно к этой среде.

1. Запускаем любым способом приложение Visual Studio.
2. В левой колонке стартового окна нажимаем ссылку *Создать проект*.

3. В левой колонке выбираем шаблон *Visual C++*, в центральной колонке тип проекта *Пустой проект*. Внизу окна указываем имя и расположение проекта (например, имя *Lab1*, расположение в терминальных классах должно быть в каталоге *D:\Prog*. Внутри этой папки желательно создать именной каталог (например, 110)). Нажимаем кнопку ОК.

4. В окне *Обозреватель решений* нажимаем правой кнопкой мыши на значке с надписью *Файлы исходного кода* и в появившемся контекстном меню выбираем действие *Добавить → Создать элемент*. В окне типов элемента выбираем элемент *Файл C++ (.cpp)*. Указываем имя файла (например, *BaseCode*).

5. В появившейся вкладке с заголовком *BaseCode* набираем текст программы:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
    cout << "Hello, World!";
    system("pause>nul");
    return 0;
}
```

6. Нажимаем клавишу F5. Если текст программы набран верно и все компоненты системы установлены корректно, то произойдёт запуск консольного приложения с текстовым сообщением "Hello, World!" и мигающим курсором справа. Для завершения работы программы нажимаем клавишу Enter, после этого системная команда `pause` завершит своё выполнение, и произойдёт выход из приложения.

Если консоль не появилась, то нужно внимательно проверить текст программы; особенно обратите внимание на знак *точка с запятой*, которым должна заканчиваться каждая команда программы.

Первая строчка программы  
`#include <iostream>`

является *директивой препроцессора*. С её помощью подключается библиотека `iostream`, предназначенная прежде всего для поддержки системы ввода/вывода.

Аналогично во второй строчке подключается библиотека `cstdlib` для поддержки функции задержки экрана `system`.

Команда  
`using namespace std;`

является инструкцией для компилятора использовать стандартное пространство имён (в том числе идентификатор стандартного выходного потока `cout`).

Работа программы состоит в выполнении главной (и единственной) функции программы со стандартным (регламентированным) названием `main` (см. главу 5). Конструкция `main()` выполняет роль входной точки программы. Операционная система предоставляет отдельный (новый) поток в распоряжение функции `main`. Как вы думаете, где в тексте программы находится выходная точка программы?

Тело функции `main` представляет собой блок команд, заключённых в фигурные скобки `{}`.

**Потоковый метод ввода и вывода на консоль данных.** Первая команда выводит на консоль текстовое сообщение. Начинается она с ключевого слова-идентификатора потокового вывода `cout` (*console output* – устройство вывода), необходимо-

го для начала формирования стандартного выходного потока. Далее следует операция << помещения в выходной поток текста, заключённого в двойные кавычки.

Наша программа выводит одну строчку текста. А как поступить, если необходимо вывести несколько строк? Для этого нужно после вывода первого сообщения выполнить перевод курсора на новую строку. Сделать это можно двумя способами:

- с помощью служебного слова `endl`  
`cout << "Hello, World!"<<endl;`

- используя непечатаемый символ перевода строки, которому соответствует *escape-последовательность* `\n`  
`cout << "Hello, World!\n";`

Существуют и другие непечатаемые символы, в частности, горизонтальная табуляция, вертикальная табуляция, перевод каретки (см. [1, гл. 1]).

Второй командой функции `main()` является функция `system("pause");` позволяющая "задержать" консоль. Эта функция, распознав кодовое служебное слово `pause`, выдаёт на консоль сообщение и ожидает нажатия на какую-либо клавишу клавиатуры, после чего управление возвращается главной функции `main()`. Следующая (третья) команда `return 0;` возвращает управление операционной системе. Это и есть точка выхода из программы. Возвращаемое операционной системе целочисленное значение "0" является формальным подтверждением того, что работа программы завершена корректно.

Если убрать команду `system("pause");`, то после запуска программы появившаяся консоль закроется "мгновенно". Чтобы в этом убедиться, не обязательно стирать команду. Можно её временно заблокировать (точнее, *закомментировать*), добавив слева двойную косую черту:

```
//system("pause");
```

Компилятор рассматривает содержимое строки, расположенное справа от двойной косой черты, как пробельные символы и, следовательно, игнорирует. С помощью последовательности `//` можно не только блокировать команды в процессе разработки программы. Поясняющие комментарии возле команд

или блоков помогают разработчику быстрее разобраться в программном коде в тех случаях, когда код содержит много команд, реализует сложный алгоритм, программист давно не работал с этой программой или с кодом работают другие программисты (например, в рамках совместного проекта). Помимо однострочного комментария `//`, можно использовать *многострочный*. Для этого перед первым символом комментируемого участка кода ставится пара символов `/*`, а замыкается участок символами `*/`.

Помимо команды `system("pause");` существуют другие способы задержки экрана – это функции, которые возвращают символ нажимаемой клавиши, например, функция `cin.get()`.

Возможна также пара команд следующего вида:

```
int pause; cin>>pause;
```

Здесь выход из консольного приложения произойдёт после ввода любого целочисленного значения и нажатия клавиши **Enter**. Принцип действия этой пары команд станет понятен по мере изучения темы *Переменные*. Последний пример иллюстрирует, что, в целях обеспечения компактности программы, несколько команд можно располагать на одной строчке. С другой стороны, это может навредить читаемости программного кода.

**Данные.** Работа любой программы в конечном счёте состоит в обработке данных. По отношению к программе данные можно разбить на три категории:

1. Литерал – это данные, присутствующие непосредственно в программном коде.
2. Переменная – именованный участок оперативной памяти, к которому программа может обращаться во время своей работы.
3. Файлы – данные, хранящиеся в постоянном запоминающем устройстве. Работа с файлами будет рассмотрена в главах 8–9.

**Литеральные константы (литералы).** Литералы могут быть числовыми, символьными и строковыми. В нашем примере

присутствует один литерал – строковая константа "Hello, World!" Он помещён непосредственно в поток вывода. Благодаря паре двойных кавычек этот набор символов интерпретируется системой как строковая константа.

Команда вывода может содержать несколько операторов вывода <<, после которых последовательно указываются данные, выводимые на экран. Так, например, приветственное сообщение можно разбить на две части:

```
cout << "Hello," << " World!\n";
```

При этом на консоль будет выведен тот же текст, что и ранее. В данном случае программа будет содержать уже две литеральные строковые константы. Для разделения двух слов на консоли первым символом второго литерала является пробел, который можно было бы поставить в конце первой строковой константы.

Такое же приветственное сообщение можно получить, используя символьные литералы:

```
cout << "Hello," << 'w' << 'o' << 'r' <<
'l' << 'd' << '!' << '\n';
```

Символьный литерал применяется для представления одиночного знака. При этом символ необходимо заключить в одинокие кавычки (апострофы).

Теперь приведём пример с целым числовым литералом:

```
cout << "It's " << 2017 << " year now\n";
```

вещественным с плавающей точкой:

```
cout << "Volume of file is" << 1.44 << "MB\n";
```

а также вещественным в экспоненциальной форме (заменим второй и третий выходные потоки):

```
...<< 1.44E+3 << "KB\n";
```

или

```
...<< 1.44E-3 << "GB\n";
```

Как вы думаете, что в последних двух примерах означают последовательности E+3 и E-3?

**Переменные.** В процессе работы программы часто возникает необходимость в сохранении некоторых данных в опера-

тивной памяти для того, чтобы иметь возможность в дальнейшем их использовать. Роль такого временного хранилища данных играет переменная. Переменная – это зарезервированная область оперативной памяти, в которой могут сохраняться (а также из которой могут считываться) данные.

Прежде чем говорить о технологии работы с переменными, приведём пример простейшей программы, в которой используется переменная. Здесь и далее для краткости будем приводить только содержимое блока операторов главной функции `main(){ }`, за исключением двух последних команд (`system("pause>nul");` и `return 0;`):

```
int age;
cout << "How old are you?" << endl;
cin >> age;
cout << "You are " << age << " years old\n";
```

Задачи первой команды:

1. Выделить 32 бита (4 байта) в оперативной памяти.
2. Связать с адресом первой ячейки идентификатор имени переменной (имя переменной) `age`.
3. Использовать в дальнейшей работе с этими ячейками бинарное кодирование целых чисел (в соответствии с идентификатором типа переменной `int`).

Таким образом, команда `int age;` является началом жизненного цикла переменной `age`. Эта процедура называется *объявление переменной*. В примере использован простейший синтаксис объявления переменной

```
тип имя;
или, если более корректно,
идентификатор_типа идентификатор_имени;
```

В табл. 1 указаны основные *скалярные* типы данных C++. О составных типах данных, для которых моделируемый объект может состоять из нескольких переменных, рассказывается в главах 10–11.

Таблица 1

## Основные типы данных C++

| Идентификатор | Описание                               | Размер<br>в битах | Диапазон                          |
|---------------|----------------------------------------|-------------------|-----------------------------------|
| char          | Символьный                             | 8                 | -127÷128                          |
| int           | Целый                                  | 32                | -2147483648÷<br>2147483647        |
| float         | Вещественный<br>с плавающей<br>точкой  | 32                | $\pm 3.4E-38 \div$<br>$3.4E+38$   |
| double        | Вещественный<br>с двойной<br>точностью | 64                | $\pm 2.2E-308 \div$<br>$1.8E+308$ |
| void          | "Пустой" тип                           | 0                 | –                                 |

Подробнее о базовых типах данных, а также об их модификациях см. [1, п. 7.6]. Для работы с вещественными числами, как правило, используется тип `double`, так как он является стандартным типом для работы с математическими функциями библиотеки `math.h`.

*Идентификатор переменной* (или *имя переменной*) – это набор символов, с помощью которых в программном коде можно обращаться к указанной области памяти. Имя переменной придумывает программист исходя из тех или иных соображений (удобства, практичности, благозвучности, лаконичности и т.д.). Существуют синтаксические ограничения на идентификаторы переменной [1, гл. 1].

Вывод сообщения "How old are you?" – это обращение к пользователю (пользователь имеет право знать, что от него требуется в тот или иной момент) с просьбой указать свой возраст (набрать на клавиатуре целочисленное значение и нажать на клавишу `Enter`).

В результате выполнения третьей команды происходит присваивание переменной `age` значения, введенного пользователем (запись в соответствующие ячейки памяти соответствующих данных в целочисленном формате кодирования). Так как

значение переменной ранее не было задано, эта операция есть не что иное, как *инициализация* переменной. Значение переменной может во время работы программы меняться многократно. В нашем примере для ввода данных использован потоковый метод, в синтаксис которого входит ключевое слово-идентификатор `cin`, оператор `>>`, а также имя переменной, связываемой с потоком ввода.

Четвёртая команда – вывод её значения переменной `age` на консоль. Здесь во втором потоке располагается имя переменной, значение которой будет выведено на консоль между двумя текстовыми фрагментами сообщения.

**Операции и выражения.** Главное предназначение любой программы – обработка данных (в том числе получение новых данных на основе имеющихся). Обработка числовых значений производится с помощью *операций* – комбинаций символов, определяющих действия по преобразованию значений.

Наиболее распространённой является операция *присваивания* "=", она присутствует в большинстве выражений. Синтаксис операции присваивания таков:

переменная = выражение ;

В результате выполнения данной команды переменной будет присвоено значение, возвращаемое при вычислении выражения. На месте выражения может находиться литерал или идентификатор переменной (разумеется, если она к этому времени уже инициализирована).

При выполнении операции присваивания может произойти *преобразование (приведение)* типа данных в правой части в соответствии с типом данных переменной:

```
int a;  
double b=3.2;  
a=b;  
cout <<"a="<<a<<"b="<<b;
```

Обратим внимание на два важных момента:

- инициализация переменной `b` (во второй строке) произошла во время её определения;

- в третьей строке произошло приведение типа данных (от `double` к `int`). При этом произошла потеря данных, так как была отброшена дробная часть значения 3.2.

Возможно многократное выполнение операции присваивания в одной команде (в порядке справа налево):

```
a=b=var=3.2;
```

Для выполнения любых расчётов используются *арифметические операторы*. К их числу относятся, во-первых, пять операторов, выполняющих стандартные математические операции: сложения, вычитания, умножения, деления и взятия остатка от деления. Они обозначаются +, −, \*, / и % соответственно. Это *бинарные* операции, они выполняют действие с *двумя операндами* (аргументами). Рассмотрим их синтаксис на примере оператора сложения:

```
операнд1 + операнд2;
```

Операция сложения возвращает значение, но для того, чтобы это значение "не пропало", её следует выполнять в одной команде вместе с операторами потока вывода:

```
cout << операнд1 + операнд2;
```

или с оператором присваивания:

```
переменная = операнд1 + операнд2;
```

Примером может быть выражение вида `a=b+c`;

Если одним из операндов в последнем примере является та же переменная, что и в левой части, например `a=a+b`; то для этой операции можно использовать более компактную конструкцию вида `a+=b`; Такое действие встречается, например, в типовых алгоритмах нахождения суммы (главы 3, 4).

Для приращения значения переменной на единицу существует специальный арифметический оператор *инкремент* ++. Приведём четыре эквивалентных выражения, приводящих к увеличению значения переменной на единицу:

|                     |                    |                   |                   |
|---------------------|--------------------|-------------------|-------------------|
| <code>a=a+1;</code> | <code>a+=1;</code> | <code>a++;</code> | <code>++a;</code> |
|---------------------|--------------------|-------------------|-------------------|

В третьем и четвёртом выражениях использован оператор инкремента соответственно в префиксной и постфиксной форме. Приведём пример, демонстрирующий отличие этих двух форм:

```
int i, j, a, b;
i=j=4;
a=++i;
b=j++;
cout <<"i="<<i<<"j="<<j;
cout <<"a="<<a<<"b="<<b;
```

Можно заметить, что отличие заключается в последовательности операции инкремента и операции присваивания. Инкремент – это *пример унарного* оператора, применяемого только к одному аргументу, а не к двум, как при сложении или умножении.

Существует аналогичный оператор *декремента* --, уменьшающий значение переменной на единицу:

|        |       |      |      |
|--------|-------|------|------|
| a=a-1; | a-=1; | a--; | --a; |
|--------|-------|------|------|

Что же касается операций умножения и деления, то, по понятным причинам, операции, аналогичные инкременту и декременту, лишены смысла. Тем не менее, для изменения значения переменной с некоторым коэффициентом *b* также можно использовать компактные способы записи (*нотации*) вида

|        |       |
|--------|-------|
| a=a*b; | a*=b; |
| a=a/b; | a/=b; |

Операция деления в C++ имеет особенность, если её операнды целочисленные. В этом случае результат будет округлён до целого, т.е. будет произведена операция *деления нацело*. Такая особенность может привести к нежелательным последствиям при работе с выражениями, содержащими целые и вещественные операнды. В качестве примера можно привести известную формулу для вычисления объёма сферы (пусть её радиус будет, например, 1.5). Сравните результат выполнения двух команд с результатом, полученным, например, с помощью калькулятора:

```
cout << "V="<<(4/3)*3.14159*1.5*1.5*1.5;
cout << "V="<<(4.0/3.0)*3.14159*1.5*1.5*1.5;
```

Для работы с целочисленными данными есть также ещё одна возможность. Это нахождение остатка от деления благодаря применению оператора %. Пример: 15%4. Чему равен результат?

При работе с арифметическими операторами следует учитывать *приоритет* операций (очерёдность выполнения компилятором действий, входящих в одно выражение). Наивысший приоритет имеют операторы ++, --. Далее следуют \*, /, %. В последнюю очередь будут выполнены операторы +, -.

Сравните результат выполнения двух команд:

```
cout << 2+2*2;
```

```
cout << (2+2)*2;
```

Существуют также *логические* операторы и операторы *отношения*. Они широко применяются в *алгоритмах ветвления* (мы их рассмотрим в главе 2).

В завершение расскажем о *стандартных математических функциях*, использование которых значительно расширяет вычислительные возможности языков программирования.

В библиотеке `math.h` содержатся такие функции, как `sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `exp(x)`, `sqrt(x)` и другие, имеющие один входной аргумент (с типом данных `double`), а также функция возведения в степень `pow(a, b)`, имеющая два аргумента. Все эти функции возвращают вещественное число. Подробнее о математических функциях см. [1, п. 7.4].

### **Задание к лабораторной работе**

После изучения темы 1 "Линейный алгоритм" предполагается выполнение лабораторной работы. В качестве примера приведём программу, вычисляющую объём сферы:

```
#include <iostream>
#include <math.h>
using namespace std;
int main() {
    double R,V,pi;
    pi=4.0*atan(1.0);
    cout << "Enter the radius of the sphere: ";
    cin >> R;
    V=4.0*pow(R,3)*pi/3.0;
    cout << "The volume of a sphere is
"<<V<<endl;
    system("pause");
    return 0;}
```

## Глава 2. Разветвляющийся вычислительный процесс и условный оператор

Можно заметить, что нашим первым программам недостаёт гибкости и умения "принимать решения". Ведь уже несложный алгоритм решения квадратного уравнения предусматривает два варианта расчёта, реальные же алгоритмы могут выдавать результаты, зависящие от десятков и сотен условий.

**Разветвляющийся вычислительный процесс (РВП)** реализуется по одному из нескольких направлений вычисления (*ветвей* алгоритма). Выбор одной из ветвей зависит от истинности или ложности некоторого условия (*логического выражения*), включённого в состав *условного оператора*. Программа должна учитывать все возможные ветви вычислений. Всегда выполняется только одна из всех возможных ветвей.

Логические выражения строятся из арифметических выражений, операций отношения, логических операций и круглых скобок.

Результатом вычисления логического выражения является одно из двух значений: TRUE (истина) или FALSE (ложь), представляемых в языке C++ как целочисленные значения 1 и 0 соответственно.

**Операции отношения** имеют следующий общий вид:

арифм. выражение1 `OO` арифм. выражение2

где `OO` – один из следующих знаков операций:

`<=`    `<`    `>`    `>=`    `==`    `!=`

Обратите внимание, что сравнение двух значений выполняется оператором `==`, в отличие от присваивания, обозначаемого одним знаком равенства `=`.

Обратите внимание, что выражения вида `if (d)` и `if (d!=0)` в языке C++ идентичны.

**Пример 1.** Составим несколько логических выражений.

- при решении квадратного уравнения выбор ветви вычислений зависит от значения дискриминанта `d`, т.е. от условия `d<0`;

- $\text{row}(x, 2) + \text{row}(y, 2) \leq \text{row}(R, 2)$  – это выражение истинно для точек с координатами  $(x, y)$ , лежащих внутри круга радиуса  $R$  с центром в начале координат;

- $\cos(x) > 1$  – выражение всегда ложно.

К вещественным значениям операция  $==$  (равно) в общем случае *неприменима* из-за неточного представления этих значений в памяти компьютера. Поэтому для вещественных переменных отношение вида  $a == b$  заменяется на  $\text{abs}(a - b) < E$ , где  $E$  – малая величина, определяющая допустимую погрешность.

**Логические операции** применимы только в логических выражениях и служат для составления сложных условий, требующих более одной операции отношения. В C++ определены три основные логические операции (табл. 2):

Таблица 2

Логические операции в C++

| Математическая запись | Запись в C++ | Название                                        |
|-----------------------|--------------|-------------------------------------------------|
| $\neg$                | !            | Операция NOT (отрицание)                        |
| $\wedge$              | &&           | Операция AND (логическое умножение, конъюнкция) |
| $\vee$                |              | Операция OR (логическое сложение, дизъюнкция)   |

Операция NOT применима к одному логическому выражению (является *унарной*). Результат истинен, если выражение ложно, и наоборот.

**Пример 2.** Выражение  $!(\sin(x) > 1)$  всегда истинно.

Операция AND связывает не менее двух логических выражений (является *бинарной*). Результат будет истинным, если истинны все выражения; если хотя бы одно из выражений ложно, то результат будет ложным.

**Пример 3.** Распишем выражение  $x \in [a, b]$ . Так как операции принадлежности в C++ нет, используем конъюнкцию и операции отношения:  $(x \geq a) \&\& (x \leq b)$

Распишем выражение  $a, b, c > 0$  (одновременно):  
 $(a > 0) \&\& (b > 0) \&\& (c > 0)$

Операция **OR** также связывает не менее двух логических выражений. Результат будет истинным, если хотя бы одно выражение истинно; если все выражения ложны, то результат будет ложным.

**Пример 4.** Распишем выражение  $x \notin [a, b]$ :  $(x < a) \vee (x > b)$   
Существует и другой способ:  $!((x \geq a) \&\& (x \leq b))$

**Пример 5.** Распишем условие "хотя бы одно из значений  $a, b, c$  положительно":  $(a > 0) \vee (b > 0) \vee (c > 0)$

**Пример 6.** Условие "только одно из значений  $a, b, c$  положительно" выглядит так:

$(a > 0) \&\& (b \leq 0) \&\& (c \leq 0) \vee$   
 $(a \leq 0) \&\& (b > 0) \&\& (c \leq 0) \vee$   
 $(a \leq 0) \&\& (b \leq 0) \&\& (c > 0)$

Логические операции можно применять и к *отдельным битам* своих аргументов. В этом случае они обозначаются  $\&$  (конъюнкция),  $\vee$  (дизъюнкция),  $\wedge$  (строгая дизъюнкция). Подробнее см. [1, гл. 2].

Приоритеты логических операций:

1.  $!$
2. Операции отношения
3.  $\&\&$
4.  $\vee \vee$

**Короткий условный оператор** имеет следующий общий вид (рис. 1):

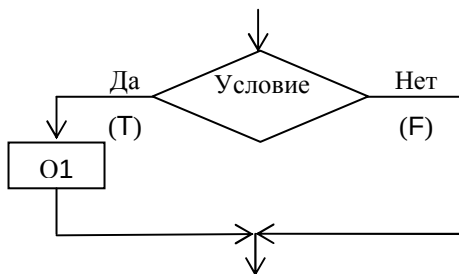


Рис. 1. Короткий условный оператор

Запись на C++ следующая:

```
if(логическое_выражение) оператор1;
```

Сначала вычисляется логическое\_выражение. Если оно истинно, то выполняется оператор1, иначе оператор1 игнорируется.

Если по условию требуется выполнить несколько операторов, их необходимо заключить в операторные скобки {...}, образуя единый *составной оператор*.

**Пример 7.** Решить квадратное уравнение.

```
double a,b,c,d,x1,x2;
cout << "Vvedite a,b,c ";
cin >> a;cin >> b;cin >> c;
d=pow(b*b-4*a*c,2);
if(d>0) {
x1=0.5*(-b+d)/a;
x2=0.5*(-b-d)/a;
cout<< "x1="<<x1<< "\nx2="<<x2;}
```

**Пример 8.** Вычислить  $y = \max(a, b, c)$ .

```
y=a;
if(b>y) y=b;
if(c>y) y=c;
```

**Полный условный оператор** имеет общий вид

```
if(логическое_выражение) оператор1;
else оператор2;
```

Если логическое\_выражение истинно, то выполняется оператор1, иначе выполняется оператор2. Всегда выполняется только один из операторов.

**Пример 9.** Вычислить  $m = \min(x, y)$ .

```
if(x<y) m=x; else m=y;
```

Существует альтернативная, более компактная форма записи алгоритма ветвления – с помощью *тернарного* (имеющего три операнда) оператора:

```
m = x<y ? x : y;
```

Общий синтаксис тернарного оператора:

```
условие ? выражение1 : выражение2
```

**Пример 10.** Если значения  $a$  и  $b$  одного знака, найти их произведение, иначе заменить нулями.

```
if(a*b>0) c=a*b;  
else {a=0; b=0;}
```

Из примера видно, что к ветви алгоритма, состоящей более чем из одного оператора, также применяются операторные скобки.

**Составной условный оператор** применяется, когда есть более двух вариантов расчёта. Общий вид оператора следующий:

```
if(логическое_выражение1) оператор1;  
else if(логическое_выражение2) оператор2;  
.  
.  
.  
else if(логическое_выражениеN) операторN;  
else оператор0;
```

Последовательно проверяются условия  $1, 2, \dots, N$ . Если некоторое условие истинно, выполняется соответствующий оператор, и управление передаётся на оператор, следующий за условным. Если все условия ложны, выполняется оператор  $0$ . Число ветвей  $N$  неограниченно, ветви `else оператор0`; может и не быть.

**Пример 11.** Вычислить значение функции, заданной графически (рис. 2):

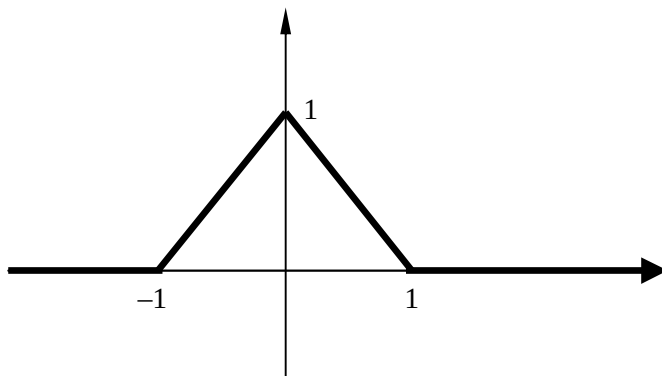


Рис. 2

```

setlocale(LC_ALL, "russian");
double x, y;
cout <<
    "Введите значение аргумента\n";
cin >> x;
if (x<-1) y = 0;
else if (x<0) y = x + 1;
else if (x<1) y = 1 - x;
else y = 0;
cout << "y="<<y<<endl;

```

Другой вариант кода:

```

if ((x<-1) || (x >= 1)) y = 0;
else if (x<0) y = x + 1;
else y = 1 - x;

```

В первой команде листинга подключается поддержка кириллицы с помощью функции `setlocale()`; которая описывается в заголовочном файле `<locale.h>`.

Когда после ключевых слов `then` и `else` вновь используются условные операторы, они называются *вложенными*. Число вложений может быть произвольным; при этом действует правило: `else` всегда относится к ближайшему `then`. Часто вложением полных условных операторов можно заменить использование составного.

**Пример 12.** Определить, в какой координатной четверти находится точка с координатами  $(x, y)$ , не лежащая на осях координат.

**Вариант 1**, без использования вложений:

```

if((x>0)&&(y>0)) p=1;
else if((x<0)&&(y>0)) p=2;
else if((x<0)&&(y<0)) p=3;
else p=4;

```

**Вариант 2**, с вложениями:

```

if(x>0) {
    if(y>0) p=1;else p=4;
}

```

```

else {
    if(y>0) p=2;else p=3;
}

```

В первом варианте программой проверяется от двух до шести условий, во втором – всегда только два условия. Здесь использование вложений дало существенный выигрыш в производительности.

**Пример 13.** Вычислить значение  $y$ , равное знаку числа  $x$ .

**Вариант 1:**

```

if(x>0) y=1;
else if(x<0) y=-1;
else y=0;

```

**Вариант 2:**

```

if (x>1) y=1;
else { if(x<0) y=-1;else y=0; }

```

Здесь в обоих вариантах проверяется не более двух условий, поэтому варианты примерно равноценны.

**Оператор выбора.** Для случаев, когда требуется выбор одного значения из конечного набора вариантов, оператор `if` удобнее заменять оператором-переключателем `switch`:

```

switch(выражение) {
    case 31: оператор1;break;
    case 32: оператор2;break;
    . . .
    case 3N: операторN;break;
    default оператор0;
}

```

Переключатель предназначен для выбора одного из нескольких альтернативных путей выполнения программы. Выполнение переключателя начинается с вычисления выражения, следующего за ключевым словом `switch`.

Значения  $31, \dots, 3N$  должны быть константными и уникальными внутри тела оператора-переключателя. Значение выражения также должно быть целочисленным.

После вычисления выражения управление передаётся тому из *операторов* 1, ..., N тела переключателя, значение константы которого совпадает с вычисленным значением.

Выполнение тела оператора-переключателя начинается с выбранного таким образом оператора и продолжается до конца тела переключателя или до тех пор, пока какой-либо оператор не передаст управление за пределы тела.

Оператор, следующий за ключевым словом `default`, выполняется, если ни одна из констант варианта не равна значению выражения. Если ветвь `default` опущена, ни один оператор в теле переключателя не выполняется.

Синтаксические конструкции `case` и `default` являются метками, однако на них нельзя передать управление по оператору `goto`. Метки `case` и `default` существенны только при начальной проверке, когда выбирается оператор для выполнения в теле переключателя. Все операторы тела переключателя, следующие за выбранным, выполняются последовательно, "не замечая" меток `case` и `default`, если только какой-либо оператор не передаст управление за пределы тела переключателя. Для выхода из тела переключателя после выполнения одной из ветвей, как правило, используется оператор разрыва `break`.

**Пример 14.** Простейший калькулятор, выполняющий одну из четырёх бинарных операций (действий с двумя операндами) – сложение, вычитание, умножение или деление. При этом операторы `+`, `-`, `*` и `/` будут представлены числами 1, 2, 3 и 4 соответственно.

```
setlocale(LC_ALL, "russian");
float a, b; int p;
cout<<"Введите код оператора:
1- '+', 2- '-', 3- '*', 4- '/' ";
cin>>p;
cout<<"Введите первый операнд a: "; cin>>a;
cout<<"Введите второй операнд b: "; cin>>b;
switch(p) {
```

```

        case 1:cout<<"a+b="<<a+b;break;
        case 2:cout<<"a-b="<<a-b;break;
        case 3:cout<<"a*b="<<a*b;break;
        case 4:cout<<"a/b="<<a/b;break;
        default:cout<<"Введён неправильный код
оператора";};cout<<"\n\n";

```

**Пример 15.** Времена года. Пользователь вводит номер месяца от 1 до 12, а программа определяет и выводит на консоль название времени года.

```

int m;
cout<<"Введите номер месяца: "; cin>>m;
switch(m) {
    case 1: case 2: case 12:
        cout<<"Зима"; break;
    case 3: case 4: case 5:
        cout<<"Весна"; break;
    case 6: case 7: case 8:
        cout<<"Лето"; break;
    case 9: case 10: case 11:
        cout<<"Осень"; break;
    default:
        cout<<"Неверный номер месяца";
}
cout<<"\n\n";

```

Приведём несколько примеров законченных программ, использующих РВП.

**Пример 16.** Проверить, может ли быть построен прямоугольный треугольник по заданным длинам сторон  $a$ ,  $b$ ,  $c$ .

Проблема с решением этой задачи – не в проверке условия теоремы Пифагора, а в том, что в условии не сказано, какая из сторон является гипотенузой. Решений возможно несколько: запрашивать у пользователя ввод данных по возрастанию длины сторон, проверять все три возможных условия теоремы Пифагора и т.п. Используем наиболее естественное решение – перед проверкой условия теоремы Пифагора упорядочим величины  $a$ ,  $b$ ,  $c$  так, чтобы выполнялись соотношения  $a \leq b \leq c$ .

Для этого применим достаточно простой приём с обменом значений переменных.

```
double a, b, c;    // Длины сторон
double s; // Буферная переменная для обмена
cout << "Введите длины сторон треугольника >0:\n";
cin >> a; cin >> b; cin >> c;
if(a>b) {s=a; a=b; b=s;}
if(a>c) {s=a; a=c; c=s;}
if(b>c) {s=b; b=c; c=s;}
if (abs(a*a + b*b - c*c) < 1e-8)
    cout << "Треугольник может быть построен\n";
else
    cout << "Треугольник не может быть построен";
```

**Пример 17.** Определить, попадает ли точка плоскости, заданная координатами  $(a, b)$ , в прямоугольник, заданный координатами двух углов –  $(x_1, y_1)$  и  $(x_2, y_2)$ .

Как и в предыдущей задаче, было бы не совсем корректно требовать от пользователя вводить данные в определённом порядке. Лучше при необходимости поменять координаты  $x$  и  $y$  прямоугольника так, чтобы пара переменных  $(x_1, y_1)$  содержала координаты левого нижнего угла прямоугольника, а  $(x_2, y_2)$  – правого верхнего.

```
double x1, y1, x2, y2, a, b;
cout << "Введите координаты первого угла\n";
cin >> x1; cin >> y1;
cout << "Введите координаты второго угла\n";
cin >> x2; cin >> y2;
if (x1>x2) {a = x1; x1 = x2; x2 = a;}
if (y1>y2) {a = y1; y1 = y2; y2 = a;}
cout << "Введите координаты точки\n";
cin >> a; cin >> b;
if ((x1 <= a) && (a <= x2) && (y1 <= b) && (b <= y2))
    cout << "Точка попадает в прямоугольник\n";
else
    cout << "Точка НЕ попадает в прямоугольник\n";
```

### Глава 3. Ввод и вывод в библиотеке `stdio.h`. Оператор цикла. Циклы с предусловием и постусловием

Прежде чем перейти к рассмотрению циклического алгоритма, познакомимся с функцией вывода `printf()` и функцией ввода `scanf_s()`, которые содержатся в библиотеке `stdio.h`. Ниже приведён листинг программы, вычисляющей объём сферы. Данная программа аналогична приведённой в конце главы 1.

```
#include <stdio.h>
#include <cstdlib>
#include <math.h>
int main() {
    double R, V, pi;
    pi = 4.0*atan(1.0);
    printf("Enter the radius of the sphere: ");
    scanf_s("%lf",&R);
    V = 4.0*pow(R, 3)*pi / 3.0;
    printf ("The volume of a sphere is %f \n",V);
    system("pause");
    return 0;}
```

Рассмотрим отличия от предыдущей версии:

1. Вывод на консоль строки с обращением к пользователю (строка 7 листинга) является примером простейшего использования функции `printf()`. В данном случае обрабатывается один аргумент – строка, которая сканируется и посимвольно выводится на консоль.

2. В десятой строке листинга функция `printf()` имеет два аргумента. Первый из них является строкой, символы которой функция последовательно выводит на консоль до тех пор, пока не встретит спецификатор преобразования `%f`. Тогда функция приостанавливает сканирование строки и обращается ко второму аргументу, которым является переменная `V`, и вы-

водит её на консоль в вещественном формате. При этом может произойти потеря точности вследствие преобразования выводимого значения из формата `double` в формат `float`. Само же значение переменной `V` не меняется. Если требуется вывод значения переменной `V` в формате двойной точности `double`, то следует использовать модифицированный спецификатор преобразования `%lf` (сокращение от `long float`). Подробнее о модификациях спецификаторов преобразования см. [1, п. 7.6]. После вывода значения переменной `V`, функция `printf()` продолжает сканирование строки и, встретив специальный символ `\n`, производит перенос строки в консоли. Так как другие символы в первом строковом аргументе отсутствуют, функция `printf()` возвращает управление функции `main()`.

3. В стандарте C++ функция форматного чтения данных из потока называется `scanf`, но в последних версиях Visual Studio рекомендуется использовать её "безопасную" версию `scanf_s`.

Функция `scanf_s()` считывает данные с клавиатуры. У неё два аргумента: первый аргумент – строка, содержащая спецификатор преобразования. Спецификатор преобразования, при использовании функции `scanf_s()`, строго должен соответствовать типу данных, которые переменная получила в момент её определения. Типу данных `double`, который имеет переменная `R`, соответствует спецификатор `%lf`. Если опустить модификацию `l` и оставить последовательность `%f`, то возникнут проблемы с кодировками, но компилятор не выдаст ошибку. Второй аргумент функции `scanf_s()` – адрес переменной, по которому будет записано введённое пользователем значение, закодированное в формате `double`. Подробнее особенности работы с функциями и адресами переменных будут описаны в главе 5 настоящего учебного пособия.

**Циклический вычислительный процесс (ЦВП)** характеризуется повторением одних и тех же вычислений над некоторым набором данных. Повторяемый блок вычислений называют

телом цикла. Однократное выполнение тела цикла называют его *шагом*.

*Оператор пошагового цикла for* имеет следующий общий вид:

`for (НВ; УВ; ВП) оператор;`

Здесь НВ — начальное выражение, служащее для инициализации параметров цикла; УВ — условное выражение, прямо или косвенно определяющее число повторений цикла (цикл выполняется, пока УВ не станет ложным); ВП — выражение приращения, используемое для модификации параметра (параметров) цикла. Любое из трёх выражений может быть опущено, а также любые два выражения или все сразу.

Цикл работает следующим образом: сначала вычисляется НВ, если оно имеется. Затем вычисляется УВ и производится его оценка следующим образом:

- если УВ истинно (не равно нулю), то выполняется тело оператора. Затем вычисляется ВП, если оно есть, и процесс повторяется;
- если УВ опущено, то его значение принимается за истину, и процесс выполнения продолжается, как описано выше. В этом случае цикл `for` бесконечен и может завершиться только при выполнении в его теле операторов `break`, `goto`, `return`;
- если УВ ложно, то выполнение цикла заканчивается, и управление передаётся следующему за ним оператору программы. Цикл `for` может завершиться также при выполнении операторов `break`, `goto`, `return` в его теле.

Например, показанный ниже цикл выполняется 10 раз:

```
for (x=1; x<11; x++) printf ("*");
```

Оператор цикла с предусловием `while`:

```
while (выражение) оператор;
```

Тело цикла `while` выполняется до тех пор, пока значение выражения не станет ложным (равным нулю). Сначала вычисляется выражение. Если оно изначально ложно, то тело

цикла не выполняется, и управление передаётся на следующий за ним оператор программы. Если выражение истинно, то выполняется тело цикла. Перед каждым следующим выполнением цикла выражение вычисляется заново. Процесс повторяется до тех пор, пока выражение не станет ложным. Оператор `while` может завершиться досрочно при выполнении `break`, `goto` или `return` внутри своего тела.

Приведённый далее цикл `while` аналогичен показанному выше `for`:

```
x=1;
while (x<11) {
    printf ("*");
    x++;}
```

Оператор *цикла с постусловием* `do` записывается в виде `do` оператор `while` (выражение);

Тело цикла `do` выполняется один или несколько раз до тех пор, пока значение выражения не станет ложным (равным нулю). Сначала выполняется тело цикла – оператор, затем вычисляется условие – выражение. Если выражение ложно, цикл завершается, и управление передаётся следующему за телом цикла оператору программы. Если выражение истинно (не равно нулю), то тело цикла выполняется вновь, и выражение вычисляется повторно. Выполнение цикла повторяется до тех пор, пока выражение не станет ложным. Цикл `do` может завершиться досрочно при выполнении в своём теле операторов `break`, `goto`, `return`. Показанный далее цикл по действию совпадает с примерами на `for` и `while`:

```
x=1;
do {
    printf ("*"); x++;
} while (x<11);
```

На рис. 3 изобразим в виде блок-схем циклы `while` (слева) и `do` (справа):

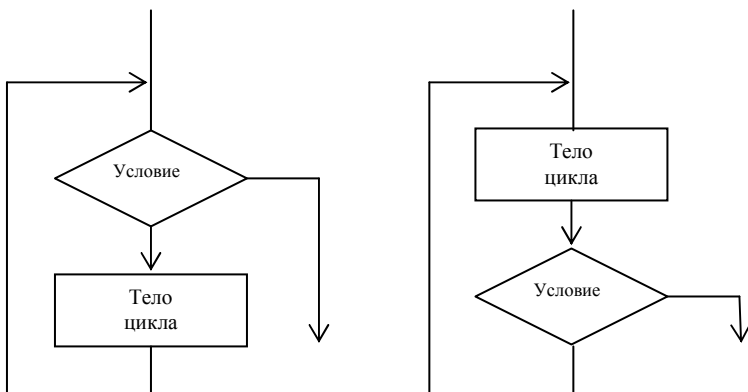


Рис. 3. Циклы с предусловием и постусловием

Итак, основное назначение циклов – обработка большого объёма данных. Математически эта обработка зачастую сводится к поиску, выбору и статистической обработке нужных величин. Практически в любой реальной задаче мы ищем максимальные и минимальные значения в наборе данных, суммируем или перемножаем требуемые данные, определяем арифметическое среднее или количество элементов, отвечающих условию. Для решения всех этих распространённых задач существуют  *типовые алгоритмы* , задающие правила выполнения соответствующих расчётов.

**Алгоритм табулирования** применяется для составления всевозможных таблиц, которыми могут быть как абстрактная таблица значений математической функции, так и конкретная таблица стоимости товара или платежей абонента сотового оператора.

В общем виде алгоритм можно описать так:

- до цикла задаётся начальное значение управляющей переменной, условием выхода из цикла служит достижение управляющей переменной конечного значения;
- в теле цикла на каждом шаге вычисляется очередное значение функции, зависящее от управляющей переменной, затем формируется строка таблицы;

- в конце шага цикла значение управляющей переменной (обозначим её  $x$ ) изменяется оператором вида  $x=x+d$ ; где  $d$  – заданный шаг по управляющей переменной.

**Пример 1.** Составим таблицу синусов в пределах от 0 до  $\pi$  с шагом по аргументу 0.25. Обозначим аргумент  $x$ , значение синуса от  $x$  обозначим  $y$ . В простейшем случае программа табулирования может выглядеть так:

```
double x, y, pi=4.0*atan(1.0);
printf("x\t sin(x) \n"); //заголовок таблицы
x = 0; //начальное значение аргумента
while (x <= pi) {
    y = sin(x); //вычисление функции
    printf("%6.4f%9.4f\n", x, y);
    x+=0.25;
} //шаг по x; завершение цикла
```

В реальной программе может понадобиться корректировка условия цикла из-за неточного представления вещественных чисел в памяти компьютера (см. главу 2).

Разумеется, другие изученные нами виды циклов также могут применяться при табулировании. Рассмотрим в качестве примера следующую задачу:

**Пример 2.** Известна стоимость единицы товара. Составить таблицу стоимости  $1, 2, \dots, k$  единиц товара, значение  $k$  вводится.

Так как число единиц товара – заведомо целое, при программировании задачи будет удобен цикл `for`:

```
double t;
int i, k;
printf
    ("Введите стоимость единицы товара t:");
scanf_s("%lf", &t);
printf("Введите количество единиц \
    товара для таблицы k: ");
scanf("%d", &k);
for(i=1; i<=k; i++) printf("%d\t%f\n", i, i*t);
```

Стоимость единицы товара обозначена  $t$ , переменная  $i$  необходима для перебора возможных значений единиц товара в цикле `for`. Поскольку счётчик цикла `for` автоматически меняется с заданным шагом 1, а функцией `printf()` можно выводить не только значения переменных, но и выражения, основной цикл программы состоит из одного оператора и не нуждается в операторных скобках.

**Алгоритм организации счётчика** применяется, когда требуется подсчитать *количество* элементов данных, отвечающих какому-либо условию (условиям). В общем виде алгоритм выглядит следующим образом:

1) описать переменную целочисленного типа, с помощью которой будет вестись подсчёт;

2) до цикла присвоить ей начальное значение 0;

3) в теле цикла, если очередной элемент данных отвечает условию подсчёта, увеличить эту переменную на 1 оператором вида `k++`;

Пока мы не начали подсчитывать количество, оно равно нулю, – этот очевидный для человека факт не очевиден для компьютера! Поэтому любой переменной, которая *может изменяться* в теле цикла, необходимо до цикла присвоить *начальное значение*, что и делает оператор вида `k=0`; Для сокращения программного кода можно совместить первый и второй пункты алгоритма организации счётчика: `int k=0`;

В качестве примера реализации алгоритма рассмотрим следующую задачу:

**Пример 3.** Последовательность  $z(i)$  задана соотношениями

$$z(i) = \begin{cases} i^2 \cdot \cos(i), & \text{если } i \text{ – чётное,} \\ \sin \frac{i}{2}, & \text{если } i \text{ – нечётное,} \end{cases} \quad i = 1, 2, \dots, 100.$$

Найти количество элементов последовательности, больших значения 0.5. Обозначив искомое количество за  $k$ , составим следующую программу:

```
double z;  
int i, k=0;
```

```

for(i=1;i<=100;i++) {
    if(i%2==0) z=pow(1.0*i,2)*cos(1.0*i);
    else z=sin(0.5*i);
    if(z>0.5) k++;
}
printf("Количество=%d\n",k);

```

Так как шаг по переменной *i* равен 1, в программе использован цикл `for`; для проверки того, является ли значение *i* чётным, использована операция `%`.

**Алгоритмы накопления суммы и произведения** применяются, когда требуется сложить или перемножить выбранные данные. В общем виде эти широко применяемые алгоритмы можно описать так:

- для подсчёта каждой суммы или произведения описать по одной переменной того же типа, что суммируемые или перемножаемые данные;
- до цикла переменной-сумме присвоить начальное значение 0, а произведению – значение 1;
- в теле цикла, если очередной элемент данных *t* отвечает условию суммирования или перемножения, сумма накапливается оператором вида `s+=t`; а произведение – оператором вида `p*=t`;

Очевидно, почему начальное значение произведения 1, а не 0. После оператора `p=0`; оператор `p*=t`; расположенный в теле цикла, будет возвращать только нули.

В качестве *t* может выступать не только переменная, но и очередной элемент массива, очередное значение функции, арифметическое выражение, зависящее от номера шага цикла, и т.д.

**Пример 4.** Найти сумму первых *n* натуральных чисел. Число *n* пользователь вводит с клавиатуры.

```

int i,n,s=0;
printf("Введите число n: ");
scanf_s("%d",&n);
for (i=1;i<=n;i++) s+=i;
printf("s=%d\n",s);

```

Немного изменив программу, можно найти произведение первых  $n$  чисел, т.е. значение  $n!$  Для вычисления арифметического среднего необходимо сумму  $S$  разделить на количество  $n$ .

**Типовые алгоритмы поиска максимума и минимума.** Рассмотрим простейшие статистические алгоритмы, главный из которых – определение максимального и минимального значения на множестве данных.

Рассмотрим алгоритм в общем виде.

Шаг 1:

- описать для каждого максимума и минимума по одной переменной того же типа, что анализируемые данные;
- до цикла максимуму присваивается заведомо малое для анализируемых данных значение либо первый элемент данных; минимуму присваивается заведомо большое для анализируемых данных значение либо первый элемент данных;
- в теле цикла каждый подходящий для поиска элемент данных  $t$  обрабатывается операторами вида  
 $\text{if}(t > \text{max}) \text{ max} = t;$  – для максимума;  
 $\text{if}(t < \text{min}) \text{ min} = t;$  – для минимума,  
где  $\text{max}$  и  $\text{min}$  – переменные, созданные для максимума и минимума соответственно.

Шаг 2 этого алгоритма требует комментариев, которые мы сделаем на примере поиска максимума. Очевидно, что сам алгоритм несложен – каждый элемент данных  $t$  последовательно сравнивается с ячейкой памяти  $\text{max}$ , и, если обнаружено значение  $t$ , большее текущего значения  $\text{max}$ , оно заносится в  $\text{max}$  оператором  $\text{max} = t$ ; Как мы помним, после описания на шаге 1 переменной  $\text{max}$ , её значение ещё не определено и может оказаться любым, откуда следует необходимость задания начального значения. Представим, что после выбора начального значения  $\text{max}$ , равного 0, при анализе встретились только отрицательные значения элементов  $t$ . В этом случае условие  $t > \text{max}$  не выполнится ни разу, ответом будет  $\text{max}$ , равное 0, что неправильно! Выбор *заведомо малого* начального значения  $\text{max}$  (например,  $-1\text{E}+30$ , т.е.  $-10^{30}$ , вряд ли встретится в любых реальных дан-

ных) гарантирует, что условие  $t > \max$  выполнится хотя бы раз и максимум будет найден. Альтернативный способ – присвоить  $\max$  значение отдельно вычисленного первого элемента последовательности данных. В этом случае ответ либо уже найден, если первый элемент и есть максимальный, либо будет найден в цикле.

Аналогичные рассуждения помогают понять, почему минимуму следует присваивать в качестве начального значения *заведомо большое* число.

**Пример 5.** Для функции  $y(x) = \sin^2(x)$ ,  $x \in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right]$ ,

$\Delta x = \frac{\pi}{24}$ , найти минимальное среди положительных и макси-

мальное значения.

Обозначив искомые значения  $\min$  и  $\max$  соответственно, составим следующую программу:

```
double pi, x, y, max, min;
pi=4.0*atan(1.0); x=-pi/3;
max=-2; min=2; //эти начальные значения -
//заведомо малое и большое для синуса
while(x<=pi/3+1e-6) {
    y=sin(x);
    if(y>0){//ищем min среди положительных
    if(y<min) min=y;
    if(y>max) max=y;
    x=x+pi/24;}}
printf("Минимум =%f\n", min);
printf("Максимум=%f\n", max);
```

## Глава 4. Массивы. Описание, ввод, вывод и обработка массивов

**Массивом** называют упорядоченный набор однотипных переменных (*элементов*). Каждый элемент имеет целочисленный порядковый номер, называемый *индексом*. Число элементов в массиве называют его *размерностью*. Массивы используются там, где нужно обработать сразу несколько переменных одного типа (например, оценки всех 20 студентов группы или координаты 10 точек на плоскости). Строку текста можно рассматривать как массив символов, а текст на странице – как массив строк.

Массив объявляется одним из следующих способов:

тип идентификатор [константное выражение];

тип идентификатор [] = {список элементов};

Элементы списка перечисляются через запятую, в этом случае размерность массива определяется по фактически указанному количеству элементов. Примеры:

```
int x[10];
```

```
float a[]={3.5,4.5,5.5}; //размерность=3
```

```
char div[3]=' ','\n','\t';
```

Элементы массивов в C++ всегда нумеруются *с нуля*.

Синтаксис индексного выражения для обращения к элементу массива имеет следующий вид:

```
выражение1[выражение2]
```

Значение индексного выражения находится по адресу, который вычисляется как сумма значений выражения1 и выражения2. Выражение1 должно иметь тип указателя на некоторый тип, например быть идентификатором массива, а выражение2, заключённое в квадратные скобки, должно иметь целый тип или преобразовываться к нему.

Индексное выражение может иметь более одного индекса, что соответствует многомерному массиву. Синтаксис такого выражения следующий:

```
выражение1[выражение2][выражение3]...
```

Такое индексное выражение интерпретируется слева направо. Сначала вычисляется самое левое индексное выражение – выражение1[выражение2]. С адресом, полученным в результате сложения выражения1 и выражения2, складывается (по правилам сложения указателя и целого) выражение3 и т.д. Выражение2 и последующие выражения имеют целый тип.

Элементы многомерного массива запоминаются *построчно*.

Примеры:

```
char a[2][3];
float matrix[10][15];
int b[3][3]={ {1,2,3}, {4,5,6}, {7,8,9} };
```

В следующем примере выполняется определение, обработка и печать *статической* матрицы, имеющей заданную непосредственно в коде размерность:

```
#include <stdio.h>
void main () {
    int b[3][3]={ {1,2,3}, {1,2,3}, {1,2,3} };
    b[0][0]=2;
    b[2][2]=b[0][0]*4;
    for (int i=0; i<3; i++) {
        printf ("\n");
        for (int j=0; j<3; j++)
            printf ("%d ",b[i][j]);
    }
}
```

Пример ниже выполняет подсчёт количества разных цифр, пробелов и остальных символов, вводимых с клавиатуры. Для завершения программы служит ввод символа EOF (комбинация клавиш Ctrl+Z). Преобразование кода символа с в целочисленный индекс элемента массива ndigit выполняется конструкцией c- '0'.

```
#include <stdio.h>
void main() {
    int c, i, nwhite, nother;
    int ndigit[10];
    nwhite = nother = 0;
```

```

for (i = 0; i < 10; ++i) ndigit[i] = 0;
while ((c = getchar()) != EOF)
    if (c >= '0' && c <= '9') ++ndigit[c-'0'];
    else if (c==' ' || c=='\n' || c=='\t')
        ++nwhite;
    else ++nother;
printf("digits =");
for (i = 0; i < 10; i++)
    printf(" %d", ndigit[i]);
printf("\nwhite space = %d, other = %d\n",
    nwhite, nother);
}

```

*Формирование массива из случайных значений* уместно, если при решении задачи массив служит лишь для иллюстрации того или иного алгоритма, а конкретные значения элементов несущественны. Для того, чтобы получить очередное случайное значение, используется стандартная функция `rand()`. Она вернёт случайное число того же типа, что тип аргумента, которое лежит в диапазоне от 0 до значения `RAND_MAX` включительно. Значение `RAND_MAX` определено в `<stdlib.h>` (обычно  $2^{15}-1=32767$ ).

Чтобы при каждом запуске программы цепочка случайных чисел была новой, перед первым вызовом `rand()` следует выполнить команду `srand(time(NULL))`; запускающую генератор случайных чисел, в котором в качестве параметра используется системное время. Приведём пример заполнения массива из 10 элементов случайными числами, лежащими в диапазоне от 0 до 32767:

```

#include <ctime> // содержит time()

...
int x[10] = {};
srand(time(NULL));
for (int i = 0; i < 10; i++) {
    x[i] = rand();
    printf("%d\n", x[i]);
}

```

Чтобы получить значения, находящиеся в диапазоне  $[0; n-1]$ , необходимо изменить одну строчку программы:

```
x[i] = rand()%n;
```

С помощью арифметических операций сложения, вычитания, умножения и деления можно преобразовать результат данной операции, чтобы получить требуемый диапазон случайных значений.

Ещё более удобный путь – чтение элементов массива из текстового или двоичного файла. Об этом рассказывается в главах 8, 9.

**Обработка массива** (в данном случае вычисляется сумма  $S$  его положительных элементов):

```
float s=0.0, b[5]; int i;
printf("Enter 5 elements of array\n");
for(i=0; i<5; i++) scanf("%f", &b[i]);
for(i=0; i<5; i++) if(b[i]>0) s+=b[i];
printf("s=%f\n", s);
```

Как видно из этого примера, к массивам *применимы все типовые алгоритмы, изученные в теме "Циклы"* (см. главу 3).

Если обработка массива осуществляется последовательно, по одному элементу, циклы ввода и обработки зачастую можно объединить.

Особенности работы с адресами элементов массива в языке C++ рассмотрены в главе 6.

Перейдём к примерам.

**Пример 1.** Найти значение минимального элемента вещественного массива  $B$  размерностью 6.

```
double min=1E+30, b[6]; int i;
printf("Enter B[1..6]\n");
for(i=0; i<6; i++) {
    scanf("%lf", &b[i]);
    if(b[i]<min) min=b[i];
}
printf("min=%f\n", min);
```

Теоретически в этой программе можно было бы обойтись и без массива, поскольку элементы `b[i]` используются только для накопления суммы и поиска максимума, так что описание массива вполне можно было заменить описанием вещественной переменной `b`. Однако в реальных задачах данные, как правило, обрабатываются неоднократно и без массивов обойтись трудно. Занимая определённый объём оперативной памяти, однажды полученные элементы массива остаются доступными весь сеанс работы программы и не требуют повторного вычисления или чтения из файла. Это порождает круг приложений, связанных с типовой обработкой наборов данных, – вычисление математических и статистических характеристик векторов, объединение массивов или поиск нужных значений в них и т.д.

Вложение одних операторов цикла в другие позволяет обработать многомерные данные, что необходимо в большинстве реальных задач.

В теле цикла может находиться любой оператор, в том числе и другой оператор цикла. Такая конструкция называется *кратными* или *вложенными* циклами. Число вложений циклов не ограничено. Если цикл содержит только один вложенный цикл, то он называется *двойным*.

Цикл, который содержит вложенный цикл, называется *внешним*, вложенный цикл называется *внутренним*. Управляющая переменная внутреннего цикла всегда меняется *быстрее*, чем переменная внешнего. Это означает, что для каждого значения переменной внешнего цикла перебираются все значения переменной внутреннего. В случае вложения нескольких циклов это правило также действует: быстрее всего меняется переменная самого внутреннего цикла.

Поясним сказанное на примере. В конструкции

```
for(i=1;i<=5;i++) { //начало внешнего цикла
    j=-5;
    while(j<=5+1e-6) {
        //начало внутреннего цикла
        j=j+0.5;
    } //конец внутреннего цикла
} //конец внешнего цикла
```

при  $i=1$  переменная  $j$  последовательно примет значения  $-5, -4.5, \dots, 5$ , затем процесс повторится при  $i=2, 3, 4$  и  $5$ . Таким образом, за время выполнения *каждого шага* внешнего цикла `for` внутренний цикл `while` выполнится 21 раз.

Двойной цикл особенно широко распространён в программировании, так как идеально подходит для обработки табличной информации, работы с матрицами, задач сортировки и т.д. Рассмотрим несколько типовых задач, связанных с двойными циклами.

**Пример 2.** Найти, сколько раз каждый элемент встречается в массиве.

Первое, что приходит в голову при решении этой распространённой задачи – кроме массива данных ввести массив частот, элементы которого будут хранить информацию о встречаемости тех или иных значений. Однако решить проблему можно и без столь расточительных трат оперативной памяти (ведь размерность массива частот будет равна размерности исходного массива!). Каждый элемент массива  $a_i$ , кроме последнего, мы будем сравнивать со следующими за ним элементами  $a_j$ , считая, сколько раз выполнилось равенство  $a_i = a_j$ . Чтобы исключить повторную обработку уже встречавшихся элементов, учтённые значения нам придётся заменять нулями. Поскольку значение 0 само может встретиться в массиве, дополнительный цикл перед обработкой подсчитает количество нулевых элементов.

```
const int Size=10,Diap=10;
int a[Size],i,n,k,j;
srand(time(NULL));
a[0]=rand()%(Diap+1);
printf("A=\n%d ",a[0]);
for(i=1;i<Size;i++) {
    a[i]=rand()%(Diap+1);
    printf(" %d ",a[i]);
}
```

```

printf("\n\n");
k=0;
for(i=0;i<Size;i++) if(a[i]==0) k++;
if(k>0) printf("0:%d\n",k);
for(i=0;i<Size-1;i++) if(a[i]!=0) {k=1;
for(j=i+1;j<Size;j++) if(a[i]==a[j]) {
    a[j]=0;k++;
}
printf("%d:%d\n",a[i],k);}printf("\n");

```

Подобный принцип сравнения "каждый с каждым" используется при решении большого числа задач.

**Пример 3.** Координаты десяти точек на плоскости заданы двумя массивами  $X_i$ ,  $Y_i$ ,  $i=1, 2, \dots, 10$ . Найти две точки, расстояние между которыми минимально.

Нам придётся искать расстояние от первой точки до всех остальных, от второй – только до точек с номерами 3, ..., 10, так как расстояние до первой уже известно, и т.д. Попробуйте написать программу самостоятельно, применив двойной цикл из предыдущего примера.

В завершение темы рассмотрим задачу, где число вложений циклов больше двух.

**Пример 4.** Номера автобусных билетов состоят из 6 цифр. "Счастливыми" называют билеты, в номерах которых сумма трёх первых цифр равна сумме трёх последних. Определить общее количество "счастливых" билетов.

Каждая из шести цифр может меняться от 0 до 9 включительно. Создав кратные циклы по шести переменным, воспользуемся известным нам алгоритмом организации счётчика:

```

int i, j, k, l, m, n;
long int kol=0;
for(i=0;i<10;i++)
    for(j=0;j<10;j++)
        for(k=0;k<10;k++)
            for(l=0;l<10;l++)

```

```

    for(m=0;m<10;m++)
        for(n=0;n<10;n++)
            if(i+j+k==l+m+n) kol++;
    printf("The number of lucky \
tickets=%ld\n\n",kol);

```

В следующем примере будет рассмотрена обработка двумерного массива с помощью двойного цикла.

**Пример 5.** Определить количество элементов матрицы, по модулю не превышающих заданного значения  $V$ .

```

#include <stdio.h>
void main () {
    int s=0,v=2;
    int b[3][3]={{-1,2,-3},{4,-2,3},{1,5,0}};
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            if(abs(b[i][j])<=v) s++;
    printf("%d ",s);}

```

## Глава 5. Функции

**Функция** – это независимая часть программы, решающая некоторую частную задачу.

Сложная программа всегда строится как совокупность функций. Это позволяет реализовать следующие возможности:

- лёгкое повторное использование однажды написанного кода;
- лучшее структурирование кода за счёт разбиения задачи на независимые подзадачи;
- оптимизацию и уменьшение размеров кода за счёт неоднократного использования (вызова) в нём функций.

Общий вид описания функции следующий:

тип имя (тип1 параметр1, ..., типN параметрN)

Перед именем функции должно быть указано, какого типа значение она вычисляет (*возвращает*). Если функция не должна вычислять единственного ответа, для неё назначается пустой тип `void`. Такую функцию нельзя будет вызвать из выражения (справа от знака `=`).

Имя функции выбирается программистом и строится по тем же правилам, что и любой другой идентификатор. Каждый параметр (*аргумент*) функции может быть любого из разрешённых в языке или добавленных программистом типов данных:

```
int count (int a, int b)
```

Здесь функция с именем `count` вычисляет целочисленную величину и имеет два целочисленных параметра, обозначенных `a` и `b`. Функция обменивается данными с программой только через свои параметры.

Если параметр *выходной* (его значение должно вычисляться внутри функции), то перед его именем ставится знак `&`:

```
void equation (float a, float b, float c,  
float &x1, float &x2)
```

Функция с именем `equation` здесь имеет три входных параметра (`a`, `b`, `c`) и два выходных, обозначенных `x1` и `x2`.

Тело функции, как и тело оператора цикла, заключается в операторные скобки `{ }`.

Например, показанная ниже функция `distance` может вычислять расстояние от точки на плоскости с координатами  $(x, y)$  до начала координат  $(0, 0)$ :

```
float distance (float x, float y) {  
    float r=sqrt(pow(x,2)+pow(y,2));  
    return r; //Вернуть из функции ответ  
}
```

Переменная, содержащая ответ (вычисленное функцией значение), должна быть именно того типа данных, который указан перед именем функции. После ключевого слова `return` можно записать также выражение, дающее при вычислении соответствующий тип данных, поэтому тело функции могло бы выглядеть и так:

```
return sqrt(pow(x,2)+pow(y,2));
```

Конечно, чаще всего функции состоят не из одного оператора, а из нескольких.

Написание функции ещё *не означает* её выполнения. Для выполнения функцию надо *вызвать*, указав её имя и список *фактических* параметров, количество и типы которых соответствуют *формальным* параметрам в заголовке функции. Например, записанную выше функцию мы могли бы вызвать таким образом:

```
float x1=3.5,y1=-1;  
float d1 = distance (x1,y1);
```

Здесь формальные параметры `x` и `y` указаны в заголовке функции, а фактические параметры `x1` и `y1` будут подставлены вместо `x` и `y` при вызове функции.

Следующий способ вызвать ту же функцию:

```
float d2 = distance (1.,1.5);
```

Здесь фактическими параметрами служат вещественные константы `1.` и `1.5`, они будут использованы функцией в качестве значений `x` и `y`.

А вот этот оператор ошибочен:

```
float d3 = distance (x1,y1,1.);
```

Здесь передаётся три фактических параметра, но функция готова принять только два.

Если функция реализована (записана) *выше* по тексту программы, чем она вызвана, никаких дополнительных действий для правильной работы этого вызова предпринимать не нужно. Если это не так, следует указать *прототип* (заголовок) функции выше по тексту программы, чем она вызвана:

```
int summa (int a, int b) { return a+b; }
int main () {
    int s1=summa (3,7);
}
```

В этой программе функция `summa` будет успешно вызвана из главной функции `main`. Во второй программе, где тело функции `raznost` записано после тела `main`, при отсутствии прототипа возникнет ошибка компиляции:

```
int raznost(int,int); //это прототип
int main(){
    //Если функцию использует только main,
    //прототип можно было бы записать и здесь
    int r1=raznost (5,1);
}
int raznost (int a,int b) { return a-b; }
```

Обычно прототипы всех функций программного модуля собраны в файл с расширением `.h` и подключены к программе оператором вида

```
#include "unit1.h"
```

где `"unit1"` – имя программного модуля. Реализации функций модуля будут в этом случае содержаться в файле с именем `unit.cpp`.

Если перед прототипом функции указано ключевое слово `extern`, это означает, что функция подключается из другого файла проекта.

**Пример 1.** Напишем функцию для проверки номера координатной четверти точки на плоскости. После этого выведем номера четвертей для двух заданных точек.

Сначала определимся с аргументами и типом функции:

- два входных параметра, координаты точки  $x$  и  $y$  (вещественные);

- один выходной параметр, целый номер четверти n, принимает значения от 1 до 4 включительно или 0, если хотя бы одна из координат равна нулю.

Так как ответ, вычисляемый функцией, всего один, он может быть возвращён непосредственно через оператор return.

```
#include <stdio>
int quadrant (float x, float y) {
    int n=0;
    if (x>0) {
        if (y>0) n=1;
        else if (y<0) n=4;
    }
    else if (x<0) {
        if (y>0) n=2;
        else if (y<0) n=3;
    }
    return n;
}
int main () {
    float x1,y1,x2,y2;
    printf ("\nВведите точку 1:");
    fflush (stdin); scanf ("%f %f",&x1,&y1);
    printf ("\nВведите точку 2:");
    fflush (stdin); scanf ("%f %f",&x2,&y2);
    int n1=quadrant(x1,y1);
    printf ("\nОТвет 1 = %d", n1);
    printf ("\nОТвет 2 = %d", quadrant
(x2,y2));
    getchar(); return 0;
}
```

По коду видно, что в такой программе была бы полезна ещё и функция для ввода с клавиатуры координат точки:

```
void point_input (float &x, float &y) {
    printf ("\nВведите коорд X:");
    fflush (stdin); scanf ("%f",&x);
```

```

printf ("\nВведите коорд Y:");
fflush (stdin); scanf("%f",&y);
return; //можно не писать для void ф-ций
}

```

Её вызов в main мог бы быть таким:

```

point_input (x1,y1);
point_input (x2,y2);

```

Главная функция превратилась бы в последовательность вызова функций приложения, решающих его частные задачи. Именно так обычно устроена хорошо написанная программа.

**Пример 2.** Так как в C/C++ размерность массива нигде не хранится, при передаче одномерного массива (вектора) в функцию следует использовать *два* параметра – целочисленную размерность вектора и адрес его нулевого элемента.

Показанная далее функция func заполняет массив нужной размерности последовательно идущими числами 1, 2, 3, ...:

```

void func (int n, int a[]) {
    //В функции работаем с массивом int a[n]
    for (int i=0; i<n; i++) a[i]=i+1;
}

```

Дважды вызовем эту функцию для различных массивов:

```

int a[3]; func (3, &a[0]);
int b[5]; func (5, b);

```

Прототип функции может иметь вид

```

void func (int, int []);

```

**Пример 3.** Найти максимальную из длин векторов  $a = (4, 5, 6, 3)$  и  $b = (1, 2, 3, 4, 5)$ . Длину вектора будем понимать в обычном декартовом смысле: как квадратный корень из суммы квадратов всех элементов вектора.

```

#include <stdio>
#include <math.h>
float length (int n, float v[]) {
    float d=0.;
    for (int i=0; i<n; i++) d += v[i]*v[i];
    return sqrt(d);
}

```

```

int main () {
    float a[] = {4, 5, 6, 3};
    float b[] = {1, 2, 3, 4, 5};
    float d1 = length (4,&a[0]);
    float d2 = length (5,b);
    printf ("\nResult=%f", (d1>d2?d1:d2));
    getchar(); return 0;
}

```

Для передачи двумерной *матрицы* в функцию существует два основных способа.

**Способ 1.** Передать две размерности (количество строк и столбцов матрицы) плюс двойной указатель, если матрица динамическая:

```

void func (int n, int m, int **a) {
    //n - количество столбцов,
    //m - количество строк,
    //**a - ссылка на данные матрицы
    //в функции работаем с матрицей a[n][m]
}
//Код в main:
int **a; //ссылка на матрицу
const int n=4, m=3;
    //количество строк и столбцов
a = new int * [n];
for (int i=0; i<n; i++) a[i] = new int [n];
//выделена память под a[n][m],
//затем элементы матрицы в двойном цикле
//вводятся с клавиатуры или иным способом

```

**Способ 2.** Описать матрицу как вектор исходя из следующих соображений. При заданной в задаче матрице  $a[n][m]$  определим вместо неё вектор  $v[n*m]$ . Тогда обращению к элементу матрицы  $a[i][j]$  соответствует обращение к элементу вектора  $v[i*m+j]$ .

## Глава 6. Указатели и строки Си

**Указатель** – это переменная, содержащая адрес некоторого объекта в оперативной памяти (ОП). Смысл использования указателей – косвенная адресация объектов в ОП, позволяющая динамически менять логику программы и управлять распределением ОП.

Основные области применения указателей:

- работа с классическими массивами и строками символов;
- прямой доступ к ОП;
- работа с динамическими объектами, под которые выделяется ОП.

Описание указателя имеет следующий общий вид:

тип \*имя;

т.е. указатель всегда *адресует определённый тип объектов!*

Например,

```
int *px;
    // указатель на целочисленные данные
char *s;
    //указатель на тип char (строку Си)
```

Опишем основные операции и действия, которые разрешены с указателями:

**Операция 1.** Сложение или вычитание с числом:

```
px++; //переставить указатель px
      //на sizeof(int) байт вперёд
s--;
      //перейти к предыдущему символу строки
      //(на sizeof(char) байт назад)
```

**Операция 2.** Указателю можно присваивать адрес объекта унарной операцией & (*взять адрес*):

```
int *px; int x,y;
px=&x;
    //теперь px показывает на ячейку памяти
    // со значением x
px=&y;
    //а теперь - на ячейку со значением y
```

**Операция 3.** Значение переменной, на которую показывает указатель, берётся унарной операцией *\** (*взять значение*):

```
x=*px;  
    //косвенно выполнили присваивание x=y  
(*px)++;  
    //косвенно увеличили значение y на 1
```

Важно, что из-за приоритетов и ассоциативности операций C++ действие

```
*px++;
```

имеет совсем другой смысл, чем предыдущее. Оно означает "взять значение y (*\*px*) и затем перейти к следующей ячейке памяти (*++*)". Расшифруем оператор

```
x=*px++;
```

Если *px* по-прежнему указывал на *y*, оператор означает "записать значение *y* в *x* и затем перейти к ячейке памяти, следующей за *px*". Именно такой подход в классическом C используется для сканирования массивов и строк.

Приведём пример связывания указателя со статическим массивом:

```
int a[5]={1,2,3,4,5};  
int *pa=&a[0];  
for (int i=0; i<5; i++)  
    cout << *pa++ << " ";  
for (int i=0; i<5; i++)  
    cout << pa[i] << " ";
```

Выделенные жирным шрифтом записи абсолютно эквивалентны, потому что в языке C конструкция *a[b]* означает не что иное, как *\*(a+b)*, где *a* – объект, *b* – смещение от начала памяти, адресующей объект. Таким образом, обращение к элементу массива *a[i]* может быть записано и как *\*(a+i)*, а присваивание указателю адреса нулевого элемента массива можно было бы записать в любом виде из четырёх:

```
int *pa=&a[0];  
int *pa=&(*(a+0));  
int *pa=&(*a);  
int *pa=a;
```

Существенно, что при любом способе записи это одна и та же операция, и она не является "присваиванием массива указателю", но лишь *установкой* указателя на нулевой элемент массива.

**Операция 4.** Сравнение указателей (вместо сравнения значений, на которые они указывают) в общем случае может быть некорректно:

```
int x;  
int *px=&x, *py=&x;  
if (*px==*py) ... //корректно  
if (px==py) ...   //некорректно!
```

Причина в том, что адресация ОП в операционной системе не обязана быть однозначной, например, в DOS одному адресу памяти могли соответствовать разные пары частей адреса – сегмент и смещение. В Windows адресация памяти линейная, и сравнение указателей обычно работает корректно.

**Операция 5.** Указатели и ссылки могут использоваться для передачи функциям аргументов *по адресу* (т.е. для выходных параметров функций). Для этого существует два способа.

**Способ 1**, со ссылочной переменной C++:

```
void swap (int &a, int &b) {  
    int c=a; a=b; b=c;  
}  
//...  
int a=3,b=5; swap (a,b);
```

Этот способ можно назвать *передача параметров по значению, приём по ссылке*.

**Способ 2**, с указателями классического языка C:

```
void swap (int *a, int *b) {  
    int c=*a; *a=*b; *b=c;  
}  
//...  
int a=3,b=5; swap (&a,&b);  
int *pa=&a; swap (pa,&b);
```

Способу соответствует *передача параметров по адресу и приём по значению*.

Одно из распространённых применений указателей – сканирование Си-строк:

```
char *s="Hello, World";
```

Это установка указателя на первый байт строковой константы, а не копирование и не присваивание.

Существенно, что, даже если размер символа `sizeof(char)` в системе равен одному байту, эта строка займёт не 12 (11 символов и пробел), а 13 байт памяти. Дополнительный байт нужен для хранения *нуль-терминатора*, символа с кодом 0, записываемого как `'\0'` (но не `'0'` – это цифра 0 с кодом 48). Многие функции работы с Си-строками автоматически добавляют нуль-терминатор в конец обрабатываемой строки:

```
char s[12];  
strcpy(s, "Hello, World");
```

В данном случае вызов стандартной функции копирования строки ошибочен, так как нет места для нуль-терминатора. Верным было бы описание

```
char s[13];
```

Длина Си-строки нигде не хранится, её можно только узнать стандартной функцией `strlen(s)`, где `s` – указатель типа `char *`. Для строки, записанной выше, будет возвращено значение 12, нуль-терминатор при определении длины строки не считается. Фактически Си-строка есть массив символов, элементов типа `char`.

Как выполнять другие операции со строками, заданными с помощью указателей `char *`? Для этого может понадобиться сразу несколько стандартных библиотек. Как правило, в новых компиляторах C++ можно подключать и классические Си-совместимые заголовочные файлы, и заголовки из более новых версий стандарта, которые ниже указаны в скобках.

**Модуль `cctype.h` (`cctype`)** содержит:

1. Прототипы функций с шаблоном имени `is*`. Они отвечают за проверку класса отдельных символов (`isalpha` – является ли символ латинской буквой, `isdigit` – цифрой и т.д.). Все функции возвращают целое число, например:

```
char d;
if (isdigit(d)) {
    //код для ситуации, когда d - цифра
}
```

Аналогичная проверка без стандартной функции могла бы быть выполнена кодом вида

```
if (d>='0' && d<='9') { /* ... */ }
```

2. Прототипы функций с шаблоном имени `to*`. Их назначение – преобразование регистра символов (например, `toupper` – к верхнему регистру). Функции возвращают преобразованный символ. Могут быть бесполезны при работе с символами национальных алфавитов.

**Модуль `string.h` (`cstring`)** предназначен для работы со строками, заданными указателем и заканчивающимися байтом `'\0'` (строками Си). Имена большинства его функций начинаются на `str`. Часть функций с именами на `mem` (`memcpy`, `memmove`, `memset`) подходит для работы с *буферами* (областями памяти с известным размером).

Разберём на примерах основные действия со строками, заданными указателями. При этом будем использовать стандартные функции модуля `<string.h>`.

**Пример 1.** Копирование строки.

```
char *s="Test string";
char s2[80];
strcpy (s2,s);
```

Обратите внимание, что `s2` – *буфер*, а не указатель. Распространённая ошибка начинающих – игнорирование того, что память, в которую мы копируем данные, должна быть статическим буфером достаточного для копирования размера или же выделена динамически самой программой.

**Пример 2.** Копирование строки с ограничением количества символов.

```
char *s="Test string";
char s2[80];
char *t=strncpy (s2,s,strlen(s));
cout << t;
```

Функция `strncpy` копирует не более `n` символов (значение `n` указывается третьим параметром функции), но не запишет нуль-терминатор, в результате чего в конце строки `t` могут оказаться случайные символы. Правильно было бы добавить после `strncpy` следующее действие:

```
t[strlen(s)]='\0';
```

т.е. непосредственную запись нуль-терминатора, второй вариант – копировать `strlen(s)+1` символов.

**Пример 3.** Копирование строки в новую память.

```
char *s="12345";  
char *s2=new char [strlen(s)+1];  
strcpy (s2,s);
```

Здесь мы безопасно скопировали строку `s` в новую память `s2`, не забыв выделить "лишний" байт для нуль-терминатора.

**Пример 4.** Напишем собственную реализацию стандартной функции `strcpy`:

```
char *strcpy_ (char *dst, char *src) {  
    char *r=dst;  
    while (*src!='\0') {  
        *dst=*src; dst++; src++;  
    }  
    *dst='\0';  
    return r;  
}
```

Вызвать эту функцию можно, например, так:

```
char *src="Строка текста";  
char dst[80];  
strcpy_ (&dst[0],&src[0]);
```

Сократим код функции `strcpy_`:

```
char *strcpy_ (char *dst, char *src) {  
    char *r=dst;  
    while (*src) *dst++=*src++;  
    *dst='\0';  
    return r;  
}
```

**Пример 5.** Сцепление строк с помощью функции `strcat`. Сначала воспроизведём типичную ошибку начинающих:

```
char *s="Test string";  
char *p = strcat(s, " + new words");
```

Так как функция `strcat` не выделяет память, поведение такого кода непредсказуемо и с вероятностью 99% приведёт к ошибке времени исполнения программы. А вот такое сцепление строк сработает:

```
char *s="Test string";  
char s2[80];  
strcpy (s2,s);  
char *p=strcat (s2, " + new words");
```

То есть *всегда должна быть память, куда писать*, – статическая из буфера или выделенная динамически.

**Пример 6.** Поиск символа или подстроки в строке. Покажем применение стандартных функций `strchr` (поиск первого вхождения символа), `strrchr` (поиск последнего вхождения символа) и `strstr` (поиск подстроки).

```
char *s="Test string";  
char *sym = strchr (s, 't');  
if (sym==NULL) puts ("Не найдено");  
else puts (sym); //выведет "t string"  
//для strrchr вывод был бы "tring"  
char *sub = strstr (s, "ring");  
puts (sub); //выведет "ring"
```

**Пример 7.** Покажем, как работает сравнение строк. Для этой цели служат функции с шаблоном имени `str*cmp` (от англ. *string comparing*).

```
char *a="abcd", *b="abce";  
int r=strcmp(a,b);
```

Значение `r` будет равно `-1`, так как символ `'d'` предшествует символу `'e'` в кодовой таблице. Соответственно, вызов `strcmp(b, a)` вернёт в данном случае значение `1`. Если строки совпадают, результат их сравнения равен `0`.

**Пример 8.** Преобразование числа в строку и строки в число. Для решения этой задачи можно использовать функции биб-

лиотеки `stdlib.h` (`cstdlib`). К сожалению, классические функции `atoi` (преобразование строки в целое число) и `atof` (преобразование строки в вещественное число) не генерируют исключений, а просто возвращают 0, если преобразование невозможно:

```
char *s="qwerty";
int i=atoi(s); //i=0
```

Решить обратную задачу, т.е. преобразовать числовое значение в строку, можно с помощью функций `itoa` (преобразование целого числа в строку Си), `ultoa` (преобразование длинного беззнакового целого в строку Си), а также группы функций с именами `fcvt`, `gcvt`, `ecvt`, предназначенных для преобразования чисел вещественных типов.

```
char buf[20];
int i=-31189;
char *t=itoa(i,buf,36);
```

В данном случае в буфере `buf` мы получили запись значения `i` в системе счисления с основанием 36.

В библиотеке `string.h` есть также готовые функции для разбора строк на лексемы, такие как `strtok`, `strspn`, `strcspn` [1, п. 7.1–7.3].

Одним из основных применений указателей служит *работа с динамической памятью*. Как правило, в программе описывается указатель нужного типа, который затем связывается с областью памяти, выделенной оператором C++ `new` или Си-совместимыми функциями для управления ОП.

Работа с динамической памятью состоит из следующих типовых действий:

1. Описать типизированный указатель на будущий динамический объект:

```
int *a;
```

Зачастую надёжнее при описании инициализировать указатель пустым значением, которое в разных компиляторах может обозначаться константой со значением `NULL`, `null`, `nullptr` или 0 из стандартной библиотеки `stdlib.h` (`cstdlib`):

```
int *a=NULL;
```

2. Оператором `new` или функциями `malloc`, `calloc` выделить нужный объём ОП:

```
a = new int [10];
```

или

```
#include < stdlib.h>
```

```
//...
```

```
a = (int *) malloc (sizeof(int)*10);
```

В некоторых компиляторах прототипы Си-совместимых функций работы с ОП описаны также в файлах `malloc.h` или `alloc.h`.

Стандартная функция `C calloc` позволяет не только выделить память, но и заполнить её байтами со значением `'\0'`:

```
a = (int *) calloc (10,sizeof(int));
```

Не рекомендуется смешивать эти два способа в одном программном модуле или проекте! В общем случае предпочтительнее оператор `new`, кроме тех ситуаций, когда нужно обеспечить заполнение памяти нулевыми байтами.

3. Проверить, удалось ли выделить память, если нет, указатель равен константе `NULL`:

```
if (a==NULL) {  
    //Обработка ошибки  
    //"Не удалось выделить память"  
}
```

4. Обработка динамического массива или строки ничем не отличается от случая, когда они статические, и выполняется поэлементно. Например, здесь динамический массив заполняется значениями 1, 2, ..., 10:

```
for (int i=0; i<10; i++) a[i]=i+1;
```

5. Когда выделенная ОП больше не нужна, её нужно освободить:

```
delete a; //Если использовали new
```

или

```
free (a);
```

Эта функция пытается освободить ОП, если для её выделения применялась функция `malloc` или `calloc`.

Всегда старайтесь придерживаться принципа стека при распределении ОП, т.е. объект, занявший ОП последним, первым её освобождает.

**Пример 9.** Динамическая матрица размерностью  $n \times m$ .

```
const int n=5,m=4;
int **a = new (int *) [n];
for (int i=0; i<n; i++) a[i] = new int [m];
```

После этого можно работать с элементами матрицы  $a[i][j]$ , например присваивать им значения. Освободить память можно так:

```
for (int i=n-1; i>-1; i--) delete a[i];
delete a;
```

### **Задачи к лабораторной работе**

1. Написать собственную функцию работы со строкой, заданной указателем, сравнить работу функции со стандартной функцией, если последняя существует.

2. Написать собственную функцию для работы с одномерным динамическим массивом, заданным указателем.

3. Написать свои версии функций преобразования строки в вещественное или целое число и вещественного или целого числа в строку.

## Глава 7. Строки класса `string`

В современном стандарте C++ определён класс с функциями (*методами*) и переменными (*свойствами*), предназначенными для организации работы со строками. В классическом языке C строк как таковых нет, есть лишь массивы символов `char`. Для начала работы с классом `string` требуется подключить следующий заголовочный файл:

```
#include <string>
```

Для работы со строками также нужно подключить стандартное пространство имён:

```
using namespace std;
```

В противном случае придётся везде указывать описатель класса `std::string` вместо `string`.

Ниже приводится пример программы, работающей с объектами класса `string` (в старых Си-совместимых компиляторах этот код не работает):

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

int main () {
    string s = "Test";
    s.insert (1, "!");
    cout << s.c_str() << endl;
    string *s2 = new string("Hello");
    s2->erase(s2->end());
    cout << s2->c_str();
    cin.get(); return 0;
}
```

Основные возможности, которыми обладает класс `string`:

- инициализация массивом символов (строкой встроеного типа) или другим объектом типа `string`. Встроенный тип не обладает второй возможностью;

- копирование одной строки в другую. Для встроенного типа приходится использовать функцию `strcpy`;
- доступ к отдельным символам строки для чтения и записи. Во встроенном массиве для этого применяется операция взятия индекса или косвенная адресация с помощью указателя;
- сравнение двух строк на равенство. Для встроенного типа используются функции семейства `strcmp`;
- *конкатенация* (сцепление) двух строк. Для встроенного типа применяется функция `strcat`, однако, чтобы получить результат в новой строке, необходимо последовательно задействовать функции `strcpy` и `strcat`, а также позаботиться о выделении памяти;
- встроенные средства определения длины строки (функции-члены класса `size` и `length`). Узнать длину строки встроенного типа можно только вычислением с помощью функции `strlen`;
- узнать, пуста ли строка.

Рассмотрим эти базовые возможности более подробно.

*Инициализация* строк при описании и вычисление длины строки (не включая завершающий нуль-терминатор) выполняется следующим образом:

```
string st("Моя строка\n");
cout << "Длина " << st << ": " << st.size()
      << " символов, включая \\n";
```

Строка может быть задана пустой:

```
string st2;
```

Для проверки того, пуста ли строка, можно сравнить её длину с нулём:

```
if (!st.size()) // пустая
```

Альтернатива – применить метод `empty()`, возвращающий значение `true` для пустой строки и `false` для не пустой:

```
if (st.empty()) // пустая
```

Третья форма создания строки инициализирует объект типа `string` другим объектом того же типа:

```
string st3 (st);
```

Строка `st3` инициализируется строкой `st`. Как мы можем убедиться, что эти строки совпадают? Воспользуемся оператором сравнения (`==`):

```
if (st == st3) // инициализация сработала
```

Скопировать одну строку в другую можно с помощью обычной операции присваивания:

```
st2 = st3; // копируем st3 в st2
```

Для сцепления строк используется операция сложения (+) или операция сложения с присваиванием (`+=`). Пусть даны две строки:

```
string s1("hello, ");  
string s2("world\n");
```

Мы можем получить третью строку, состоящую из первых двух, таким образом:

```
string s3 = s1 + s2;
```

Если же мы хотим добавить `s2` в конец `s1`, мы должны написать:

```
s1 += s2;
```

Операция сложения может сцеплять объекты класса `string` не только между собой, но и со строками встроенного типа. Можно переписать пример, приведённый выше, так, чтобы специальные символы и знаки препинания представлялись встроенным типом `char *`, а значимые слова – объектами класса `string`:

```
const char *pc = ", ";  
string s1("hello");  
string s2("world");  
string s3 = s1 + pc + s2 + "\n";  
cout << endl << s3;
```

Подобные выражения работают потому, что компилятор "знает", как автоматически преобразовывать объекты встроен-

ного типа в объекты класса `string`. Возможно и простое присваивание встроенной строки объекту `string`:

```
string s1;  
const char *pc = "a character array";  
s1 = pc; // правильно
```

Обратное преобразование при этом не работает. Попытка выполнить следующую инициализацию строки встроенного типа вызовет ошибку компиляции:

```
char *str = s1; // ошибка компиляции
```

Чтобы осуществить такое преобразование, необходимо явно вызвать функцию-член с названием `c_str()` (строка Си):

```
const char *str = s1.c_str();
```

Функция `c_str()` возвращает указатель на символьный массив, содержащий строку объекта `string` в том виде, в каком она находилась бы во встроенном строковом типе. Ключевое слово `const` здесь предотвращает опасную в современных визуальных средах возможность непосредственной модификации содержимого объекта через указатель.

К отдельным символам объекта типа `string`, как и встроенного типа, можно обращаться с помощью операции взятия индекса. Вот, например, фрагмент кода, заменяющего все точки символами подчёркивания:

```
string str ("www.disney.com");  
int size = str.size();  
for (int i = 0; i < size; i++)  
    if (str[i] == '.') str[ i ] = '_';  
cout << str;
```

Альтернативный путь решения подобных задач – встроенные в современный стандарт C++ готовые *алгоритмы*. Например, предыдущее действие мы могли бы выполнить вызовом одной-единственной функции `replace()`:

```
replace(str.begin(), str.end(), '.', '_');
```

Здесь использован не метод `replace` класса `string`, а одноимённый алгоритм:

```
#include <algorithm>
```

Поскольку объект `string` ведёт себя как контейнер, к нему могут применяться и другие алгоритмы. Это позволяет решать задачи, не решаемые напрямую функциями класса `string`.

В табл. 3 приводится краткое описание основных операторов и функций класса `string`. Более полный список возможностей класса можно посмотреть, например, в [8, 9].

Таблица 3

Основные методы и операторы класса `string`

| <b>Задание символов в строке</b>                |                                                                          |
|-------------------------------------------------|--------------------------------------------------------------------------|
| <code>operator=</code>                          | Присваивает значения строке                                              |
| <code>assign</code>                             | Назначает символы строке                                                 |
| <b>Доступ к отдельным символам</b>              |                                                                          |
| <code>at</code>                                 | Получает указанный символ и проверяет выход индекса за границы           |
| <code>operator[]</code>                         | Получает указанный символ                                                |
| <code>front</code>                              | Получает первый символ                                                   |
| <code>back</code>                               | Получает последний символ                                                |
| <code>data</code>                               | Возвращает указатель на первый символ строки                             |
| <code>c_str</code>                              | Возвращает немодифицируемый массив символов C, содержащий символы строки |
| <b>Проверка на вместимость строки</b>           |                                                                          |
| <code>empty</code>                              | Проверяет, является ли строка пустой                                     |
| <code>size</code><br>или<br><code>length</code> | Возвращает количество символов в строке                                  |
| <code>max_size</code>                           | Возвращает максимальное количество символов                              |
| <code>reserve</code>                            | Резервирует место под хранение                                           |

| <b>Операции над строкой</b>    |                                                 |
|--------------------------------|-------------------------------------------------|
| <code>clear</code>             | Очищает содержимое строки                       |
| <code>insert</code>            | Вставляет символы                               |
| <code>erase</code>             | Удаляет символы                                 |
| <code>push_back</code>         | Добавляет символ в конец строки                 |
| <code>pop_back</code>          | Удаляет последний символ                        |
| <code>append</code>            | Добавляет символы в конец строки                |
| <code>operator+=</code>        |                                                 |
| <code>compare</code>           | Сравнивает две строки                           |
| <code>replace</code>           | Заменяет каждое вхождение указанного символа    |
| <code>substr</code>            | Возвращает подстроку                            |
| <code>copy</code>              | Копирует символы                                |
| <code>resize</code>            | Изменяет количество хранимых символов           |
| <code>swap</code>              | Обменивает содержимое                           |
| <b>Поиск в строке</b>          |                                                 |
| <code>find</code>              | Находит символы в строке                        |
| <code>rfind</code>             | Находит последнее вхождение подстроки           |
| <code>find_first_of</code>     | Находит первое вхождение символов               |
| <code>find_first_not_of</code> | Находит первое вхождение отсутствия символов    |
| <code>find_last_of</code>      | Находит последнее вхождение символов            |
| <code>find_last_not_of</code>  | Находит последнее вхождение отсутствия символов |

Следует обратить внимание на то, что у любой функции класса `string` может быть несколько *перегрузок* – разновидностей с одинаковыми именами, отличающихся между собой списками и типами аргументов.

В качестве недостатков объекта `string` можно отметить:

- отсутствие в классе встроенных средств для разбора строк по набору разделителей (аналога функции `strtok` для строк `char *`);

- возможное замедление быстродействия по отношению к `char *` при сложной обработке данных.

### **Задание к лабораторной работе**

Ниже приведён код для разбора введённой с клавиатуры строки `string` на слова. Доработать код, исключив знаки препинания, стоящие последними символами строк `vecstr[i]`, а также слова, не содержащие ни одного алфавитно-цифрового символа.

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
using namespace std;

int main() {
    cout << "Enter the string: ";
    string str;
    getline(cin, str);

    vector <string> vecstr;
    string word;
    stringstream s(str);

    while (s >> word) vecstr.push_back(word);

    int vsize = vecstr.size();
    for (int i = 0; i < vsize; i++)
        cout << vecstr[i] << endl;

    cin.get();
    return 0;
}
```

При написании программы дополнительно использованы возможности стандартных классов `vector` [10] и `stringstream` [11].

Можно реализовать классический разбор на слова (*парсинг*) с помощью встроенных методов класса `string`, в простейшем случае так:

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

int main() {
    string s0 =
        " I'm a\text from some words. ";
    string c,c0,w;
    bool inword=false;
    string divs=" \t\r\n";
    string s=" "+s0+" ";
    int len = s.size(), start;
    for (int i=1; i<len; i++) {
        string c0 = s.substr(i-1,1);
        string c = s.substr(i,1);
        if (divs.find(c0)!=string::npos &&
            divs.find(c)==string::npos) {
            inword=true; start=i;
        }
        if (divs.find(c)!=string::npos) {
            if (inword) {
                w = s.substr(start,i-start+1);
                //здесь есть очередная лексема w
                cout << w << endl;
            }
            inword = false;
        }
    }

    cin.sync(); cin.get(); return 0;
}
```

## Глава 8. Работа с текстовыми и бинарными файлами

В главе рассмотрен классический способ работы с файлами в C/C++, использующий библиотеку `stdio.h` и доступ к данным через структуру `FILE`. Альтернативный современный механизм работы с файлами в языке C++ на основе потоков и библиотек `<fstream>`, `<ifstream>`, `<ofstream>` будет изучен в следующей главе.

Базовые функции для работы с файлами описаны в библиотеке `stdio.h`. Вся работа с файлом выполняется через *файловую переменную*-указатель на структуру типа `FILE`, определённую в стандартной библиотеке:

```
FILE *fp;
```

Открыть файл можно функцией `fopen`, имеющей два параметра:

```
FILE *fopen  
(char *имя_файла, char *режим_доступа)
```

Параметр `имя_файла` может содержать относительный или абсолютный путь к открываемому файлу. Приведём примеры его значений:

- `"data.txt"` – открывается файл с именем `data.txt` из текущей папки. Следует понимать, что, при запуске исполняемого файла приложения не из IDE, текущая папка – та, где находится исполняемый файл. При отладке в IDE папка может быть иной, например, в Visual Studio при открытом консольном решении с именем `Console`, содержащем одноимённый проект, файл следует разместить в папке `Console/Console`, а при запуске исполняемого файла не из IDE – в папке `Console/Debug`;

- `"f:\\my.dat"` – открывается файл `my.dat` из головной папки диска `f:`. Так как обратный слеш `'\\'` в C/C++ является служебным символом, при записи пути к файлу его нужно удваивать.

Имя файла также может быть запрошено у пользователя:

```
char buf[80];  
printf ("\nВведите имя файла:");  
fflush (stdin);  
gets (buf);
```

Параметр `режим_доступа` определяет, какие действия будут разрешены с открываемым файлом. Примеры его возможных значений:

- `"rt"` – открывается для чтения текстовый файл;
- `"r+b"` – открывается для произвольного доступа (как чтения, так и записи) бинарный файл;
- `"at"` – открывается текстовый файл для добавления данных в конец файла;
- `"w"` – открывается файл для новой записи без указания того, текстовый он или бинарный.

По факту указание режима доступа `"b"` или `"t"` не накладывает каких-либо ограничений на методы, которые будут применяться для чтения или записи данных.

После открытия файла следует обязательно проверить, удалась ли эта операция. Для этого существует два основных подхода:

- 1) использовать стандартный обработчик ошибок `ferror` ([1, с. 76–77]);
- 2) сравнить указатель, который вернула `fopen`, с константой `NULL` (`nullptr`) стандартной библиотеки.

Во втором случае код может быть таким:

```
fp = fopen ("text.txt", "r+b");  
if (fp==NULL) {  
    //Обработка ситуации "Не удалось открыть"  
}
```

**Пример 1.** Приложение проверяет, удалось ли открыть файл из текущей папки, имя файла запрашивается у пользователя (код для Visual Studio).

```
#include <windows.h>  
#include <locale.h>  
#include <stdio.h>  
#include <stdlib.h>
```

```

int main(void) {
    setlocale(LC_ALL, "Rus");
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    FILE *fp;
    char buf[80];
    printf ("\nВведите имя файла:");
    fflush (stdin); gets (buf);
    fp = fopen (buf, "r+b");
    if (fp==NULL) {
        printf ("\nне удалось открыть файл");
        getchar();
        exit (1); //Выйти с кодом завершения 1
    }
    fflush(stdin); getchar(); return 0;
}

```

Функции, возвращающие указатель, в том числе `fopen`, считаются небезопасными в ряде новых компиляторов, например в Visual Studio 2015. Если их использование приводит не к предупреждению, а к генерации ошибок, есть два основных способа решения проблемы:

1. В соответствии с рекомендациями компилятора, заменить старые названия функций на их безопасные версии, например `strcpy` на `strcpy_s` и `fopen` на `fopen_s`. При этом может измениться и способ вызова функций, например,

```
FILE *out; fopen_s(&out, "data.txt", "wt");
```

вместо

```
FILE *out = fopen_s("data.txt", "wt");
```

2. В начале файла (до всех `#include`) указать директиву `#define _CRT_SECURE_NO_WARNINGS`

Если используется предкомпиляция, то можно определить этот макрос в заголовочном файле `stdafx.h`.

Выбор *способа чтения или записи данных* зависит от того, какой должна быть структура файла.

Если файл *форматированный*, т.е. является текстовым и состоит из лексем, разделённых стандартными разделителями

(пробел, табуляция, перевод строки), обмен данными с ним можно выполнять следующими методами:

- `fscanf` – для чтения;
- `fprintf` – для записи.

Первым параметром этих функций указывается файловая переменная, в остальном работа совпадает со стандартными `scanf` и `printf`.

**Пример 2.** Файл `text.txt` в текущей папке приложения имеет следующий вид:

```
1 1.5 -3.5
2 3.5
```

Прочитаем его как последовательность вещественных чисел.

```
FILE *fp = fopen ("text.txt", "r");
if (fp==NULL) {
    printf ("\nНе удалось открыть файл");
    getchar(); exit (1);
}
float a;
while (1) {
    fscanf (fp, "%f", &a);
    if (feof(fp)) break;
    //обработка очередного значения a
    printf ("%0.2f ", a);
}
fclose(fp);
```

При работе с форматированными файлами важно учитывать следующие моменты:

1. Функции семейства `scanf` возвращают целое число – количество значений, которые успешно прочитаны в соответствии с указанным форматом. В реальных приложениях эту величину следует проверять в коде:

```
int i=fscanf (fp, "%f", &a);
if (i!=1) {
    //не удалось получить одно значение
}
```

2. На "восприятие" программой данных может влиять установленная в приложении локаль. Например, если до показанного кода выполнен оператор

```
setlocale(LC_ALL, "Rus");
```

результат работы кода может измениться (для русской локали разделителем целой и дробной части числа является запятая, а не точка).

3. Очередное чтение данных изменяет внутренний *файловый указатель*. Этот указатель в любой момент времени, пока файл открыт, показывает на следующее значение, которое будет прочитано. Благодаря этому наш код с бесконечным `while` не заиклился.

4. Код показывает, как читать из файла заранее неизвестное количество значений. Это позволяет сделать стандартная функция `fEOF` (проверка, достигнут ли конец файла; вернёт не ноль, если прочитано всё).

5. Распространённый в примерах из Интернета код вида

```
while (!feof(fp)) {  
    fscanf (fp, "%f",&a);  
    //обработка числа a  
}
```

под Windows может породить неточности при интерпретации данных. Например, этот код может прочитать как последнее значение завершающий перевод строки в файле, благодаря чему последнее прочитанное значение удвоится.

**Пример 3.** В качестве примера *форматной записи* в файл сохраним массив `a` из 10 целочисленных значений в файле с именем `result.txt` по 5 элементов в строке:

```
const int n=10;  
int a[n],i;  
FILE *fp=fopen ("result.txt","wt");  
if (fp==NULL) {  
    puts ("Не удалось открыть файл!");  
    getchar(); exit (1);  
}  
for (i=0; i<n; i++) a[i]=i+1;
```

```

for (i=0; i<n; i++) {
    fprintf (fp, "%5d ", a[i]);
    if ((i+1)%5==0) fprintf (fp, "\n");
}
fclose (fp);

```

Обратите внимание на последнюю строчку листинга. Если файл открывался для записи, всегда закрывайте его программно, не полагаясь на то, что файлы будут автоматически закрыты при завершении приложения.

Важно понимать, что ввод/вывод функциями библиотеки `stdio.h` *буферизован*, т.е. данные "пропускаются" через область памяти заданного размера, обмен происходит не отдельными байтами, а "порциями". Поэтому перед чтением данных желательно очищать буфер от возможных "остатков" предыдущего чтения методом `fflush`, а после записи данных следует обязательно закрывать файл методом `fclose`, иначе данные могут быть потеряны. Заметим, что консольный ввод/вывод методами `scanf` и `printf` также буферизован.

Теперь рассмотрим текстовый файл, состоящий из *неструктурированных* строк (абзацев) текста, разделённых символами перевода строки. При работе с такими данными могут потребоваться следующие функции:

- `fgetc` и `fputc` – для посимвольного чтения и посимвольной записи данных;
- `fgets` и `fputs` – для чтения и записи строк с указанным максимальным размером.

Как и в случае с функциями для чтения форматированных данных, у всех этих методов имеются аналоги для работы со стандартным вводом/выводом.

**Пример 4.** Читая файл, определить длину каждой строки в символах. Для решения задачи воспользуемся тем фактом, что строки завершаются символом "перевод строки" ('\n'). Предполагается, что файл уже открыт для чтения.

```

int c; int len=0, cnt=0;
while (1) {
    c=fgetc(fp);

```

```

if (c=='\n') {
    printf("\nString %d, len.=%d", ++cnt, len);
    len=0;
}
else len++;
if (feof(fp)) break;
}
if (len)
    printf ("\nString %d, len.=%d", ++cnt, len);

```

Из-за особенностей реализации `fgetc` под Windows, без последней проверки за телом цикла код мог "не обратить внимания", например, на последнюю строку файла, состоящую только из пробелов и не завершающуюся переводом строки.

**Пример 5.** Читаем построчно файл с известной максимальной длиной строки. Предполагается, что файл уже открыт для чтения.

```

char buf[128];
while (1) {
    fgets(buf, 127, fp);
    if (feof(fp)) break;
    int len = strlen(buf);
    if (buf[len-1]=='\n') buf[len-1]='\0';
    puts (buf);
    //Вывести прочитанные строки на экран
}

```

Без дополнительной обработки прочитанные из файла строки при выводе будут содержать "лишние" пустые строки между строками данных. Это происходит потому, что функция `fgets` читает строку файла вместе с символом перевода строки (точнее, под Windows – с парой символов `"\r\n"`, интерпретируемых как один), а функция `puts` добавляет к выводимой строке ещё один перевод строки.

Если максимальная длина строки принципиально не ограничена, помочь может либо предварительное посимвольное чтение файла для её определения, либо работа с файлом как с *бинарными* данными.

Бинарный файл отличается от текстового тем, что не обязательно состоит из печатаемых символов со стандартными разделителями между ними. Следовательно, для него не имеет смысла понятие "строки данных", а основной способ работы с ним – чтение и запись наборов байтов указанного размера. Основные функции для чтения и записи бинарных данных – `fread` и `fwrite` соответственно. В базовой реализации они имеют по четыре параметра:

- `void *buffer` – нетипизированный указатель на место хранения данных;
- `size_t size` – размер элемента данных в байтах, тип данных `size_t` обычно определён как `unsigned`;
- `size_t count` – максимальное количество элементов, которое требуется прочитать или записать;
- `FILE *stream` – указатель на структуру `FILE`.

**Пример 6.** Целочисленный массив `a` запишем в двоичный файл.

```
FILE *fp=fopen ("data.dat", "wb");
if (fp==NULL) {
    puts ("Не удалось открыть файл");
    getchar(); exit (1);
}
const int n=10;
int a[n];
for (int i=0; i<n; i++) a[i]=i+1;
for (int i=0; i<n; i++)
    fwrite (&a[i], sizeof(int), 1, fp);
fclose (fp);
```

Здесь 10 элементов массива записываются в файл по одному, при этом, если `sizeof(int)==2`, файл будет иметь размер 20 байт, а при `sizeof(int)==4` – 40 байт.

Учитывая, что данные массива хранятся в последовательно идущих адресах памяти, цикл `for` для записи можно заменить одним оператором:

```
fwrite (&a[0], sizeof(int), n, fp);
```

Подход к чтению данных с помощью функции `fread` аналогичен. Например, если файл уже открыт для чтения в режиме `"rb"`, возможны следующие операторы:

```
unsigned char c;
//...
fread (&c,1,1,fp); //читаем по 1 байту
unsigned char buf[512];
//...
fread (&buf,1,512,fp);
//читаем по 1 сектору (по 512 байт)
```

Для файлов, открытых в режиме `"r+b"`, разрешены и чтение, и запись (произвольный доступ). Поэтому при работе с такими файлами нужны функции позиционирования файлового указателя:

- функции `fgetpos` и `ftell` позволяют выполнить чтение текущей позиции указателя в файле;
- функции `fseek` и `fsetpos` позволяют осуществить переход к нужной позиции в файле.

**Пример 7.** Определить размер файла в байтах. Предположим, что файл уже открыт в режиме чтения или произвольного доступа.

```
fseek (fp, 0, SEEK_END);
//0 байт от конца файла
long int pos;
pos = ftell (fp);
//Получили текущую позицию
if (pos<0) puts ("\nОшибка");
else if (!pos) puts ("\nФайл пуст");
else printf ("\nВ файле %ld байт",pos);
```

Для ввода/вывода через цветную консоль во многих источниках используются методы библиотеки `conio.h`. Следует учитывать, что её реализации в компиляторах от Borland и Microsoft значительно отличаются, а в компиляторах под Unix/Linux реализации `conio.h` могут отсутствовать.

Как вариант, в компиляторах Visual Studio можно использовать аналоги `conio.h` от сторонних разработчиков, например открытый проект `conio.w.h` [12].

**Пример 8.** Приведём законченный пример кода, реализующего несложное консольное меню для Visual Studio. Предполагается, что к проекту подключены заголовочный файл `conio.w.h` и файл исходного кода `conio.w.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <windows.h>
#include <locale.h>
#include "conio.w.h"
typedef void (* FUN) (void);
typedef struct ITEM { /* элемент меню */
    int x,y; //Столбец и строка консоли
    char *str; //Наименование пункта
    FUN f; //Функция, привязанная к пункту
};
typedef struct WINDOW { //Окно вывода
    int x1,y1,x2,y2,back,color;
    //Координаты, фоновый цвет, цвет текста
};
void Exit () { //Восстановить консоль и выйти
    window (1,1,80,25); textbackground(BLACK);
    textcolor(LIGHTGRAY); clrscr(); exit(0);
}
void DrawWindow (WINDOW w) { //Нарисовать окно w
    char c[]={'+',' ','+', '!', '!', '+', '=', '+'};
    window (1,1,80,25);
    textbackground(w.back); textcolor(w.color);
    gotoxy (w.x1-1,w.y1-1); cprintf ("%c",c[0]);
    for (int i=w.x1; i<=w.x2; i++)cprintf ("%c",c[1]);
    cprintf ("%c",c[2]);
    for (int j=w.y1; j<=w.y2; j++) {
        gotoxy (w.x1-1,j); cprintf ("%c",c[3]);
        for (int i=w.x1; i<=w.x2; i++) cprintf (" ");
        cprintf ("%c",c[4]);
    }
    gotoxy (w.x1-1,w.y2+1); cprintf ("%c",c[5]);
}
```

```

    for (int i=w.x1; i<=w.x2; i++)cprintf ("%c",c[6]);
    cprintf ("%c",c[7]);
}
void flush(void) { fflush(stdin); }
void DrawMenu(int n,ITEM *m,WINDOW w) {
    int sel=0, back=WHITE, inactivecolor=DARKGRAY,
        activecolor=RED;
    DrawWindow (w);
    textbackground(back);
    for (int i=0; i<n; i++) {
        gotoxy(m[i].x,m[i].y);
        textcolor(inactivecolor);
        cprintf ("%s",m[i].str);
    }
    while (1) {
        //Бесконечный цикл обработки нажатий клавиш
        gotoxy(m[sel].x,m[sel].y);
        textcolor(activecolor);
        cprintf ("%s",m[sel].str); //активный пункт
        flush();
        int ch=getch();
        gotoxy(m[sel].x,m[sel].y);
        textcolor(inactivecolor);
        cprintf ("%s",m[sel].str); //убрать выделение
        if (!ch) { //Это расширенный код?
            ch=getch();
            switch(ch) {
                //Обработка расширенных кодов клавиш
                case 72: case 75: if(sel)sel--; else sel=n-1;
                    break; //Стрелки вверх и влево
                case 80: case 77: if(sel<n-1)sel++; else sel=0;
                    break; //Стрелки вниз и вправо
            }
        }
    }
    else { //Обработка нерасширенных кодов клавиш
        switch(ch) {
            case 13: //Нажата Enter
                textbackground(w.back);
                textcolor(w.color);
                window (w.x1,w.y1,w.x2,w.y2);
                m[sel].f(); //Вызов пункта меню

```

```

        clrscr();
        textbackground(back); window (1,1,80,25);
        break;
    case 27: Exit(); //По Esc выход из приложения
    }
}
}
}
//Ниже начинается часть пользователя
void File() { //Функция меню File
    long int i=0;
    while (!kbhit()) {
        delay (200);
        cprintf ("Работа функции File, шаг %ld\r\n",++i);
    }
}
void Do() { //Функция меню Do
    cprintf ("Введите целое число: ");
    int d;
    cscanf ("%d",&d);
    cprintf ("\r\nВы ввели значение %d\r\n\
Нажмите клавишу для выхода...",d);
    flush(); getch();
}
int main () {
    setlocale (LC_ALL,"Russian");
    setlocale(LC_CTYPE, ".65001"); //Юникод
    SetConsoleCP(65001);
    SetConsoleOutputCP(65001);
    //Пример для горизонтального меню
    ITEM menu1[3]={ //Описали меню
        {1,1,"File",File},
        {6,1,"Do",Do},
        {9,1,"Exit",Exit}
    };
    WINDOW w={3,3,77,23,BLUE,YELLOW};
    clrscr();
    DrawMenu (3,menu1,w); return 0;
/* //Пример для вертикального меню
    ITEM menu1[3]={ //Описали меню
        {2,2,"File",File},

```

```

    {2, 3, "Do", Do},
    {2, 4, "Exit", Exit}
};
WINDOW w={8, 2, 78, 24, BLUE, YELLOW};
textbackground (WHITE);
clrscr();
DrawMenu (3, menu1, w); return 0; */
}

```

Очистка буфера клавиатуры, выполняемая методом `fflush(stdin)` перед вызовом функций ввода с консоли, вынесена в отдельную функцию потому, что в новых компиляторах, например в Visual Studio 2015, это действие может быть реализовано следующим альтернативным кодом:

```
scanf_s("%*c");
```

### **Задание к лабораторной работе**

Базовое задание включает две задачи, первая из которых предполагает обработку файла как текстовых данных, вторая – как бинарных. В качестве дополнительной, третьей, задачи может быть предложена реализация одной из первых двух задач, содержащая консольный интерфейс и меню.

## Глава 9. Работа с файлами с помощью потоков и классов ввода/вывода

Механизм ввода/вывода, разработанный для обычного языка C, не соответствует общепринятому сегодня стилю объектно-ориентированного программирования. Кроме того, он активно использует операции с указателями, считающиеся потенциально небезопасными в современных защищённых средах выполнения кода. Альтернативой при разработке прикладных приложений является механизм классов ввода/вывода, предоставляемый стандартом языка C++.

Наиболее часто применяются классы `ifstream` для чтения, `ofstream` для записи и `fstream` для модификации файлов.

Все поточные классы ввода/вывода являются косвенными производными от общего предка `ios`, полностью наследуя его функциональность. Так, режим открытия файлов задаёт член данных перечисляемого типа `open_mode`, который определяется следующим образом:

```
enum open_mode {  
    app, binary, in, out, trunc, ate };
```

В табл. 4 приведены возможные значения флагов и их назначение.

Таблица 4

Режимы открытия файлов

| Режим               | Назначение                                                                                                                                                  |
|---------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>in</code>     | Открыть для ввода (выбирается по умолчанию для <code>ifstream</code> )                                                                                      |
| <code>out</code>    | Открыть для вывода (выбирается по умолчанию для <code>ofstream</code> )                                                                                     |
| <code>binary</code> | Открыть файл в бинарном виде                                                                                                                                |
| <code>app</code>    | Присоединить данные; записать в конец файла                                                                                                                 |
| <code>ate</code>    | Установить файловый указатель на конец файла                                                                                                                |
| <code>trunc</code>  | Уничтожить содержимое, если файл существует (выбирается по умолчанию, если флаг <code>out</code> указан, а флаги <code>ate</code> и <code>app</code> – нет) |

Например, чтобы открыть файл с именем `test.txt` для чтения данных в бинарном виде, следует написать:

```
ifstream file;  
file.open ("test.txt", ios::in|ios::binary);
```

Оператор логического ИЛИ (`|`) позволяет составить режим с любым сочетанием флагов. Так, чтобы, открывая файл для записи, случайно не затереть существующий файл с тем же именем, нужно использовать следующую форму записи:

```
ofstream file;  
file.open ("test.txt", ios::out|ios::app);
```

Предполагается, что к проекту подключён соответствующий заголовочный файл:

```
#include <fstream.h>
```

Для проверки того, удалось ли открыть файл, можно применять конструкцию

```
if (!file) {  
    //Обработка ошибки открытия файла  
}
```

Переопределённый в классах работы с файлами *оператор включения* (`<<`) записывает данные в файловый поток. Как только вы открыли файл для записи, можно записывать в него текстовую строку целиком:

```
file << "Это строка текста";
```

Можно также записывать текстовую строку по частям:

```
file << "Это " << "строка " << "текста";
```

Оператор `endl` завершает ввод строки символом перевода строки:

```
file << "Это строка текста" << endl;
```

С помощью оператора включения несложно записывать в файл значения переменных или элементов массива:

```
ofstream file ("temp.txt");  
char buff[] =  
    "Текстовый массив содержит переменные";  
int    vx = 100;  
float  pi = 3.14159;
```

```
file << buff << endl <<  
vx << endl << pi << endl;
```

В результате выполнения кода образуется три строки текстового файла `temp.txt`:

```
Текстовый массив содержит переменные  
100  
3.14159
```

Обратите внимание, что числовые значения записываются в файл в виде текстовых строк, а не двоичных значений.

*Оператор извлечения (>>)* производит обратные действия. Казалось бы, чтобы извлечь символы из файла `temp.txt`, записанного ранее, нужно написать код наподобие следующего:

```
ifstream file ("Temp.txt");  
char buff[100];  
int vx;  
float pi;  
file >> buff >> vx >> pi;
```

Однако оператор извлечения остановится на первом попавшемся разделителе (символе пробела, табуляции или новой строки). Таким образом, при разборе предложения "Текстовый массив содержит переменные" только слово "Текстовый" запишется в массив `buff`, пробел игнорируется, слово "массив" станет значением целой переменной `vx`, а исполнение кода окажется непредсказуемым, с неминуемым нарушением структуры данных. Далее при обсуждении класса `ifstream` будет показано, как правильно организовать чтение файла из предыдущего примера.

Как следует из расшифровки названия, класс `ifstream` предназначен для ввода файлового потока. В табл. 5 перечислены основные методы класса. Большая часть из них унаследована от класса `istream` и перегружена с расширением родительской функциональности. Например, функция `get`, в зависимости от параметра вызова, способна считывать не только одиночный символ, но и символьный блок.

Таблица 5

Основные методы класса `ifstream`

| Метод                | Описание                                                                                                 |
|----------------------|----------------------------------------------------------------------------------------------------------|
| <code>open</code>    | Открывает файл для чтения                                                                                |
| <code>get</code>     | Читает один или более символов из файла                                                                  |
| <code>getline</code> | Читает символьную строку из текстового файла или данные из бинарного файла до определённого ограничителя |
| <code>read</code>    | Считывает заданное число байтов из файла в память                                                        |
| <code>eof</code>     | Возвращает ненулевое значение ( <code>true</code> ), когда указатель потока достигает конца файла        |
| <code>peek</code>    | Выдаёт очередной символ потока, но не выбирает его                                                       |
| <code>seekg</code>   | Перемещает указатель позиционирования файла в заданное положение                                         |
| <code>tellg</code>   | Возвращает текущее значение указателя позиционирования файла                                             |
| <code>close</code>   | Закрывает файл                                                                                           |

**Пример 1.** Теперь понятно, как нужно модифицировать код, чтобы использование оператора извлечения данных давало ожидаемый результат:

```
ifstream file("Temp.txt");
char buff[100];
int vx;
float pi;
file.getline(buff, sizeof(buff));
file >> vx >> pi;
```

Метод `getline` прочитает первую строку файла до конца, а оператор `>>` присвоит значения переменным.

**Пример 2.** Покажем добавление данных в текстовый файл с последующим чтением всего файла. Цикл `while (1)` применяется вместо `while(!file2.eof())` по причинам, которые обсуждались в главе 8.

```
#include <iostream>
#include <fstream>
```

```

using namespace std;
int main() {
    ofstream file;
    file.open("test.txt", ios::out|ios::app);
    if (!file) {
        cout << "Can't open to write data!";
        cin.sync(); cin.get(); return 1;
    }
    for (int i=0;i<10;i++) file << i << endl;
    file.close();
    ifstream file2;
    file2.open("test.txt", ios::in);
    if (!file2) {
        cout << "Can't open to read data!";
        cin.sync(); cin.get(); return 2;
    }
    int a,k=0;
    while (1) {
        file2 >> a;
        if (file2.eof()) break;
        cout << a << " "; k++;
    }
    cout << endl << "K=" << k << endl;
    file2.close();
    cin.sync(); cin.get(); return 0;
}

```

**Пример 3.** Покажем цикл считывания строк из файла test.txt и их отображения на консоли.

```

#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream file; file.open("test.txt");
    if (!file) return 1;
    char str[80];
    while
        (!file.getline(str, sizeof(str)).eof())

```

```

        cout << str << endl;
        cin.sync(); cin.get(); return 0;
    }

```

Этот код под Windows также зависит от наличия в последней строке файла символа перевода строки. Надёжнее было бы сделать так:

```

while (1) {
    if (file.eof()) break;
    file.getline(str, sizeof(str));
    cout << str << endl;
}

```

Явные вызовы методов `open` и `close` не обязательны. Описание файла с аргументом-именем позволяет сразу же, в момент создания потокового объекта `file`, открыть файл:

```
ifstream file("test.txt");
```

Вместо метода `close` можно использовать оператор `delete`, который автоматически закроет файл.

Класс `ofstream` предназначен для вывода данных из файлового потока. В табл. 6 перечислены основные методы данного класса.

Таблица 6

Основные методы класса `ofstream`

| Метод              | Описание                                                     |
|--------------------|--------------------------------------------------------------|
| <code>open</code>  | Открывает файл для записи                                    |
| <code>put</code>   | Записывает одиночный символ в файл                           |
| <code>write</code> | Записывает заданное число байтов из памяти в файл            |
| <code>seekp</code> | Перемещает указатель позиционирования в указанное положение  |
| <code>tellp</code> | Возвращает текущее значение указателя позиционирования файла |
| <code>close</code> | Закрывает файл                                               |

Описанный ранее оператор включения удобен для организации записи в текстовый файл:

```

ofstream file ("temp.txt");
if (!file) return;

```

```

for (int i=1; i<=3; i++)
file << "Строка " << i << endl;
file.close();

```

В принципе, бинарные данные обслуживаются наподобие текстовых. Отличие состоит в том, что если бинарные данные записываются в определённой логической структуре, то они должны считываться из файла в переменную того же структурного типа.

Первый параметр методов `write` и `read` (адрес блока записи/чтения) должен иметь тип символьного указателя `char *`, поэтому необходимо произвести явное преобразование типа адреса структуры `void *`. Второй параметр указывает, что бинарные блоки файла имеют постоянный размер байтов независимо от фактической длины записи.

**Пример 4.** Приведём пример создания и отображения данных простейшей записной книжки. Затем записи файла последовательно считываются и отображаются на консоли.

```

#include <iostream>
#include <fstream>
#include <locale>
using namespace std;
struct Notes {      // структура данных
    char Name[60];   // Ф.И.О.
    char Phone[16];  // телефон
    int  Age;        // возраст
};
int main() {
    setlocale(LC_ALL, "Russian");
    Notes Note1={"Грозный Иоанн Васильевич",
        "не установлен", 60 };
    Notes Note2={"Годунов Борис Фёдорович",
        "095-111-2233 ", 30 };
    Notes Note3={"Романов Пётр Михайлович",
        "812-333-2211 ", 20 };
    ofstream ofile ("Book.dat", ios::binary);
    ofile.write((char*)&Note1, sizeof(Notes));

```

```

    ofile.write((char*)&Note2, sizeof(Notes));
    ofile.write((char*)&Note3, sizeof(Notes));
    ofile.close();
    ifstream ifile ("Book.dat", ios::binary);
    Notes Note;
    char str[80];
    while (!ifile.read
        ((char*)&Note, sizeof(Notes)).eof()) {
        sprintf(str, "%s\\tТел: %s\\tВозраст: %d",
            Note.Name, Note.Phone, Note.Age);
        cout << str << endl;
    }
    ifile.close();
    cin.sync(); cin.get(); return 0;
}

```

В результате выполнения этого кода образуется бинарный файл `Book.dat` из трёх блоков размером по 80 байт каждый.

Естественно, можно использовать другие поточные методы и выполнять любые операции над полями определённой структуры данных.

Теперь предположим, что в нашей записной книжке накопилось 100 записей, а мы хотим считать 50-ю. Конечно, можно организовать цикл и прочитать все записи с первой по заданную. Очевидно, что более целенаправленное решение – установить указатель позиционирования файла `pos` прямо на запись 50 и считать её:

```

    ifstream ifile("Book.dat", ios::binary);
    int pos = 49 * sizeof(Notes);
    ifile.seekg(pos); // поиск 50-й записи
    Notes Note;
    ifile.read((char*)&Note, sizeof(Notes));

```

Подобные операции поиска эффективны, если файл состоит из записей известного и постоянного размера. Чтобы заменить содержимое произвольной записи, надо открыть поток вывода в режиме модификации:

```

ofstream ofile("Book.dat",
    ios::binary|ios::ate);

```

```

int pos = 49 * sizeof(Notes);
ofile seekp(pos); // поиск 50-й записи
Notes Note50={"Ельцин Борис Николаевич",
  "095-222-3322", 64};
ofile.write((char*)&Note, sizeof(Notes));

```

Если не указать флаг `ios::ate` (или `ios::app`), то при открытии бинарного файла `Book.dat` его предыдущее содержимое будет стёрто! Наконец, можно открыть файл одновременно для чтения/записи, используя методы, унаследованные поточным классом `fstream` от своих предшественников. Поскольку класс `fstream` произведён от `istream` и `ostream` (родителей `ifstream` и `ofstream` соответственно), все упомянутые ранее методы становятся доступными в приложении.

**Пример 5.** Покажем перестановку первой и третьей записей файла `Book.dat`.

```

#include <iostream>
#include <fstream>
#include <locale>
using namespace std;
struct Notes {
char Name[60]; char Phone[16]; int Age;};
int main() {
  setlocale(LC_ALL, "Russian");
  Notes Note1, Note3;
  fstream file("Book.dat",
    ios::binary | ios::in | ios::out);
  file.seekg(2 * sizeof(Notes));
  file.read((char*)&Note3, sizeof(Notes));
  file.seekg(0);
  file.read((char*)&Note1, sizeof(Notes));
  file.seekg(0);
  file.write((char*)&Note3, sizeof(Notes));
  file.seekg(2 * sizeof(Notes));
  file.write((char*)&Note1, sizeof(Notes));
  char str[80];
  file.seekg(0);
  while (!file.read

```

```

        ((char*)&Note1, sizeof(Notes)).eof()) {
            sprintf(str, "%s\tТел: %s\tВозраст: %d",
                Note1.Name, Note1.Phone, Note1.Age);
            cout << str << endl;
        }
        file.close();
        cin.sync(); cin.get(); return 0;
    }
}

```

В конструкторе объекта `file` нужно указать флаги `ios::in` и `ios::out`, разрешая одновременное выполнение операций чтения и записи. В результате выполнения этого кода первая и третья записи бинарного файла `Book.dat` поменяются местами.

**Пример 6.** Реализация задачи на обработку текстовых данных. Прочитать посимвольно текстовый файл и найти процент символов, являющихся русскими гласными (приведена реализация в Visual Studio, файл `data.txt` должен находиться в текущей папке, там же, где модуль программы).

```

#include <iostream>
#include <fstream>
#include <string>
#include <windows.h>
using namespace std;
int main() {
    setlocale(LC_ALL, "RUS");
    SetConsoleCP(1251); SetConsoleOutputCP(1251);
    ifstream file; file.open("data.txt");
    if (!file) return 1;
    string str;
    string vowels = "аеёиоуыэюяАЕЕИОУЫЭЮЯ";
    int all = 0, cnt = 0;
    while (1) {
        if (file.eof()) break;
        getline(file, str);
        //Читаем, не заботясь о длине строк!
        cout << str << endl;
        //Медленно для больших файлов!
    }
}

```

```

int len = str.size();
for (int i = 0; i < len; i++) {
    all++;
    string c = str.substr(i, 1);
    if (vowels.find(c)!=string::npos) cnt++;
}
}
cout<<((cnt + 0.)/all*100.)<<"%"<<endl;
cin.sync(); cin.get(); return 0;
}

```

**Пример 7.** Реализация задачи на обработку бинарного файла. Прочитать файл в ОП (файл интерпретируется как бинарный, использован файл предыдущей задачи).

```

#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream is("data.txt", ios::binary);
    if (is) {
        is.seekg(0, is.end);
        int len = is.tellg(); is.seekg(0, is.beg);
        char *buf = new char[len];
        if (!buf) { return 1; /* No memory */ }
        is.read(buf, len);
        if (is) cout << "All read";
        else cout << "Only "
            << is.gcount() << " char(s) read";
        is.close();
        //Делаем что-то с содержимым файла в buf
        delete[] buf;
    }
    cin.sync(); cin.get(); return 0; }

```

### **Задание к лабораторной работе**

Реализовать с использованием классов файлового ввода/вывода C++ первую или вторую задачу предыдущей лабораторной работы (по выбору студента или указанию преподавателя; см. с. 80 настоящего учебного пособия).

## Глава 10. Составные типы данных (структуры)

Представим себе, что нам нужно программно обработать информацию о студентах. Данные о каждом студенте могут включать в себя величины нескольких типов: *строку* с фамилией и инициалами; *целое число*, показывающее возраст студента в годах; *средний балл*, являющийся *вещественным значением*, и т.д.

Простейший путь решения задачи – описать в программе набор соответствующих переменных:

```
char name[20]; //фамилия и инициалы
int age;       //возраст в годах
float ball;    //средний балл
```

Когда студентов становится несколько, нам придётся перейти от переменных к массивам:

```
#define ST 5 /* количество студентов */
char name[ST][20];
int age[ST];
float ball[ST];
```

Чтобы извлечь информацию о последнем студенте, нам нужно обработать величины `name[4]`, `age[4]` и `ball[4]`. Таким образом, информация об одном объекте оказывается распределённой по нескольким контейнерам-массивам, что делает крайне неудобными операции со "студентом" как единым объектом. Преодолеть это неудобство может *структура*, позволяющая объединять в одном объекте разнотипные данные.

Итак, структура – это основной составной тип данных. В отличие от массива, она объединяет в одном объекте значения, которые могут иметь разные типы данных. Это делает структуры основным средством моделирования в программе объектов реального мира:

```
struct student {
    char name[20]; //фамилия и инициалы
    int age; //возраст в годах
    float ball; //средний балл
};
```

Для совместимости с языком С перед ключевым словом `struct` часто записывают оператор определения типа `typedef`. На С++ этот оператор является излишним.

Опишем переменные "нового" типа данных `student`:

```
student s, s2;
```

Никакой разницы с описанием переменных простых типов `int`, `float` и т.д. нет.

Для доступа к отдельным данным о студенте, перечисленным в описании структурного типа (*полям* структуры), служит операция `.` (точка), применяемая в записи вида `структура.поле`. Например, `s.name` будет означать поле "имя" студента, информация о котором сохранена в переменной с именем `s`. При работе с полем структуры разрешены все те же операции, что и с обычной переменной соответствующего типа данных:

```
strcpy (s.name, "Ivanov");  
s.age = 19; s.ball = 3.5;  
printf  
("\n%s,%d,%.2f", s.name, s.age, s.ball);
```

При работе со структурами как с единым целым разрешены следующие операции:

- передача в качестве аргумента функции;
- возврат из функции в качестве её значения;
- получение адреса структуры в памяти и создание указателя на неё;
- присваивание структур одинакового типа.

**Пример 1.** Рассмотрим первые два действия. Функция `st_print` выводит информацию о структуре типа `student`, переданной в качестве параметра. Функция `st_new` заполняет новую структуру `student` данными по умолчанию и возвращает структуру. За счёт того, что поддерживается присваивание структур, полученная из функции информация дублируется в структуре `s2` функции `main`.

```
#define _CRT_SECURE_NO_WARNINGS  
#include <iostream>  
using namespace std;  
struct student {
```

```

    char name[20]; int age; float ball;
};
void st_print (student s) {
    cout << s.name << ", age=" << s.age <<
    ", ball=" << s.ball << endl;
}
student st_new () {
    student s;
    strcpy (s.name, "Noname");
    s.age=17; s.ball=0.;
    return s;
}
int main() {
    student s1 = {"Petrov", 19, 4.2 };
    st_print (s1);
    student s2 = st_new();
    st_print (s2);
    cin.get(); cin.sync(); return 0;
}

```

Заметим, что передача в качестве параметров функции и возврат из функции структур большого размера могут быть чреваты переполнением стека. Определить количество байтов, занимаемое структурой в ОП, можно стандартными средствами, с помощью операции `sizeof(student)`.

Для сохранения информации о 20 студентах группы естественным выглядит описание *массива структур*:

```
student group[20];
```

Как и любой массив, массив структур тоже можно обрабатывать поэлементно. К примеру, вычислим средний балл группы (предполагая, что данные уже введены или прочитаны из файла):

```

float average = 0.;
for (int i=0; i<20; i++)
    average+=group[i].ball;
average /= 20;

```

Доступ к структуре можно осуществлять и *через указатель*:

```
student *ptr = &group[0];
```

После этого можно выполнять операцию `ptr++` для сканирования списка или массива структур таким образом, как мы поступали с указателями на простые типы данных.

Доступ к полю структуры через указатель имеет следующий вид:

```
(*ptr).field
```

Круглые скобки здесь необходимы из-за приоритетов операций языка C++. Запись выглядит громоздкой, поэтому вместо неё, как правило, применяют мнемоническое сокращение

```
ptr->field
```

**Пример 2.** Функция `show` печатает информацию о студенте `s` (без указателя):

```
void show (student s) {  
    printf ("\n%s,%d,%.2f",  
        s.name,s.age,s.ball);  
}
```

Та же функция с указателем:

```
void show (student *s) {  
    printf ("\n%s,%d,%.2f",  
        s->name,s->age,s->ball);  
}
```

В первом случае через стек передано `sizeof(student)=sizeof(char)*20+sizeof(int)+sizeof(float)` байт (от 28 байт, если символы `char` однобайтовые), во втором случае – 4 или 8 байт (в зависимости от размера указателя в 32-разрядной или 64-разрядной архитектуре ЭВМ).

Для `i`-го элемента нашего массива структур вызов первой функции имел бы вид

```
show (group[i]);
```

а второй – вид

```
show (&group[i]);
```

**Пример 3.** Показанная в примере 1 функция `st_new` с использованием указателя приняла бы вид

```
student *st_new () {  
    student *s = new student;  
    strcpy (s->name,"Noname");
```

```

s->age=17;
s->ball=0.;
return s;
}

```

Её вызов мог бы быть следующим:

```

student *s2 = st_new();
st_print (*s2);

```

Поскольку оператор `new` выделяет память в куче, а не в стеке, память затем можно освободить оператором `delete`.

Остановимся на особенностях присваивания структур. Следует понимать, что если структурный тип содержит указатели или динамические поля (под которые мы выделяли ОП), операция присваивания таких структур будет работать *некорректно*. Например, если после операции `a=b` объект `b` удалён, а скопированный указатель в объекте `a` продолжает показывать на несуществующий адрес.

**Пример 4.** В программе примера 1 все поля структуры `student` – статические, под них не выделялась динамическая память, поэтому присваивание таких структур корректно:

```

student s1 = {"Petrov", 19, 4.2};
student s2 = s1; //Так делать можно

```

Если статическое поле `name` заменить динамическим, под поле "имя" каждого студента нужно будет выделять динамическую память. Присваивание таких структур оператором вида `s2=s1` будет означать установку указателя `s2.name` на тот же адрес ОП, который содержится в `s1.name`.

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
struct student {
    char *name; int age; float ball;
};
void st_print (student s) {
    cout << s.name << ", age=" << s.age
        << ", ball=" << s.ball << endl;
}

```

```

int main() {
    student s1;
    s1.name = new char [20];
    //выделяем память!
    strcpy (s1.name, "Petrov");
    s1.age=19; s1.ball=4.2;
    student s2 = s1; //так делать нельзя!
    delete s1.name;
    st_print (s2); //крах программы
    cin.get(); cin.sync(); return 0;
}

```

Как вариант, поставим вместо строки

```
delete s1.name;
```

строку

```
strcpy (s1.name, "Ivanov");
```

Так как указатели `s1.name` и `s2.name` показывают на один и тот же адрес памяти, после вызова `strcpy` изменится и "имя" второго студента!

Элемент структуры может быть *структурой другого структурного типа* или *указателем на структуру своего типа*. Последнее важное свойство структур позволяет организовывать динамические структуры данных, такие как стеки, очереди или деревья. Динамическим структурам будет посвящена глава 12.

Наконец, заметим, что наиболее естественным выглядит сохранение статических структур в *бинарных файлах данных* (так как размер такой структуры в байтах – фиксированный) и обмен данными с этими файлами с помощью методов `fread/fwrite` из `stdio.h` или `read/write` из `fstream`.

**Пример 5.** Чтение 20 записей типа `student` из бинарного файла.

```

for (int i=0; i<20; i++)
    fread (&group[i], sizeof(student), 1, f);
//Перейти к записи k (нумерация с нуля):
long int pos = sizeof(student)*k;
fsetpos (f, &pos);

```

**Пример 6.** К структурам с указателями в общем случае сказанное неприменимо. В качестве альтернативы напишем программу, позволяющую прочитать массив структур `student` с динамическим полем `name` из текстового файла `data.txt` следующего вида:

```
Ivanov 19 4.2
Petrova 19 3.9
Popov 20 4.1
```

Предполагается, что файл находится в текущей для исполнения файла приложения папке. Максимальное число записей ограничим константой `ST`.

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
using namespace std;
#define ST 10
struct student {
    char *name; int age; float ball;
};
int main() {
    student st[ST];
    FILE *f = fopen ("data.txt","rt");
    if (!f) {
        cout << "Can't open data.txt!";
        cin.get(); cin.sync(); return 1;
    }
    char buf[80],name[80];
    int i,k=0;
    while (1) {
        if (k>ST-1) break;
        fgets (buf, 79, f);
        int i=sscanf (buf,"%s %d %f",
            name,&st[k].age,&st[k].ball);
        if (i!=3) {
            cout << "Bad line " << (k+1) <<
                "in file, skipped" << endl;
            break;
        }
    }
}
```

```

    st[k].name = new char [strlen(name)+1];
    strcpy (st[k].name, name);
    k++;
    if (feof(f)) break;
    buf[0]='\0';
}
cout << "All data:" << endl;
for (i=0; i<k; i++)
    cout << st[i].name << ", " <<
        st[i].age << ", " << st[i].ball << endl;
cin.get(); cin.sync(); return 0;
}

```

Программа также содержит простейшую защиту от неверных данных в исходном файле.

### **Задание к лабораторной работе**

Согласно варианту задания реализовать приложение, работающее с массивом структур.

## Глава 11. Составные типы данных (перечисления, объединения, структуры структур)

Стандартным составным типом данных является *перечисление*. Перечисление определяет список целочисленных значений, которые может принимать переменная, и позволяет присвоить этим значениям названия. Например, определим именованное перечисление `mode` (режим работы чего-то в программе) и перечислим его возможные состояния:

```
enum mode {LAZY, WAIT, BUZY, ERROR};
```

Теперь в программе можно создать переменную, имеющую тип перечисления `mode` и использовать её:

```
mode mymode = LAZY;  
//...  
switch (mymode) {  
    case LAZY: /* ... */ break;  
    case WAIT: /* ... */ break;  
    /* ... */  
}
```

Здесь "говорящие" значения `WAIT` или `BUZY` напомним о том, какую информацию сейчас хранит переменная, гораздо лучше, чем безликие числовые значения 1 или 2.

Все константы перечисления должны быть допустимыми идентификаторами C++. Обычно, как и для других констант, их имена пишут большими буквами (это соглашение, а не правило языка).

Тип `enum`, как правило, интерпретируется компилятором как `int` (или `unsigned`), первый элемент списка по умолчанию равен 0, если не указано иного.

Основное применение перечислений – удобные мнемонические имена для переменных, имеющих фиксированный набор значений, таких как цвета для рисования, дни недели, месяцы или сезоны года, состояния автомата, социальные статусы человека и т.д.

Основные недостатки перечислений – отсутствие встроенных средств для вывода строковых значений констант перечисления, а также не всегда очевидная итерация по членам перечисления.

**Пример 1.** Перечисление "дни недели".

```
enum day {
    SATURDAY, SUNDAY=0, MONDAY, TUESDAY,
    WEDNESDAY, THURSDAY, FRIDAY
}; //SATURDAY также == 0!
day today = SATURDAY;
cout << today << endl; //0
//today++;
//ошибка компиляции - к типу неприменимо ++
today = FRIDAY;
cout << today << endl; //5
int workday = SATURDAY;
for (int i=0; i<7; i++) {
    cout << workday++ << " ";
    // ++ работает, но напечатается 0 1 ... 6
}
```

К составным типам данных можно отнести также *объединение* (union), позволяющее интерпретировать один и тот же объект (*одну и ту же память*) как данные различных типов:

```
union uchar {
    char c[2];
    unsigned short int u;
};
```

Здесь элемент типа `uchar`, в зависимости от того, для чего он нам нужен, может быть интерпретирован и как массив из двух символов типа `char`, и как беззнаковое целое 2-байтовое значение:

```
cout << sizeof(uchar) << endl; //2
uchar n;
n.u=0x6121;
cout << n.c[0] << n.c[1] << endl; //!a
cout << n.u << endl; //24865
```

Контроль за соответствием размерности полей в `union` возлагается на программиста. Следует учитывать, что компилятор может выравнивать данные на границу слова или двойного слова.

Нечасто используемой в прикладном программировании, но полезной возможностью структур являются *битовые поля*. Они позволяют "упаковать" поле структуры в указанное количество *битов* вместо байтов:

```
тип идентификатор: константное_выражение;
```

Целочисленное константное\_выражение, записанное после двоеточия, определяет число битов, выделяемых под переменную.

Указанный в шаблоне идентификатор необязателен. Неименованное битовое поле означает пропуск указанного числа битов перед размещением следующего элемента структуры. Неименованное битовое поле, для которого указан размер 0, имеет специальное назначение: оно гарантирует, что память для следующей переменной в этой структуре будет начинаться на границе машинного слова (размер машинного слова зависит от архитектуры ЭВМ и обычно равен разрядности регистров процессора). Это относится и к следующему битовому полю.

Битовое поле не может выходить за границу ячейки памяти объявленного для него типа. Например, битовое поле `unsigned int` упаковывается либо в пространство, оставшееся в текущей ячейке размерностью `sizeof(unsigned int)*8` бит от размещения предыдущего битового поля, либо, если предыдущий элемент структуры не был битовым полем или памяти в текущей ячейке недостаточно, в новую ячейку `unsigned int`.

На практике битовые поля обычно используются:

- для экономии памяти, поскольку позволяют плотно упаковать значения не по границам байтов;
- в системном программировании, например, для организации удобного доступа к регистрам внешних устройств, в которых различные биты могут иметь самостоятельное функциональное назначение, при доступе к элементам файловых таблиц и т.п.

В качестве примера покажем структуру, описывающую отдельную позицию экрана текстовой консоли и дающую возможность доступа к отдельным свойствам экранной позиции через битовые поля.

```
struct texel {
    int background: 8; //фон
    int color: 4; //цвет
    int underline: 1; //подчёркивание
    int blink: 1; //мерцание
};
union bits {
    texel t;
    int d;
};
```

Объединение `bits` позволит манипулировать со структурой как с обычным целым числом, в частности, выводить его на экран консоли в виде цепочки битов. Для работы показанного ниже кода нужно подключить стандартное пространство имён и библиотеку `bitset`:

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

Класс `bitset`, доступный в современных стандартах C++, удобен для решения задач, связанных с манипулированием отдельными битами или с булевой алгеброй.

```
texel t;
cout << "Size of texel = "
    << sizeof(texel) << " byte(s)" << endl;
cout << "Size of bits = "
    << sizeof(bits) << " byte(s)" << endl;
t.background = 0xFF;
t.color = 0;
t.underline = 1;
t.blink = 0;
bits b; b.t = t;
cout << "T=" << hex << b.d <<
    " (hex)" << endl;
```

```
cout << "T=" << bitset<sizeof(b.d)*8>(b.d)
    << " (bin)" << endl;
```

Вывод этой программы на 32-разрядном компьютере получился бы таким:

```
Size of texel = 4 byte(s)
```

```
Size of bits = 4 byte(s)
```

```
T=cccccd0fff (hex)
```

```
T=11001100110011001101000011111111 (bin)
```

Обратите внимание, что не используемые в структуре биты могут оказаться заполненными "мусором".

**Пример 2.** В качестве законченного примера ступенчатого моделирования структур, включающих в себя другие структурные типы, решим следующую задачу. Описать структурный тип "персона" с полями "имя" (память под поле выделяется динамически), "социальный статус" (со значениями "не определён", "иждивенец", "учащийся", "работающий", "пенсионер"), "номер паспорта", "дата рождения". Прочитать из текстового файла массив структур указанной размерности, проиллюстрировать добавление новой записи в файл.

Начнём определять типы для нашей будущей структуры. "Социальный статус" здесь прямо-таки напрашивается стать перечислением:

```
enum status {
    undefined,
    dependent, pupil, working, pensioner };
```

Логично, что статус "не определён" будет соответствовать нулевому значению, а остальные статусы упорядочены по возрасту.

К сожалению, `enum` всегда интерпретируется как `int`, поэтому "строковое" значение именованной константы вывести просто так нельзя. Чтобы это всё-таки сделать, нам придётся написать дополнительную функцию, в которой те же самые социальные статусы будут перечислены в строковом массиве:

```
char *get_status (status s) {
    char *status_strings[] = {
        "undefined", "dependent", "pupil",
```

```

    "working", "pensioner" };
    return status_strings[s];
}

```

Заметим, что при нарушении естественного порядка нумерации 0, 1, 2, ... в перечислении придётся изменить и логику этой функции.

Показанный подход к получению строковых "имен" констант перечисления очень примитивен, но альтернативой было бы довольно трудоёмкое использование макросов или написание отдельных классов для решения задачи конвертирования перечислений в их строковые аналоги.

Номер паспорта состоит из двух частей – 4-значной серии и 6-значного номера. Опишем его отдельной структурой, не забывая, что 6-значное число в тип `int` может "не войти", если он 2-байтовый (хотя на современной ЭВМ такого уже не бывает):

```

struct passport {
    unsigned short int series;
    unsigned long int number;
};

```

Для даты рождения не будем создавать три отдельных поля, лучше упакуем её в 2 байта с помощью битовых полей. День месяца может принимать значения от 1 до 31, так что на него понадобится 5 бит; 4 бита займёт месяц со значениями от 1 до 12; оставшиеся 7 бит (при условии, что размер целого – 2 байта) позволят закодировать  $2^7=128$  разных лет. Поэтому примем какой-либо год нашей эры за начало отсчёта, скажем, 1970-й – начало эпохи Unix. Самые старшие биты займёт год, затем месяц, затем день – это гарантирует, что большее число всегда будет соответствовать более поздней дате:

```

struct date {
    unsigned short int year :7;
    //Год 1970=0...до 2098
    unsigned short int month :4;
    unsigned short int day :5;
};

```

Возможно, дата понадобится нам не только как совокупность битовых полей, но и как 2-байтовое целое число. Здесь

как раз удобным решением окажется объединение, позволяющее интерпретировать одни и те же данные двумя способами:

```
union udate {
    date d;
    unsigned short int u;
};
```

Наконец, структурный тип "персона" будет объединять введённые ранее типы данных:

```
struct person {
    char *name; //или name[30]
    status status;
    passport passport;
    date date;
};
```

Код потребует подключения следующих стандартных библиотек:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Напишем функцию-обработчик ошибок, принимающую параметр "сообщение об ошибке" (msg):

```
void error (char *msg) {
    puts ("\n"); puts (msg);
    puts ("\nEnter to EXIT");
    fflush (stdin); getchar(); exit(0);
}
```

Все остальные действия демопрограммы поместим непосредственно в функцию main:

```
int main () {
    const int maxrecords=5;
    person p[maxrecords];
    char buf[128],name[40];
    FILE *f=fopen ("data.txt","r+t");
    if (!f) error ("Can't open data.txt");
    int count=0; //Счётчик записей
    while (1) {
```

```

fgets (buf,128,f); //Читаем строку
sscanf(buf,"%s %d %d %ld %u",
    name,&p[count].status,
    &p[count].passport.series,
    &p[count].passport.number,
    &p[count].date);
//и разбираем её по полям структуры
p[count].name = new char[strlen(name)+1];
    //выделяем память под поле "имя"
if (p[count].name==NULL) {
    sprintf (buf,
        "No memory for rec. no. %d",count+1);
    error (buf);
}
strcpy(p[count].name,name);
count++;
if (feof(f) || count>maxrecords-1) break;
}
//Допишем в файл 1 запись,
//если нет последней записи Newname
if (count &&
    strcmp(p[count-1].name,"Newname")) {
    udate d = { 2013-1970, 12, 31 };
    fseek (f, 0, SEEK_END);
    fprintf (f,"\n%s %d %d %ld %u",
        "Newname",1,5004,456789,d.u);
}
fclose (f);
for (int i=0; i<count; i++)
    printf ("\n%d. %s, %s, %04u \
        %06lu, %02d.%02d.%d",i+1, p[i].name,
        get_status(p[i].status),
        p[i].passport.series,
        p[i].passport.number,p[i].date.day,
        p[i].date.month, p[i].date.year+1970);
error ("");
}

```

Файл с данными `data.txt` находится в той же папке, что и приложение, он может иметь такой вид:

```
Ivanov 1 5004 096345 10000
Petrov 2 5001 233245 20000
Sidorov 0 5111 233456 30000
```

Нашему решению не хватает возможности преобразования "сжатых" дат в числа и обратно. Напишем соответствующее решение отдельной демопрограммой; функции `date_to_int` и `int_to_date` дают нужные преобразования формата:

```
#include <iostream>
using namespace std;
typedef struct date {
    unsigned short int year :7;
    unsigned short int month :4;
    unsigned short int day :5;
};
unsigned short int date_to_int (date d) {
    return ((d.year<<9)|(d.month<<5)|
    (d.day&0x001F)); }
date int_to_date (unsigned short int n) {
    date d;
    d.year=(n&0xFE00)>>9;
    d.month=(n&0x01E0)>>5;
    d.day=n&0x001F;
    return d; }
int main () {
    date d = {2013-1970, 12, 31};
    cout << "\n" << date_to_int(d);
    d=int_to_date (22431);
    cout << "\n" << d.day << "." << d.month
    << "." << d.year+1970;
    cin.sync(); cin.get(); return 0;
}
```

## Глава 12. Динамические структуры данных

Мы знаем, что элементами структур могут быть скалярные величины, массивы или другие структуры. При этом так же, как для простых типов данных, возможна организация массивов из структур. Сделав эти массивы *динамическими*, мы можем реализовать любые *динамические структуры данных*. Таким образом, пользуясь средствами динамического распределения памяти из кучи, программа может изменять текущую размерность набора структур.

При современной организации адресации ОП, теоретически можно выделить три основных класса динамических структур, отличающихся между собой двумя характеристиками (табл. 7):

- Имеет ли значение для доступа к данным физический порядок расположения элементов в ОП?
- Возможны ли произвольная вставка и удаление элементов в любом месте последовательности?

Таблица 7

Классификация динамических структур данных

| Название класса динамических структур    | Значим ли физический порядок элементов в ОП | Возможны ли произвольная вставка и удаление элементов | Примеры реализации структуры |
|------------------------------------------|---------------------------------------------|-------------------------------------------------------|------------------------------|
| Упорядоченные совокупности               | Да                                          | Нет                                                   | Динамические массивы         |
| Неупорядоченные совокупности (коллекции) | Нет                                         | Да                                                    | Списки, деревья, хэши        |
| Последовательности                       | Нет                                         | Нет                                                   | Стеки, очереди               |

Комбинация характеристик "Да"–"Да" в данном случае (см. табл. 7) противоречила бы линейной адресации памяти – в конце концов, чтобы физически добавить элемент в середину

массива, мы должны будем сдвинуть в ОП все его элементы, начиная с точки вставки.

Возможны два основных подхода к реализации динамических структур:

1. Использовать готовые библиотеки среды программирования или стандартные средства языка программирования, например, библиотеку стандартных шаблонов `stl` из стандарта C++ или библиотеку `classlib` из старых реализаций Borland C++ для DOS.

2. "Вручную" организовывать динамические наборы структур, связанных между собой указателями на свой структурный тип.

Именно второй подход обсуждается в этой главе.

Опишем структуру `student`, подобную тем, что рассмотрены в главе 10:

```
#define MAX 30 /* макс. длина фамилии */
#define ST 4 /* макс. кол-во студентов */
struct student {
    unsigned char f[MAX];
    int ekz[4];
};
student group[ST]; //Размер списка задан!
```

Недостаток такого подхода очевиден: у нас нет средств добавить в список (ST+1)-го студента. Мы могли бы просто описать указатель `student *group`, сделав наш массив студентов динамическим и выделяя под него память так же, как для скалярного массива.

Но проблема состоит в том, что, когда наши структуры становятся достаточно громоздкими, а отношения между ними – сложнее простого следования друг за другом, получается крайне невыгодно обходиться одними массивами. Например, при сортировке элементов массива требуется физическая перестановка значений его элементов в памяти, что трудоёмко, если каждый элемент – большая структура. Время поиска в массиве нужного элемента всегда растёт линейно по отношению к количеству

элементов в нём, а для более сложных структур данных это может быть не так. Кроме того, отношения между моделируемыми объектами обычно таковы, что не укладываются в простую упорядоченную совокупность (вспомните примеры из теории баз данных).

Основным механизмом моделирования коллекций и последовательностей служит включение в состав описывающей объект структуры *указателя на свой же структурный тип*, который будет хранить адреса одного или нескольких объектов, связанных с текущим. Конкретное количество этих указателей программист определяет исходя из возможного количества связей любого конкретного объекта с остальными.

Так, для простейшего односвязного списка, каждый элемент которого связан с одним следующим за ним элементом, нам достаточно было бы добавить в структуру один-единственный указатель:

```
struct student {  
    unsigned char f[MAX];  
    int ekz[4];  
    student *next;  
};
```

Указатель *next* будет просто хранить адрес того места в ОП, где находятся данные о следующем студенте.

Таким образом, *один указатель* может моделировать *одну связь*. Для двусвязного списка, каждый элемент которого связан с одним последующим и одним предыдущим элементом, нам понадобилось бы включить в структуру два указателя, на предыдущего и следующего студента:

```
student *prev, *next;
```

В случае бинарного дерева, каждый узел которого имеет левого и правого потомка, добавление к структуре было бы точно таким же:

```
student *left, *right;
```

Наконец, захоти мы по каким-то причинам организовать записи о студентах в произвольное дерево, похожее на дерево папок Windows (каждый узел может иметь произвольное количество узлов-потомков), нам пришлось бы использовать указа-

тель на указатель, хранящий адрес *динамического массива адресов узлов-потомков*:

```
student **childs;
```

Вернёмся к моделированию односвязного списка. Примем, что глобальный указатель `student *head`, описанный в программе, всегда показывает на начало списка, а у последнего элемента поле `next==NULL`. Тогда метод просмотра всего списка мог бы выглядеть так:

```
int show (student *head) {
    int count=0;
    while (1) {
        //вывести данные очередного элемента
        printf ("\n%s", head->f);
        for (int i=0; i<4; i++)
            printf (" %d",head->ekz[i]);
        count++;
        //проверка на конец списка:
        if (head->next == NULL) break;
        //переход к следующему элементу:
        head = head->next;
    }
    printf ("\nAll=%d",count);
    return count;
}
```

Так как длина списка в нашей реализации нигде не хранится, функция `show` будет её возвращать, что можно использовать в будущем. Разумеется, это лишь один из множества возможных вариантов реализации, но суть дела всегда одинакова: получив доступ к полям текущего объекта через указатель `head`, мы затем переставляем его на адрес следующего объекта, пользуясь как раз тем, что в структуру включено поле-указатель соответствующего типа:

```
head = head->next;
```

Напишем небольшую демопрограмму, которая позволит нам ввести данные списка с консоли и показать его на экране. Данные для списка будут браться из файла `data.txt`, распо-

ложенного в текущей папке и имеющего простой формат "одна строка – одна запись", например, такого:

```
Ivanov 5 4 4 5
Petrov 3 4 4 3
Sidorov 3 4 4 5
```

Текст функции main:

```
int main (void) {
    FILE *f=fopen("data.txt","rt");
    if (f==NULL) {
        printf ("\nFile data.txt not found");
        exit(1);
    }
    char buf[40]; //буфер
    student *current = NULL, //текущий элемент
    *prev = NULL, //есть ли предыдущий?
    *head; //начало списка
    do {
        fgets (buf, 40, f);
        if (strlen(buf)<10) continue;
        //Убираем слишком короткие строки
        if (current!=NULL) prev=current;
        //если был тек.эл-т, запомнили его
        current = (student *)
            malloc(sizeof(student));
        if (current==NULL) {
            printf ("\nNo memory"); exit(2);
        }
        current->next = NULL;
        //Добавляемый эл-т пометили как последний
        if (prev!=NULL) prev->next = current;
        //Если current не первый, предыдущий
        //эл-т будет показывать на него
        else head = current;
        //Иначе запомнили, что он - начало списка
        sscanf (buf,"%s %d %d %d %d",current->f,
            &current->ekz[0], &current->ekz[1],
            &current->ekz[2], &current->ekz[3]);
    }
```

```

    } while (!feof(f));
    show (head);
    fclose (f);
    fflush (stdin); getchar (); return 0;
}

```

К программе добавим обычные заголовки:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
using namespace std;
#define MAX 30 /* макс. длина фамилии */

```

Здесь мы добавляли записи в список последовательно, по мере чтения их из файла. Существенно то, что цикл прохождения по списку, реализованный в функции `show`, вовсе не предполагает физически последовательного расположения данных в ОП. Например, мы можем реализовать функции для добавления записей в начало, конец списка или произвольно выбранное место списка. Оперативная память при этом будет выделяться последовательно, но истинный порядок следования элементов, "защитый" в цепочке указателей, по которой мы перемещаемся, избавляет нас от необходимости заботиться о том, в какой именно ячейке ОП находится какой элемент (в отличие от массива).

Добавим к приложению метод, который находит в списке фамилию, содержащую в себе указанную строку `st`. Логично, что он вернёт указатель на найденную запись или `NULL`, если поиск невозможен или искомая фамилия не найдена:

```

student *search (student *head, char *st) {
    if (head==NULL) return NULL; //список пуст
    student *next = head;
    st=strlwr((char *)st);
    //поиск не зависит от регистра символов
    do {
        char *find = strstr (
            strlwr((char *)next->f), st);
        if (find!=NULL) return next;
        if (next->next==NULL) break;
    }
}

```

```

    next = next->next;
} while (1);
return NULL; //не найдено
}

```

Следует помнить, что стандартная реализация функции `strlwr` преобразования строки в нижний регистр может не обработать символы национальных алфавитов. Тогда нужна собственная аналогичная функция или выполнение соответствующих настроек компилятора и/или локали.

Тест функции `search` мог бы быть таким:

```

char *st="rov";
student *ptr=head;
do {
    //возможность неоднократного поиска
    ptr=search(ptr,st);
    if (ptr) {
        printf ("\n%s",ptr->f);
        ptr = ptr->next;
        //следующий поиск - со след. элемента
    }
} while (ptr);

```

Напишем функцию `add1` для добавления элемента `st` в начало списка, заданного указателем `head`, и функцию `add2` для аналогичного добавления элемента в конец списка. Так как в функции будут передаваться адреса уже размещённых в ОП структур типа `student`, нам понадобится также служебная функция `copy0`, способная переписать элемент списка из исходного адреса расположения `from` в указанный адрес `to`:

```

void copy0(student *to, student *from) {
    strcpy ((char *)to->f,(char *)from->f);
    for (int i=0; i<4; i++)
        to->ekz[i]=from->ekz[i];
}
student *add1 (student *head,student *st) {
    student *current =
        (student *) malloc (sizeof(student));

```

```

    copy0 (current,st);
    if (head==NULL) current->next=NULL;
    else {
        current->next = head;
        head = current;
    }
    return current;
}
student *add2 (student *head,student *st) {
    student *last=NULL;
    if (head) {
        last=head;
        while (last->next) last=last->next;
    }
    student *current =
        (student *) malloc (sizeof(student));
    copy0 (current,st);
    current->next=NULL;
    if (last) last->next = current;
    return current;
}

```

Протестировать функции из main можно так:

```

student test1 = {"Popova",4,4,4,4};
head=add1 (head,&test1);
student test2 = {"Vasilenko",5,5,5,4};
add2 (head,&test2);
show (head);

```

Обратите внимание, что add1 возвращает новую голову списка, поэтому мы сохраняем её в переменной head, всё время работы программы служащей для этой цели.

Проверку на наличие записи st в списке, начинающемся с head, можно реализовать так:

```

int check (student *head, char *s) {
    while (head) {
        if (strcmp((char *)head->f,s)==0)
            return 1;
    }
}

```

```

    head = head->next;
}
return 0;
}

```

Здесь проверяется только совпадение фамилии, с той же оговоркой для `strcmp`, что и для `strlwr` выше. Разумеется, эта функция может сканировать и часть списка, если передать вместо `head` указатель на какой-то другой элемент. Тест функции:

```

printf ("\n%d",
    check(head, (char *)test1.f)); //1
printf ("\n%d",
    check(head, "NeponyatnoKto")); //0

```

Реализуем сортировку списка методом вставок. Существенно то, что записи не меняются местами в памяти, а только переставляются указатели. Для больших объёмов данных это может быть важно. Сортировка будет упорядочивать текущий список по алфавиту (полно фамилии `f`) и возвращать указатель на начало изменённого списка. Код этой функции нелёгок для восприятия, но его тщательное изучение поможет вам разобратся с перестановкой указателей.

Указатель `q` в функции обозначает элемент, с которого мы начинаем очередной шаг, `p` – очередной элемент, с которым сравнивается `q`, `pr` – предыдущий элемент, указатель `ph` служит для контроля на конец списка (по-прежнему предполагаем, что у последнего элемента ссылка на следующий должна быть равна `NULL`). Стандартная функция `strcmp` вернёт значение больше нуля, если первая строка-аргумент "больше" второй, т.е. следует за ней по алфавиту, и их нужно переставить.

```

student *sort (student *ph) {
    student *q, *out=NULL, *p, *pr;
    //out - выход - сначала пуст
    while (ph !=NULL) {
        q = ph; ph = ph->next;
        //исключить очередной элемент
        for (p=out, pr=NULL; p!=NULL &&
            strcmp((char *)q->f, (char *)p->f)>0;

```

```

        pr=p,p=p->next) ;
        //ищем, куда включить очередной элемент;
        //strcmp задаёт критерий сравнения
        //элементов, возможен и другой критерий
        if (pr==NULL) { q->next=out; out=q; }
        //включение в начало
        else { q->next=p; pr->next=q; }
        //или после предыдущего
    }
    return out;
}

```

Так как функция возвращает указатель на начало отсортированного списка, не забудем при вызове сохранить его в head:

```

head = sort (head);
show (head);

```

Удаление из списка будем выполнять по фамилии студента (в реальности это следует делать по некоторому первичному ключу записи). Метод удаления будет возвращать указатель на голову списка, учитывая случай удаления первого элемента.

```

student *exclude (student *head, char *f) {
    if (head==NULL) return NULL;
    student *current = head, //текущая запись
            *start = head, //начало списка
            *prev = NULL; //предыдущая запись
    while (current) {
        if (strcmp((char *)current->f,f)==0) {
            if (prev) prev->next = current->next;
            else start=current->next;
            free(current); //1
            break; //2
        }
        prev = current;
        current = current->next;
    }
    return start;
}

```

Обратим внимание на операторы, помеченные комментариями 1 и 2.

1. Записи для студентов Popova и Vasilenko были добавлены как локальные *переменные стека* test1 и test2, а не динамические переменные кучи, которые получаются при распределении памяти оператором new или функциями malloc, calloc языка C. Попытка удаления этих переменных оператором delete или функцией free является некорректным действием и может привести к краху программы. Соблюдайте следующие правила при работе с динамическими данными:

- не пытайтесь освободить память, которую вы не выделяли явно, т.е. не применяйте delete или free к данным, для которых явно не выполнялся оператор new или функции malloc/calloc;

- не смешивайте в одной программе разные способы выделения/освобождения динамической памяти, т.е. new/delete и malloc/free;

- помните, что наиболее потенциально опасное действие при работе с динамическими данными – не выделение, а именно освобождение ОП, проверяйте в первую очередь delete и free.

2. После удаления одной подходящей записи функция не продолжает поиск следующей записи, а завершает работу.

Тест функции exclude:

```
head = exclude (head, "Petrov");  
show (head);
```

Вообще говоря, для односвязного списка, в отличие от стека, физическое удаление данных из памяти – трудоёмкая задача, такая же, как для массива. Часто физическое удаление заменяют перестановкой указателей, аналог которой – пометка записи на удаление в СУБД, физически ничего не удаляющая. На практике время от времени для таких данных, в которых из-за множественных удалений образуются "пустоты", следует выполнять физическую упаковку файлов с данными.

### **Задание к лабораторной работе**

Для структуры из вашего варианта задания сформировать динамический односвязный список (или другую динамическую структуру, если вариант задания предполагает это).

Обязательные элементы приложения:

- В структуре есть поле первичного ключа.
- Реализован просмотр всего списка.
- Реализовано добавление элементов в начало или в конец списка или по возрастанию ключевого поля. Обосновать выбор именно такого способа добавления элементов.
- Реализовано удаление элементов, выбранных по первичному ключу.
- Реализована правка элемента списка, выбранного по первичному ключу.

На выбор студента:

- Данные хранятся в текстовом или бинарном файле.
- Интерфейс приложения.

Возможны дополнительные вопросы при защите работы.

## Глава 13. Вопросы к экзамену и примеры экзаменационных задач

Экзаменационный билет содержит две задачи, примерное время выполнения – 2 астрономических часа. Сдаётся и проверяется письменная работа, а не файл. В случае спорной оценки возможна устная беседа по темам задач билета.

Тема задачи 1 – работа со строками, указателями или динамическими массивами.

Тема задачи 2 – работа с текстовыми или бинарными файлами.

Предполагается владение следующими основными знаниями и алгоритмическими приёмами:

- Типовые алгоритмы расчёта для последовательностей и массивов (вычисление суммы, арифметического среднего, произведения, количества, максимума или минимума, обработка всех элементов вектора и матрицы).

- Динамическое выделение памяти под вектор и матрицу, работа с динамическими данными.

- Сравнение элементов вектора или последовательности по принципу "каждый с каждым", выполняемое в двойном цикле (сортировка, перестановка элементов).

- Стандартные функции работы со строками, заданными указателями и строками класса `string`.

- Посимвольная обработка строк с помощью указателей.
- Разбиение строк на слова или по формату.
- Посимвольное, форматное, построчное чтение данных из текстовых файлов. Запись данных в файлы.

- Чтение и запись бинарных данных.

Если размеры строки или файла не указаны в условии, они предполагаются произвольными, иначе следует соблюдать размеры и ограничения из условия.

Программа обязательно тестируется, для теста достаточно заданных в коде примеров, ввод с консоли необязателен.

Проверка корректности данных нужна, если ограничения на их значения указаны в условии. Разбиение кода на функции обязательно только в том случае, если явно указано в условии, иначе оно оставляется на усмотрение студента.

**Пример 1.** Текстовый файл содержит произвольные слова на английском языке, по одному слову в строке, не более 40 слов в файле, длина слова ограничена 80 символами. Переписать слова в другой файл, отсортировав их в алфавитном порядке.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    const int SIZE = 40;
    char s[SIZE][80];
    char *p[SIZE], *temp;
    FILE *fp;
    int i, j, n = 0;
    fp = fopen("data.txt", "rt");
    if (fp == NULL) {
        printf("\nCan't open file to read!");
        exit(1);
    }
    while (1) {
        fscanf(fp, "%s", &s[n][0]);
        p[n++] = &s[n][0];
        if (feof(fp) || n > SIZE - 1) break;
    }
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++) {
            if (_stricmp(p[i], p[j]) > 0) {
                //strcmp - без учёта регистра символов
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
}
```

```

fclose(fp);
fp = fopen("res01.txt", "wt");
if (fp == NULL) {
    printf("\nCan't open file to write!");
    exit(1);
}
for (i = 0; i<n; i++) fprintf(fp, "%s\n",
p[i]);
fclose(fp);
fflush(stdin); getchar(); return 0;
}

```

**Пример 2.** Подсчитать количество строк и символов в произвольном файле, а также процент символов, являющихся пробельными (пробел, табуляция, возврат каретки и перевод строки). Строка завершается парой символов "возврат каретки" и "перевод строки". При подсчёте количества символов пара символов "возврат каретки" и "перевод строки" не интерпретируется как один символ.

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    FILE *fp;
    char c;
    long count = 0, lines = 0, divs = 0;
    float percent;
    fp = fopen("data.txt", "rb");
    if (fp == NULL) {
        printf("\nCan't open file");
        exit(1);
    }
    while (1) {
        fread(&c, 1, 1, fp);
        if (feof(fp)) break;
        count++;
    }
}

```

```

        if (c == '\n') lines++;
        if (strchr(" \t\r\n", c)) divs++;
    }
    percent = divs * 100 / count;
    printf("\n Count=%ld", count);
    printf("\n Lines=%ld", lines + 1);
    printf("\n Dirs =%4.2f%", percent);
    fclose(fp);
    fflush(stdin); getchar(); return 0;
}

```

**Пример 3.** Задана строка *s* произвольного размера. Реализовать подпрограмму удаления символов из строки, начиная с символа номер *n* ( $n \geq 0$ ) до символа номер *k* включительно ( $k \geq n$ ). Подпрограмма возвращает указатель на полученную строку или NULL, если удаление невозможно.

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
char *str2(char *s, int n, int k) {
    char *p = s;
    int i, len, l;
    len = strlen(s);
    if (n<0||n>len-1||k<0||k>len-1||n>k)
        return NULL;
    s += n;
    l = k - n + 1;
    for (i = n; i<len; i++,s++) *s = *(s + l);
    *s = '\0';
    return p;
}
int main() {
    char *s = new char[20];
    //не char *s="Hello, World!";
    //- нельзя будет менять строку!
    strcpy(s, "Hello, World!");
}

```

```

char *p = str2(s, 3, 6);
if (p == NULL) printf("\n Impossible!");
else printf("\n Result is \"%s\"", p);
fflush(stdin); getchar(); return 0;
}

```

**Пример 4.** Прочитать из бинарного файла данных динамический массив, состоящий из целых чисел. Определить, сколько раз в нём встречается максимальное значение.

Реализовать пример средствами поточной библиотеки `fstream`.

```

#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream file;
    file.open("data.txt",
        ios::in|ios::binary);
    if (!file) { return 1; }
    file.seekg(0, file.end);
    long int size = file.tellg();
    int n = (int)size / sizeof(int);
    file.seekg(0, file.beg);
    int *a = new int[n];
    if (!a) { return 2; }
    file.read((char *)&a[0], sizeof(int)*n);
    file.close();
    int max = a[0], kol = 0;
    for (int i = 0; i<n; i++) {
        printf("%d ", a[i]);
        if (a[i]>max) { max = a[i]; kol = 1; }
        else if (a[i] == max) kol++;
    }
    printf("\nMax=%d,kol=%d", max, kol);
    fflush(stdin); getchar();return 0;
}

```

**Пример 5.** Посимвольно считать абзацы (строки) из текстового файла и записать в вектор.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
    /* необходимо для cout << string */
using namespace std;
int main() {
    const char *filename = "data.txt";
    ifstream file(filename);
    int ch = 0;
    string s = (string)"";
    vector <string> v;
    while ((ch = file.get()) != EOF) {
        if (char(ch) != '\n') s += char(ch);
        else {
            v.push_back(s);
            s.clear();
        }
    }
    if (!s.empty()) v.push_back(s);
    //учесть последнюю строку,
    //если она не пуста
    file.close();
    for (unsigned i = 0; i < v.size(); i++) {
        //здесь обрабатываем элементы вектора v
        cout << (string)v[i] << '\n';
    }
    cin.get(); return 0;
}
```

## **Заключение**

В учебном пособии рассмотрены основы языка C++ и основные приёмы процедурного программирования на этом языке, представляющем собой мощный и гибкий инструмент для решения системных и прикладных задач.

Приведён синтаксис языка, примеры решения практически значимых задач, а также задания к лабораторным работам. Изучение дополнительной литературы, указанной в библиографическом списке, поможет читателю глубже освоить программирование на языке C++, остающемся одним из самых актуальных и востребованных средств разработки приложений.

Авторы будут рады вашим отзывам, пожеланиям или замечаниям, присылать которые можно на e-mail [kis@sibstrin.ru](mailto:kis@sibstrin.ru) (Кисленко Николай Петрович) и [zyryanov@sibstrin.ru](mailto:zyryanov@sibstrin.ru) (Зырянов Кирилл Игоревич).

## Библиографический список

1. Кисленко, Н. П. Основы программирования на языке Си : учеб. пособие / Н. П. Кисленко ; Новосиб. гос. архитектур.-строит. ун-т (Сибстрин). – Новосибирск : НГАСУ (Сибстрин), 2008. – 136 с.
2. Информатика : индивид. задания и метод. указания / Новосиб. гос. архитектур.-строит. ун-т (Сибстрин) ; сост.: И. Н. Мухина, Н. П. Кисленко. – Новосибирск : НГАСУ (Сибстрин), 2008. – 75 с.
3. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – Москва : Вильямс, 2013. – 304 с.
4. Шилдт, Г. С++. Базовый курс / Г. Шилдт. – Москва : Вильямс, 2015. – 624 с.
5. Страуструп, Б. Язык программирования С++ / Б. Страуструп. – Москва : Бином-Пресс, 2011. – 1136 с.
6. Васильев, А. Н. Самоучитель С++ с примерами и задачами / А. Н. Васильев. – Санкт-Петербург : Наука и Техника, 2015. – 480 с.
7. Тимофеев, В. В. С++ как он есть : самоучитель / В. В. Тимофеев. – Москва : Бином, 2014. – 336 с.
8. string (C++) [Электронный ресурс]. – Режим доступа: [https://ru.wikipedia.org/wiki/String\\_\(C%2B%2B\)](https://ru.wikipedia.org/wiki/String_(C%2B%2B)). – Загл. с экрана.
9. class std::string [Электронный ресурс]. – Режим доступа: <http://www.cplusplus.com/reference/string/string/>. – Загл. с экрана.
10. class template std::vector [Электронный ресурс]. – Режим доступа: <http://www.cplusplus.com/reference/vector/vector/>. – Загл. с экрана.
11. class std::stringstream [Электронный ресурс]. – Режим доступа: <http://www.cplusplus.com/reference/ssstream/stringstream/>. – Загл. с экрана.
12. conio.h [Электронный ресурс]. – Режим доступа: <http://sourceforge.net/projects/conioh/files/>. – Загл. с экрана.

Учебное издание

Зырянов Кирилл Игоревич  
Кисленко Николай Петрович

# ПРОГРАММИРОВАНИЕ НА C++

Учебное пособие

ISBN 978-5-7795-0817-9



Темплан 2017 г.

Редактор А.В. Тренина

Санитарно-эпидемиологическое заключение

№ 54.НС.05.953.П.006252.06.06 от 26.06.2006 г.

Подписано к печати 12.07.2017. Формат 60×84 <sup>1</sup>/<sub>16</sub> д.л.

Гарнитура Таймс. Бумага офсетная. Ризография.

Объем 6 уч.-изд.л.; 8,25 п.л. Тираж 40 экз. Заказ №

---

Новосибирский государственный  
архитектурно-строительный университет (Сибстрин)  
630008, Новосибирск, ул. Ленинградская, 113

---

Отпечатано