Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Оренбургский государственный университет»

С. А. Сильвашко

ОСНОВЫ ПРОГРАММИРОВАНИЯ МИКРОКОНТРОЛЛЕРОВ НА C++

Учебное пособие

Рекомендовано ученым советом федерального государственного бюджетного образовательного учреждения высшего образования «Оренбургский государственный университет» для обучающихся по образовательным программам высшего образования по направлениям подготовки 11.03.02 Инфокоммуникационные технологии и системы связи, 11.03.03 Конструирование и технология электронных средств и 11.03.04 Электроника и наноэлектроника

Оренбург
2019

УДК 004.42(075.8)
ББК 32.973я73
С 36

Рецензент – профессор, доктор технических наук В. Н. Булатов

Сильвашко, С. А.
С 36 Основы программирования микроконтроллеров на С++ [Электронный ресурс]: учебное пособие / С. А. Сильвашко ; Оренбургский гос. ун-т. – Оренбург : ОГУ, 2019. – 126 с.
ISBN 978-5-7410-2398-3

В учебном пособии изложены основные сведения о языке программирования С++ и основы его использования при подготовке программ для микроконтроллеров.

Учебное пособие предназначено для обучающихся по программам высшего образования по направлениям подготовки 11.03.02 Инфокоммуникационные технологии и системы связи, 11.03.03 Конструирование и технология электронных средств и 11.03.04 Электроника и наноэлектроника.

УДК 004.42(075.8)
ББК 32.973я73

ISBN 978-5-7410-2398-3

Содержание

Введение

В настоящее время в различных электронных устройствах и системах, как промышленного (например, измерительные приборы, автоматизированные системы контроля и управления и др.), так и бытового назначения (видеокамеры, принтеры, цифровые телевизоры, микроволновые печи и др.), находят широкое применение однокристальные микроконтроллеры. Функционирование микроконтроллера определяется программой, подготовленной разработчиком электронного устройства (системы).

На заре возникновения микроконтроллеров программное обеспечение для них разрабатывали на языке ассемблера. Основной недостаток ассемблерных языков состоит в том, что каждый из них ориентирован на набор команд конкретного микроконтроллера. Кроме этого язык ассемблера достаточно сложен в освоении. В последнее время при написании программ для микроконтроллеров все чаще используют язык высокого уровня С (Си). Программа, написанная на языке С, может быть использована для любого микроконтроллера, для которого есть компилятор с языка С.

В учебном пособии изложены основы программирования на языке С++, который является развитием языка С и поддерживает операторы этого языка, а также особенности подготовки программ для микроконтроллеров.

Учебное пособие предназначено для обучающихся по программам высшего образования по направлениям подготовки 11.03.02 Инфокоммуникационные технологии и системы связи, 11.03.03 Конструирование и технология электронных средств и 11.03.04 Электроника и наноэлектроника. Может быть рекомендовано всем начинающим программистам, желающим самостоятельно освоить основы программирования на С++.

1 Краткие сведения о среде программирования Microsoft Visual C++ 2010

Интегрированная среда разработки программ Microsoft Visual C++ 2010 (разработчик – корпорация Майкрософт (Microsoft Corp.)) представляет собой среду программирования на языке высокого уровня C++ (бесплатная версия – MS Visual C++ 2010 Express). Основу среды разработки составляет платформа **Microsoft .NET Framework** – встроенный компонент Windows, который поддерживает создание и выполнение приложений нового поколения и веб-служб. Основными компонентами .NET Framework являются **общеязыковая среда выполнения** (CLR) и **библиотека классов** .NET Framework, которая включает ADO.NET, ASP.NET, Windows Forms и Windows Presentation Foundation (WPF) [2].

Основными компонентами Microsoft Visual C++ 2010 являются:

– ***редактор исходного текста***;

– ***редактор ресурсов***;

– ***компилятор кода***;

– ***компилятор ресурсов*** (предназначен для компиляции текстовых файлов с описанием ресурсов (RS) в двоичные RES – файлы);

– ***компоновщик*** (служит для формирования исполняемого *.exe файла);

– ***отладчик*** (предназначен для трассировки программы (пошаговое выполнение) с целью поиска ошибок в программе);

– ряд **библиотек**.

Рассмотрим интерфейс среды разработки на основе MS Visual C++ 2010 Express.

1.1 Интерфейс среды программирования MS Visual C++ 2010 Express

Главное окно программы, которое открывается после ее запуска, имеет вид стандартного окна приложений Windows (рисунок 1.1). В верхней части окна расположено **главное меню** программы (элементы меню ***Файл***, ***Правка***, ***Вид***,

Отладка и др., рисунок 1.2). При вызове команд главного меню открываются соответствующие **ниспадающие меню**, содержащие наборы команд, сгруппированных по определенным общим признакам (примеры некоторых меню показаны на рисунке 1.3).

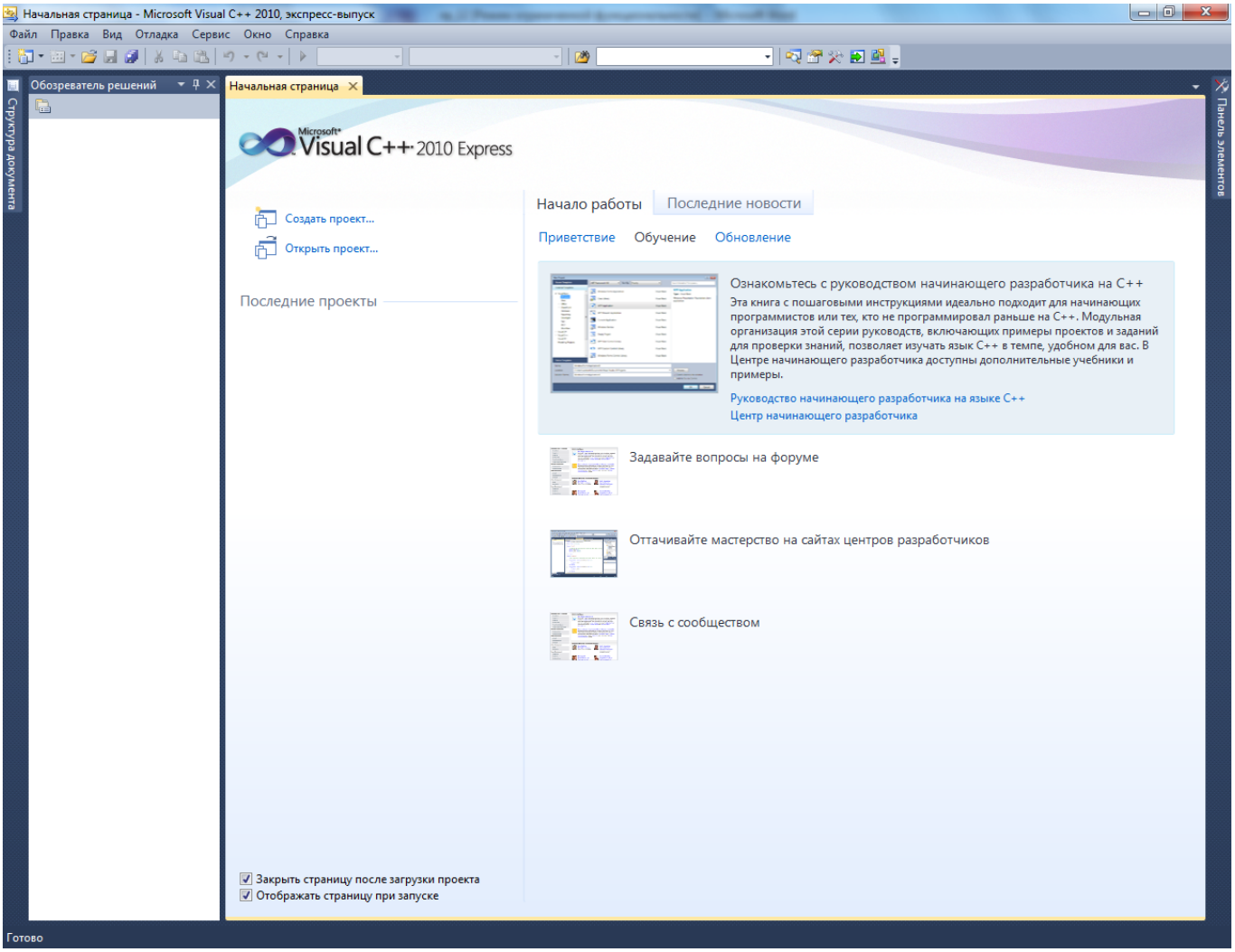


Рисунок 1.1 – Главное окно программы MS Visual C++ 2010 Express

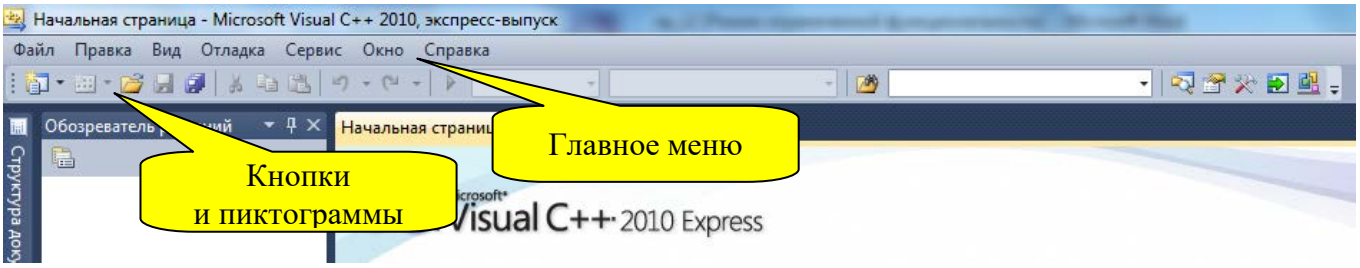


Рисунок 1.2 – Главное меню и кнопки быстрого вызова команд

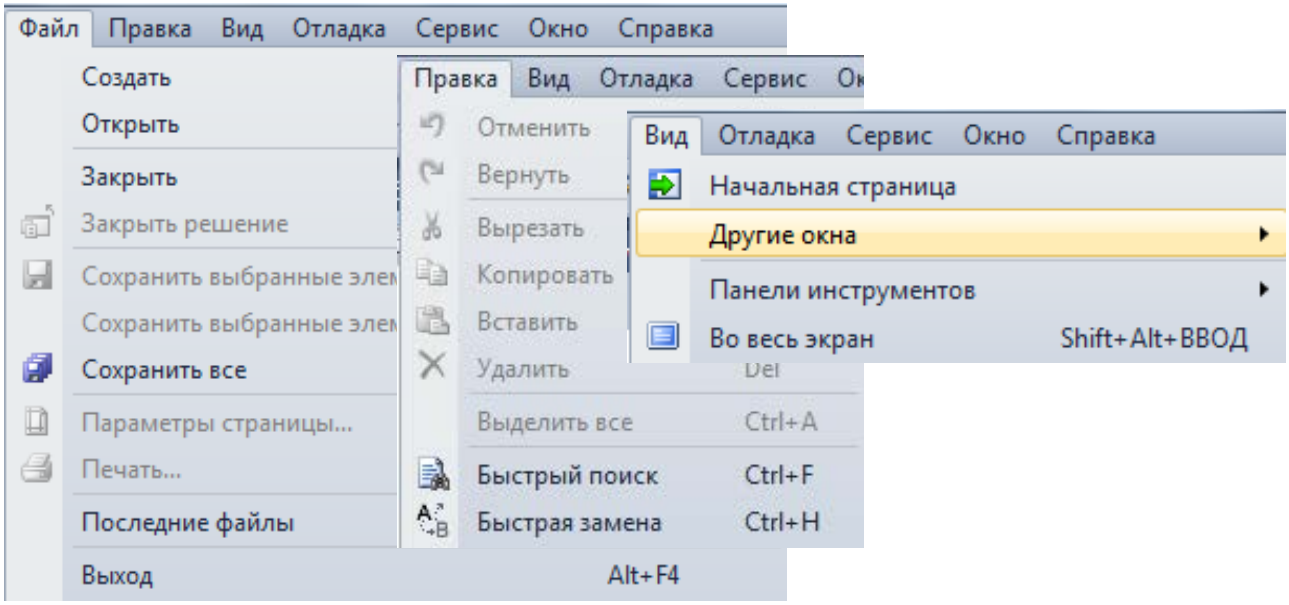


Рисунок 1.3 – Команды некоторых меню

Под главным меню расположены **кнопки** быстрого доступа к некоторым диалоговым окнам и **пиктограммы** наиболее часто используемых команд (кнопки и пиктограммы сопровождаются всплывающими подсказками).

Вид главного окна программы **изменяется после начала разработки приложения**.

Рабочий стол программы MS Visual C++ 2010 Express формируется из набора окон [5], каждое из которых имеет стандартную заголовочную полосу в верхней части. Положение окна на рабочем столе можно изменять, протягивая мышью при нажатой левой кнопке, если указатель курсора мыши расположен на заголовочной полосе. Щелчком правой кнопки мыши в области заголовочной полосы или нажатием на треугольник (кнопка <**Положение окна**>), расположенный на полосе, можно открыть диалоговое окно со свойствами соответствующего окна. На рисунке 1.4 в качестве примера показан перечень свойств окна ***Обозреватель решений***.

Сущность свойств окна:

– ***Плавающая область*** – окно с таким свойством можно разместить в любом месте рабочего стола;

– ***Закрепить*** – окно закрепляется в определенном месте рабочего стола;

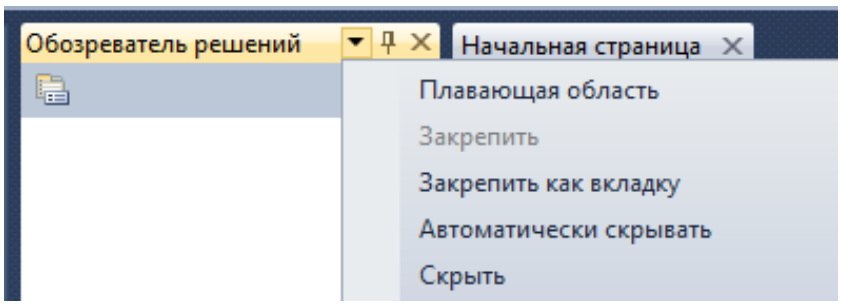


Рисунок 1.4 – Свойства окна ***Обозреватель решений***

– ***Закрепить как вкладку*** – окно размещается в качестве вкладки в основное окно рабочего стола (первоначально в нем в качестве вкладки располагается ***Начальная страница***);

– ***Автоматически скрывать*** – окно автоматически сворачивается и закрепляется в качестве вкладки на ближайшей боковой стороне основного окна рабочего стола. При наведении курсора мыши на имя окна оно автоматически всплывает (например, окно ***Обозреватель решений***, закрепленное в качестве вкладки на левой боковой стороне основного окна, автоматически всплывает при наведении на него указателя курсора, рисунок 1.5). Задать указанное свойство можно также с помощью кнопки  на заголовочной полосе с одноименным названием;

– ***скрыть*** – окно с таким свойством исчезает с экрана. Восстановить окно на рабочем столе можно с помощью меню ***Вид*** → ***Другие окна***.

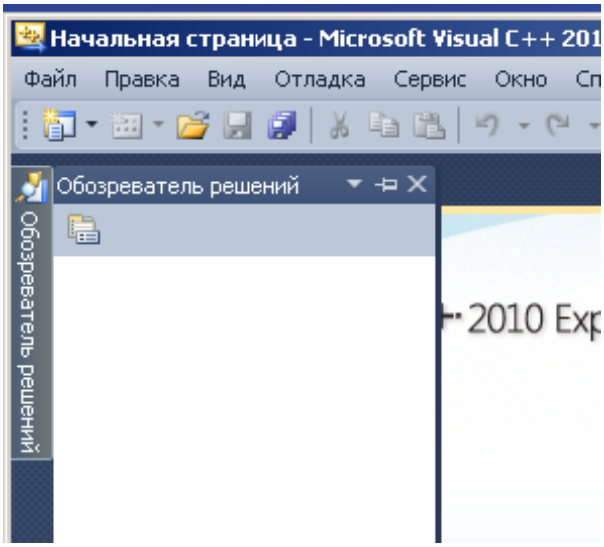


Рисунок 1.5

Программы, создаваемые в MS Visual C++, называются **приложениями**. Приложения строятся средой в виде **проектов**, представляющих собой ***совокупность нескольких файлов***. Программа на языке C++ – это ***совокупность функций***, отвечающих некоторым требованиям. При этом **приложение** является **главной функцией**, внутри которой помещаются операторы, реализующие алгоритм приложения.

Запуск любой программы начинается с запуска **главной функции**, включающей в себя всю остальную часть программы. В программе могут быть использованы как **функции**, **созданные** самим **программистом**, так и **библиотечные функции**, поставляемые со средой программирования.

В системе MS Visual C++ 2010 Express можно разрабатывать два вида приложений – **консольные** и **оконные**.

Консольное приложение – это приложение **без графического интерфейса**, которое взаимодействует с пользователем **через командную строку** (как при функционировании компьютера под управлением операционной системы MS DOS). Устройством ввода для консольного приложения является клавиатура, а устройством вывода – монитор. Пример окна программы с консольным приложением показан на рисунке 1.6.

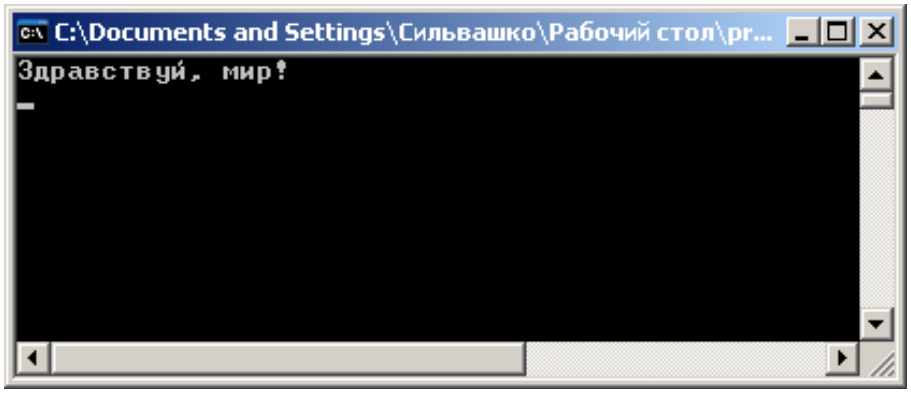


Рисунок 1.6 – Окно консольного приложения

Оконное приложение (Windows-приложение) – это приложение, в котором используется **Windows-интерфейс GUI** (Graphical User Interface – графический интерфейс пользователя). В настоящее время – это основной вид приложений для

операционной системы семейства Windows. В виде консольных приложений создают, в основном, различные утилиты, программы для тестирования технических средств компьютера и др. Консольные приложения также удобно использовать на начальной стадии изучения языка программирования.

Приложения в MS Visual C++ 2010 Express создают с помощью специальных шаблонов, доступных из диалогового окна ***Создать проект*** (рисунок 1.7).

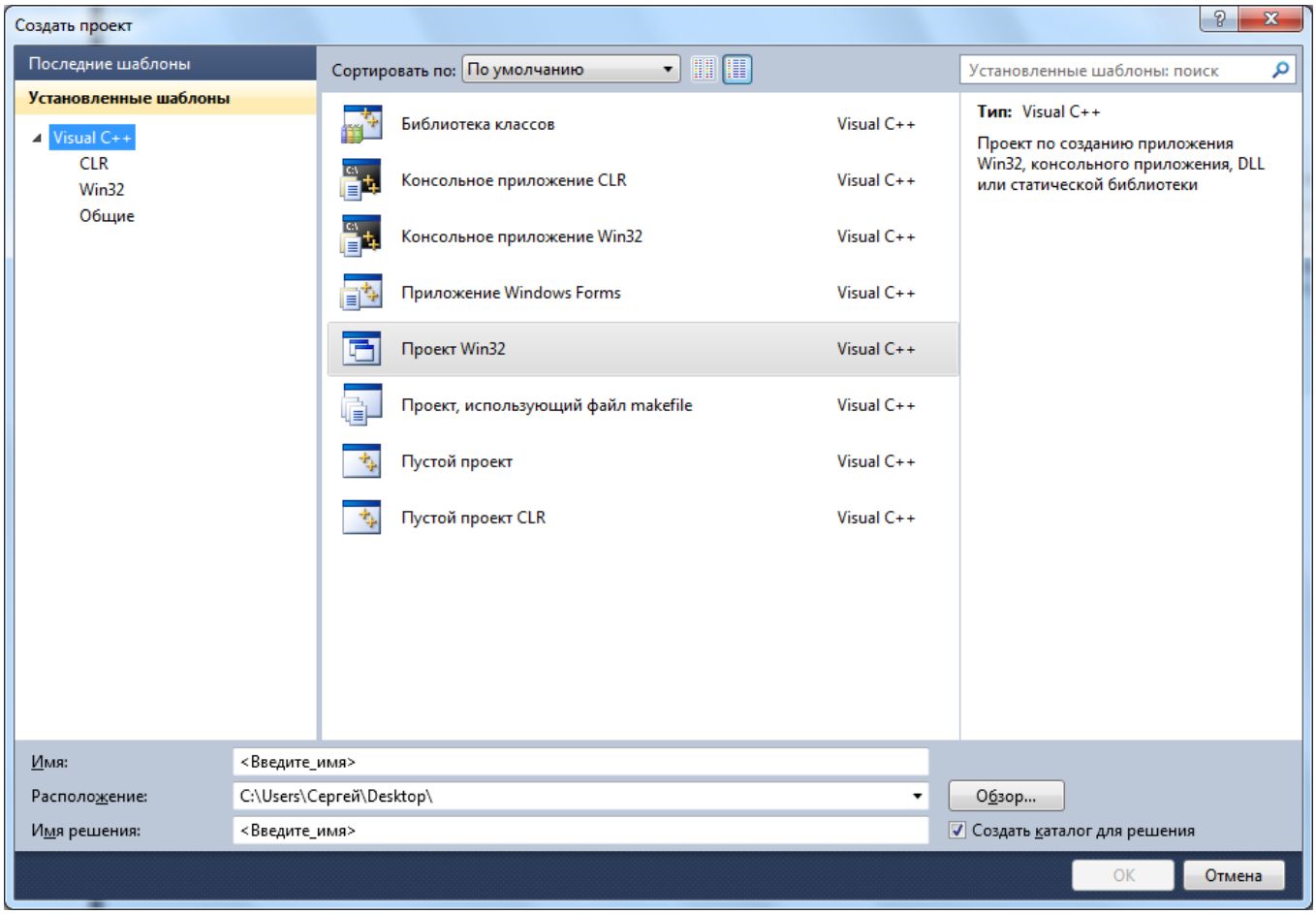


Рисунок 1.7 – Диалоговое окно ***Создать проект***

Открыть диалоговое окно ***Создать проект*** можно либо выбором команды ***Файл*** → ***Создать*** → ***Проект***, либо нажатием на кнопку <**Создать проект…**> вкладки ***Начальная страница***, (рисунок 1.1), которая открывается после запуска среды разработки MS Visual C++ 2010 Express в ее главном окне, если установлен флажок в позиции «**Отображать страницу при запуске**».

В левой части главного окна среды разработки (рисунок 1.1) размещается окно ***Обозревателя решений***. С его помощью **после создания проекта** можно осуществ-

лять навигацию по файлам, входящим в проект, отображать их содержимое в окне редактора кода, а также добавлять новые файлы к проекту.

Большую часть главного окна среды разработки при работе над проектом занимает окно ***Редактора кода*** – место, где вводят и редактируют исходный код и другие компоненты программы. До начала создания проекта сразу после запуска среды разработки в месте, отведенном для редактора кода, отображается ***Начальная страница***.

Изучение языка C++ начнем с разработки консольных приложений. Оконные приложения отличаются большим объемом программного кода, что не позволяет сосредоточиться на изучении особенностей языка программирования. Освоив C++ на консольных приложениях, перейдем к разработке оконных Windows-приложений.

1.2 Методика создания консольного приложения

Для создания консольного приложения в диалоговом окне ***Создать проект*** (рисунок 1.7) следует выбрать «***Консольное приложение CLR***».

Примечание – **CLR** (Common Language Runtime – *общеязыковая среда выполнения*) – это стандартизованная среда выполнения программ, написанных на высокоуровневых языках программирования, включая Visual Basic, C# и C++ (компонент пакета Microsoft .NET Framework).

В нижней части окна ***Создать проект*** в поле «**Имя**» задают **имя** будущего **проекта**. Такое же имя отобразится в поле «**Имя решения**». При желании решению можно присвоить **другое имя**. С помощью кнопки <**Обзор**> следует выбрать папку, в которую будет помещен проект. Если в позиции «**Создать каталог для решения**» установлен флажок, то в папке, указанной в позиции «**Расположение**», будет создана папка с именем решения. В эту папку помещается папка с именем проекта (содержащая файлы проекта), а также файлы, относящиеся к решению. При снятом флажке **файлы проекта и файлы решения помещаются в одну папку** с именем проекта.

После заполнения всех полей станет активной кнопка <**OK**> в нижней части окна (эта часть окна примет вид, как показано, например, на рисунке 1.8), нажатие

на которую вызовет создание **заготовки** будущего приложения. Вид окна программы MS Visual C++ 2010 Express после нажатия на кнопку <**OK**> показан на рисунке 1.9.

Рисунок 1.8 – Присвоение имени проекту

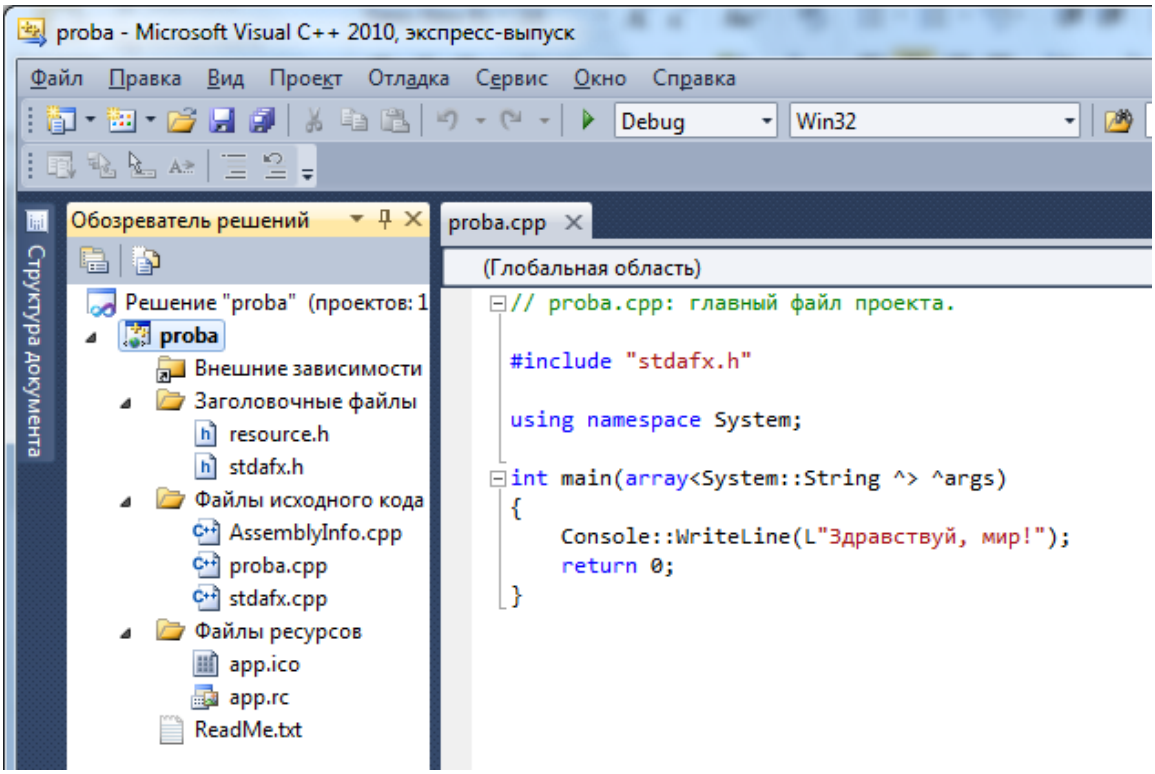


Рисунок 1.9 – Вид окна программы при работе с проектом

В окне ***Редактора кода*** на вкладке с именем «***proba.cpp***» (рисунок 1.9) отображается **заготовка** будущего консольного приложения. Слева в окне ***Обозревателя решений*** отображаются все файлы, входящие в решение (и, соответственно, в проект).

Среда MS Visual C++ 2010 Express оформляет создаваемое приложение в виде **двух контейнеров**, **вложенных один в другой**. Первый (главный) контейнер называется ***Решение***, второй (вложенный) – ***Проект***. **Проект** объединяет **группу**

файлов, имена которых отражены в окне ***Обозреватель решений***. Каждый проект содержит несколько подконфигураций, в том числе **отладочную** (Debug) и **обычную** (Release). Задать вид подконфигурации можно в ниспадающем меню, которое открывается щелчком по кнопке <**Конфигурация решения**> (по умолчанию выбрана опция «**Debug**» (отладка) (рисунок 1.9).

Проекты являются частью другого контейнера, который называется ***Решение*** и который **отражает взаимосвязь между проектами**. Для работы с группой проектов используется специальный инструмент, который называется ***Обозреватель решений***.

Примечание – Одно ***Решение*** может содержать **множество проектов**, а проект содержит множество элементов.

Как видно из рисунка 1.9, проект включает несколько файлов, в частности:

– ***proba.cpp*** – главный исходный файл и точка входа в создаваемое приложение (**proba** – это имя созданного проекта);

– ***stdafx.cpp*** – подключает для компиляции файл **stdafx.h**;

– ***stdafx.h*** – если для проекта требуются какие-то дополнительные заголовочные файлы, то они задаются пользователем в этом файле;

– ***ReadMe.txt*** – файл, описывающий некоторые из созданных шаблоном консольного приложения файлы проекта.

Посмотреть содержимое файлов можно в соответствующих окнах, открываемых через контекстные меню (щелчок правой кнопкой мыши на имени файла) выбором команды ***Открыть***.

Чтобы разрабатываемый проект можно было использовать как приложение, необходимо выполнить его **компиляцию**. Компиляция проекта осуществляется с помощью команды «***Начать отладку***», воспользоваться которой можно несколькими способами: командой меню ***Отладка → Начать отладку***; с помощью кнопки быстрого вызова команды  ; с помощью клавиши <**F5**>. Процесс компиляции проекта отображается в окне ***Вывод***, которое открывается после начала отладки приложения. Если в ходе компиляции в тексте программы обнаружены ошибки, то

исполняемый файл приложения **не создается**. Обнаруженные при компиляции ошибки отображаются в окне ***Вывод*** (рисунок 1.10).

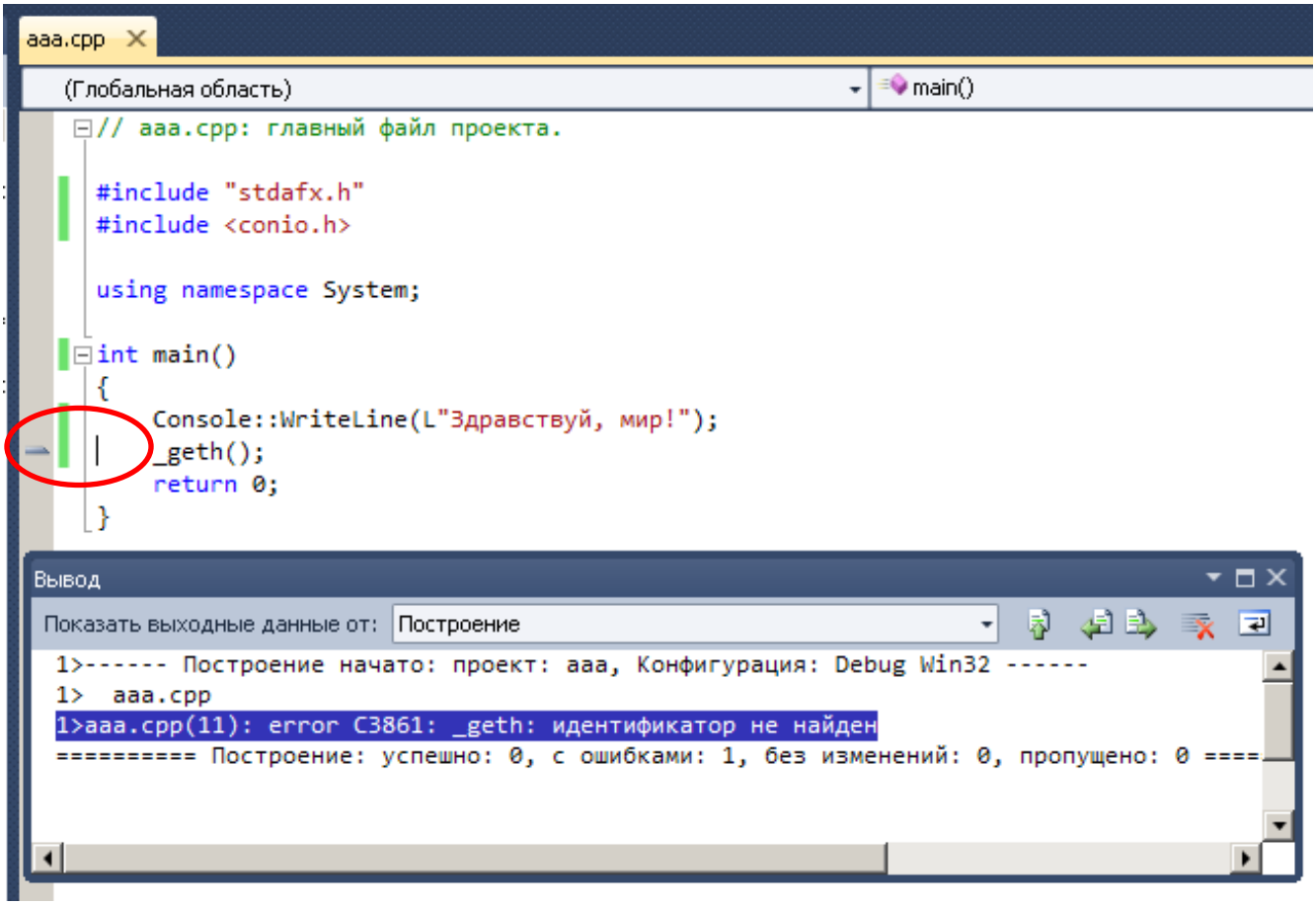


Рисунок 1.10 – Отображение результата компиляции

Кроме этого ознакомиться с ошибками, обнаруженными в ходе компиляции, можно с помощью окна ***Список ошибок***, которое открывают командой меню ***Вид*** → ***Другие окна*** → ***Список ошибок*** (рисунок 1.11).

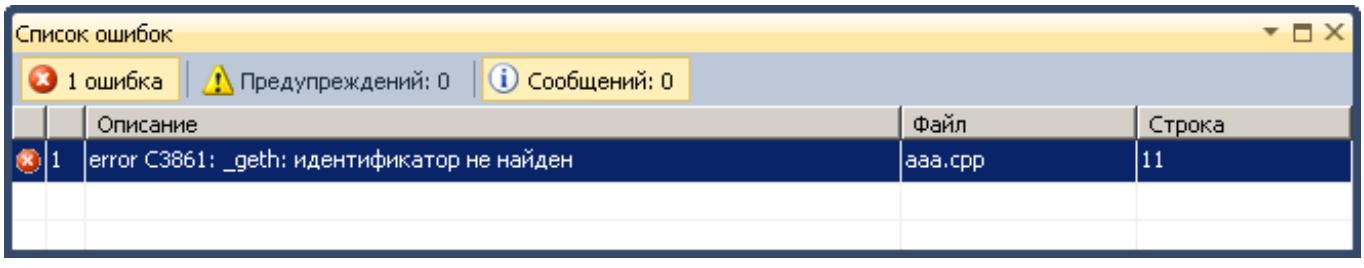


Рисунок 1.11 – Окно ***Список ошибок***

Если дважды щелкнуть по строке с сообщением об ошибке (в окне ***Вывод*** или в окне ***Список ошибок***), то указатель курсора отображается в строке текста про-

граммы (в окне редактора кода), в которой обнаружена ошибка, и в свободном поле слева (в поле подшивки) напротив этой строки появится **отметка** в виде горизонтальной черты серого цвета (рисунок 1.10).

В случае если самостоятельно понять, в чем суть ошибки, не удается, можно воспользоваться документацией, в которой изложено описание ошибок. Каждое сообщение об ошибке сопровождается номером, по которому и находят ее описание в документации. Чтобы обратиться к названной документации, нужно щелкнуть указателем курсора в окне ***Вывод*** на строке с номером ошибки, а затем нажать клавишу <**F1**>. Будет открыто новое окно с более подробным описанием ошибки.

После завершения компиляции в папку ***Debug*** проекта будет сохранен исполняемый файл приложения ****.exe*** и, одновременно, в окне интерпретатора командной строки будет выполнено консольное приложение.

Запустить консольное приложение на выполнение можно двойным щелчком по файлу ****.exe***, расположенному в папке ***Debug*** проекта.

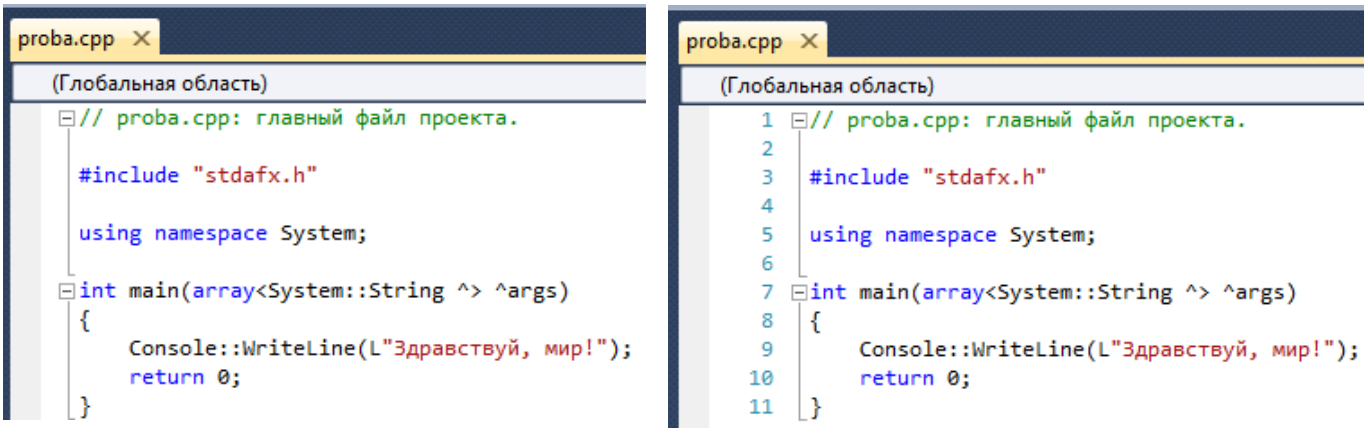
Если возникает необходимость выполнить редактирование ранее созданного проекта, то необходимо в окне программы MS Visual C++ 2010 Express командой ***Файл → Открыть → Решение или проект…*** открыть файл ****.sln***. В результате этого в окне ***Обозревателя решений*** будут открыты файлы и папки проекта, а в окне ***Редактора кода*** – файл ****.cpp*** с текстом программы (приложения). После редактирования проекта можно выполнить его компиляцию с созданием новой версии исполняемого файла ****.exe***.

В некоторых случаях нет необходимости открывать проект, а достаточно только открыть файл с текстом программы (файл ****.cpp***), например, чтобы скопировать часть текста в другой создаваемый проект. В этом случае нужно воспользоваться командой ***Файл → Открыть → Файл…*** и в папке с файлами проекта выбрать главный файл ****.cpp***. В окне ***Редактора кода*** будет открыт файл с текстом программы. Его можно скопировать, отредактировать, сохранить после редактирования, но при этом нельзя выполнить компиляцию файлов проекта. **Компиляция возможна только тогда, когда открыт проект, а не отдельные файлы из проекта.**

1.3 Структура программы, написанной на C++

Как отмечено ранее, для создания консольного приложения будем использовать шаблон «***Консольное приложение CLR***». Приложение, созданное с помощью шаблона **CLR**, отличается от приложения, созданного с помощью шаблона «***Консольное приложение Win32***» тем, что его заготовка обеспечивает подключение к приложению **специального системного пространства System**, содержащего объекты, размещение в памяти которых надо автоматически регулировать. Режим CLR обеспечивает работу с управляемой областью памяти компьютера, размещение в которой объемов данных и освобождение ее от этих объемов происходит под управлением среды.

Заготовка консольного приложения, создаваемого с использованием шаблона «***Консольное приложение CLR***», имеет вид, представленный на рисунке 1.12, *а*.



а *б*

Рисунок 1.12 – Стартовый вид окна ***Редактора кода***

Для обеспечения более удобной работы с текстом программы в окне ***Редактора кода*** можно добавить нумерацию строк (рисунок 1.12, *б*). С этой целью командой главного меню ***Сервис → Параметры…*** нужно открыть диалоговое окно ***Параметры*** (рисунок 1.13), выбрать параметры ***C/C++ → Общие*** и в правой панели в группе параметров «**Показывать**» установить флажок напротив параметра ***Номера строк***. Завершить операцию изменения параметров нажатием на кнопку <**OK**>.

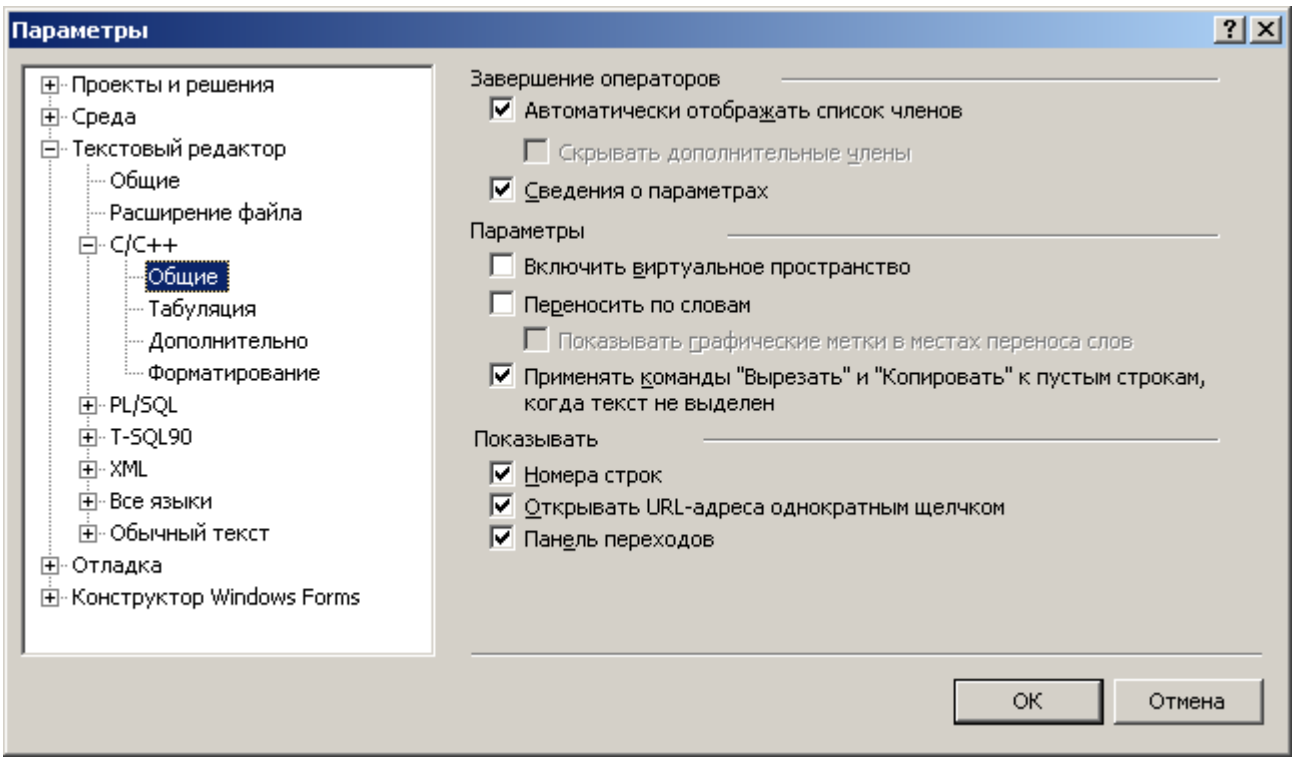


Рисунок 1.13 – Диалоговое окно ***Параметры***

Главный файл проекта на C++ содержит следующие структурные элементы:

директивы препроцессора
using namespace (название пространства имен)
int main(аргументы функции)
{
тело программы
}

В первой строке текста программы (рисунок 1.12) записан **комментарий**, который при компиляции программы игнорируется и может быть удален без каких-либо негативных последствий для самой программы. Первыми обязательными элементами в структуре любой программы являются **директивы препроцессора**. **Директива препроцессора** – это *инструкция*, которая *включает в текст программы файл, содержащий описание функций, используемых в теле программы*. Это позволяет правильно компилировать программу. **Все директивы** начинаются с фразы **#include**, после которой следует название включаемого файла. Встретив директиву

препроцессора, компилятор заменяет ее полным текстом файла, на который она ссылается. В приведенном примере (рисунок 1.12) директивой включается один стандартный заголовочный файл **stdafx.h**. Обычно в программах используется несколько директив препроцессора. Следует иметь в виду, что после директивы препроцессора (в конце строки) **точка с запятой не ставится**.

Строка программы ***using namespace***, записанная после директив препроцессора, указывает на то, какое **пространство имен** должен использовать компилятор при трансляции программы. ***Пространство имен*** (*namespace*) создает декларативную область, в которой могут размещаться различные элементы программы. **Ключевое слово *using*** информирует компилятор об использовании заявленного пространства имен. В примере (рисунок 1.12) используется пространство имен **System**.

Почти все средства библиотеки .NET (типы данных, имена переменных, стандартные функции и др.), которые используются в программе на C++, находятся в пространстве имен **System**.

Функция с именем **main(…)** является главной функцией в программе и служит так называемой **точкой входа** в программу. Наличие круглых скобок после слова ***main*** свидетельствует о том, что это имя функции. Если содержимое круглых скобок отсутствует или если в скобках записано служебное слово **void**, то это означает, что в функцию **main**() **не передается никаких аргументов**.

Аббревиатура **int** перед именем функции означает, что она является целочисленной (то есть возвращает целое число).

В **фигурных скобках** после функции **main** записывают тело программы. ***Тело программы*** *– программный блок, содержащий операторы описания, присваивания, ввода-вывода, функции и др*.

Инструкция **return 0** в последней строке тела программы (рисунок 1.12) указывает на то, что выполнение функции **main()** закончено и что в систему возвращается значение нуль (целое число). Нуль используется в соответствии с соглашением об индикации успешного завершения программы.

Если скомпилировать и выполнить программу, представленную на рисунке 1.12, то положительного результата, скорее всего, не будет. Окно консоли мигнет и

закроется. Чтобы задержать его на экране монитора, в тело программы следует добавить, например, функцию **_getch()** (для некоторых компиляторов – **getch()**). При этом следует также добавить заголовочный файл <**conio.h**>, который поддерживает функцию **_getch()**. Эта функция предназначена для приема сообщения о нажатии какой-либо клавиши на клавиатуре, после которого окно консоли закрывается. Пример программы, отредактированной с учетом приведенных выше замечаний, представлен на рисунке 1.14.

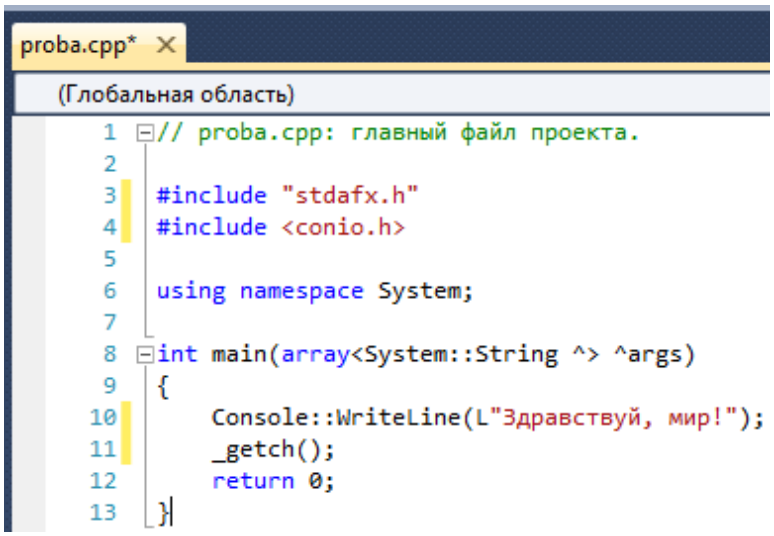


Рисунок 1.14

В разрабатываемом консольном приложении может потребоваться вывод некоторого текстового сообщения на кириллице (на русском языке). После запуска приложения может оказаться, что вместо желаемого текста в консоли отображается последовательность бессмысленных символов (рисунок 1.15).

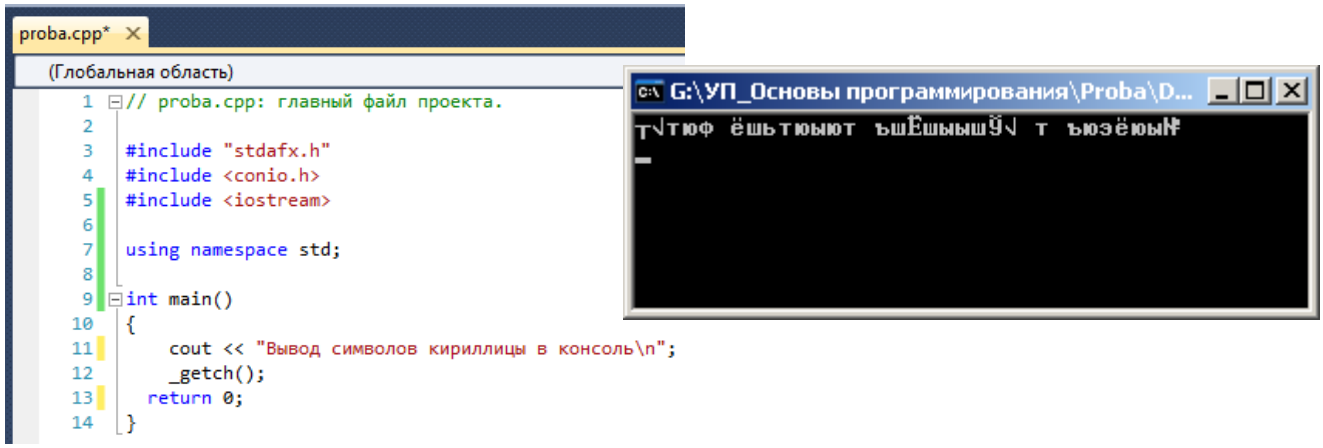


Рисунок 1.15 – Вывод текста на кириллице в консоль

Причина неправильного вывода текста на кириллице кроется в том, что в Windows используется кодировочная таблица символов **cp1251**, а в командной строке все символы кодируются с использованием кодировочной таблицы **cp866**. Причем поменять кодировку в командной строке Windows нельзя.

Решить возникающую проблему можно следующим образом: перед тем, как передать текст в консоль, необходимо его перекодировать из стандарта cp1251 в стандарт кодирования символов cp866. Одним из способов преобразования кодов символов из одного стандарта в другой является настройка **локали** (локаль – это набор параметров: набор символов, язык пользователя, страна, часовой пояс и др.). В C++ используется функция **setlocale()**, которая выполняет перекодировку символов с учетом используемого языка. Эта функция определена в заголовочном файле <**clocale**>. Пример использования функции **setlocale()** показан на рисунке 1.16. Вместо аргумента функции "**rus**" можно использовать "**Russian**". Если оставить двойные кавычки пустыми, то будет использован набор символов, который используется в качестве основного в операционной системе (то есть в русскоязычной Windows – набор символов русского языка).

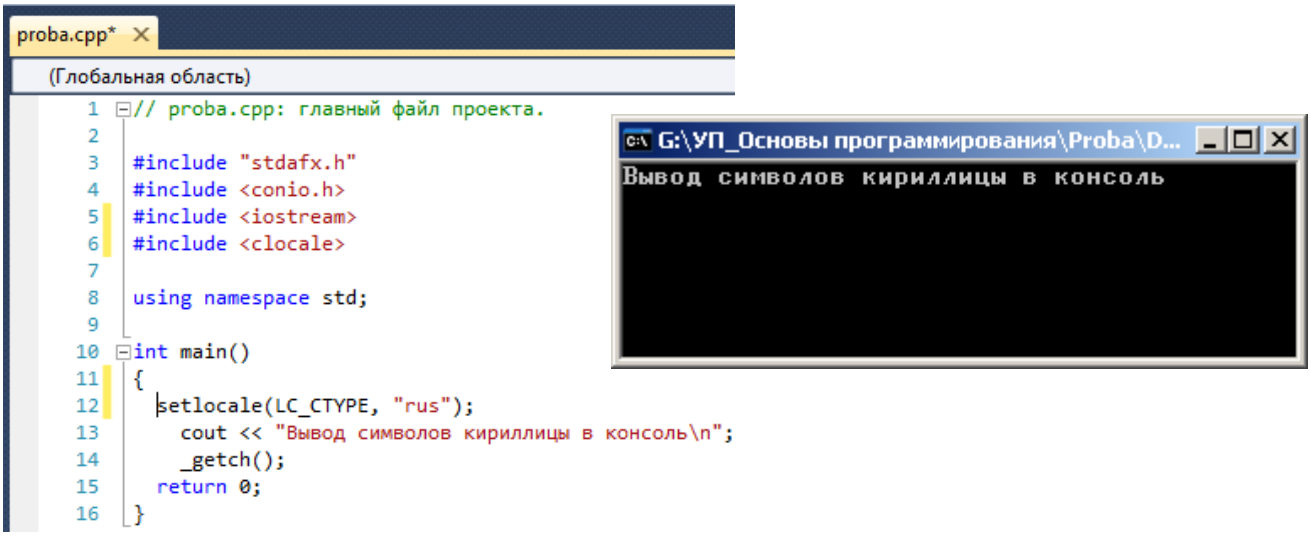


Рисунок 1.16 – Использование функции **setlocale**

1.4 Вопросы для самоконтроля

1.4.1 Какие компоненты включает интегрированная среда разработки программ Visual C++ 2010 Express?

1.4.2 Как открыть диалоговое окно со свойствами окон программы Visual C++ 2010 Express?

1.4.3 Как открыть существующий проект на C++ в Visual C++ 2010 Express? Как создать новый проект?

1.4.4 Что называется решением в среде Visual C++ 2010 Express? Чем отличается решение от проекта?

1.4.5 Какие виды проектов могут быть созданы в среде Visual C++ 2010 Express?

1.4.6 Как создать консольное приложение в среде Visual C++ 2010 Express?

1.4.7 Как запустить на выполнение консольное приложение?

1.4.8 Где отображается информация об ошибках, выявленных в ходе компиляции программы?

1.4.9 Как найти обнаруженную компилятором ошибку в тексте программы?

1.4.10 Какова структура программы на C++? Поясните назначение основных элементов в структуре программы.

1.4.11 Как обеспечить задержку на экране монитора окна консоли после запуска приложения?

1.4.12 Как обеспечить вывод текста на русском языке в консоли?

2 Основные понятия языка программирования C++

2.1 Алфавит и лексемы языка программирования

Основными элементами языка программирования C++ являются ***символы***, ***лексемы***, ***выражения*** и ***операторы***. **Лексемы** образуются из **символов**, **выражения** – из лексем и символов, а **операторы** – из символов, выражений и лексем (рисунок 2.1) [4].

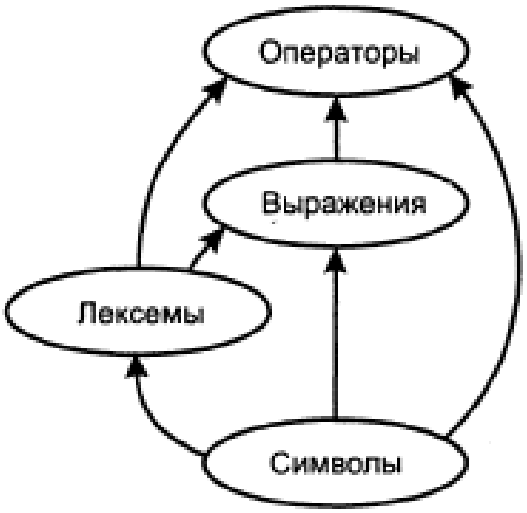


Рисунок 2.1 – Основные элементы языка C++

Символы составляют ***алфавит*** языка программирования. Все множество символов, используемых в языке C++, можно разделить на несколько групп:

1) символы, используемые для образования ***ключевых слов*** и ***идентификаторов***. К ним относятся **прописные** и **строчные** буквы латинского алфавита, **арабские цифры**, а также **символ подчеркивания** (в некоторых случаях можно использовать **прописные** и **строчные** буквы русского алфавита – в комментариях, в строковых константах, а также в названиях файлов, если это допускает среда выполнения);

2) **знаки нумерации** и **специальные символы**:

( ) [ ] { } , . : ; | \ / ' " ^ ~ _ + - * % ? < = > ! & #

Эти символы используются для организации процесса вычислений и для передачи компилятору определенного набора инструкций;

3) **пробельные символы** (пробел, символы табуляции, символы перехода на новую строку);

4) **управляющие последовательности**. Это специальные символьные комбинации, используемые в функциях ввода и вывода информации.

Управляющая последовательность начинается с *обратной дробной черты* (\), после которой следует один из допустимых символов или комбинация цифр (таблица 2.1).

Таблица 2.1 – Управляющие последовательности

Изображение	Наименование
\a	Звуковой сигнал
\b	Возврат на шаг
\f	Перевод страницы (формата)
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта
\'	Апостроф
\”	Кавычка
\?	Вопросительный знак
\ooo	Восьмеричный код символа (o – цифры от 0 до 7)
\xhh	Шестнадцатеричный код символа (h – цифры от 0 до F)
Примечание – В таблице ooo – это строка, содержащая от одной до трех восьмеричных цифр (любое целое восьмеричное число от нуля до 377), а hh – строка, содержащая одну или две шестнадцатеричных цифры (любое целое шестнадцатеричное число от нуля до FF). И в первом и во втором случаях числа представляют собой коды символов в таблице кодирования ASCII.	

Символ обратной дробной черты (\) перед буквой латинского алфавита указывает компилятору на то, что следующий за ним символ нужно интерпретировать не как соответствующую букву, а как некоторый управляющий код.

Управляющая последовательность интерпретируется компилятором как **одиночный символ**. Если непосредственно за обратной косой чертой следует символ, не предусмотренный таблицей 2.1, то **результат интерпретации не определен**. Как видно из таблицы, с помощью цифровых кодов можно вывести только символы, ко-

ды которых размещены в базовой части таблицы ASCII (буквы латинского алфавита, цифры, знаки препинания и др. – всего 256 символов).

Управляющие последовательности можно использовать, в том числе, в строковых константах. Например, если внутри строковой константы требуется записать кавычку, ее предваряют косой чертой: **"Телевизор \"Samsung\""**.

Из символов алфавита образуют **лексемы** языка программирования:

– ***идентификаторы***;

– ***ключевые слова***;

– ***знаки операций***;

– ***константы***;

– ***разделители*** (скобки, точка, запятая, пробельные символы).

Рассмотрим более подробно сущность названных лексем.

2.2 Переменные, константы, идентификаторы, ключевые слова

Переменная представляет собой величину, значение которой **во время выполнения программы** может изменяться. По своей сути **переменная** – это **именованная область памяти**, в которой хранятся данные определенного типа. Перед использованием в программе каждая переменная должна быть **объявлена** (указан тип переменной). Тем самым дается указание компилятору, какой объем памяти требуется выделить для хранения значения переменной.

Константа – величина, значение которой во время выполнения программы не изменяется. До начала использования в программе константа должна быть **объявлена** и **инициализирована** (ей должно быть присвоено значение).

Различают **целые**, **вещественные**, **символьные** и **строковые константы**. Форматы констант, соответствующие каждому типу, приведены в таблице 2.2.

Примечания

1 **Пробелы внутри числа не допускаются**, а для отделения целой части от дробной используется **точка**.

2 Элементы в формате вещественного числа, заключенные в квадратные скобки (таблица 2.2), могут отсутствовать.

Таблица 2.2 – Форматы констант C++

Константа	Формат	Примеры
Целая	**Десятичный**: последовательность десятичных цифр, начинающаяся не с нуля, если это не число нуль	12, 0, 912623
	Восьмеричный: нуль, за которым следуют восьмеричные цифры (0, 1, 2, 3, 4, 5, 6, 7)	03, 052, 063713
	Шестнадцатеричный: 0x или 0X, за которым следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)	0x4B, 0xC, 0XE07A
Вещественная	**Десятичный**: **[цифры].[цифры]**	12.3, .401, 127.
	Экспоненциальный: **[цифры] [.] [цифры]{E|e}[+|-] [цифры]**	3.17E4, .253e-3, 4E8
Символьная	Один или два символа, заключенных в апострофы	'л', 'Да', 'si', '\046', '\x17\x2D'
Строковая	Последовательность символов, заключенная в кавычки	"Я изучаю программирование", "\tЗначение r=\xF5\n"

Символьные константы, состоящие из одного символа, занимают в памяти **один байт** (одну ячейку памяти). Двухсимвольные константы занимают в памяти компьютера **два байта**, при этом первый символ размещается в байте (в ячейке памяти) с меньшим адресом.

Идентификатором называют ***имя*** константы, переменной, типа данных, функции, используемых в программе. В идентификаторе можно использовать **латинские буквы** (прописные и строчные), **цифры** и **знак подчеркивания**. При этом прописные и строчные буквы различаются, то есть ***proba***, ***Proba*** и ***proba*** – три различных идентификатора. **Цифра не может быть первым символом** идентификатора. Кроме этого **внутри** идентификатора **не допускается использовать пробелы**.

Некоторые рекомендации по выбору идентификатора:

1) использовать имена с учетом программируемой задачи;

2) давать короткие осмысленные имена, отражающие назначение *программного объекта* (переменной, функции, константы и т. п.);

3) **не начинать** идентификатор **с символа подчеркивания**, поскольку таким способом принято именовать объекты, помещенные в библиотеки системы программирования;

4) следовать единой системе именования;

5) идентификатор **не должен совпадать** с **ключевыми словами** и именами **стандартных объектов**, используемых в языке программирования.

Ключевые слова – это **зарезервированные идентификаторы**, которые используются для именования типов констант (переменных), операторов и т. п. Ключевые слова нельзя использовать в качестве имен переменных, вводимых пользователем. Список ключевых слов, используемых в языке C++, приведен в таблице 2.3 [4].

Таблица 2.3 – Ключевые слова C++

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

2.3 Знаки операций, разделители и комментарии

Знак операции – это один или несколько символов, определяющих действие над операндами. Знаки операций обеспечивают формирование и последующее выполнение выражений. Один и тот же знак операции может по-разному интерпретироваться в разных выражениях в зависимости от контекста.

Внутри знака операции (если он состоит из нескольких символов) **пробелы не допускаются**.

Все операции, допустимые в C++, с учетом количества участвующих в них операндов можно разделить на **унарные** (один операнд), **бинарные** (два операнда) и **тернарную** (три операнда).

Основные знаки операций C++ приведены в таблице 2.4 [4].

Таблица 2.4 – Основные операции языка C++

Знак операции	Операция	Группа операций
Унарные операции		
++	инкремент (увеличение значения на 1)	
--	декремент (уменьшение значения на 1)	
!	логическое отрицание	
-	унарный минус	
+	унарный плюс	
Бинарные и тернарная операции		
*	умножение	арифметические мультипликативные
/	деление	
%	остаток от деления	
+	сложение	арифметические аддитивные
-	вычитание	
<<	сдвиг влево	операции сдвига
>>	сдвиг вправо	
<	меньше	операции отношения
<=	меньше или равно	
>=	больше или равно	
==	равно	
!=	не равно	
&&	логическое И	логические операции
||	логическое ИЛИ	

Продолжение таблицы 2.4

Знак операции	Операция	Группа операций
=	присваивание	операции присваивания
*=	умножение с присваиванием	
/=	деление с присваиванием	
%=	остаток от деления с присваиванием	
-=	вычитание с присваиванием	
+=	сложение с присваиванием	
<<=	сдвиг влево с присваиванием	
>>=	сдвиг вправо с присваиванием	
?:	условная операция (тернарная)	

Рассмотрим особенности использования некоторых знаков операций.

Если константа, используемая в программе, имеет отрицательное значение, то при ее инициализации перед числом (целым или вещественным) ставится знак операции **унарный минус**. Например: **a = -138**; **c = -.022**; **d = -0.17e4**.

Операндом при использовании операций инкремент и декремент **не может быть константа**. Кроме этого операции **инкремент** (++) и **декремент** (--) могут быть **префиксными** (изменение значения операнда на единицу **до его использования**) и **постфиксными** (изменение значения операнда на единицу **после его использования**).

Примеры

1 **int a = 4, b = 5;**

c = a++ * b++;

В результате получим c = 20, a = 5, b = 6.

2 **int a = 4, b = 5;**

c = ++a * ++b;

В результате получим c = 30, a = 5, b = 6.

Если операндами при использовании операции **деление** (/) являются **целые числа**, то результат **округляется до целого**.

Пример –

int a = 10, b = 3;

c = a / b;

Получим c = 3.

Для всех операций присваивания вида **O1 x= O2**, где **O1** – левый операнд, **O2** – правый операнд, **x** – знак операции *****, **/**, **%** и т. д., эквивалентным является выражение **O1 = O1 x O2**. Например, вместо выражения **a = a + 2** можно записать **a += 2**. В этом случае к значению левого операнда (**a**) прибавляется значение правого операнда (**2**) и результат помещается в левый операнд.

В тексте программы применяют различные **разделители**. К ним относятся следующие лексемы языка C++ [6]: **[] () {} , ; : … * = # &**.

Квадратные скобки **[]** используют для указания размера одно- или многомерного массива. Например, **A[5]** – одномерный массив из пяти элементов, **B[2][3]** – матрица размерностью 2x3 (две строки, три столбца). Кроме этого с помощью квадратных скобок задают **индекс** элемента массива при использовании его значения в выражении.

Пример –

```
int x, A[] = {3, 1, 2, 5, 8, 4} // объявление и инициализация одномерного
                                 // массива из шести элементов
int B[2][3]                      // объявление двумерного массива (матрицы)
    x = A[3]                     // переменной x присваивается значение 5
```

Круглые скобки **()**:

– используют для выделения условных выражений в операторе разветвления;

– входят как обязательные элементы в определение и описание любой функции;

– используются для изменения естественной последовательности выполнения операций;

– входят как обязательные элементы в операторы циклов.

Фигурные скобки **{}** используют для обозначения начала и конца составного оператора (блока), а также при инициализации массивов и структур при их определении.

Запятая используется:

– для разделения элементов списков (начальных значений, присваиваемых индексированным элементам массивов; формальных и фактических параметров и их спецификаций в функциях);

– в качестве разделителя в заголовке оператора цикла **for**;

– в описаниях и определениях объектов (переменных) одного типа.

Точка с запятой **;** завершает каждый оператор, каждое определение (кроме определения функции) и каждое описание. Любое допустимое выражение, за которым следует точка с запятой, воспринимается как оператор.

Двоеточие **:** служит для отделения (соединения) метки и помечаемого ею оператора. Второе применение двоеточия – описание производного класса (имя производного класса отделяется от списка базовых классов двоеточием).

Многоточие (**три точки без пробелов между ними**) используется для обозначения переменного числа параметров у функции при ее определении и описании.

Звездочка ***** используется в качестве знака операции умножения и знака операции **разыменования** (получения значения через указатель).

Знак **=** используется:

– для реализации операции присваивания;

– для отделения описания объекта от списка его инициализации в определении;

– для указания на выбираемое по умолчанию значение аргумента в списке формальных параметров функции.

Символ # используется для обозначения директив препроцессора.

Символ **&** играет роль разделителя при определении переменных типа ссылки.

Примеры использования разделителей в тексте программы будут рассмотрены при изучении операторов и других конструкций языка C++.

Для пояснения отдельных частей или всей программы принято использовать ***комментарии***. Для введения **однострочного комментария** используют пару символов //, после которых следует **поясняющий текст** (в одной строке). **Многострочн-**

ный комментарий начинается с пары символов **/*** и заканчиваются парой символов ***/**. **Внутри комментария** *можно использовать любые допустимые на данном компьютере символы*, а не только символы из алфавита языка С++, поскольку комментарий не является частью кода программы и компилятор его игнорирует.

Символы комментариев можно использовать для временного исключения блоков кода программы при ее отладке.

Рекомендации по использованию комментариев:

– начинать программу с кратких комментариев, описывающих основные этапы алгоритма, переменные для хранения исходных данных, промежуточных и выводимых результатов;

– писать комментарии в терминах постановки задачи и выбранного метода решения;

– не вставлять комментарии в середину строки программы;

– не писать очевидных комментариев.

2.4 Типы данных С++

Любая переменная, используемая в программе, характеризуется двумя параметрами – **именем** и **значением**. Имени соответствует адрес участка (ячейки) памяти, выделенного переменной, а значением является содержимое этого участка. Именем переменной служит ее **идентификатор**. Значение переменной соответствует **типу** переменной, определяющему **множество допустимых значений и набор операций**, для которых переменная может служить операндом [6].

Множество допустимых значений переменных в основном совпадает с множеством допустимых значений констант того же типа (таблица 2.2). Различают **целые**, **вещественные** и **символьные** переменные. К ним следует добавить еще **логические** переменные.

Все типы данных языка С++ можно разделить на ***основные*** и ***составные***. Определено шесть **основных** типов данных для представления целых, веществен-

ных, символьных и логических величин, для описания которых определены следующие **ключевые слова**:

– **int** (целый);

– **char** (символьный);

– **wchar_t** (расширенный символьный);

– **bool** (логический);

– **float** (вещественный);

– **double** (вещественный с двойной точностью).

Четыре первых типа называют **целочисленными** (целыми), последние два – типами **с плавающей точкой**.

Используют четыре **спецификатора типа**, уточняющих внутреннее представление и диапазон значений основных типов:

– **short** (короткий);

– **long** (длинный);

– **signed** (знаковый);

– **unsigned** (беззнаковый).

Два первых спецификатора (short, long) называют **спецификаторами размера**, а два последних (signed, unsigned) – **спецификаторами знака**.

В таблице 2.5 приведены размеры в памяти и диапазоны допустимых значений основных типов данных.

Правила использования спецификаторов:

– спецификатор записывается **перед названием типа**;

– **если не указан спецификатор знака**, то по умолчанию **подразумевается** signed;

– с базовым типом **float** модификаторы **не употребляются**;

– модификатор short применим **только** к базовому типу int.

Расширенный символьный тип **wchar_t** предназначен для работы с набором символов, для кодировки которых используется более одного байта (Unicode). Строковые константы типа **wchar_t** записывают с префиксом **L**, например, **L”String”**.

Таблица 2.5 – Типы данных C++

Тип данных	Размер, байт	Диапазон значений	Назначение типа
bool	1	true или false	Логические выражения
unsigned char	1	0 … 255	Небольшие целые числа и коды символов
char (signed char)	1	-128 … 127	Очень малые целые числа и ASCII-коды
unsigned short int	2	0 … 65535	Большие целые и счетчики циклов
int (signed short int)	2	-32768 … 32767	Небольшие целые, управление циклами
unsigned long int	4	0 … 4294967295	Астрономические расстояния
signed long int	4	-2147483648 … … 2147483647	Большие числа, популяции
float	4	3.4E-38 … 3.4E+38	Научные расчеты (7 значащих цифр)
double	8	1.7E-308 … 1.7E+308	Научные расчеты (15 значащих цифр)
long double	10	3.4E-4932 … 1.1E+4932	Финансовые расчеты (19 значащих цифр)

Величины логического типа **bool** могут принимать только два значения: **true** и **false**. Внутренняя форма представления значения **false** – 0 (нуль). Любое другое значение интерпретируется как **true**. При преобразовании к целому типу **true** имеет значение 1 (единица).

Применение в определениях типов отдельных ключевых слов **int**, **char**, **short** и **long** эквивалентно **signed int**, **signed char**, **signed short** и **signed long** соответственно. Поэтому ключевое слово **signed** можно опускать в определениях и описаниях. Использование при задании типа только одного **unsigned** эквивалентно **unsigned int**.

От выбора типов переменных зависит скорость вычислений и объем памяти, занимаемой программой.

К основным типам относят также тип **void**, который используется:

– для определения функций, которые не возвращают значения;

– для указания пустого списка аргументов функции;

– в качестве базового типа для указателей;

– в операции приведения типов.

Константы с плавающей точкой по умолчанию имеют тип **double**. При этом тип константы можно указать явно с помощью суффиксов **F**, **f** (float) или **L**, **l** (long). Суффикс записывают после значения константы. Например, константа **3.12e+8L** будет иметь тип **long double**, а константа **1,27f** – тип **float**.

Из основных типов с помощью операций *****, **&**, **[]**, **()** и механизмов определения типов структурированных данных (классов, структур, объединений) создают **производные** типы [6]. Все возможные производные типы разделяют на **скалярные**, **агрегатные** и **функции**. К скалярным относят **арифметические** типы, **перечислимые** типы, **указатели** и **ссылки**. Агрегатные (структурированные) типы включают **массивы**, **структуры**, **объединения** и **классы**.

2.5 Вопросы для самоконтроля

2.5.1 Какие символы можно использовать при программировании на C++?

2.5.2 Что называют управляющими последовательностями? Приведите примеры.

2.5.3 Что называется переменной в программе на C++? Что называется константой?

2.5.4 Перечислите виды констант C++ и приведите их форматы.

2.5.5 Что называется идентификатором? Как правильно записать идентификатор переменной в C++?

2.5.6 Что называют ключевыми словами? Приведите примеры.

2.5.7 Приведите примеры знаков операций? Какие различают виды операций с учетом количества участвующих в них операндов?

2.5.8 Поясните особенности использования операций инкремент и декремент.

2.5.9 Поясните особенности использования операции деление (/).

2.5.10 Что называется комментарием? Как записать комментарий в программе?

2.5.11 Что определяет тип переменной?

2.5.12 Какие переменные различают в C++?

2.5.13 Перечислите основные типы данных C++.

3 Базовые конструкции структурного программирования

3.1 Простейшие операторы языка C++

Во многих случаях разработанное приложение предназначено для производства вычислений. Вычисления реализуют с помощью **выражений**. В общем случае выражение состоит из **операндов**, **знаков операций** и **скобок**. ***Операнды*** задают ***данные*** для вычислений. В качестве операнда может использоваться **выражение**, **константа**, **переменная** или **вызов функции**.

Знаки операций, как отмечено ранее, определяют действие (операцию) над операндами. Операции выполняются в соответствии с **приоритетами** (в таблице 2.4 основные операции языка C++ расположены сверху вниз в порядке убывания приоритетов). Для изменения порядка выполнения операций используются ***круглые скобки***. Если в одном выражении записано несколько операций одинакового приоритета, то унарные операции, условная операция и операции присваивания выполняются **справа налево**, остальные – **слева направо**.

Примеры выражений:

3.4 – x

(x + 3.14) / c

(sin(x) – 1.05e4) / ((2 * b + 5) * (2 * b + 2.3))

a = b = c // означает a = (b = c)

a + b + c // означает (a + b) + c

Любое выражение, завершающееся **точкой с запятой**, рассматривается как **оператор**. ***Оператор*** – это предложение, описывающее одно действие по обработке данных или действия программы на очередном шаге ее исполнения. Точка с запятой помечает конец оператора, но **не конец строки**. Следовательно, **один оператор может распространяться на несколько строк**, и, наоборот, **несколько операторов могут находиться в одной строке** (каждый должен заканчиваться точкой с запятой).

Пример –

```
i++;                  // выполняется операция инкремент
a *= b + c;           // выполняется умножение с присваиванием
i++; S = S + i*i;     // два оператора записаны в одной строке
fun(i, k);            // выполняется вызов функции
```

Все операторы языка С++ можно условно разделить на две группы:

– операторы **преобразования данных**;

– операторы **управления ходом выполнения программы**.

К **операторам преобразования данных** относятся **выражения** и оператор **обращения к функции**. Функцию оператора присваивания в языке С++ выполняет **выражение**, в составе которого используется операция **присваивания**.

Операцией оператора **вызова функции** являются круглые скобки, а операндами – имя функции и список ее параметров.

К операторам **управления ходом выполнения программы** относятся: условные операторы (оператор **условия** (if) и оператор **выбора** (switch)), операторы **цикла** (for, while, do while), операторы **перехода** (goto, break, continue) и оператор **возврата** (return).

К простейшим операторам языка С++ можно отнести оператор **присваивания** и операторы **ввода-вывода**.

Оператор присваивания имеет следующий формат:

имя_переменной = выражение;.

Пример –

```
x = 2*b + 5;
c = sqrt(a*a + b*b);
```

Для ввода-вывода значений переменных в С++ используют стандартные объекты-потоки **cin** (сокращение от console input – ввод с консоли) для ввода с клавиатуры и **cout** (сокращение от console output – вывод на консоль) для вывода на экран,

а также операции помещения в поток << и извлечения из потока >> [4]. Объекты-потоки **cin** и **cout**, а также операции << и >> определены в заголовочном файле <**iostream**>, содержащем описание набора классов для управления вводом-выводом. Для подключения библиотеки, связанной с потоками ввода-вывода, используют директиву препроцессора **#include <iostream>**.

Операции помещения в поток (**<<**) и извлечения из потока (**>>**) указывает в том направлении, куда передаются данные (например, **cin >> x1;** – данные с клавиатуры помещаются в переменную x1, **cout << x1;** – значение переменной x1 выводится в командную строку на экране монитора).

В заголовочном файле <**iostream**> классы объявлены в пространстве имен **std** (стандартное). Поэтому прежде чем использовать в программе объекты-потоки **cin** и **cout**, нужно указать, что используется стандартное пространство имен (**using namespace std;**). Объекты-потоки **cin** и **cout** с библиотекой **iostream** могут быть использованы и в пространстве имен **System** (**using namespace System;**), но в этом случае либо при каждом их использовании нужно писать **std::cout** (**std::cin**), что не очень удобно, либо после директив препроцессора следует записать **using std::cout;** (**using std::cin;**). В последнем случае **cin** и **cout** в тексте программы можно использовать без указания квалификатора (наименования пространства имен).

Два двоеточия, отделяющих имя пространства имен от имени элемента, образуют оператор, называемый **оператором разрешения области видимости**.

На рисунке 3.1 приведен пример фрагмента программы с использованием **cin** и **cout** в пространстве имен **std** (рисунок 3.1, *а*) и в пространстве имен **System** (рисунок 3.1, *б*).

Для ввода значений переменных с клавиатуры используют оператор ввода вида:

cin >> x1;.

Одним оператором можно ввести значения нескольким переменным, например:

cin >> x1 >> x2 >> d >> p;.

```
// квадрат.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>
    using namespace std;
    int main() {
        setlocale(LC_CTYPE, "rus");
    float x, n;
        cout << "Введите значение x" << endl;
     cin >> x;
        cout << "x=" << x << endl;
```

а

```
// квадрат.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>
    using namespace System;
    using std::cin;
    using std::cout;
    using std::endl;
    int main() {
        setlocale(LC_CTYPE, "rus");
    float x, n;
        cout << "Введите значение x" << endl;
     cin >> x;
        cout << "x=" << x << endl;
```

б

Рисунок 3.1 – Способы использования **cin** и **cout** в программе

Вводимые с клавиатуры данные должны совпадать с **типами** соответствующих переменных (на практике это могут быть **целые** или **вещественные** числа, **символьные** или **строковые** переменные). Вводимые значения переменных следует отделять пробелом (или любым другим пробельным символом). Операция потокового ввода завершается нажатием на клавишу <**Enter**>. После этого выполнение программы продолжается **со следующего оператора**.

Для вывода значений переменных на консоль используется оператор вида

cout << xl << x2 << d << p;.

Операция помещения в поток << вставляет указанные в операторе данные (в примере – значения переменных *x*1, *x*2, *d* и *p*) в выходной поток. Если требуется при выводе перевести курсор в начало следующей строки, то для этого в операторе вывода можно использовать ключевое слово **endl** (end of line – конец строки), или управляющую последовательность символа новой строки **\n**, заключив ее в кавычки (или в апострофы). Например:

cout << xl << endl;

или

cout << xl << "\n";.

Как показано выше, одним оператором можно вывести в консоль значения нескольких переменных. При этом все они будут выведены последовательно без каких-либо разделительных символов. Чтобы отделить друг от друга выводимые данные, можно, например, вставить в поток вывода пробел (или подряд несколько пробелов), заключив его в кавычки:

cout << xl << " " << x2 << " " << d << " " << p;.

Кроме этого для сдвига выводимых данных можно использовать управляющую последовательность символа табуляции **\t**:

cout << xl << "\t" << x2 << "\t" << d << "\t" << p;.

На рисунке 3.2, *а* приведен пример использования в программе ключевого слова **endl** и управляющих последовательностей для управления выводом на консоль хода решения задачи, а на рисунке 3.2, *б* – вид окна консоли с результатом работы программы.

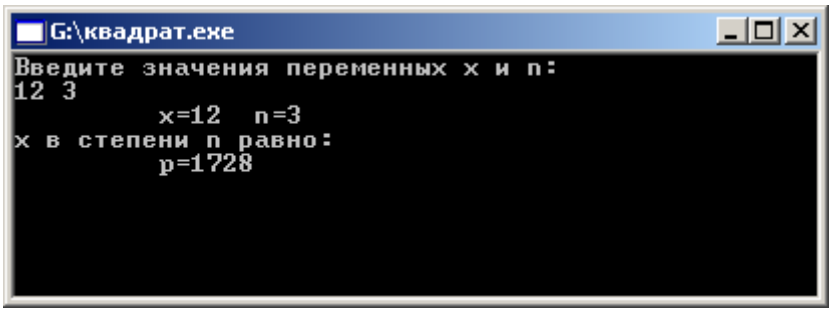
При решении прикладных задач часто возникает необходимость вывода результатов вычислений в виде массива чисел (например, при программировании циклических алгоритмов). В этом случае удобно представить результаты вычислений в виде таблицы. Как правило, предположить заранее, как будут выглядеть результаты вычислений (например, будут ли это целые или вещественные числа, сколько значащих цифр будет в целой части числа, сколько – в дробной и т. п.), невозможно. В связи с этим выведенная на экран таблица может иметь неприглядный вид. Исправить положение позволяют специальные ***манипуляторы*** и функции, предназначенные для изменения способа управления выводом данных в поток (или вводом из потока), которые определены в заголовочном файле <**iomanip**>. Чтобы в тексте программы можно было использовать манипуляторы, нужно добавить директиву препроцессора **#include <iomanip>**.

```
// квадрат.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>
    using namespace std;

    int main() {
        setlocale(LC_CTYPE, "rus");
    float x, n;
        cout << "Введите значения переменных x и n:" << endl;
     cin >> x >> n;
        cout << "\t x=" << x << "   " << "n=" << n << "\n";
     cout << "x в степени n равно:" << endl;
        cout << "\t p=" << powf(x, n);
     _getch();
     return 0;
    }
```

а



б

Рисунок 3.2 – Пример использования операторов ввода-вывода в C++

Для управления выводом информации (значений переменных) на консоль можно использовать манипулятор **setw(*n*)**, или функцию **cout.width(*n*)**, где ***n*** – количество позиций символов в поле вывода в окне консоли. Например, с помощью **setw(*n*)** или **cout.width(*n*)** можно задать ширину поля в столбце таблицы. При этом выводимые символы (например, число) выравниваются **по правому краю поля**. Манипулятор **setw(*n*)** выводит **одно** следующее за ним в команде вывода значение. Следовательно, при выводе в одной строке значений нескольких переменных перед идентификатором каждой переменной нужно записать манипулятор **setw(*n*)**. Например:

cout << "x1 = " << setw(12) << x1 << " " << "x2 = " << setw(8) << x2 << endl;.

На рисунке 3.3 приведен пример вычисления значений функции $y(x) = exp(x)$ на заданном интервале ($x \in [x_{min}, x_{max}]$) и при заданном количестве (N) значений аргумента x. Вычисленные значения функции y и соответствующие им значения аргумента x выведены в виде таблицы (рисунок 3.3, *б*). Форматирование ширины столбцов таблицы выполнено с помощью манипулятора **setw(*n*)** (рисунок 3.3, *а*).

С помощью манипуляторов **fixed** и **setprecision(*n*)** можно задать формат вывода вещественных чисел: **fixed** определяет вывод чисел с плавающей точкой в фиксированной форме, **setprecision(*n*)** – количество цифр (n) после десятичной точки. На рисунке 3.4 приведен пример использования функции **cout.width(*n*)** и указанных манипуляторов при решении ранее рассмотренной задачи.

Как видно из рисунка 3.4, функцию **cout.width(*n*)** и манипуляторы **fixed** и **setprecision(*n*)**, как и манипулятор **setw(*n*)**, при выводе в одной строке значений нескольких переменных нужно записывать перед идентификатором каждой переменной.

3.2 Программирование линейного алгоритма

Программу для решения задачи любой сложности можно составить только из трех видов алгоритмических структур: **следование** (линейный алгоритм), **ветвление** (разветвляющийся алгоритм) и **цикл** (циклический алгоритм). Следование, ветвление и цикл называют базовыми конструкциями **структурного программирования** [4].

Независимо от того, какой алгоритм вычислительного процесса (линейный, разветвляющийся или циклический) реализуют в программе, ее структура будет одинаковой (см. подраздел 1.3). Различия будут иметь место только в **теле программы**.

Наиболее простым в реализации на любом языке программирования является **линейный алгоритм**. Тело программы, реализующей линейный алгоритм, содержит операторы, которые выполняются последовательно один за другим.

```
// cikl.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>
#include <iomanip>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
        float x0, xk, x, Dx;
        int N, n = 0;
cout << "Введите диапазон изменения аргумента и число точек: x0, xk, N \n";
      cin >> x0 >> xk >> N;
cout << "x0=" << x0 << "  " << "xk=" << xk << "  " << "N=" << N << endl;
    cout << "-----------------------\n";
    cout << "| n |    x    |   y   |\n";
    cout << "-----------------------\n";
       Dx = xk / (N - 1);
            for (x = x0; x <= xk; x += Dx)
{n++;
    cout <<"|" << setw(3) << n << "|" << setw(9) << x << "|" << setw(7) << exp(x) << "|\n";}
    cout << "-----------------------\n";
     _getch();
     return 0;
}
```

а

```
G:\cikl_1\Debug\cikl.exe
Введите диапазон изменения аргумента и число точек: x0, xk, N
0 3 20
x0=0  xk=3  N=20
-----------------------
| n |    x    |   y   |
-----------------------
|  1|        0|      1|
|  2| 0.157895|1.17104|
|  3| 0.315789|1.37134|
|  4| 0.473684| 1.6059|
|  5| 0.631579|1.88058|
|  6| 0.789474|2.20224|
|  7| 0.947368|2.57891|
|  8|  1.10526|3.02002|
|  9|  1.26316|3.53657|
| 10|  1.42105|4.14148|
| 11|  1.57895|4.84985|
| 12|  1.73684|5.67938|
| 13|  1.89474| 6.6508|
| 14|  2.05263|7.78837|
| 15|  2.21053|9.12051|
| 16|  2.36842|10.6805|
| 17|  2.52632|12.5073|
| 18|  2.68421|14.6466|
| 19|   2.8421|17.1518|
| 20|         3|20.0855|
-----------------------
```

б

Рисунок 3.3 – Пример форматированного вывода данных

```
// cikl.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>
#include <iomanip>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
        float x0, xk, x, Dx;
        int N, n = 0;
cout << "Введите диапазон изменения аргумента и число точек: x0, xk, N \n";
      cin >> x0 >> xk >> N;
cout << "x0=" << x0 << "  " << "xk=" << xk << "  " << "N=" << N << endl;
    cout << "---------------------\n";
    cout << "| n |   x   |   y   |\n";
    cout << "---------------------\n";
       Dx = xk / (N - 1);
            for (x = x0; x <= xk; x += Dx)
{n++;
    cout << "|"; cout.width(3); cout << n;
    cout << "|"; cout.width(7); cout << fixed << setprecision(3) << x;
    cout << "|"; cout.width(7); cout << fixed << setprecision(3) << exp(x); cout << "|\n";}
    cout << "---------------------\n";
_getch();
    return 0;
}
```

а

```
G:\cikl\Debug\cikl.exe
Введите диапазон изменения аргумента и число точек: x0, xk, N
0 3 20
x0=0  xk=3  N=20
---------------------
| n |   x   |   y   |
---------------------
|  1|  0.000|  1.000|
|  2|  0.158|  1.171|
|  3|  0.316|  1.371|
|  4|  0.474|  1.606|
|  5|  0.632|  1.881|
|  6|  0.789|  2.202|
|  7|  0.947|  2.579|
|  8|  1.105|  3.020|
|  9|  1.263|  3.537|
| 10|  1.421|  4.141|
| 11|  1.579|  4.850|
| 12|  1.737|  5.679|
| 13|  1.895|  6.651|
| 14|  2.053|  7.788|
| 15|  2.211|  9.121|
| 16|  2.368| 10.681|
| 17|  2.526| 12.507|
| 18|  2.684| 14.647|
| 19|  2.842| 17.152|
| 20|  3.000| 20.086|
---------------------
```

б

Рисунок 3.4 – Пример форматированного вывода данных

Основными операторами, используемыми в программе, реализующей линейный алгоритм, являются **операторы ввода-вывода** и **операторы присваивания**. В качестве операндов могут использоваться **выражения**, **константы**, **переменные** или **вызов функций**.

Каждая переменная перед использованием в выражениях программы должна быть **объявлена** и **инициализирована** (то есть переменной должно быть присвоено какое-либо значение) [4]. **Объявление** переменной – это оператор программы, который задает, как минимум, **имя** и **тип переменной**. Например:

int x;

float size;.

Объявление переменной в программе необходимо для того, чтобы зарезервировать соответствующее типу переменной количество ячеек памяти (как показано ранее, значение переменной может занимать объем от одного до десяти байтов).

Инициализация переменной может быть выполнена одновременно с объявлением. Например:

float x=5.28;.

Инструкция объявления переменной (**оператор описания переменной**), в общем случае, имеет формат [4]:

[класс памяти] [const] тип_переменной имя_переменной [инициализатор];.

Элементы оператора описания, помещенные в квадратные скобки, являются необязательными.

В одной строке программы можно объявить несколько переменных. При этом в одном операторе описания можно объявить несколько переменных **одного типа**, разделяя их запятыми. Например:

float size, x, y=12.67; int z;.

Рассмотрим правила задания составных частей оператора описания. Необязательный элемент **класс памяти** определяет *время жизни* и *область видимости* пе-

ременной. Если класс памяти явно не указан, он определяется компилятором исходя из контекста объявления [4].

Время жизни переменной может быть **постоянным** (в течение выполнения программы) или **временным** (в течение выполнения блока).

Описание переменной явным или неявным (по умолчанию) образом задает ***область действия*** этой переменной, то есть ту часть программы, в которой идентификатор переменной можно использовать для доступа к области памяти компьютера, связанной с идентификатором (выделенной для хранения значения переменной, именованной этим идентификатором). ***Область действия*** зависит как от вида описания, так и от места размещения описания в тексте программы.

В зависимости от области действия различают **локальные** и **глобальные** переменные. Если переменная определена внутри блока (часть текста программы, ограниченная фигурными скобками), то она называется ***локальной***. Область действия локальной переменной распространяется от места ее описания до конца блока, **включая все вложенные блоки**. Объявления локальных переменных помещают, как правило, в начале функции, сразу за заголовком функции.

Если переменная определена вне блока, она называется ***глобальной*** и областью ее действия считается файл (то есть вся программа), в котором она определена, от точки описания до его конца.

Областью видимости идентификатора переменной называется часть текста программы, из которой возможно обращение к области памяти компьютера, связанной с идентификатором. Как правило, область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке описана переменная с таким же именем. В этом случае **внешняя переменная во вложенном блоке невидима**, хотя он и входит в ее **область действия**. Если такая переменная объявлена как глобальная, то к ней можно обратиться, используя операцию доступа к области видимости **::**.

Класс памяти может принимать одно из значений: **auto**, **extern**, **static** и **register** [4]:

– **auto** – *автоматическая* переменная. Время жизни такой переменной – с момента описания до конца блока. Освобождение памяти от значения, которым инициализирована переменная, происходит при выходе из блока, в котором она описана. **Для глобальных переменных этот спецификатор не используется, а для локальных он принимается по умолчанию**. Поэтому его можно не задавать явным образом;

– **extern** – означает, что переменная определяется в другом месте программы (например, дальше по тексту). Используется для создания переменных, доступных во всех модулях программы, в которых они объявлены;

– **static** – *статическая* переменная. Время жизни такой переменной постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть глобальными и локальными. Глобальные статические переменные видны только в том модуле, в котором они описаны;

– **register** – аналогично auto, но память выделяется по возможности в регистрах процессора. Если такой возможности у компилятора нет, переменные обрабатываются как auto.

Пример –

```
int a;                  // глобальная переменная a
int main(){
int b;                  // локальная переменная b
extern int x;           // переменная x определена в другом месте
static int c;           // локальная статическая переменная c
a = 1;                  // инициализация глобальной переменной a
int a;                  // локальная переменная a
a = 5;                  // инициализация локальной переменной a
::a = 3;                // инициализация глобальной переменной a
return 0;
}
int x = 4;              // определение и инициализация переменной x
```

Необязательный модификатор **const** указывает на то, что значение переменной изменять нельзя (данные используются **только для чтения**). При описании переменной ей можно **присвоить начальное значение** (выполнить ее **инициализацию**). Инициализатор можно записывать в двух формах: со знаком равенства («**= значение**») или в круглых скобках («**(значение)**»).

Примечание – **Константу обязательно инициализировать при объявлении**.

Результат вычисления выражения характеризуется **значением** и **типом**. Тип результата зависит от типов переменных, используемых в выражении. Например, если ***a*** и ***b*** – переменные целого типа и описаны в программе следующим образом

int a = 2, b = 5;,

то выражение ***a* + *b*** имеет значение **7** и тип **int** (то есть совпадает с типом переменных).

Примечание – Следует иметь в виду, что если переменная, которой присваивается выражение, и переменные, которые являются операндами, определены как int, то результат вычисления будет округлен до целого значения. Например:

```
int a = 3, b = 5, c = 2, d, f;
d = a / b;
f = a / c;
```

После вычисления выражений получим: **d = 0, f = 1**.

В общем случае в выражение могут входить **операнды**, относящиеся к **разным типам**. В этом случае перед вычислениями **выполняется преобразование типов** (*приведение типов*) по определенным правилам.

Любое выражение, которое должно быть обработано в ходе выполнения программы, разбивается на последовательно выполняемые операции с двумя операндами. Поэтому правила приведения операндов применяются только в отношении пар операндов [9]. Для любой пары операндов разного типа проверяются приоритеты, приведенные в списке ниже по убыванию:

– long double;

– double;

– float;

– unsigned long long;

– long long;

– unsigned long;

– long;

– unsigned int;

– int.

Операнд, тип которого расположен в списке ниже, приводится к типу операнда, тип которого указан в списке выше. Например, если один из операндов относится к типу **int**, а второй – к типу **float**, то первый операнд будет приведен к типу **float** и результат вычисления будет относиться к типу **float**.

Величины типов **char**, **signed char**, **unsigned char**, **short int** и **unsigned short int** преобразуются в тип **int**, если он может представить все значения, или в **unsigned int** в противном случае [4]. При желании программист может задать преобразования типа явным образом (см. рекомендуемую литературу по программированию на C++).

При написании программы для выполнения сложных расчетов могут использоваться различные математические функции (созданные пользователем или библиотечные). Для подключения библиотеки математических функций C++ необходимо использовать директиву **#include <cmath>**, с помощью которой в точку исходного файла, где записана эта директива, при компиляции вставляется содержимое файла **cmath** с описанием библиотечных функций. В таблице 3.1 приведены описания некоторых наиболее часто используемых функций математической библиотеки C++.

В выражениях программы на C++ можно использовать некоторые константы, числовые значения которых описаны в файле **_USE_MATH_DEFINES**. Чтобы использовать библиотечные константы в программе, необходимо перед директивой препроцессора, вставляющей в текст программы содержимое заголовочного файла <**cmath**>, записать директиву **#define** (определяет подстановку в тексте программы некоторого текста (например, числа) вместо записанного имени, значением которого этот текст является) с именем файла **_USE_MATH_DEFINES**:

#define _USE_MATH_DEFINES

#include <cmath>.

Таблица 3.1 – Математические функции C++

Запись	Значение
abs(x)	Возвращает абсолютное значение комплексного числа.
acos(x)	Вычисляет арккосинус (в радианах).
asin(x)	Вычисляет арксинус (в радианах).
atan(x)	Вычисляет арктангенс (в радианах).
cos(x)	Вычисляет значение косинуса (аргумент задавать в радианах).
cosh(x)	Вычисляет значение гиперболического косинуса (аргумент задавать в радианах).
exp(x)	Вычисляет экспоненциальное значение e в степени x.
fmod(x, y)	Остаток от деления нацело x на y.
log(x)	Вычисляет натуральный логарифм аргумента.
log10(x)	Вычисляет десятичный логарифм аргумента.
norm(x)	Вычисляет квадрат абсолютного значения аргумента.
pow(x, y)	Вычисляет x в степени y.
sin(x)	Вычисляет синус x (аргумент задавать в радианах).
sinh(x)	Вычисляет гиперболический синус (аргумент задавать в радианах).
sqrt(x)	Вычисляет положительный квадратный корень.
tan(x)	Вычисляет тригонометрический тангенс x (аргумент задавать в радианах).
tanh(x)	Вычисляет гиперболический тангенс (аргумент задавать в радианах).

Символы (идентификаторы) констант, описанных в **_USE_MATH_DEFINES**, и их числовые значения представлены в таблице 3.2.

Таблица 3.2 – Константы в C++

Символ	**Выражение**	**Значение**
M_E	*e*	2.71828182845904523536
M_LOG2E	*log*2(*e*)	1.44269504088896340736
M_LOG10E	*log*10(*e*)	0.434294481903251827651
M_LN2	*ln*(2)	0.693147180559945309417
M_LN10	*ln*(10)	2.30258509299404568402

Продолжение таблицы 3.2

Символ	Выражение	Значение
M_PI	*pi*	3.14159265358979323846
M_PI_2	*pi*/2	1.57079632679489661923
M_PI_4	*pi*/4	0.78539816339744830961 6
M_1_PI	1/*pi*	0.31830988618379067153 8
M_2_PI	2/*pi*	0.63661977236758134307 6
M_2_SQRTPI	2/*sqrt*(*pi*)	1.12837916709551257390
M_SQRT2	*sqrt*(2)	1.41421356237309504880
M_SQRT1_2	1/*sqrt*(2)	0.70710678118654752440 1

3.3 Программирование разветвляющегося алгоритма

Разветвляющимся вычислительным процессом называется *процесс, направление вычислений в котором зависит от результата проверки некоторого условия (условий)*.

Алгоритм разветвляющегося вычислительного процесса предусматривает **выбор** одной из нескольких возможных **альтернатив** (как минимум – одной из двух) в зависимости от значения исходных данных или результатов промежуточных вычислений. Каждую из альтернатив называют **ветвью алгоритма**.

Схема алгоритма разветвляющегося вычислительного процесса представлена на рисунке 3.5. Она содержит, по крайней мере, один блок **РЕШЕНИЕ**. В схеме разветвляющегося алгоритма может быть реализован **полный** (рисунок 3.5, *а*) или **неполный** (рисунок 3.5, *б*) **выбор**.

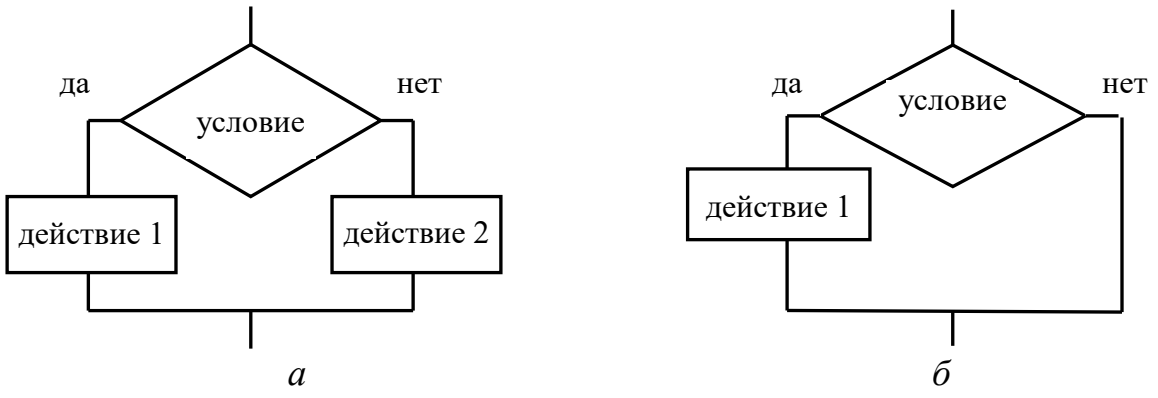


Рисунок 3.5 – Схема алгоритма разветвляющегося вычислительного процесса

В схеме алгоритма **с полным выбором** в зависимости от результата проверки условия выполняется только действие ветви **ДА** (true), если условие выполняется, или только действие ветви **НЕТ** (false), если условие не выполняется. В схеме алгоритма **с неполным выбором** выполняется действие ветви **ДА** (true) только тогда, когда условие выполняется.

Некоторой разновидностью схемы разветвляющегося вычислительного процесса с полным выбором (рисунок 3.5, *а*) является схема **ВЫБОР** (рисунок 3.6). В ней, в зависимости от результатов пошаговой проверки выполнения условий, реализуется одно из предусмотренных действий, удовлетворяющих этим условиям.

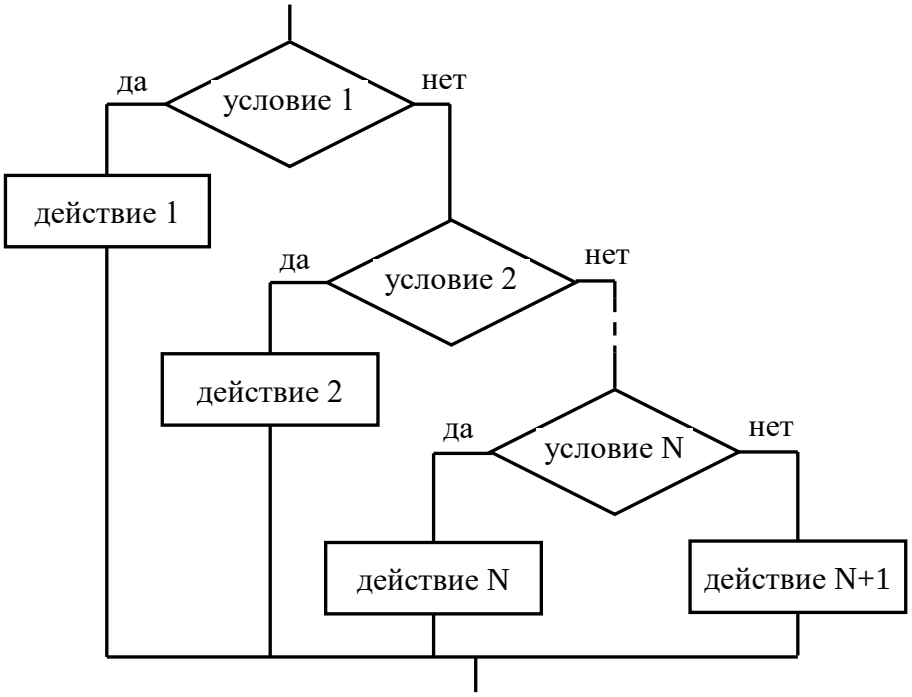


Рисунок 3.6 – Схема алгоритма **ВЫБОР**

В C++ для реализации разветвляющегося вычислительного процесса на два направления (рисунок 3.5, *а*) используется *условный оператор* **if**. Формат оператора:

if (выражение) оператор_1; [else оператор_2;].

Если реализуется разветвляющийся алгоритм **с неполным выбором** (рисунок 3.5, *б*), то содержимое в квадратных скобках не используется.

Выполнение условного оператора начинается с вычисления выражения в скобках, следующих за ключевым словом **if**. **Выражение** в условном операторе

представляет собой некоторое **условие**, содержащее операции **отношения** (таблица 3.3) и **логические операции** (таблица 3.4). Результатом операции отношения является целое число (0 – **ложь** (**fa1se**) или 1 – истина (**true**)). Если в результате вычисления выражения (проверки условия) получено значение **true**, то выполняется **оператор_1**. Если же результатом вычисления выражения явилось значение **fa1se**, то выполняется **оператор_2**. После этого **управление передается на оператор, следующий за условным**.

Таблица 3.3 – Операции отношения

Обозначение	Значение
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно
==	равно (проверка на равенство)
!=	не равно

Таблица 3.4 – Логические операции

Поразрядные логические операции		Логические операции	
обозначение	значение	обозначение	значение
&	and (и)	&&	and (и)
|	or (или)	||	or (или)
~	not (не – отрицание)	!	not (не – отрицание)
^	xor (исключающее или)		-

Если при использовании условного оператора в программе ключевое слово **else** отсутствует, то при невыполнении условия в скобках (получено значение **fa1se**) **оператор_1** не выполняется и управление сразу передается на оператор, следующий за условным.

В таблице 3.4 приведены два вида логических операций, которые могут быть использованы в выражении условного оператора. Отличие логических операций от поразрядных логических операций состоит в том, что **поразрядные логические операции** применяются только к целочисленным операндам, представленным дво-

ичными кодами. При выполнении таких операций действие выполняется с битами в одноименных разрядах (например, при выполнении операции & первый бит первого операнда логически умножается на первый бит второго операнда, второй бит первого операнда – на второй бит второго операнда и т. д.). Операнды логических операций && (and) и || (or) могут иметь арифметический тип или быть указателями. При этом каждый операнд оценивается с точки зрения его эквивалентности нулю (операнд, равный нулю, рассматривается как **false**, не равный нулю – как **true**). Результатом логической операции также является **true** или **false** [4] (например, результат операции && имеет значение **true** только тогда, когда оба операнда имеют значение **true**).

Пример –

```
if (t >= 0 && t < 3) u = (U / 3) * t;                  // реализация неполного выбора
if (t >= 0 && t < 3) u = (U / 3) * t; else u = 0;  // реализация полного выбора
```

Если в какой-либо ветви требуется выполнить не один, а несколько операторов, их необходимо **заключить в блок** (блок выделяют фигурными скобками). Блок может содержать **любые** операторы (например, операторы **описания**, операторы **вывода**, **условные** операторы и др.). После каждого оператора внутри блока ставится точка с запятой. **После закрывающей фигурной скобки точка с запятой не ставится**. Если в блоке используется условный оператор, то он называется **вложенным оператором if**. Вложенный оператор **if** может также содержать еще один вложенный оператор **if**. Таким способом можно реализовать схему алгоритма **ВЫБОР** (рисунок 3.6).

Вложенный условный оператор может быть как полным, так и сокращенным (без ключевого слова **else**). При этом следует иметь ввиду, что при вложениях условных операторов каждое ключевое слово **else** соответствует **ближайшему** к нему **предшествующему** ключевому слову **if**. Поэтому для исключения неправильного выполнения программы при использовании вложенного сокращенного условного оператора его следует **заключать в фигурные скобки**.

Пример – Составить фрагмент программы для вычисления значений функции:

$$y(x)=\begin{cases}0, \text{ при } x<0,\\ 2x, \text{ при } 0\le x<2,\\ 4, \text{ при } 2\le x<4,\\ -4x+20, \text{ при } 4\le x\le 6,\\ 0, \text{ при } x>6.\end{cases}$$

Ниже приведена часть текста программы, в которой реализовано разветвление алгоритма:

```
float x, y;
cout << "Введите значение переменной x: \n" << "x=";
        cin >> x;
if (x > 0)
        { if(x >= 0 && x < 2) y = 2*x;
        else if(x >= 2 && x < 4) y = 4;
                else if(x >= 4 && x <= 6) y = -4*x + 20; }
else y = 0;
cout << "y= " << y;
```

При использовании в выражении оператора **if** проверки переменной на равенство некоторой константе рекомендуется **константу записывать слева от операции сравнения** (**if (3 == x) y = 0**).

В некоторых случаях для реализации разветвления процесса вычислений на несколько направлений удобно использовать оператор **switch** (переключатель). Формат оператора:

```
switch (переключающее выражение)
{
        case константное_выражение_1: операторы_1;
        case константное_выражение_2: операторы_2;
        …
```

```
        case константное_выражение_n: операторы_n;
        default: операторы;
}
```

Выполнение оператора начинается с вычисления переключающего выражения (оно **должно быть целочисленным** или его значение **приводится к целочисленному**). Затем управление передается к тому оператору из списка, помеченного с помощью **case**, константное выражение которого совпало с вычисленным значением переключающего выражения. После этого, **если выход из переключателя явно не указан**, последовательно выполняются все остальные ветви.

Если значение переключающего выражения не совпало ни с одним из константных выражений, то выполняется переход к оператору, помеченному меткой **default** (а при его отсутствии – к следующему за **switch** оператору).

Для выхода из переключателя можно использовать операторы **break** или **return**.

Ниже приведен пример фрагмента программы для вычисления значений функции $y(x)$ (из рассмотренного выше примера) с помощью оператора **switch**:

```
float x, y; int d;
cout << "Выберите диапазон d изменения переменной x: \n"
        << "1, если 0 <= x < 2 \n"
        << "2, если 2 <= x < 4 \n"
        << "3, если 4 <= x <= 6 \n"
        << "4, если x < 0 или x > 6 \n" << "d=";
                cin >> d;
cout << "Введите значение переменной x: \n" << "x=";
                cin >> x;
        switch (d)
        {case 1: y = 2*x; break;
        case 2: y = 4; break;
```

```
        case 3: y = -4*x + 20; break;
        case 4: y = 0; }
cout << "y= " << y;
```

3.4 Программирование циклического алгоритма

Циклическим называется вычислительный процесс, в котором получение результата обеспечивается путем **многократного повторения некоторой последовательности действий.**

Циклические алгоритмы используются **для организации многократно повторяющихся вычислений**. Любой **цикл** состоит из **тела цикла**, то есть тех операторов, которые выполняются несколько раз, **начальных установок**, **модификации параметра цикла** и **проверки условия продолжения выполнения цикла**.

Один проход цикла называется **итерацией**. Поэтому схема алгоритма циклического вычислительного процесса получила название **ИТЕРАЦИЯ** (или **ЦИКЛ)**. Различают несколько разновидностей схем **ЦИКЛ**: **цикл с параметром, цикл с предусловием** и **цикл с постусловием.**

Схема алгоритма **цикла с параметром** представлена на рисунке 3.7, где приняты следующие сокращения: ИП – имя ячейки памяти, в которую заносится значение параметра; НЗ – начальное значение параметра; КЗ – конечное значение параметра; ШАГ – величина приращения параметра после каждого выполнения **тела цикла**.

Тело цикла может представлять собой вычислительный процесс любого из трех видов (линейный, разветвляющийся или циклический) или их сочетаний. Операторы в теле цикла выполняются столько раз, сколько разных значений примет параметр в заданных пределах от НЗ до КЗ. При использовании цикла с параметром **число повторений тела цикла известно заранее**.

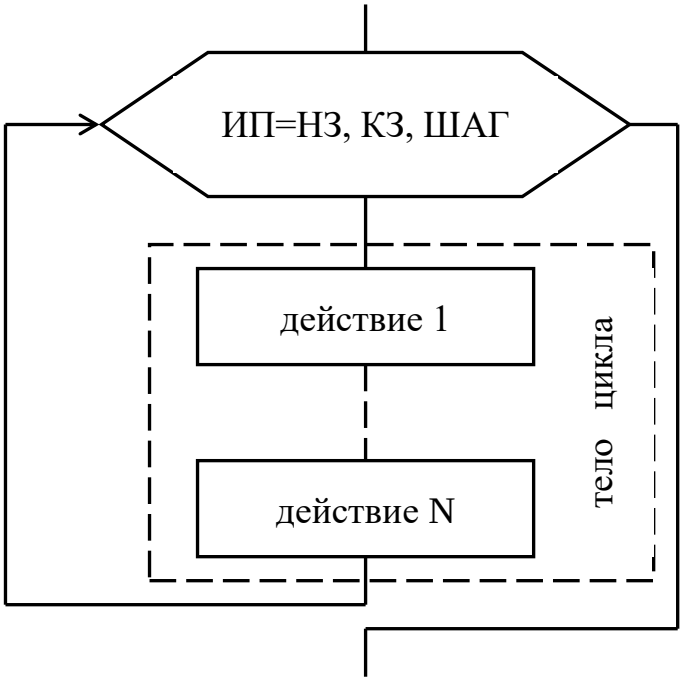


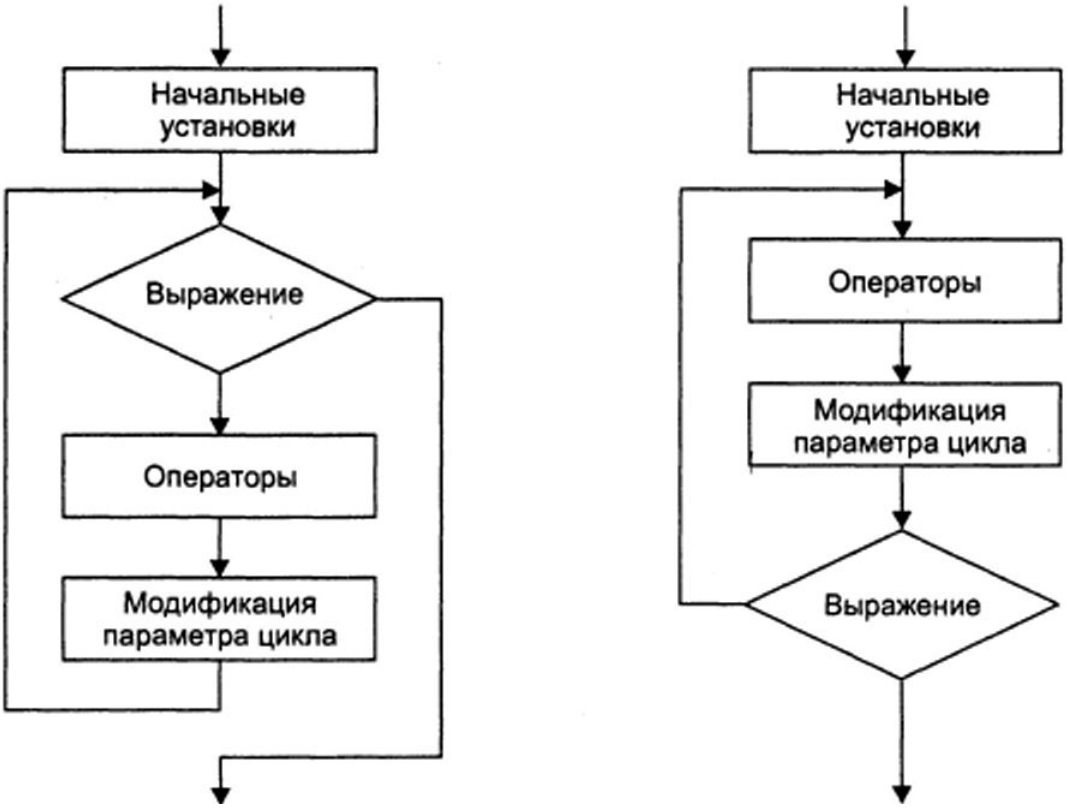
Рисунок 3.7 – Схема алгоритма **цикла с параметром**

В **цикле с предусловием** и **цикле с постусловием** число повторений тела цикла **заранее не известно**, а выход из цикла осуществляется при выполнении определенного условия, связанного с проверкой значения монотонно изменяющейся (возрастающей или убывающей) в теле цикла величины. Схема **цикла с предусловием** представлена на рисунке 3.8, *а*, а схема **цикла с постусловием** – на рисунке 3.8, *б*.

Проверка условия выполняется на каждой итерации либо **до тела цикла** (в **цикле с предусловием**), либо **после тела цикла** (в **цикле с постусловием**). Основное различие между циклом с предусловием и циклом с постусловием состоит в том, что **тело цикла с постусловием всегда выполняется хотя бы один раз**, в то время как тело **цикла с предусловием** может быть **не выполнено ни разу** [4].

Переменные, изменяющиеся в теле цикла и используемые при проверке условия продолжения цикла, называются **параметрами цикла**. Целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации, называются **счетчиками цикла**.

Цикл завершается, если условие его продолжения не выполняется.



а *б*

Рисунок 3.8 – Схемы циклических алгоритмов

В С++ для программирования циклических алгоритмов используют **три оператора цикла: while**, **do while** и **for**. Формат операторов:

– цикла с предусловием:

while (выражение-условие) {тело_цикла (выражения);};

– цикла с постусловием:

do {тело_цикла (выражения);} while (выражение-условие);;

– цикла с параметром:

for (инициализация параметра; выражение-условие; модификация параметра) {тело цикла (выражения);}.

Если тело цикла в приведенных выше операторах состоит из одного выражения, то фигурные скобки не требуются.

Выражение-условие в операторах цикла **while** и **do while** определяет условие повторения тела цикла. Выполнение оператора цикла **while** начинается с вычисления выражения-условия. Если оно истинно (**true**), выполняется тело цикла. Выражение-условие вычисляется перед каждой итерацией цикла. Если же при первой проверке выражение-условие равно **false**, цикл не выполнится ни разу и управление передается первому оператору, следующему за телом цикла.

При реализации оператора цикла **do while** сначала выполняется тело цикла (простой или составной оператор), а затем вычисляется выражение-условие. Если оно истинно (**true**), тело цикла выполняется еще раз. Цикл завершается, когда выражение-условие станет равным **false** или в теле цикла будет выполнен какой-либо оператор передачи управления. **Тип** выражения-условия должен быть **арифметическим** или **приводимым к нему**.

В операторе цикла с параметром **for инициализация** используется для присвоения начальных значений величинам, используемым в цикле. Одновременно можно и объявить используемые величины. В области инициализации можно записать несколько операторов, разделенных запятой, например:

```
int k, m;
for (k = 1, m = 0; …; …)
```

или

```
for (int k = 1, m = 0; …; …).
```

Выражение-условие определяет условие выполнения цикла: если его результат равен **true**, цикл выполняется. Как только выражение-условие принимает значение **false**, происходит выход из цикла.

Модификация выполняется после каждой итерации цикла и служит для изменения **параметра** цикла.

Любая из частей оператора **for** может быть опущена, но **точки с запятой надо оставить на своих местах**. Например, сумму целых чисел, принадлежащих

диапазону от пяти до двадцати, с помощью оператора цикла **for** можно вычислить следующим образом:

```
int max = 20, sum = 0;
        for(int i = 5; i <= max; i++)
sum += i;
```

или

```
int i = 5, max = 20, sum = 0;
        for( ; i <= max; sum += i++);.
```

Пример вычисления значений функции $y(x) = exp(x)$ на заданном интервале ($x \in [x_{min}, x_{max}]$) с использованием цикла с параметром **for** приведен на рисунках 3.3 и 3.4.

Для принудительного завершения цикла в C++ служат *операторы передачи управления* **break**, **continue**, **return** и **goto**.

Операторы цикла в общем случае взаимозаменяемы. Но на практике в каждом конкретном случае один оператор может иметь преимущество перед другими. С учетом этого следует учитывать некоторые рекомендации по выбору операторов цикла [4]:

– оператор **while** целесообразно использовать в случаях, когда **число итераций заранее не известно**, очевидных параметров цикла нет или модификацию параметров удобнее записывать не в конце тела цикла;

– оператор **do while** целесообразно использовать, когда цикл **требуется** обязательно **выполнить хотя бы раз** (например, если в цикле производится ввод данных);

– оператор **for** предпочтительнее в большинстве остальных случаев (в частности, он незаменим, если требуется организовать цикл со счетчиком).

В C++ допускается вложение одного цикла в другой (причем число вложенных циклов не ограничивается). Важно только, чтобы выход из внутреннего цикла был раньше, чем из внешнего.

Если цикл **for** вкладывается в другой цикл **for**, то допускается во внутреннем цикле в области инициализации определять переменную с таким же именем, какое имеет переменная, определенная в области инициализации внешнего цикла. Принято, что область действия имен в цикле **for** – от места размещения оператора **for** до конца блока, в котором используется этот цикл. Однако на практике с целью исключения ошибок при программировании, рекомендуется во внутренних и внешнем циклах использовать разные идентификаторы параметров циклов.

3.5 Операторы передачи управления

К операторам передачи управления в C++ относят четыре оператора, позволяющих изменить естественный порядок выполнения вычислений:

– оператор безусловного перехода **goto**;

– оператор возврата из функции **return**;

– оператор выхода из цикла или переключателя **break**;

– оператор перехода к следующей итерации цикла **continue**.

Оператор безусловного перехода **goto** имеет формат:

goto метка;,

где *метка* – это идентификатор (буква или сочетание нескольких букв), заканчивающийся двоеточием и расположенный в той же функции, где используется оператор **goto**.

Оператор **goto** передает управление на **оператор**, помеченный **меткой**. Передача управления разрешена на любой помеченный оператор в пределах функции, где используется оператор **goto**. Но при этом следует иметь в виду, что **нельзя «перескакивать» через описания**, содержащие инициализацию переменных, используемых в этой функции. Вместе с тем это ограничение не распространяется на вложенные блоки (их можно обходить целиком). **Не следует** передавать управление внутрь операторов **if**, **switch** и циклов.

Ниже приведен пример правильного и ошибочного использования оператора **goto**:

```
goto m;                          // не правильно
        float x, y;
        cin >> x;
if (x > 0)
goto m;                          // правильно
        { … }
m: cout << "x= " << x;.
```

Рекомендуется использовать оператор безусловного перехода **goto** в следующих случаях [4]:

– для принудительного выхода вниз по тексту программы из нескольких вложенных циклов или переключателей;

– для перехода из нескольких мест одной функции к одному участку программы.

В остальных случаях целесообразно использовать другие операторы передачи управления.

Оператор возврата из функции **return** имеет формат:

return [выражение];.

Он завершает выполнение функции и передает управление в точку ее вызова. Выражение в квадратных скобках является необязательным. Если тип возвращаемого функцией значения описан как **void**, выражение должно отсутствовать. Если оператор **return** используется с выражением, то оно должно иметь **скалярный тип**.

Ниже приведен пример использования оператора **return** без выражения:

```
void y(float x)
{ cout << "y=" << exp(x);
        return;
}.
```

В приведенном примере описанная функция ***y*** выводит на экран дисплея значения функции ***exp(x)***, но не возвращает в точку ее вызова никакого значения.

Оператор **break** используется для принудительного выхода из цикла или переключателя **switch** и передачи управления к оператору, находящемуся непосредственно за оператором (цикла или переключателя), внутри которого находится оператор **break**.

Оператор **continue** используется только в операторах цикла. Он пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации (то есть на проверку условий начала следующей итерации).

3.6 Указатели и ссылки

Как отмечено ранее, при объявлении переменной ей выделяется область в оперативной памяти компьютера (объем выделенной памяти зависит от типа переменной), которая заполняется конкретными данными после инициализации переменной (одним из способов, например, вводом с клавиатуры, инициализацией при объявлении переменной, вычислением значения при выполнении программы и т. п.). Каждая область памяти имеет **адрес**, по которому процессор обращается к конкретным данным в процессе вычислений по заданному алгоритму.

В памяти можно хранить не только данные, относящиеся к конкретной переменной, но и адрес области памяти, в которой хранятся эти данные. Для этой цели в программах на C++ служат специальные объекты, называемые **указателями**.

Указатель – это переменная, которая хранит адрес другой переменной. Переменная-указатель, как и любая другая переменная, имеет **имя** и **тип**. Тип указателя определяет тип данных, на которые он может указывать.

Переменные-указатели, в основном, **используют** при работе **с динамической памятью** (свободная часть оперативной памяти компьютера, в которой во время выполнения программы можно выделять место в соответствии с потребностями). Доступ к выделенным участкам динамической памяти производится только через указатели [4]. Кроме этого в некоторых случаях указатели удобно использовать при выполнении операций с данными, хранимыми в массиве, а также для обеспечения

доступа функций к объемным блокам данных (например, к массивам), определенным вне этих функций [9].

Различают три вида указателей:

– указатели на объект;

– указатели на функцию;

– указатели на void.

Перечисленные виды указателей отличаются свойствами и набором допустимых операций.

Указатель на функцию используют **для косвенного вызова функции** (не через ее имя, а через обращение к переменной, хранящей ее адрес), а также для передачи имени функции в другую функцию в качестве параметра.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа.

***Указатель на* void** применяется, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Для доступа к переменной, на которую указывает переменная-указатель, используют **оператор косвенного доступа (*)**, который записывают перед именем указателя. Простейшее объявление указателя на объект имеет вид:

тип *имя;.

Звездочка относится непосредственно к имени указателя. Одновременно можно объявить несколько указателей, относящихся к одному типу, но при этом звездочку требуется ставить перед именем каждого указателя. Например:

```
int *a, *x, y;
```

В примере описываются два указателя на целые переменные с именами **a** и **x**, а также целая переменная **y**.

Инициализацию указателя можно выполнить несколькими способами [4], одним из которых является присваивание ему адреса существующего объекта с помощью операции получения адреса:

int a = 3; // описание и инициализация целой переменной
int *p = &a; // в указатель записывается адрес ячейки памяти, в которой
// хранится значение переменной a
int *p(&a); // то же, но другим способом.

В рассмотренном примере для инициализации указателя адресом переменной используется **оператор получения адреса &**.

Более подробную информацию об использовании переменных-указателей можно получить, например, из [4].

Ссылка представляет собой **другое имя** (*синоним имени*) уже существующего объекта. Формат объявления ссылки:

тип & имя_ссылки инициализатор;,

где **тип** – это тип величины, на которую указывает ссылка, **&** – оператор ссылки, означающий, что следующее за ним имя является именем переменной ссылочного типа.

Инициализация ссылки может быть выполнена одним из способов:

тип& имя_ссылки = выражение;

или

тип& имя_ссылки(выражение);.

В качестве выражения, инициализирующего ссылку, должно использоваться имя объекта, ранее объявленного (то есть объекта, которому отведено место в памяти). Значением ссылки после объявления становится адрес этого объекта.

Пример –

int x = 4; // объявление и инициализация переменной x
int& mx = x; // объявление и инициализация ссылки mx.

Ссылка не является полноценным объектом (например, как переменная). После инициализации значение ссылки изменить нельзя.

При использовании ссылок необходимо придерживаться следующих правил [4]:

– переменная-ссылка должна инициализироваться явным образом при ее описании, кроме случаев, когда она является параметром функции, описана как **extern** или ссылается на поле данных класса;

– после инициализации ссылке нельзя присвоить другую переменную;

– тип ссылки должен совпадать с типом величины, на которую она ссылается;

– не разрешается определять указатели на ссылки, создавать массивы ссылок и ссылки на ссылки.

Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем ранее определенной величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

3.7 Обработка массивов в C++

Под ***массивом*** понимают **именованную последовательность однотипных величин**. При обработке данных на компьютере массив, по сути, это некоторое множество мест в памяти, называемых *элементами массива*, каждый из которых может хранить единицу данных определенного типа и к которым можно обращаться по одному имени переменной. Каждый элемент массива имеет свой номер (**индекс**), который представляет собой целое число (порядковый номер элемента в массиве). **Нумерация элементов массива начинается с нуля**.

Объявление массива имеет формат:

тип имя_массива [количество_элементов];.

В приведенной записи параметр в квадратных скобках является обязательным. Вместе с типом элементов он задает требуемый объем памяти, выделяемый для массива. **Многомерные массивы** задают указанием каждого измерения в квадратных

скобках. Например, двумерный массив **a** вещественных элементов размерностью 4×6 (четыре строки, шесть столбцов) в программе имеет описание:

float a [4][6];.

Тип элементов массивов может быть любым, как и обычных переменных.

Инициализация массива может быть выполнена как непосредственно в программе (например, при его объявлении), так и посредством ввода значений элементов извне (например, с помощью клавиатуры). При инициализации массива во время объявления значения его элементов записывают в фигурных скобках. Если элементов в массиве больше, чем значений, записанных в фигурных скобках, оставшимся элементам присваиваются нулевые значения. Например:

float a[3] = {1.25, 4.0, 2.43}; // a[0]=1,25; a[1]=4; a[2]=2,43
int b[5] = {7, 1, 3}; // b[0]=7; b[l]=1; b[2]=3; b[3]=0; b[4]=0.

Инициализацию многомерного массива можно выполнить двумя способами:

– представить его как массив из одномерных массивов (в этом случае каждый одномерный массив заключается в свои фигурные скобки);

– задать в одних фигурных скобках список элементов в том порядке, в котором они располагаются в памяти (элементы первой строки, элементы второй строки и т. д.).

Пример –

int mass [3][2] = { {21, 8}, {3, 12}, {11, 2} };

или

int mass [][2] = { {21, 8}, {3, 12}, {11, 2} };

или

int mass [3][2] = {21, 8, 3, 12, 11, 2};.

Если при объявлении массив инициализируется, то его размерность (количество элементов) можно не указывать. Но наличие квадратных скобок обязательно. Например:

int a[] = {3, 7, 2, 2, 4, 1};.

Для доступа к элементу массива после его имени записывают номер элемента (его индекс) в квадратных скобках, например:

```
int y, a[] = {2, 5, 3, 1, 6};
y = a[1] + a[0] * a[4];.
```

Идентификатор массива является константным указателем на его нулевой элемент. Например, для массива из приведенного выше примера имя **a** – это то же самое, что и **&a[0]**. При этом к *i*-му элементу массива можно обратиться, используя выражение ***(a+i)**.

Можно описать указатель, присвоить ему адрес начала массива и работать с массивом через указатель. Ниже приведен пример копирования элементов массива **a** в массив **b** [4]:

```
int a[10], b[10];
int *pa = &a[0];
int *pb = b;
for(int i = 0; i < 10; i++)
*pb++ = *pa++;            // или pb[i] = pa[i];.
```

Массив, размер которого может изменяться во время исполнения программы, называется **динамическим**. Динамическое выделение памяти необходимо для эффективного использования памяти компьютера. В C++ для динамического распределения памяти предназначены операции **new** и **delete**. Операция **new** создает объект заданного типа, выделяет ему память **из области свободной памяти** и возвращает указатель правильного типа на данный участок памяти, а операция **delete** высвобождает выделенную память после завершения обработки находящейся в ней информации. Выделение памяти возможно под любой тип данных: **int**, **float**, **double**, **char** и т. д.

Если память выделить невозможно (например, в случае отсутствия свободных участков), то указатель вернет значение нуль.

При создании динамического массива необходимо указать его тип и размерность, например:

int n = 100;
float *a = new float [n];.

В приведенном примере для массива в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель **a**.

Динамические массивы **нельзя при создании инициализировать**. Объем памяти, выделяемой под динамический массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива осуществляется так же, как к элементам статического массива. Например, к элементу массива с номером 5 в приведенном выше примере можно обратиться как **a[5]** или ***(a+5)**.

Память, зарезервированная под динамический массив с помощью **new []**, должна освобождаться оператором **delete []**, например

delete [] a;.

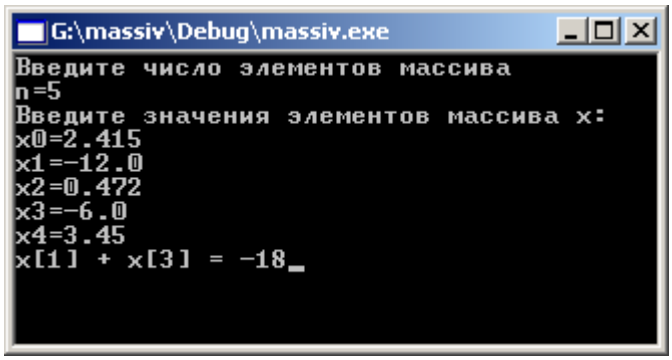
Размерность массива в операции **delete** не указывается, но наличие квадратных скобок обязательно.

На рисунке 3.9 приведен пример создания динамического массива, организации ввода значений элементов массива с клавиатуры с помощью оператора цикла **for**, а также использования значений элементов в арифметической операции сложения. Показан вид консоли при выполнении программы.

3.8 Использование функций при программировании на C++

При программировании сложных вычислительных процессов часто возникает необходимость использовать одну и ту же группу операторов в нескольких местах программного кода. С целью упрощения программного кода, избавления от необходимости многократно переписывать одинаковые строки программы (и, что немало-

важно, уменьшения объема занимаемой программой памяти компьютера) была предложена концепция **подпрограммы**.




```
// massiv.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <clocale>

    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
    int n;   // описание размерности динамического массива
    float s;
        cout << "Введите число элементов массива\n" << "n="; cin >> n;
    float *x = new float [n];  // выделение динамической памяти для массива x
        cout << "Введите значения элементов массива x: \n";
    for (int i = 0; i < n; i++)
        {cout << "x" << i << "="; cin >> x[i]; }
            cout << "x[1] + x[3] = " << x[1] + x[3];
    delete [] x;
    _getch();
        return 0;
}
```

Рисунок 3.9 – Создание динамического массива

Подпрограмма – это именованная, логически законченная группа операторов, которую можно вызвать для выполнения любое количество раз из различных мест программы. В языке С++ подпрограммы реализованы в виде **функций**. Под ***функцией*** понимают именованную последовательность описаний и операторов, выполняющую какое-либо законченное действие [4]. Информация, передаваемая в функцию для обработки, называется ***параметром***, а результат, полученный при вычислении функции, ее ***значением***. Обращение к функции называют ***вызовом***.

Любая программа на С++ состоит из функций, одна из которых (главная функция) должна иметь имя **main**. С этой функции начинается выполнение программы. Если среди операторов функции **main** встречается **вызов** какой-нибудь функции, то управление передается операторам этой функции. Когда все операторы вызванной функции будут выполнены, **управление возвращается оператору, следующему за вызовом функции**.

В **стандартной библиотеке** C++ находятся определения типов, констант, макросов, **функций** и классов. Функции стандартной библиотеки можно разбить на группы по их назначению: ввод/вывод, обработка строк, математические функции (см. таблицу 3.1), работа с динамической памятью, поиск и сортировка и т. д. Чтобы использовать их в программе, требуется с помощью директивы **#inc1ude** включить в исходный текст программы заголовочные файлы, в которых находятся объявления соответствующих функций.

Кроме функций стандартной библиотеки в программе можно использовать функции, созданные самостоятельно (иногда их называют функциями пользователя). Для этого функцию нужно **объявить** и **определить**. Объявление функции должно находиться в тексте программы раньше ее **вызова**, с которого начинается выполнение функции.

Определение функции включает **объявление** и **тело** функции. ***Объявление функции*** задает ее **имя**, **тип** возвращаемого значения и **список** передаваемых **параметров**. ***Тело функции*** представляет собой **последовательность операторов** и **описаний**, заключенных в **фигурные скобки**.

Формат определения функции:

[класс] тип имя_функции ([список_параметров])[throw (исключения)]
{ тело функции }.

Назначение составных частей определения:

– необязательный модификатор **класс** позволяет явным образом задать область видимости функции с помощью ключевых слов **extern** (глобальная видимость во всех модулях программы (используется по умолчанию)) или **static** (видимость только в пределах модуля, в котором определена функция);

– **тип** возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение, то указывается тип **void**;

– **список параметров** определяет перечень величин (**аргументов**), которые требуется передать в функцию при ее вызове. Элементы списка параметров отделя-

ются друг от друга запятыми. Для каждого параметра указывается его тип и имя. Если функция не имеет аргументов, то в скобках либо указывают тип **void**, либо скобки оставляют пустыми.

Об использовании необязательного параметра в формате определения функции **throw (исключения)** можно получить информацию в [4].

Примечание – В **определении**, в **объявлении** и **при вызове** одной и той же функции ***типы*** и ***порядок следования*** параметров должны совпадать.

Для **вызова** функции в простейшем случае нужно указать ее имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов. Вызов функции может находиться в любом месте программы. Если тип возвращаемого функцией значения не **void**, она может входить в состав выражений или располагаться в правой части оператора присваивания. Для того чтобы функция вернула какое-либо **значение**, в ней должен быть оператор:

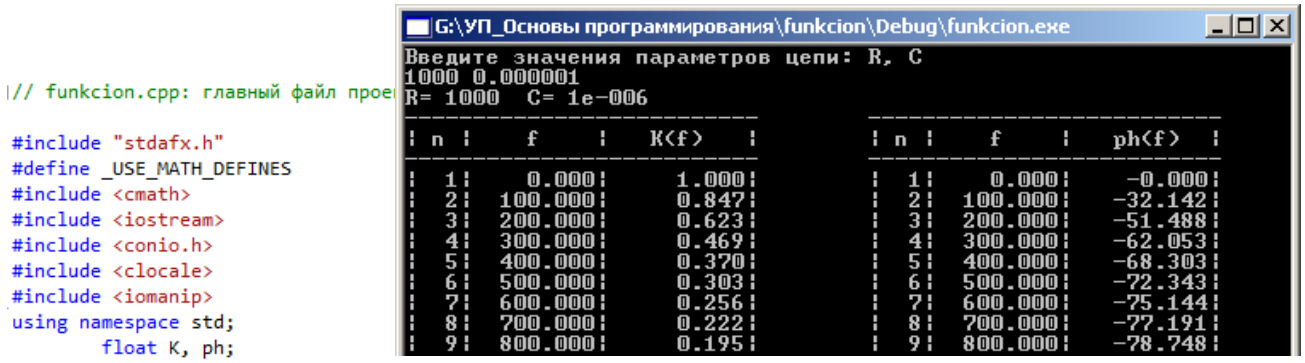
return (выражение);.

Если в программе используются функции, разработанные самостоятельно, то ее структура в общем случае может иметь вид:

```
директивы препроцессора
using namespace (название пространства имен)
тип имя_1(список_переменных)
{ тело_функции_1; }
тип имя_2(список_переменных)
{ тело_функции_2; }
...
тип имя_n(список_переменных)
{ тело_функции_n; }
        int main(аргументы функции)
{                                  // может содержать операторы вызова
        тело программы      // функций: имя_1, имя_2, …, имя_n
}.
```

Допускается объявлять функции пользователя до функции **main**, а определять – в теле этой функции (в теле программы).

На рисунке 3.10 приведен пример определения и вызова функций пользователя для вычисления значений модуля (функция **module**) и аргумента (функция **argum**) комплексного коэффициента передачи интегрирующей цепи. Параметрами функций являются частота ***f*** и постоянная времени цепи ***tau***.




```
// funkcion.cpp: главный файл прое

#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <conio.h>
#include <clocale>
#include <iomanip>
using namespace std;
        float K, ph;
        float module(float f, float tau)
            {K = 1 / (sqrt(1+pow(2*M_PI*f*tau, 2)));
            return K;}
        float argum(float f, float tau)
            {ph = 180 * (-atan(2*M_PI*f*tau)) / M_PI;
            return ph;}
int main(){
    setlocale(LC_ALL, "rus");
    float R, C, f;
      cout << "Введите значения параметров цепи: R, C \n";
        cin >> R >> C;
      cout << "R= " << R << "  " << "C= " << C << endl;
      cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
      cout << "| n |    f    |   K(f)   |"; cout << "        "; cout << "| n |    f    |   ph(f)  |\n";
      cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
      float tau = R * C;
      int n = 0;
            for (f = 0; f <= 5000; f += 100) {n++;
        cout << "|"; cout.width(3); cout << n;
        cout << "|"; cout.width(9); cout << fixed << setprecision(3) << f;
        cout << "|"; cout.width(10); cout << fixed << setprecision(3) << module(f, tau); cout << "|"; cout.width(9);
        cout << "|"; cout.width(3); cout << n;
        cout << "|"; cout.width(9); cout << fixed << setprecision(3) << f;
        cout << "|"; cout.width(10); cout << fixed << setprecision(3) << argum(f, tau); cout << "|\n";}
        cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
            _getch();
    return 0;
}
```

Рисунок 3.10 – Пример использования функции пользователя

В теле функции **module** используются стандартные математические функции извлечения квадратного корня и возведения в степень (конкретно – возведения в квадрат), в теле функции **argum** – стандартная математическая функция арктангенс. Описание названных функций находится в заголовочном файле <**cmath**>. Кроме

этого используется константа π (в программе – **M_PI**), описанная в файле **_USE_MATH_DEFINES**.

Вызов функций для вычисления выполняется внутри цикла **for**. Обратите внимание на то, что все переменные должны быть объявлены и инициализированы до их использования в выражениях.

На рисунке 3.10 приведен также фрагмент консоли с результатами вычисления значений модуля ($K(f)$) и аргумента ($ph(f)$) комплексного коэффициента передачи электрической цепи.

3.9 Работа с файлами

При решении прикладных задач может возникать необходимость обмена массивами данных разработанной программы с внешними устройствами. Например, результаты вычислений требуется передать в MS Excel для их представления в виде диаграммы (графика) или использовать в программе данные, полученные при проведении физического эксперимента. Как правило, информация, о которой идет речь, хранится на внешних носителях (жестком магнитном диске, FLASH-накопителе и т. п.) в виде файлов. Кроме этого файлы удобно использовать для длительного хранения данных и их многократного использования.

Под *файлом* в информатике понимают *именованную область на носителе информации,* содержащую данные одного типа произвольной длины, доступ к которой обеспечивается средствами операционной системы. В общем случае в файле может храниться как программа для решения каких-либо задач, так и данные, сгенерированные программой (приложением) или предназначенные для ввода в программу.

Как показано в подразделе 3.1, для ввода-вывода значений переменных в C++ используют стандартные средства библиотеки ввода-вывода, построенной на основе механизма классов. Эти средства описаны в заголовочном файле **iostream** и могут быть использованы в программе для извлечения данных из потоков и для включения данных в потоки. При этом под ***потоком*** понимают *последовательность байтов* (символов), передаваемых между двумя устройствами в процессе обмена данными

(например, оперативной памятью компьютера и внешним устройством – накопителем на жестком диске, принтером, дисплеем, клавиатурой, FLASH-накопителем и т. п.).

Используемые в программах потоки логически делятся на три типа [6]:

– входные, из которых читается информация;

– выходные, в которые вводятся данные;

– двунаправленные, допускающие как чтение, так и запись.

Все потоки библиотеки ввода-вывода **последовательные**, т. е. в каждый момент для потока определены позиции записи и (или) чтения, и эти позиции после обмена перемещаются по потоку на длину переданной порции данных.

В зависимости от того, какие устройства участвуют в обмене данными, потоки принято делить на *стандартные, консольные, строковые* и *файловые.* **Стандартные** и **консольные** потоки соответствуют передаче данных от клавиатуры к экрану дисплея. Если символы потока образуют символьный массив (строку) в основной памяти компьютера, то такой поток называется **строковым**. Если при использовании потока его символы размещаются на внешнем носителе данных, то имеет место **файловый** поток (файл).

Библиотека ввода-вывода Си++ включает средства для работы с **последовательными** файлами. Логически последовательный файл можно представить как именованную последовательность байтов, имеющую начало и конец. Чтение из последовательного файла или запись в такой файл ведутся байт за байтом от начала к концу. В каждый момент позиции в файле, откуда выполняется чтение (или куда производится запись), определяются значениями указателей позиций записи (чтения) файла. Позиционирование указателей записи и чтения (т. е. установка на нужные байты) выполняется либо автоматически, либо за счет явного управления их положением [6].

Потоки для работы с последовательными файлами создаются как объекты следующих классов:

– **ofstream** – для вывода данных в файл;

– **ifstream** – для ввода данных из файла;

– **fstream** – для двунаправленного обмена данными (чтения и записи данных).

Чтобы можно было использовать названные классы, необходимо директивой препроцессора в текст программы включить заголовочный файл **fstream**.

Для вывода данных (например, вычисленных значений функции) в файл, в простейшем случае, необходимо создать выходной файловый поток, относящийся к типу **ofstream**, и присоединить его к файлу с явно заданным именем. Например:

ofstream o_file(”FileName.txt”);.

В приведенном примере **o_file** – имя выходного файлового потока, **FileName.txt** – имя текстового файла, в который будут выведены данные. Если файл с именем **FileName.txt** не существует, то он будет создан, открыт и соединен с потоком **o_file**. Если же файл уже существует, то он будет удален и с указанным именем будет создан пустой файл. Имя файла должно содержать, в том числе, путь к его размещению (имя логического диска, имя папки и т. п.). Если же путь к файлу не указан, то по умолчанию файл сохраняется в папке, в которой хранятся файлы проекта.

Как отмечено ранее, данные при выводе в файл записываются последовательно по одному символу (байту). При этом с помощью оператора вывода можно управлять способом отображения данных, записанных в файл, в окне текстового редактора (в виде строки или в виде столбца). На рисунках 3.11 и 3.12 приведены примеры вывода в текстовый файл (с именами **expon.txt** и **expon_1.txt** соответственно) значений функции $f(x) = exp(x)$ на заданном интервале значений аргумента, а также вид содержимого файлов в текстовом редакторе **Блокнот**. Как видно из рисунков, вычисление значений функции и формирование выходного файлового потока **o_file** реализовано в цикле **for**. После записи в файл всех данных, файл нужно закрыть (разорвать связь выходного файлового потока с файлом). Для этого служит функция **close()**. В программах, текст которых приведен на рисунках 3.11 и 3.12, для закрытия файла записана строка:

o_file.close();.

```
// file.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <conio.h>
#include <clocale>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
        float x0, xk, x, Dx, f; int N;
        ofstream o_file("G:\expon.txt");
cout << "Введите диапазон изменения аргумента и число точек: x0, xk, N \n";
      cin >> x0 >> xk >> N;
cout << "x0=" << x0 << "  " << "xk=" << xk << "  " << "N=" << N;
       Dx = xk / (N - 1);
            for (x = x0; x <= xk; x += Dx)
                {f=exp(x);
                o_file << f << "  ";}
                o_file.close();
         getch();
```

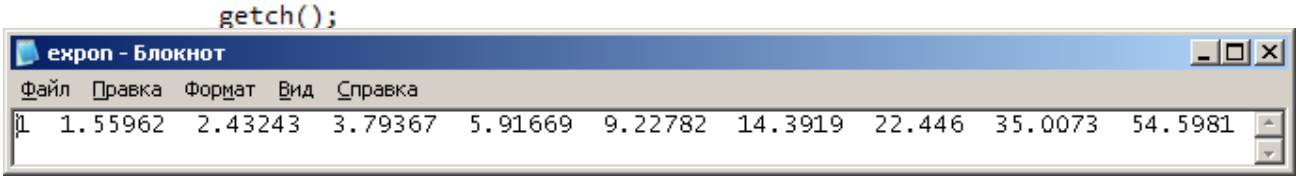


Рисунок 3.11 – Вывод данных в файл

```
// file.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <conio.h>
#include <clocale>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
        float x0, xk, x, Dx, f; int N;
        ofstream o_file("G:\expon_1.txt");
cout << "Введите диапазон изменения аргумента и число точек: x0, xk, N \n";
      cin >> x0 >> xk >> N;
cout << "x0=" << x0 << "  " << "xk=" << xk << "  " << "N=" << N;
       Dx = xk / (N - 1);
            for (x = x0; x <= xk; x += Dx)
                {f=exp(x);
                o_file << f << endl;}
                o_file.close();
        _getch();
    return 0;
}
```

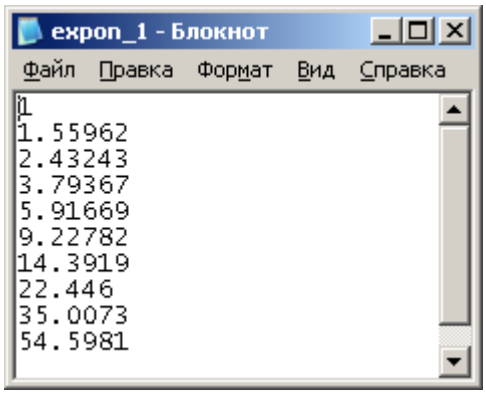


Рисунок 3.12 – Вывод данных в файл

В уже существующий файл (ранее созданный) можно добавлять данные. Чтобы открыть файл в режиме добавления данных, нужно использовать оператор

ofstream имя_выходного_потока(”FileName.txt”, ios::app);.

Параметр выходного файлового потока **ios::app** обеспечивает расположение файлового указателя в конце открытого файла. После этого в файл можно записать новые данные, не удаляя уже существующие в нем. Например, в файл *expon_1.txt*, созданный при выполнении программы, приведенной на рисунке 3.12, добавим строку «Вектор значений экспоненциальной функции». Программа, обеспечивающая добавление данных в файл, и результат ее работы приведены на рисунке 3.13.

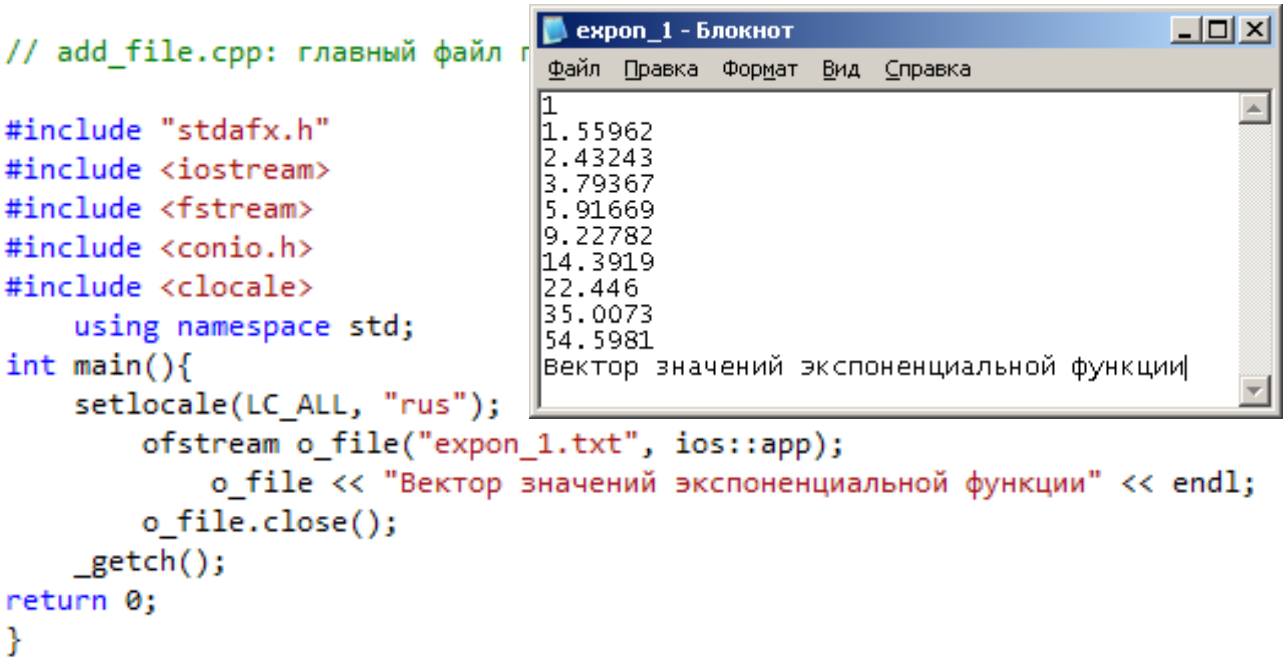


Рисунок 3.13 – Добавление данных в файл

Для ввода данных из файла в программный код используется оператор

ifstream имя_входного_потока(”FileName.txt”);,

где **FileName.txt** – имя существующего файла, из которого считываются данные во входной поток.

Прежде чем считать данные из файла, нужно убедиться в его существовании. Для проверки результата выполнения операции с потоком можно использовать,

например, функцию-член **fail**. Эта функция возвращает значение 1 (true), если файл не удалось открыть, или 0 (false), если операция выполнена успешно. Функция **fail** может быть использована как с входным, так и с выходным файловыми потоками. Вызов функции производится следующим образом:

имя_входного_потока.fail()
имя_выходного_потока.fail()

Приведенные выше записи представляют собой логические выражения и могут быть использованы в управляющих структурах (например, в цикле **while** или в операторе **if … else**).

Проверить открытие файла для чтения (записи) можно также с помощью перегруженной унарной операции **!**. Если применить операцию **!** к потоку (например, **!имя_потока**), то при ошибке открывания файла результат операции будет ненулевым. Если операция завершена успешно, то выражение **!имя_потока** имеет нулевое значение. Например, если имя входного файлового потока **i_file** и имя выходного файлового потока **o_file**, то проверку открывания файла в тексте программы можно выполнить следующим образом:

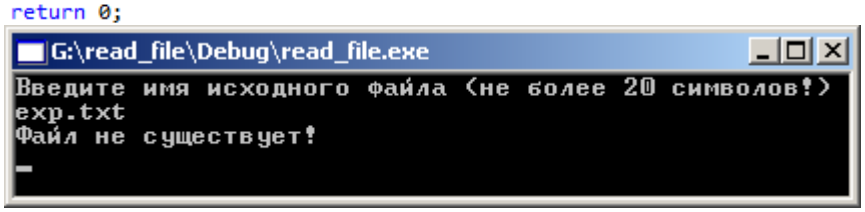
```
. . .
if(!o_file)
        {cout << "Ошибка при создании файла! \n";
        exit(1);} // Завершение работы программы
. . .
if(!i_file)
        {cout << "Файл не существует! \n";
        exit(1);} // Завершение работы программы
. . .
```

На рисунке 3.14 приведен пример программы, обеспечивающей считывание числовых данных из текстового файла и вывод считанных данных на экран консоли в табличной форме. На рисунке представлен вид окна консоли для двух случаев: при

отсутствии файла с введенным именем и при успешном открывании файла (файл с введенным именем существует и открыт для обмена данными).

```
// read_file.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <conio.h>
#include <clocale>
#include <iomanip>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
     char mas_name[20]; // Создание массива для имени файла
    cout << "Введите имя исходного файла (не более 20 символов!) \n";
        cin >> mas_name; // Ввод имени файла
    ifstream i_file(mas_name);
        if (i_file.fail()) // Проверка существования файла
            {cout << "Файл не существует! \n";
                _getch();
            exit(1);}  // Завершение работы программы
        cout << "Файл открыт успешно \n";
            cout << "--------------\n";
            cout << "| n | exp(x) |\n";
            cout << "--------------\n";
        float y; int n = 1;
        while(!i_file.eof()) // Считывание данных из файла и вывод на экран
        {i_file >> y;
            cout << "|"; cout.width(3); cout << n;
cout << "|"; cout.width(8); cout << fixed << setprecision(3) << y; cout << "|" << endl;
                n++;}
        i_file.close();
    _getch();
return 0;
```



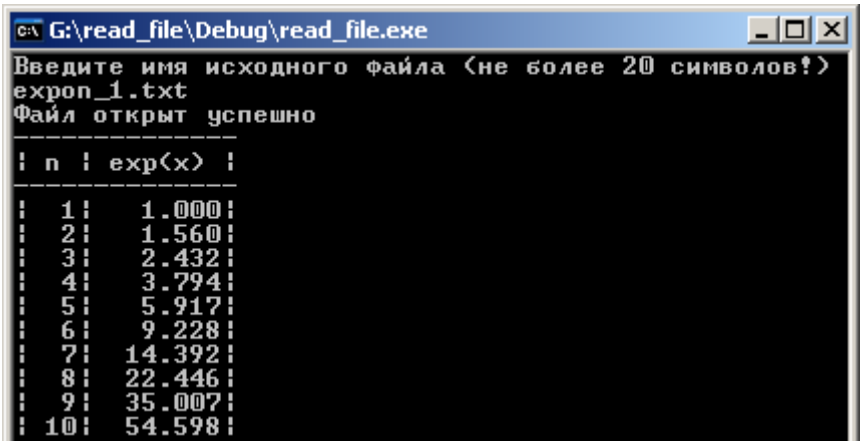


Рисунок 3.14 – Считывание данных из файла

Считывание данных из файла в представленной программе организовано с помощью оператора цикла **while**. Условием завершения цикла является равенство единице функции в скобках (**!i_file.eof()**). Функция **eof** возвращает значение 1 (true), если файл пустой (в приведенной программе позволяет обнаружить конец файла).

Данные, считанные из файла, могут быть использованы в выражениях. Для этого при считывании данных из файла из них нужно сформировать массив, а затем использовать в выражениях элементы массива. На рисунке 3.15 приведен пример считывания данных из файла в массив и выполнения арифметических операций над элементами массива.

```
// read_file_1.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <conio.h>
#include <clocale>
#include <iomanip>
    using namespace std;
int main(){
    setlocale(LC_ALL, "rus");
     char mas_name[20]; // Создание массива для имени файла
    cout << "Введите имя исходного файла (не более 20 символов!) \n";
        cin >> mas_name; // Ввод имени файла
    ifstream i_file(mas_name);
        if (i_file.fail()) // Проверка существования файла
            {cout << "Файл не существует! \n";
                _getch();
            exit(1);}  // Завершение работы программы
        cout << "Файл открыт успешно \n";
        float y[20]; // Объявление массива
        int i = 0;
        while(!i_file.eof()) // Считывание данных из файла в массив
        {i_file >> y[i];
            i++;}
        i_file.close();
        cout << "В массиве  " << i << "  элементов \n";
    cout << "y1 + y3 = " << y[1] + y[3] << ",  y4 - y2 = " << y[4] - y[2] << endl;
    _getch();
return 0;
}
```

```
G:\read_file_1\Debug\read_file_1.exe
Введите имя исходного файла (не более 20 символов!)
expon_1.txt
Файл открыт успешно
В массиве  11  элементов
y1 + y3 = 5.35329,  y4 - y2 = 3.48426
```

Рисунок 3.15 – Использование данных из файла в вычислениях

3.10 Вопросы для самоконтроля

3.10.1 Что понимают под оператором в C++?

3.10.2 Как правильно записать оператор в тексте программы?

3.10.3 Назовите операторы управления ходом выполнения программы.

3.10.4 Приведите формат и примеры оператора присваивания.

3.10.5 Как выполнить ввод исходных данных с клавиатуры при выполнении программы?

3.10.6 Как вывести информацию на экран монитора (на консоль)?

3.10.7 Какие вы знаете способы управления выводом на консоль?

3.10.8 Назовите основные операторы, используемые при программировании линейного алгоритма.

3.10.9 Поясните сущность объявления и инициализации переменной в программе.

3.10.10 Приведите формат инструкции объявления переменной.

3.10.11 Можно ли в одной инструкции объявить сразу несколько переменных?

3.10.12 Поясните понятия: время жизни, область видимости и область действия переменной.

3.10.13 Что такое модификатор типа? Приведите примеры комбинаций базовых типов данных и модификаторов.

3.10.14 В чем сущность операции приведения типов?

3.10.15 Опишите структуру программы на C++.

3.10.16 Что такое директива препроцессора?

3.10.17 Какой вычислительный процесс называется разветвляющимся?

3.10.18 Приведите схему разветвляющегося алгоритма.

3.10.19 Приведите схему алгоритма ВЫБОР.

3.10.20 Приведите формат условного оператора **if**.

3.10.21 Приведите список операций отношения, используемых в условном операторе **if**.

3.10.22 Приведите список логических операций, используемых в условном операторе **if**.

3.10.23 В каком случае в разветвляющемся вычислительном процессе используют оператор **switch**.

3.10.24 Какой вычислительный процесс называется циклическим?

3.10.25 Приведите схему алгоритма цикла с параметром.

3.10.26 Приведите схему алгоритма цикла с предусловием.

3.10.27 Приведите схему алгоритма цикла с постусловием.

3.10.28 Приведите и поясните формат оператора цикла **for**.

3.10.29 Приведите и поясните формат оператора цикла **while**.

3.10.30 Приведите и поясните формат оператора цикла **do while**.

3.10.31 Приведите условия предпочтительного использования того или иного оператора цикла.

3.10.32 Какие циклы называют вложенными? Поясните правила программирования вложенных циклов.

3.10.33 Назовите операторы передачи управления и поясните, в каких случаях их используют.

3.10.34 Что такое указатель? В чем разница между указателем и переменной?

3.10.35 В каких случаях целесообразно использовать указатели?

3.10.36 Поясните способы инициализации указателей.

3.10.37 Что называется ссылкой? Приведите формат объявления ссылки.

3.10.38 Приведите формат объявления одномерного и многомерного массивов. Как выполнить инициализацию массива?

3.10.39 Какой массив называют динамическим?

3.10.40 Что называется функцией в C++?

3.10.41 Приведите формат определения функции. Как использовать функцию пользователя в программе?

3.10.42 Как обеспечить возвращение значения, вычисленного функцией?

3.10.43 Как использовать в программе функции из стандартной библиотеки C++?

3.10.44 Как сохранить результаты вычислений в файл?

3.10.45 Как открыть файл для записи в него данных?

3.10.46 Как добавить данные в существующий файл?

3.10.47 Как открыть файл для ввода из него данных в программу?

4 Основы объектно-ориентированного программирования

4.1 Основные сведения об объектно-ориентированном программировании

Дальнейшим развитием модульного программирования, основанного на модульном построении программы (модуль – множество взаимосвязанных процедур (подпрограмм) вместе с данными, которые эти процедуры обрабатывают), явилось объектно-ориентированное программирование.

Базовыми понятиями объектно-ориентированного программирования (ООП) являются **объекты** и **классы**. ***Объект*** – это часть окружающей нас реальности. Содержательно объект можно представить как что-то ощущаемое или воображаемое и имеющее хорошо определенное поведение. Любой объект можно характеризовать совокупностью присущих ему свойств.

Всегда можно выделить некоторую группу объектов, имеющих часть общих для всех их свойств. Например, объекты автомобили могут отличаться цветом и типом кузова, размером колес, типом и мощностью двигателя и т. п. В то же время такие свойства как кузов, двигатель, колесо, фара и некоторые другие являются общими для всех объектов, являющихся автомобилями.

Множество объектов, имеющих общую структуру и общее поведение, образуют **класс**. ***Класс*** является **типом данных**, определяемым пользователем. В классе структуры данных и функции их обработки объединяются. С помощью классов можно задавать свойства и поведение различных объектов, а также функции для работы с ними.

Объекты взаимодействуют между собой, посылая и получая **сообщения**. ***Сообщение*** – это запрос на выполнение действия, содержащий набор необходимых параметров. Механизм сообщений реализуется с помощью **вызова** соответствующих функций.

К базовым принципам объектно-ориентированного стиля программирования относятся: **инкапсуляция**, **наследование, полиморфизм** и **передача сообщений**.

Инкапсуляция (пакетирование) предполагает соединение в одном объекте **данных** и **функций**, которые манипулируют этими данными. Инкапсуляция позволяет создавать библиотеки классов для применения во многих программах.

Наследование позволяет создавать иерархии классов, в которых потомки наследуют все свойства своих предков, могут их изменять и добавлять новые. Свойства при наследовании повторно не описываются, что сокращает объем программы. Иерархия классов представляется в виде древовидной структуры, в которой более общие классы располагаются ближе к корню, а более специализированные – на ветвях и листьях. В C++ каждый класс может иметь сколько угодно потомков и предков.

Полиморфизм – это возможность использовать в различных классах иерархии одно имя для обозначения сходных по смыслу действий и гибко выбирать требуемое действие во время выполнения программы. Полиморфизм позволяет использовать одни и те же функции для решения разных задач.

Передача сообщений выражает основную методологию построения объектно-ориентированных программ. Такие программы представляются в виде набора объектов и передачи сообщений между этими объектами.

С учетом выше изложенного, ***объектно-ориентированное программирование*** можно определить как *метод построения программ в виде множества взаимодействующих объектов, структура и поведение которых описаны соответствующими классами, и все эти классы являются компонентами иерархии классов, выражающей отношения наследования*.

4.2 Классы

Как определено выше, **класс** – это **тип данных** (аналогично определенным в C++ типам int, char, float и др.), определяемый пользователем. В нем объединяются данные и функции для работы с ними. **Данные** класса называют **полями**, а **функции** класса – **методами**. Поля и методы называются **элементами класса**.

Описание класса имеет вид [4]:

class имя {

[private:]

описание скрытых элементов

public:

описание доступных элементов

};.

Примечание – Описание класса заканчивается точкой с запятой после закрывающей фигурной скобки.

Описание скрытых (доступных) элементов в описании класса представляет собой совокупность выражений, каждое из которых заканчивается **точкой с запятой**. Скрытые элементы, описанные после служебного слова **private**, видимы только внутри класса. Спецификатор **private** в описании класса можно не записывать, поскольку он действует по умолчанию. Все доступные извне элементы класса записывают после ключевого слова **public**.

Можно задавать несколько секций **private** и **public**, порядок их следования значения не имеет.

Поля класса [4]:

– могут иметь любой тип, **кроме типа этого же класса** (например, элемент класса Cube может иметь тип int, char, float и др., но не может быть элементом типа Cube). Допускается включать в качестве элемента класса **указатель** или **ссылку** на переменную типа Cube;

– могут быть описаны с модификатором **const**;

– могут быть описаны с модификатором **static**, но не как **auto**, **extern** и **register**.

Инициализация (присваивание значений) **полей при описании не допускается**.

Различают классы **глобальные** и **локальные**. ***Глобальными*** являются классы, объявленные вне любого блока, а ***локальными*** – объявленные внутри блока (например, внутри функции или другого класса).

Ниже приведены некоторые особенности, характеризующие локальный класс:

– внутри локального класса можно использовать типы, статические (**static**) и внешние (**extern**) переменные, внешние функции и элементы перечислений из области, в которой он описан. Запрещается использовать автоматические переменные из этой области;

– локальный класс не может иметь статических элементов;

– методы класса могут быть описаны только внутри этого же класса;

– если один класс вложен в другой класс, они не имеют каких-либо особых прав доступа к элементам друг друга и могут обращаться к ним только по общим правилам.

После того, как объявлен класс, можно объявить переменную, представляющую собой **экземпляр** этого **класса**. Например, по аналогии с объявлением переменной типа **int** можно объявить переменную типа **Cube**, являющуюся экземпляром класса **Cube**:

Cube bigCube;.

На рисунке 4.1 приведен пример создания класса **Cube**, методами которого являются вычисление площади полной поверхности и объема куба.

Функции (методы класса), заявленные в классе (в примере – s и v) должны быть описаны. Описание функций производится после объявления класса. Принадлежность функции к конкретному классу указывают с помощью оператора разрешения области видимости **::** (**Cube::s**, **Cube:v**). Благодаря этому оператору различные классы могут использовать одинаковые имена функций.

Доступ из основной программы к методам класса обеспечивается с помощью оператора «**точка**». Чтобы вызвать функцию, принадлежащую классу, нужно к имени объекта, являющегося экземпляром класса, через точку приписать имя функции, возвращающей значение этого объекта, с фактическим значением ее параметра. Например, **S.s(L)** (рисунок 4.1) возвращает площадь полной поверхности, а **V.v(L)** – объем куба с длиной ребра, равной L.

```
// class.cpp: главный файл проекта.

#include "stdafx.h"
#include <cmath>
#include <iostream>
#include <conio.h>
#include <clocale>
#include <iomanip>

using namespace std;
    class Cube
    {public:
        float s(float); // Объявление функций – членов класса Cube
        float v(float);
    };
    float Cube::s(float l) // Описание функций – членов класса Cube
        {return 6*pow(l, 2);}    // Возвращает площадь поверхности куба
    float Cube::v(float l)
        {return pow(l, 3);}  // Возвращает объем куба
int main(){
    setlocale(LC_ALL, "rus");
    Cube S;      // Объявление объекта S – экземпляра класса Cube
    Cube V;      // Объявление объекта V – экземпляра класса Cube
    float L;
    cout << "Программа для вычисления площади поверхности" << endl;
    cout << "                и объема куба              \n";
        cout << "Введите значение длины ребра куба в см, L="; cin >> L;
        cout << "Площадь поверхности куба: S=" << S.s(L) << " кв. см;" << endl;
        cout << "Объем куба: V=" << V.v(L) << " куб. см.";
    getch();
        return 0;
}
```

```
G:\class\Debug\class.exe
Программа для вычисления площади поверхности
                и объема куба
Введите значение длины ребра куба в см, L=10
Площадь поверхности куба: S=600 кв. см;
Объем куба: V=1000 куб. см.
```

Рисунок 4.1 – Пример описания класса **Cube** и доступа к его элементам

Значение полям экземпляра класса можно присвоить обычным способом с помощью оператора присваивания. Например:

```
class Box
        { public:
        float Length;  // Длина ящика
        float Width;   // Ширина ящика
        float Height;  // Высота ящика
        };
Box box1;   // Объявление экземпляра box1 класса Box
Box box2;   // Объявление экземпляра box2 класса Box
        box1.Length = 24.0; // Присваивание значений полям
        box1.Width = 10.0;  // экземпляра класса
        box1.Height = 6.0;.
```

Как отмечено в п. р. 4.1, полиморфизм можно определить как свойство, позволяющее использовать одно имя для обозначения сходных по смыслу действий, общих для родственных классов. В C++ полиморфизм реализован через **механизм перегрузки** (функций и операций), **виртуальные функции** и **шаблоны**. На рисунке 4.2 приведен пример использования механизма перегрузки функций.

```
// klass_peregruzka.cpp: главный файл проекта.

#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <conio.h>
#include <clocale>
#include <iomanip>

using namespace std;
    class circle
{public:
    float l(float);        // Объявление функции l от одной переменной
    float l(float, float); // Объявление функции l от двух переменных
    float s(float);
};
// Описание функций - членов класса
    float circle::l(float r) // Описание функции l от одной переменной (r - радиус окружности)
        { return 2*M_PI*r; }   // Вычисление длины окружности
    float circle::l(float r, float a) // Описание второй функции l от двух переменных
        { return M_PI*r*a/180; }      // Вычисление длины дуги (a - угол сектора)
    float circle::s(float r) // Описание функции s от одной переменной
        { return M_PI*r*r; }   // Вычисление площади круга
    int main(){
        setlocale(LC_ALL, "rus");
    circle L;      // Объявление объекта L - экземпляра класса circle
    circle S;      // Объявление объекта S - экземпляра класса circle
        float R, A;
            cout << "Программа для вычисления параметров окружности" << endl;
                cout << "Введите значение радиуса окружности в см, R="; cin >> R;
                cout << "Введите значение угла сектора в градусах, A="; cin >> A;
            cout << "Длина окружности равна: L=" << L.l(R) << " см;" << endl;
            cout << "Длина дуги равна: L1=" << L.l(R, A) << " см;" << endl;
            cout << "Площадь круга равна: S=" << S.s(R) << " кв. см.";
        getch();
    return 0;
}
```

```
G:\klass_peregruzka\Debug\klass_peregruzka.exe
Программа для вычисления параметров окружности
Введите значение радиуса окружности в см, R=10
Введите значение угла сектора в градусах, A=45
Длина окружности равна: L=62.8319 см;
Длина дуги равна: L1=7.85398 см;
Площадь круга равна: S=314.159 кв. см.
```

Рисунок 4.2 – Пример перегрузки функции

4.3 Класс комплексных чисел в C++

Класс комплексных чисел стандартной библиотеки C++ (шаблонный класс std::complex) представляет собой хороший пример использования объектной модели. Благодаря перегруженным арифметическим операциям объекты этого класса

можно использовать так же, как и любые объекты, принадлежащие одному из встроенных типов данных. В арифметических операциях могут одновременно принимать участие и переменные встроенного арифметического типа, и комплексные числа.

При использовании в выражениях комплексных чисел нужно подключить заголовочный файл **#include <complex>**. В большинстве случаев в качестве параметра можно использовать тип **double**, хотя, если это обеспечивает требуемую точность получаемого результата, можно использовать и тип **float**.

Объявление комплексной переменной имеет формат:

complex < тип > имя_комплексной переменной[(re, im)];,

где **re** – действительная часть комплексного числа;

im – мнимая часть комплексного числа.

Инициализация комплексной переменной может быть выполнена при объявлении, или в основной программе. В последнем случае содержимое в квадратных скобках опускают. Значение комплексного числа (результат вычисления в комплексной форме) выводится в виде вектора: (**re**, **im**). На рисунке 4.3 приведен пример объявления комплексного числа и вывода его значения.

```
// compl.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <complex>
#include <conio.h>
#include <clocale>

    using namespace std;

    int main(){
        setlocale(LC_ALL, "rus");
    complex < float > z( 23.0, 12.0 );      // z = 23 + 12i
        cout << "Введено комплексное число z=" << z << endl;
      _getch();
    return 0;
}
```

```
G:\compl\Debug\compl.exe
Введено комплексное число z=(23,12)
```

Рисунок 4.3 – Пример объявления комплексного числа

В выражениях с комплексными числами можно использовать операторы: **!=** (проверка на неравенство), **==** (проверка на равенство), ***** (умножение), **+** (сложение), **-** (вычитание), **/** (деление). Кроме этого класс комплексных чисел поддерживает четыре **составных оператора** присваивания: **+=**, **-=**, ***=** и **/=**.

Комплексные и арифметические типы разрешается смешивать в одном выражении, например, можно прибавить (вычесть) к комплексному числу вещественное, или умножить (разделить) комплексное число на вещественное.

В таблице 4.1 приведены основные функции стандартной библиотеки C++, предусмотренные для работы с комплексными числами, а на рисунке 4.4 – пример использования некоторых функций в программе.

Таблица 4.1 – Функции для работы с комплексными числами

Запись	Значение
abs(z)	Вычисляет модуль комплексного числа z
arg(z)	Извлекает аргумент из комплексного числа z
conj(z)	Возвращает комплексно-сопряженную величину комплексному числу z
cos(z)	Возвращает косинус комплексного числа z
exp(z)	Возвращает экспоненциальную функцию комплексного числа z
imag(z)	Извлекает мнимую часть комплексного числа z
log(z)	Возвращает натуральный логарифм комплексного числа z
log10(z)	Возвращает десятичный логарифм комплексного числа z
polar(abs(z), arg(z))	Возвращает комплексное число z в декартовой форме (в виде пары (real(z), imag(z)))
pow(z1, z2)	Вычисляет комплексное число, получаемое в результате возведения основания (комплексное число z1) в степень комплексного числа z2
real(z)	Извлекает вещественную часть комплексного числа z
sin(z)	Возвращает синус комплексного числа z
sqrt(z)	Возвращает квадратный корень комплексного числа z
tan(z)	Возвращает тангенс комплексного числа z

```
// compl.cpp: главный файл проекта.

#include "stdafx.h"
#include <iostream>
#include <complex>
#include <conio.h>
#include <clocale>

    using namespace std

    int main(){
        setlocale(LC_ALL, "rus");
    complex <float> z;      // Объявление комплексной переменной z
    cout << "  Введите комплексное число z "; cin >> z;
        cout << "  Введено комплексное число z=" << z << endl;
            cout << "  Комплексно-сопряженное число z1=" << conj(z) << endl;
                cout << "  Действительная часть числа z равна " << real(z) << endl;
                cout << "  Мнимая часть числа z равна " << imag(z) << endl;
            cout << "  Модуль числа z равен " << abs(z) << endl;
        cout << "  Аргумент числа z в градусах равен " << arg(z) * 180 / 3.141592 << endl;
    cout << "  Комплексное число z в декартовой форме равно " << polar(abs(z), arg(z));
        _getch();
    return 0;
}
```

```
G:\compl\Debug\compl.exe
  Введите комплексное число z (25, 18)
  Введено комплексное число z=(25,18)
  Комплексно-сопряженное число z1=(25,-18)
  Действительная часть числа z равна 25
  Мнимая часть числа z равна 18
  Модуль числа z равен 30.8058
  Аргумент числа z в градусах равен 35.7539
  Комплексное число z в декартовой форме равно (25,18)
```

Рисунок 4.4 – Пример использования функций стандартной библиотеки C++

Рассмотрим пример использования класса комплексных чисел для вычисления амплитудно-частотной (АЧХ) и фазочастотной (ФЧХ) характеристик четырехполюсника, комплексный коэффициент передачи которого описывается выражением:

$$\dot{K}(\omega)=\frac{1+j\omega RC_1}{1+j\omega R(C_1+C_2)}. \qquad (4.1)$$

Умножим числитель и знаменатель выражения (4.1) на выражение, комплексно сопряженное знаменателю, и выделим действительную и мнимую части комплексного выражения:

$$\dot{K}(\omega)=\frac{1+\omega^2R^2\left(C_1^2+C_1C_2\right)}{1+\left[\omega R(C_1+C_2)\right]^2}-j\frac{\omega RC_2}{1+\left[\omega R(C_1+C_2)\right]^2}. \qquad (4.2)$$

Выражение (4.2) используется в программе для вычисления АЧХ и ФЧХ (рисунок 4.5) четырехполюсника.

```
// mod.cpp: главный файл проекта.

#include "stdafx.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <iostream>
#include <complex>
#include <conio.h>
#include <clocale>
#include <iomanip>
    using namespace std;

    int main(){
        setlocale(LC_ALL, "rus");
        cout << "  Программа для вычисления АЧХ и ФЧХ четырехполюсника \n";
        double R, c1, c2, w;
    cout << "  Введите сопротивление резистора R= "; cin >> R;
    cout << "  Введите емкость конденсатора c1= "; cin >> c1;
    cout << "  Введите емкость конденсатора c2= "; cin >> c2;
        cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
        cout << "| n |    f    |   K(f)   |"; cout << "        "; cout << "| n |    f    |   ph(f)  |\n";
        cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
          int n = 0;
            for (int f = 0; f <= 1000; f += 50)
            {n++; w = 2*M_PI*f;
// Объявление и инициализация комплексной переменной k
    complex <double> k((1+pow(w*R,2)*(c1*c1+c1*c2))/(1+pow(w*R*(c1+c2),2)), -w*R*c2/(1+pow(w*R*(c1+c2),2)));
        cout << "|"; cout.width(3); cout << n;
            cout << "|"; cout.width(9); cout << fixed << setprecision(3) << f;
        cout << "|"; cout.width(10); cout << fixed << setprecision(3) << abs(k); cout << "|"; cout.width(9);
        cout << "|"; cout.width(3); cout << n;
            cout << "|"; cout.width(9); cout << fixed << setprecision(3) << f;
        cout << "|"; cout.width(10); cout << fixed << setprecision(3) << arg(k)*180/M_PI; cout << "|\n";}
        cout << "--------------------------"; cout << "        "; cout << "--------------------------\n";
        _getch();
    return 0;
}
```

```
G:\mod\Debug\mod.exe
  Программа для вычисления АЧХ и ФЧХ четырехполюсника
  Введите сопротивление резистора R= 1000
  Введите емкость конденсатора c1= 0.000001
  Введите емкость конденсатора c2= 0.000001
--------------------------        --------------------------
| n |    f    |   K(f)   |        | n |    f    |   ph(f)  |
--------------------------        --------------------------
|  1|        0|     1.000|        |  1|        0|    -0.000|
|  2|       50|     0.888|        |  2|       50|   -14.701|
|  3|      100|     0.735|        |  3|      100|   -19.346|
|  4|      150|     0.644|        |  4|      150|   -18.750|
|  5|      200|     0.594|        |  5|      200|   -16.815|
|  6|      250|     0.565|        |  6|      250|   -14.825|
|  7|      300|     0.547|        |  7|      300|   -13.091|
|  8|      350|     0.536|        |  8|      350|   -11.643|
|  9|      400|     0.528|        |  9|      400|   -10.445|
| 10|      450|     0.522|        | 10|      450|    -9.449|
| 11|      500|     0.518|        | 11|      500|    -8.614|
| 12|      550|     0.515|        | 12|      550|    -7.906|
| 13|      600|     0.513|        | 13|      600|    -7.301|
| 14|      650|     0.511|        | 14|      650|    -6.779|
| 15|      700|     0.509|        | 15|      700|    -6.324|
| 16|      750|     0.508|        | 16|      750|    -5.924|
| 17|      800|     0.507|        | 17|      800|    -5.571|
| 18|      850|     0.506|        | 18|      850|    -5.257|
| 19|      900|     0.506|        | 19|      900|    -4.976|
| 20|      950|     0.505|        | 20|      950|    -4.722|
| 21|     1000|     0.505|        | 21|     1000|    -4.493|
--------------------------        --------------------------
```

Рисунок 4.5 – Пример программы для вычисления АЧХ и ФЧХ четырехполюсника

4.4 Вопросы для самоконтроля

4.4.1 В чем состоят различия между модульным и объектно-ориентированным программированием?

4.4.2 Перечислите этапы разработки программ в объектно-ориентированном стиле.

4.4.3 Что понимают под классом в C++?

4.4.4 Назовите базовые принципы объектно-ориентированного программирования.

4.4.5 Поясните сущность инкапсуляции.

4.4.6 В чем сущность наследования?

4.4.7 В чем сущность полиморфизма?

4.4.8 Приведите структуру описания класса.

4.4.9 Поясните различия между глобальными и локальными классами.

4.4.10 Что такое экземпляр класса?

4.4.11 Каким образом осуществляется доступ к элементам данных и методам класса или объекта?

4.4.12 С помощью каких методов выполняется создание и удаление объектов?

4.4.13 Назовите статусы доступа к компонентам класса.

4.4.14 Как обеспечивается доступ из основной программы к методам класса?

4.4.15 Приведите формат объявления комплексной переменной.

4.4.16 Какие операторы можно использовать в выражениях с комплексными числами?

4.4.17 С помощью каких функций можно вычислить модуль и аргумент комплексного числа?

4.4.18 Как выполнить инициализацию комплексной переменной?

5 Программирование микроконтроллеров PIC на C++

5.1 Краткие сведения о компиляторах для микроконтроллеров с языка C

Под **компилятором** в программировании понимают программу (или техническое средство), выполняющую компиляцию. В процессе компиляции осуществляется трансляция модулей программы, написанных на языке программирования высокого уровня (или языке ассемблера), в эквивалентные программные модули на машинном языке и последующая сборка исполняемой машинной программы.

Микроконтроллеры PIC (производитель – компания Microchip Technology Inc.) основаны на модифицированной гарвардской RISC-архитектуре и поставляются в самых разнообразных аппаратных конфигурациях. Их отличительной особенностью является совместимость «снизу вверх». Это означает, что программы, написанные для более старых моделей микроконтроллеров, могут быть запущены и в более новых моделях. Базовый набор команд микроконтроллеров PIC содержит всего лишь 33 инструкции, и большинство моделей (за исключением самых современных) использует один и тот же набор команд [10].

Для компиляции программ, написанных на языке C, в машинные коды микроконтроллеров PIC разработаны несколько компиляторов, в частности:

– CCS-PICC (производитель Custom Computer Services, Inc., ccsinfo.com/ccs-product-catalog.php);

– mikroC (разработанный компанией MikroElektronica, www.mikroe.com);

– C30 и C32 (разработки компании Microchip, www.microchip.com).

Компилятор CCS-PICC доступен через интегрированную среду разработки (IDE), позволяющую создавать проекты из одного или нескольких файлов исходного кода, настраивать опции компилятора, а также компилировать исходный код в исполняемые файлы, предназначенные для загрузки в микроконтроллер. Спроектирован для работы совместно с отладчиком Microchip MPLAB (загружается бесплатно с сайта компании Microchip – www.microchip.com), представляющим собой программный имитатор микроконтроллеров PIC.

Компилятор mikroC, как и рассмотренные ранее средства программирования, предоставляет интегрированную среду разработки (IDE) и возможность выполнять эмуляцию программ.

Компиляторы C30 и C32 разработаны для создания программ на языке C для новых микроконтроллеров семейств PIC24 и PIC32 соответственно. Они, как и ранее рассмотренные, используются совместно с интегрированной средой MPLAB.

Все перечисленные компиляторы являются платными, поэтому их использование возможно только **после приобретения** соответствующей **лицензии**.

После компиляции программы, написанной на языке C, создается hex-файл (исполняемый шестнадцатеричный код), который с помощью программатора может быть загружен в память программ микроконтроллера.

Примечание – При разработке программы для микроконтроллера на языке C предполагается, что пользователь ознакомлен со структурой и особенностями функционирования конкретного микроконтроллера.

5.2 Стандартные функции ввода-вывода языка C

Стандартные функции ввода-вывода языка C построены на основе функций посимвольного ввода-вывода [10]. В обычных программах на языке C эти функции используются для ввода данных с клавиатуры и вывода на экран (или принтер). В программах для микроконтроллеров функции ввода-вывода применяются для обмена данными через приемопередатчик UART/USART (по умолчанию) или последовательный порт. Поэтому прежде, чем начать ввод-вывод, в программе должны быть выполнены соответствующие настройки с помощью директив препроцессора.

Простейшими функциями ввода-вывода в языке C являются функции **getchar()** и **putchar()** (объявлены в заголовочном файле **stdio.h**), которые предназначены для посимвольного обмена данными. Функция **getchar()** возвращает символ, принятый от приемопередатчика UART/USART или по последовательному интерфейсу, а функция **putchar()** выводит символ во внешнее устройство.

Функции **puts()** и **printf()** используют функцию **putchar()** для посимвольного вывода строки в порт RS232, назначенный последним. Функция **puts()** обеспечивает более простой вывод строковой переменной. Например, если определена строка символов

char s[] = ”abcde”;,

то вызов функции для вывода этой строки имеет вид:

puts(s);.

Функция **printf()** обеспечивает форматированный вывод. Формат функции:

printf("строка, в которую подставляются переменные", список_переменных);.

Каждую переменную в списке отделяют запятой.

В выводимой строке можно использовать **спецификации форматирования значений переменных**, переданных в качестве параметров. Каждая спецификация начинается со знака "**%**", после которого следуют обозначения параметров форматирования:

– c – вывод параметра в виде символа ASCII;

– d – вывод параметра в виде целого числа;

– e – вывод параметра в экспоненциальном формате;

– f – вывод параметра в виде вещественного числа;

– ld – вывод параметра в виде длинного целого со знаком;

– lu – вывод параметра в виде беззнакового длинного целого;

– Lx – вывод параметра в виде беззнакового длинного целого в шестнадцатеричном представлении с использованием нижнего регистра символов;

– LX – вывод параметра в виде беззнакового длинного целого в шестнадцатеричном представлении с использованием верхнего регистра символов;

– s – вывод параметра в виде строки;

– u – вывод параметра в виде беззнакового целого;

– x – вывод параметра в виде беззнакового целого в шестнадцатеричном представлении с использованием нижнего регистра символов;

– X – вывод параметра в виде беззнакового целого в шестнадцатеричном представлении с использованием верхнего регистра символов;

– % – вывод знака процента.

Количество спецификаций должно совпадать с количеством переменных.

При выводе числовых значений после знака "**%**" можно указать количество выводимых символов и формат отображения числа, например:

– %5 – пять знаков;

– %6.2 – всего шесть знаков, два знака после десятичной точки;

– %06 – шесть знаков с дополнением нулями слева.

Пример –

```
int a = 157;
float b = 12.48;
char str[] = "Microchip";
printf("a = %d, b = %8.3f, str = %s", a, b, str);
printf("a = 0x%04x", a);.
```

В результате работы оператора **printf()** будет выведено:

a = 157, b = 12.480, str = Microchip

a = 0x009d.

В последней строке примера значение константы **a** выведено (с учетом выбранного параметра форматирования) в шестнадцатеричном представлении с использованием нижнего регистра символов.

Функции **gets()** и **scanf()** выполняют действия, обратные функциям **puts()** и **printf()**: считывают посимвольно строку через назначенный последним последовательный интерфейс.

Формат функции **scanf()** аналогичен формату функции **printf()**. Функция считывает из входного потока значения в формате, определенном в первом параметре функции, и сохраняет их в переменных, адреса которых переданы ей в качестве последующих параметров. Спецификации форматирования для функции

scanf() такие же, как и для функции **printf()**. В программе функции **scanf()** и **printf()** используются в паре.

Пример –

```
scanf("%f, %x \n", &x, &y); // Считывает в память два числа: одно
                            // вещественное в десятичной форме, а
                            // второе – в шестнадцатеричной форме
printf("x = %f, y = %04x", x, y); // Вывод чисел.
```

5.3 Директивы препроцессора

Как отмечено в п. р. 1.3, чтобы в тексте программы можно было использовать библиотечные функции или различные операторы, необходимо с помощью директив препроцессора подключить соответствующие заголовочные файлы, содержащие их определения (описания). Препроцессор представляет собой программу, выполняющую предварительную обработку данных до начала компиляции программы. Стандартной директивой языка С (С++), используемой практически во всех программах, является директива **#include**. Она может использоваться в двух формах:

```
#include <имя_файла>
```

или

```
#include "имя_файла"
```

Если имя заголовочного файла помещено между знаками < и >, то он считается частью стандартной библиотеки, поставляемой вместе с компилятором. Если же имя заголовочного файла заключено в двойные кавычки, то файл расположен в папке с остальными файлами проекта.

Еще одной стандартной директивой языка С (С++) является директива **#define**. Она определяет идентификатор и последовательность символов, которой будет замещаться данный идентификатор при его обнаружении в тексте программы. Идентификатор еще называется именем **макроса**, а процесс замещения – подстановкой макроса. Формат директивы **#define**:

```
#define имя_макроса последовательность_символов.
```

Например, если в программе нужно использовать константу π, с помощью директивы **#define** можно определить ее значение:

#define pi 3.1415927.

После этого если компилятор обнаружит в тексте программы макрос **pi**, он заменит этот макрос на число **3.1415927**.

При использовании директивы **#define** необходимо учитывать следующее:

– при создании комментариев в строке с **#define** всегда используется комментарий вида /* ... */;

– следует помнить, что конец строки – это конец **#define**, и весь текст слева заменит текст справа;

– для переноса текста подстановки на другую строку используется символ обратной косой черты «\»;

– для отмены определения используется директива **#undef**, например:

#define pi 3.1415927

…

#undef pi

#define pi 3.14.

Кроме стандартных директив каждый компилятор поддерживает множество встроенных нестандартных директив, специфичных для конкретного компилятора. Рассмотрим некоторые директивы, характерные **для компилятора CCS-PICC**.

Директива препроцессора **#bit** используется для получения доступа к отдельным разрядам регистров или переменных. Ее формат:

#bit идентификатор = x.y,

где x – константа, определяющая регистр, или переменная;

y – константа от 0 до 7 (номер разряда).

Например, если переменную T0IF нужно объявить как разряд 2 регистра по адресу 0x0B, то в программе нужно записать:

#bit T0IF = 0x0B.2.

Директива **#byte** имеет следующий формат:

#byte идентификатор = X,

где X – имя переменной или константы.

Если идентификатор соответствует ранее объявленному имени переменной, то компилятор помещает эту переменную по адресу X, в противном случае (если переменная не объявлена) компилятор создает новую переменную и помещает ее по адресу X как восьмиразрядное целое число. Например [10]:

```
char c;          //Переменная с размещается в памяти компилятором
#byte a = c;  //Переменной a назначен тот же адрес, что и переменной c
```

Директива **#device** определяет, для какого типа микроконтроллеров написана программа. Формат директивы:

#device имя_микроконтроллера [параметр].

Параметр является необязательным и используется для **управления памятью**, **аналого-цифровым преобразованием** и **возможностью отладки**. В частности, параметр управления памятью позволяет пользователю задать количество разрядов, используемых для хранения указателя памяти. Возможны следующие значения параметра управления памятью: *=5 (для всех семейств микроконтроллеров PIC), *=8 (для 14-ти- и 16-тиразрядных микроконтроллеров), *=16 (для 14-тиразрядных микроконтроллеров). Например, приведенная ниже директива определяет для микроконтроллера PIC16F877 16-тиразрядный указатель памяти:

#device PIC16F877 *=16.

Для управления АЦП используется параметр **ADC = x**, где x – количество разрядов, считываемых из преобразователя с помощью внутренней функции **read_adc()**.

Если необходимо, чтобы сгенерированный код был совместим с отладочным программным обеспечением ICD от компании Microchip, в директиву **#device** следует включить параметр **ICD = TRUE**.

Директива **#fuse** определяет, какие предохранители должны быть установлены при программировании микроконтроллера. Формат директивы:

#fuse параметры.

Набор параметров для каждого микроконтроллера свой. Одновременно в одной директиве можно записать несколько параметров, отделяя их запятыми.

Директива препроцессора **#locate** по своему назначению аналогична директиве **#byte** за тем исключением, что назначает переменной ячейку памяти, в которую не может быть помещена никакая другая переменная. Формат директивы:

#locate идентификатор = X,

Например, с помощью директивы **#locate** переменная **c** размещается по адресу **0x50**:

#locate c = 0x50.

Директива **#priority** может быть использована для установки порядка, в котором опрашиваются флаги прерываний. Когда возникает прерывание, управление передается по адресу, указанном в векторе прерываний. По умолчанию, компилятор CCS-PICC помещает по этому адресу диспетчерскую подпрограмму, которая опрашивает флаги прерываний, чтобы определить, какое из них имело место, и вызвать соответствующую подпрограмму обслуживания прерывания.

Формат директивы **#priority**:

#priority список_прерываний.

Прерывания в списке отделяют запятыми.

Директива препроцессора **#rom** позволяет вставлять данные в файл **.hex**. Ее формат:

#rom адрес = {список}

где **адрес** – адрес в памяти контроллера;

список – список слов через запятую.

Пример –

#rom 0x1B08 = {5, 2, 4, 2, 1, 3, 1}.

Директива препроцессора **#type** позволяет переопределять типы, поддерживаемые компилятором. Ее формат:

#type стандартный_тип = размер, стандартный_тип = размер, ...

Например, по умолчанию компилятор CCS-PICC отводит под переменную типа **char** восемь бит (один байт), а под переменную типа **int** – 16 бит (два байта). Размер указанных типов переменных (объем памяти, выделяемой под каждую из них) можно изменить с помощью директивы **#type**, например:

#type char = 16, int = 32.

Директива **#use delay** предназначена для задания рабочей частоты процессора и разрешает использование в программе внутренних функций **delay_ms()** (задержка в миллисекундах) и **delay_us()** (задержка в микросекундах). Формат директивы имеет две формы:

#use delay(частота)

#use delay(частота, restart_WDT).

Частоту задают в герцах. Параметр **restart_DWT** указывает компилятору на то, что нужно перезагрузить сторожевой таймер во время программных задержек.

Директива препроцессора **#use rs232** используется для инициализации последовательного порта, работающего по стандарту RS232. В зависимости от того, как заданы выводы передачи и приема, реализация порта может быть аппаратной или программной. Если выводы соответствуют приемопередатчику USART, то используется аппаратный порт, в противном случае – программный UART. Формат директивы:

#use rs232 (параметр, параметр, параметр, ...).

В директиве могут быть использованы следующие параметры:

– **BAUD = x** – установка скорости передачи;

– **BITS = x** – установка разрядности в x, где x – от пяти до девяти в случае программного UART и от восьми до девяти в случае аппаратного UART;

– **ENABLE = вывод** – во время передачи компилятор переводит указанный вывод в состояние высокого уровня;

– **ERRORS** – ошибки приема сохраняются в переменной RS232_ERRORS;

– **FLOAT_HIGH** – использование на выходе схем с открытым коллектором;

– **INVERT** – инвертирует полярность выводов последовательного порта (используется только с программным приемопередатчиком UART);

– **PARITY = x** – контроль по четности: x = N – отсутствует; x = E – контроль по четности; x = O – контроль по нечетности;

– **RCV = вывод** – установка вывода для приема данных;

– **RESTART_WDT** – сброс сторожевого таймера при ожидании символа в функции **getchar()**;

– **STREAM = имя_потока** – ассоциирует идентификатор потока с портом RS232 (этот идентификатор может использоваться в функциях, наподобие **fputc()**);

– **XMIT = вывод** – установка вывода для передачи данных.

Директива препроцессора **#zero_ram** указывает компилятору обнулять перед началом выполнения программы все внутренние регистры, которые могут быть использованы для хранения переменных.

5.4 Обработка прерываний

В среде компилятора CCS-PICC для определения подпрограмм обработки прерываний используются директивы препроцессора **#int_default**, **#int_global** и **#int_xxx** (указываются непосредственно перед функцией) [10]. Во многих микроконтроллерах PIC используется единственный вектор прерывания. Поэтому при возникновении прерывания вызывается только одна подпрограмма (диспетчер

прерываний), которая отвечает за опрос флагов прерываний и вызов соответствующих подпрограмм обслуживания прерываний. Если обнаружено прерывание, но ни один из флагов прерывания не установлен, то диспетчер вызывает функцию, обозначенную с помощью директивы **#int_default**.

Директива **#int_global** позволяет определить функцию, заменяющую диспетчер компилятора. Директива **#int_xxx** определяет функцию, отвечающую за обслуживание того или иного прерывания, например:

– **#int_ad** – аналого-цифровое преобразование завершено;

– **#int_adof** – задержка аналого-цифрового преобразования;

– **#int_buscol** – конфликт шины;

– **#int_button** – нажатие кнопки;

– **#int_ccp1** – захват или совпадение в модуле CCP1;

– **#int_ccp2** – захват или совпадение в модуле CCP2;

– **#int_comp** – прерывание от аналогового компаратора;

– **#int_eeprom** – запись в EEPROM завершена;

– **#int_ext** – внешнее прерывание;

– **#int_ext1** – внешнее прерывание 1;

– **#int_ext2** – внешнее прерывание 2;

– **#int_i2c** – прерывание от шины I^2C;

– **#int_lcd** – ЖК-дисплей активен;

– **#int_lowvolt** – обнаружено низкое напряжение;

– **#int_psp** – запись данных в порт PSP;

– **#int_rb** – изменение состояния выводов 4-7 порта B;

– **#int_rc** – изменение состояния выводов 4-7 порта C;

– **#int_rda** – доступен прием данных по интерфейсу RS232;

– **#int_rtcc** – переполнение таймера TMR0;

– **#int_ssp** – активность интерфейса SPI или I^2C;

– **#int_tbe** – буфер передачи по интерфейсу RS232 пуст;

– **#int_timer0** – переполнение таймера TMR0;

– **#int_timer1** – переполнение таймера TMR1;

– **#int_timer2** – переполнение таймера TMR2;

– **#int_timer3** – переполнение таймера TMR3.

В программах, предназначенных для компилятора mikroC, вся обработка прерываний функции **interrupt**:

```
void interrupt ()
{
…
}.
```

В случае, если в программе может возникать несколько различных видов прерывания, это необходимо проверять внутри функции interrupt путем опроса соответствующих флагов, например [10]:

```
void interrupt() {
if (INTCON.TMR0IF)
{counter++;
TMRO = 96;
INTCON.TMR0IF = 0;}
else if (INTCON.RBIF)
{counter++;
TMRO = 96;
INTCON.RBIF = 0;}
}.
```

В программах, предназначенных для компилятора C30 (микроконтроллеры PIC24), для связывания некоторой функции с тем или иным вектором прерывания служит следующий оператор:

```
void_ISR_ОбозначениеТипаПрерывания(void)
{
…
}.
```

Для обозначения типа прерывания служат макроопределения, заданные в соответствующем файле сценария компоновки **.gld**.

5.5 Использование ассемблерного кода в программе на С

В программу на языке С, написанную для микроконтроллера, можно вставлять команды на ассемблере. Это в некоторых случаях позволяет оптимизировать код программы. Рассмотрим средства различных компиляторов, обеспечивающие возможность обработки таких программ.

В программах, разрабатываемых в среде CCS-PICC, для вставки ассемблерных фрагментов используются директивы препроцессора **#asm** и **#endasm**:

#asm
ассемблерный код
#endasm.

Если требуется, чтобы компилятор не выполнял автоматического переключения банков памяти для переменных, к которым нельзя получить доступ из текущего банка, в директиве **#asm** следует использовать параметр **ASIS**:

#asm ASIS
ассемблерный код
#endasm.

Без параметра ASIS выполняется расширенное ассемблирование, в результате которого доступ к переменным всегда реализуется корректно, благодаря добавлению (там, где это необходимо) переключения банков.

В программах, предназначенных для компилятора mikroC, ассемблерный код вставляют с помощью ключевого слова **asm**:

asm {
ассемблерный код
}.

В компиляторах C30 и C32 ассемблерные команды добавляются в программу на С по одной, с помощью функции **asm**. Например:

asm("mov.b [w1++], w0");
asm("ze w0, w0");.

Для того чтобы на этапе оптимизации компилятор не изменял порядок следования команд и позицию встроенного ассемблерного кода, к каждому вызову функции **asm** необходимо добавить атрибут **volatile**:

volatile asm("mov.b [wl++], w0");
volatile asm("ze w0, w0");.

5.6 Примеры программ на С для компилятора mikroC

Рассмотрим несколько примеров программ для микроконтроллера PIC18F452, рассчитанных на компиляцию с помощью mikroC [10].

Пример 1 Разработать программу для моделирования игральных костей. Одну игральную кость можно реализовать с помощью семи светодиодов, расположенных в соответствии с правилом размещения точек на гранях реальной кости (рисунок 5.1). Устройство моделирует бросание двух костей.

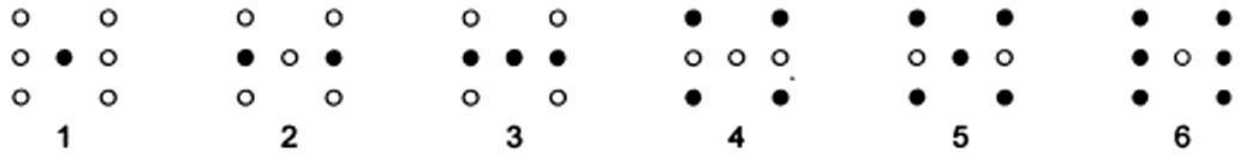


Рисунок 5.1 – Схема размещения светодиодов (черной точке соответствует включенный светодиод)

Алгоритм работы устройства состоит в следующем. При запуске устройства все светодиоды обеих «костей» отключены. При нажатии кнопки микроконтроллер генерирует два случайных числа в диапазоне от единицы до шести для каждой из «костей». Числа отображаются с помощью соответствующей группы светодиодов в

течение трех секунд. По истечении трех секунд все светодиоды опять гаснут и устройство находится в режиме ожидания до следующего нажатия кнопки.

Схема подключения светодиодов к порту микроконтроллера показана на рисунке 5.2.

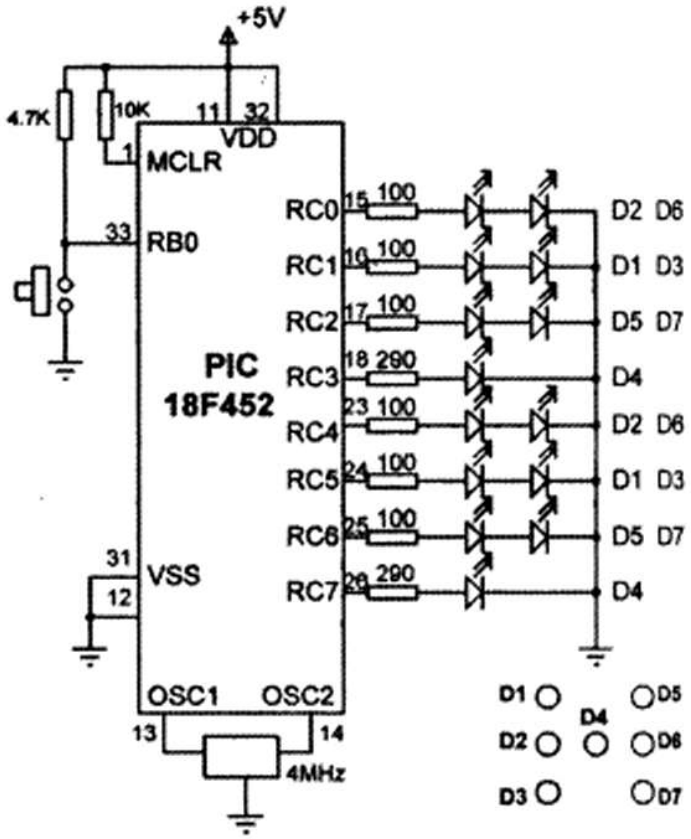


Рисунок 5.2 – Схема подключения светодиодов и кнопки к микроконтроллеру

Светодиоды первой и второй «костей» сгруппированы таким образом, чтобы использовать минимально необходимое количество выводов микроконтроллера (используется только один порт С, линии которого работают в режиме потребления тока через ограничительные резисторы сопротивлением 100 Ом и 290 Ом, рисунок 5.2). Кнопка для «бросания костей» подключена к выводу 0 порта В с использованием подтягивающего резистора.

Чтобы понять, как используются светодиоды при индикации цифр от единицы до шести, достаточно проанализировать рисунки 5.1 и 5.2. Видно, что для управления семью светодиодами одной «кости» достаточно четыре линии, поэтому управлять двумя «костями» можно с помощью одного восьмиразрядного порта. На рисун-

ке 5.2 первой «кости» соответствуют выводы RC0 – RC3, а второй – линии RC4 – RC7.

В таблице 5.1 представлены двоичные коды, выводимые в линии порта С в зависимости от числа, выпавшего на одной из двух «костей», и их шестнадцатеричные эквиваленты.

Таблица 5.1 – Двоичные коды, выводимые в линии порта С

Число	RC3/RC7	RC2/RC6	RC1/RC5	RC0/RC4	В шестнадцатеричном виде
1	1	0	0	0	8
2	0	0	0	1	1
3	1	0	0	1	9
4	0	1	1	0	6
5	1	1	1	0	E
6	0	1	1	1	7

Вариант программы, реализующей «бросание костей» [10]:

```
#include <18F452.h>          // Подключение файла с описанием PIC18F452
#define Switch PORTB.F0   // Кнопка
#define Pressed 0             // Если нажата, RB0 = 0
// Функция, генерирующая псевдослучайное число от 1 до Lim
unsigned char Number(int Lim, int Y)
{
static unsigned int Y;
Y = (Y * 32719 + 3) % 32749;
return ((Y % Lim) + 1);
}
void main() {
unsigned char LED1, LED2, Seed = 1;
// Определяем массив с числами, соответствующими граням "кости"
unsigned char Dice[] = {0, 0x8, 0x1, 0x9, 0x6, 0xE, 0x7};
TRISC = 0;        // Выводы порта С – выходы
TRISB = 1;        // RB0 – вход
PORTC = 0;        // Выключаем все светодиоды
```

```
while(1)            // Бесконечный цикл
{
if(Switch == Pressed)  // Кнопка нажата?
{
LED1 = Dice[Number(6, seed)]; // Получаем шаблон для кости 1
LED2 = Dice[Number(6, seed)]; // Получаем шаблон для кости 2
PORTC = 16*LED2 + LED1;    // Включаем светодиоды костей
Delay_ms(3000);                  // Задержка на 3 секунды
PORTC = 0;                        // Отключаем светодиоды
                }
        }
}
```

Пример 2 Разработать вольтметр, измеряющий напряжения в диапазоне от нуля до пяти вольт с индикацией результата на жидкокристаллическом дисплее. Измеряемое напряжение подается на один из аналоговых входов микроконтроллера PIC18F452, а результат аналого-цифрового преобразования отображается с помощью ЖК-модуля, совместимого с HD44780 от компании Hitachi. Модуль индикации подключается к выводам порта С и работает в четырехразрядном режиме.

Схема подключения микроконтроллера к внешним устройствам приведена на рисунке 5.3 [10].

Поскольку аналого-цифровой преобразователь (АЦП) микроконтроллера P1C18F452 – 10-разрядный, опорное напряжение 5 В можно разделить максимум на 1024 уровня дискретизации. При этом шаг квантования по уровню соответствует 4,88 мВ. С учетом этого переменная **Vin** (измеряемое напряжение), полученная в результате опроса вывода AN0, умножается на 488, а затем целочисленно делится на 100 для получения целой и дробной частей значения в милливольтах.

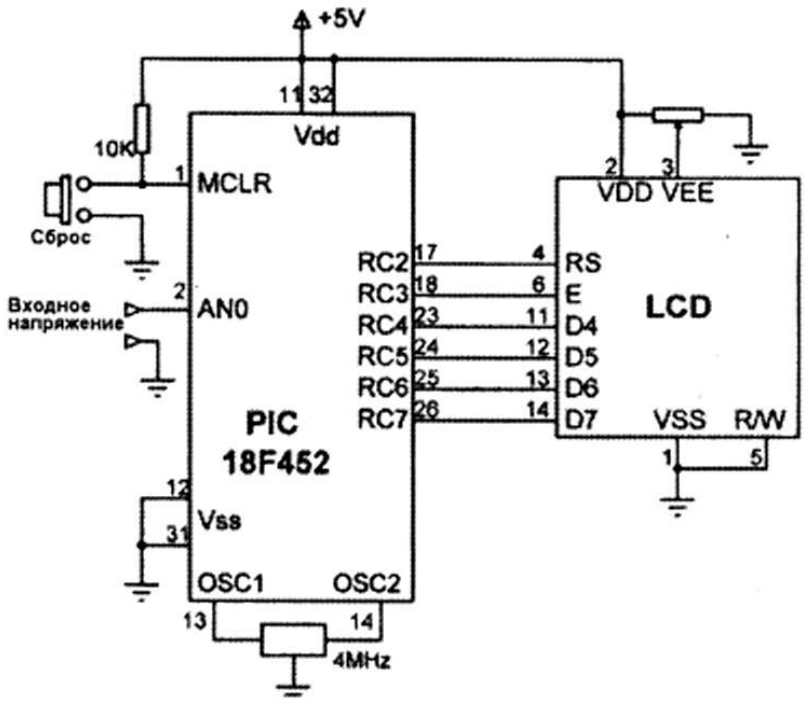


Рисунок 5.3 – Функциональная схема цифрового вольтметра

Вариант программы, реализующей вольтметр [10]:

```
void main() {
unsigned long Vin, mV, Vdec, Vfrac;
unsigned char op[12];
unsigned char i, j, lcd[5], ch1, ch2;
TRISC = 0;                          // Выводы порта C – выходы
TRISA = 0xFF;                       // Выводы порта A – входы
// Конфигурирование ЖК-модуля
Lcd_Init(&PORTC);                   // Подключен к порту C
Lcd_Cmd(LCD_CLEAR);                 // Очистка дисплея
Lcd_Out(1, 1, "VOLTMETER");         // Вывод строки
Delay_ms(2000);                     // Задержка на 2 секунды
// Конфигурирование АЦП: канал ANO, Vref = +5 В
ADCON1 = 0x80;
while(1)
{
```

```
Lcd_Cmd(LCD_CLEAR);              // Очистка дисплея
Vin = Adc_Read(0) * 488,         // Напряжение на выводе AN0
Vdec = Vin / 100;                // Целая часть
Vfrac = Vin % 100;               // Дробная часть
LongToStr(Vdec, op);             // Преобразование Vdec в строку
// Удаление лишних пробелов
j = 0;
for(i = 0; i <= 11; i++)
{
        if(op[i] != ' ')
        {
                lcd[j] = op[i];
                j++;
        }
}
// Вывод результата на ЖК-индикатор
Lcd_Out (1, 1, "mV = ");
Lcd_Out(1, 6, lcd);
Lcd_Out_Cp(".");
ch1 = Vfrac /10;                 // Формирование дробной части
ch2 = Vfrac % 10;
Lcd_Chr_Cp(48 + ch1);            // Отображение дробной части
Lcd_Chr_Cp(48 + ch2);            // '0' + . . .
Delay_ms(1000);                  // Задержка 1 секунда
        }
}
```

Пример 3 Разработать калькулятор на основе клавиатурной матрицы 4x4 и ЖК-модуля, совместимого с HD44780 от компании Hitachi. Клавиатура будет подключена к выводам порта B, а ЖК-модуль – к выводам порта C (рисунок 5.4).

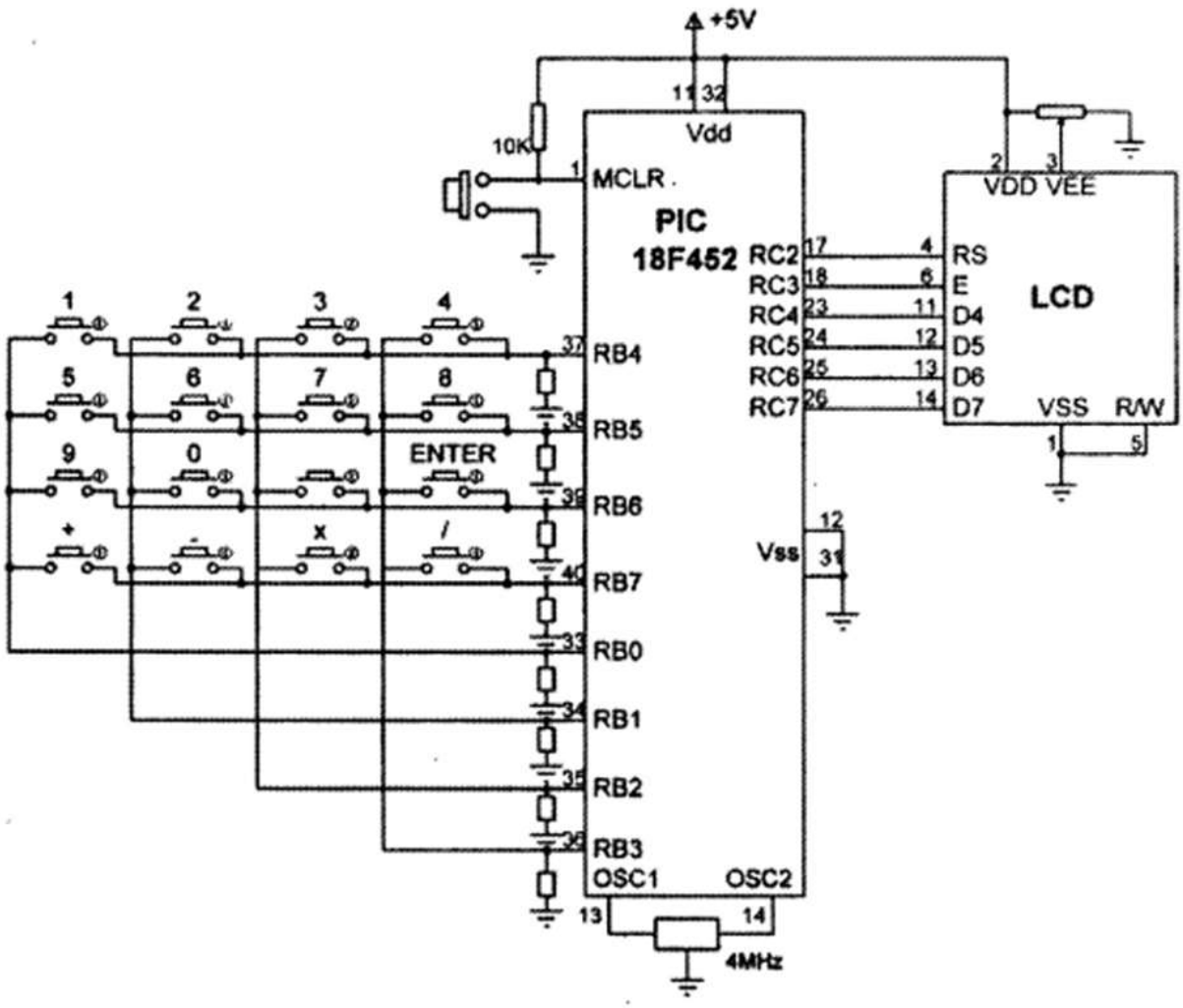


Рисунок 5.4 – Функциональная схема калькулятора

Алгоритм работы калькулятора состоит в следующем [10]:

1) в первой строке дисплея отображается «No1:»;

2) пользователь вводит с помощью клавиатуры первое число и нажимает клавишу <**Enter**>;

3) во второй строке дисплея отображается «No2:»;

4) пользователь вводит с помощью клавиатуры второе число и нажимает клавишу <**Enter**>;

5) дисплей очищается и в первой строке отображается «Op:»;

6) пользователь нажимает клавишу, соответствующую требуемой операции;

7) дисплей очищается и в первой строке в течение пяти секунд выводится результат операции, после чего дисплей очищается и процесс можно повторить с начала.

Нажатие той или иной клавиши определяется по ее порядковому номеру от нуля до пятнадцати по следующей схеме: 0 – «1»; 1 – «2»; 2 – «3»; 3 – «4»; 4 – «5»; 5 – «6»; 6 – «7»; 7 – «8»; 8 – «9»; 9 – «0»; 10 – « »; 11 – «Enter»; 12 – «+»; 13 – «–»; 14 – «x»; 15 – «/».

Вариант программы для микроконтроллера, функционирующего в режиме калькулятора:

```
#include <18F452.h>

// Определение номеров управляющих клавиш

#define Enter 12
#define Plus 13
#define Minus 14
#define Multiply 15
#define Divide 16
void main() {
unsigned char Key, i, j, lcd[5], op[12];
unsigned long Calc, Op1, Op2;
TRISC = 0;                          // Выводы порта C – выходы
Lcd_Init(&PORTC);                   // ЖК-модуль подключен к порту C
Lcd_Cmd(LCD_CLEAR);                 // Очистка дисплея
Lcd_Out(1, 1, "CALCULATOR"); // Вывод строки
Delay_ms(2000);                     // Задержка 2 секунды
Lcd_Cmd(LCD_CLEAR);                 // Очистка дисплея
Keypad_Init(&PORTB);                // Клавиатура подключена к порту B
while(1)
{
        Key = 0;
        Op1 = 0;
        Op2 = 0;
Lcd_Out(1, 1, "No1:");              // Ввод 1-го числа
while(1)
```

```
{
        do
        Key = Keypad_Released();    // Считывание номера нажатой клавиши
        while(!Key);                // Пока клавиша не нажата, ожидание
        if(Key == Enter) break;     // Если нажато <Enter>, то выход
        if(Key == 10) Key = 0;      // Если нажат пробел, считать за «1»
        Lcd_Chr_Cp(Key + '0');      // Вывод числа
        Op1 = 10*Op1 + Key;         // Получение первого операнда
}
Lcd_Out(2, 1, "No2:");              // Ввод 2-го числа
while (1)
{
        do
        Key = Keypad_Released();
        while(!Key);
        if(Key == Enter) break;
        if(Key == 10) Key = 0;
        Lcd_Chr_Cp(Key + '0');
        Op2 = 10*Op2 + Key;         // Получение второго операнда
}
Lcd_Cmd(LCD_CLEAR);                 // Очистка дисплея
Lcd_Out(1, 1,"Op:");                // Ввод операции
while (1)
do
        Key = Keypad_Released();
        while(!Key);
Lcd_Cmd(LCD_CLEAR);                 // Очистка дисплея
Lcd_Out(1, 1, "Res=");              // Вывод результата
switch(Key)
{
```

```
case Plus: Calc = Op1 + Op2;          // Сложение
        break;
case Minus: Calc = Op1 - Op2;         // Вычитание
        break;
case Multiply: Calc = Op1 * Op2;    // Умножение
        break;
case Divide: Calc = Op1 / Op2;        // Деление
        break;
}
LongToStr(Calc, op);                  // Преобразование результата в строку
// Удаление лишних пробелов
j = 0;
for (i = 0; i <= 11; i++)
{
        if(op[i] != ' ')
        {
                lcd[j] = op[i];
                j++;
        }
}
Lcd_Out_Cp(lcd);                      // Вывод результата
Delay_ms(5000);                       // Задержка 5 секунд
Lcd_Cmd(LCD_CLEAR);                   // Очистка дисплея
        }
}
```

5.7 Вопросы для самоконтроля

5.7.1 Что называется компилятором? Перечислите задачи, решаемые компилятором.

5.7.2 Назовите наиболее известные компиляторы с языка С в машинные коды для микроконтроллеров PIC.

5.7.3 Назовите функции ввода языка С. В каком заголовочном файле они определены?

5.7.4 Назовите функции вывода языка С.

5.7.5 Для чего используют спецификации форматирования значений переменных?

5.7.6 Приведите формат функции printf() с использованием спецификаций.

5.7.7 Приведите стандартные директивы препроцессора языка С.

5.7.8 В чем различие между директивами #include <имя_файла> и #include "имя_файла"?

5.7.9 Назовите встроенные директивы компилятора CCS-PICC. Какое их назначение?

5.7.10 Как добавить в программу на языке С ассемблерный код?

Список использованных источников

1 Аверкин, В. П. Программирование на С++ : учебное пособие / В. П. Аверкин, А. И. Бобровский, В. В. Веснич, В. Ф. Радушинский, А. Д. Хомоненко; под ред. проф. А. Д. Хомоненко. – СПб. : Корона-Принт, 1999. – 252 с.

2 Зиборов, В. В. MS Visual C++ 2010 в среде .NET. Библиотека программиста / В. В. Зиборов. – СПб. : Питер, 2012. – 320 с. – ISBN 978-5-459-00786-2.

3 Культин, Н. Б. Основы программирования в Microsoft® Visual C++ 2010 / Н. Б. Культин. – СПб. : БХВ-Петербург, 2010. – 384 с. – ISBN 978-5-9775-0520-8.

4 Павловская, Т. А. C/C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2003. – 461 с. – ISBN 5-94723-568-4.

5 Пахомов, Б. И. C/C++ и MS Visual C++ 2010 для начинающих / Б. И. Пахомов. – СПб. : БХВ-Петербург, 2011. – 736 с. – ISBN 978-5-9775-0599-4.

6 Подбельский, В. В. Язык Си++ : учеб. пособие / В. В. Подбельский. – 5-е изд. – М. : Финансы и статистика, 2003. – 560 с. – ISBN 5-279-02204-7.

7 Прокопенко, В. С. Программирование микроконтроллеров ATMEL на языке С / В. С. Прокопенко. – Киев : «МК-Пресс»; СПб. : «КОРОНА-ВЕК», 2012. – 320 с. – ISBN 978-5-7931-0906-2 («КОРОНА-ВЕК»); ISBN 978-966-8806-73-5 («МК-Пресс»).

8 Слабнов, В. Д. Программирование на С++ [Электронный ресурс] : лекции / В. Д. Слабнов; Институт экономики, управления и права (г. Казань). – Казань : Познание, 2012. – 136 с. – ISBN 978-5-8399-0386-9. – Режим доступа: http://biblioclub.ru/index.php?page=book&id=364222. – ЭБС «Университетская библиотека онлайн».

9 Хортон, А. Visual C++ 2010 : полный курс / А. Хортон; пер. с англ. В. А. Коваленко. – М. : ООО «И.Д. Вильямс», 2011. – 1216 с. – ISBN 978-5-8459-1698-3 (рус.).

10 Шпак, Ю. А. Программирование на языке С для AVR и PIC микроконтроллеров. Изд. 2-е, переработанное и дополненное / Ю. А. Шпак. – Киев : «МК-Пресс»; СПб. : «КОРОНА-ВЕК», 2011. – 544 с. – ISBN 978-5-7931-0842-3 («КОРОНА-ВЕК»); ISBN 978-966-8806-67-4 («МК-Пресс»).

Приложение А

(*справочное*)

Функции компилятора mikroC

Таблица А.1 – Математические функции компилятора

Функция	Библиотека	Заголовок	Описание
1o		unsigned short lo(long n)	Возвращает младший (первый) байт числа n
hi		unsigned short hi(long n)	Возвращает второй байт числа n
higher		unsigned short higher(long n)	Возвращает третий байт числа n
highest		unsigned short highest(long n)	Возвращает старший (четвертый) байт числа n
max	stdlib.h	int max(int a, int b)	Возвращает большее из двух чисел a и b
min	stdlib.h	int min(int a, int b)	Возвращает меньшее из двух чисел a и b

Таблица А.2 – Функции компилятора для работы со строками

Функция	Библиотека	Заголовок	Описание
sprinti	sprint.h	int sprinti (char *buf, const char *fmt [, arg1, arg2, …])	Функция, которая отличается от функции sprintf только тем, что не поддерживает целые числа типа long
sprintf	sprint.h	int sprintf (char *buf, const char *fmt [, arg1, arg2, …])	Форматирует набор строк и числовых значений, хранимых в буфере buf, согласно правилам форматирования, заданным строкой fmt, и помещает результирующую строку в буфер. Возвращает количество символов, фактически записанных в буфер
sprint1	sprint.h	int sprint1 (char *buf, const char *fmt [, argl, arg2, ...])	Функция, которая отличается от функции sprintf только тем, что не поддерживает вещественные числа
strcspn	string.h	char strcspn(char *s1, char *s2)	Вычисляет длину наибольшего начального сегмента строки s1, состоящего исключительно из символов, отсутствующих в строке s2. Возвращает длину сегмента
strncat	string.h	char *strncat(char *sl, char *s2, int n)	Добавляет не более n символов из строки s2 к строке s1. Первый символ s2 затирает символ "\0" в конце s1. Этот же символ автоматически добавляется в конце результирующей строки. Функция возвращает указатель на s1
strpbrk	string.h	char *strpbrk(char *s1, char *s2)	Ищет в строке s1 первое вхождение любого символа из строки s2 и возвращает индекс такого символа в s1. Если символ не найден, возвращает значение $FF.

Продолжение таблицы А.2

Функция	Библиотека	Заголовок	Описание
strspn	string.h	int strspn(char *sl, char *s2)	Возвращает длину наибольшего начального сегмента s1, состоящего исключительно из символов, входящих в состав строки s2
strtok	string.h	char *strtok(char *s1, char *s2)	Просматривает строку s1 в поиске первого символа, не входящего в строку s2. При этом предполагается, что s1 состоит из последовательности подстрок, разделенных символами из строки s2. Первый вызов функции вернет указатель на первый символ-разделитель, после которого в строке будет вставлен символ "\0". Последующие вызовы функции будут возвращать следующие подстроки со вставкой вместо разделителя символа "\0" до тех пор, пока не будут просмотрены все разделители. После этого возвращается NULL

Таблица А.3 – Функции компилятора для управления микроконтроллером

Функция	Библиотека	Заголовок	Описание
Clock_Khz		unsigned Clock_Khz(void)	Возвращает значение тактовой частоты микроконтроллера в кГц, округленное до ближайшего целого
Clock_Mhz		unsigned short Clock_Mhz(void)	Возвращает значение тактовой частоты микроконтроллера в МГц, округленное до ближайшего целого
Delay_Cyc		void Delay_Cyc(char C)	Создает задержку, равную 10 машинным циклам, умноженным на параметр С
Delay_ms		void Delay_ms (const time)	Создает задержку на число миллисекунд time
Delay_us		void Delay__us (const time)	Создает задержку на число микросекунд time
Time_dateDiff	timelib.h	long Time_dateDiff (TimeStruct *tl, TimeStruct *t2)	Возвращает разницу в секундах между датами t1 и t2
Vdelay_ms		void Vdelay_ms (unsigned time)	Аналог функции Delay_ms, однако в данном случае параметр time – не константа, а переменная

Таблица А.4 – Функции преобразований

Функция	Библиотека	Заголовок	Описание
Bcd2Dec		char Bcd2Dec(char bcdnum)	Преобразовывает BCD-число bcdnum в его десятичный эквивалент
Bcd2Decl6		unsigned Bcd2Dec (unsigned bcdnum)	Преобразовывает BCD-число bcdnum в его десятичный эквивалент
ByteToStr		void ByteToStr (unsigned short number, char *output)	Создает строку output длиной в три символа из числа number (0…256)
Dec2Bcd		char Dec2Bcd(char decnum)	Преобразовывает десятичное число decnum в BCD-число
Dec2Bcdl6		unsigned Dec2Bcdl6 (unsigned decnum)	Преобразовывает десятичное число decnum в BCD-число
FloatToStr		void FloatToStr (float number, char *output)	Создает строку output из вещественного числа number. Результирующая строка представлена в экспоненциальном формате с пятью знаками после запятой
IntToStr		void IntToStr (int number, char *output)	Создает строку output длиной в шесть символов из числа number
LongToStr		void LongToStr (long number, char *output)	Создает строку output длиной в одиннадцать символов из числа number
ShortToStr		void ShortToStr (short number, char *output)	Создает строку output длиной в четыре символа из числа number
WordToStr		void WordToStr (unsigned number, char *output)	Создает строку output длиной в пять символов из числа number
xtoi	stdlib.h	int xtoi(char *s)	Преобразовывает строку s, состоящую из шестнадцатеричных символов в соответствующее ей число

Таблица А.5 – Функции компилятора для работы с периферией

Функция	Заголовок	Описание
Adc_Read	unsigned Adc_Read (unsigned short ch)	Возвращает 10-разрядное значение, считанное из заданного параметром ch канала АЦП
Pwm_Change_Duty	void Pwm_Change_Duty (unsigned short ratio)	Устанавливает коэффициент заполнения для ШИМ. Значению ratio = 0 соответствует коэффициент 0%, ratio = 127 – 50%, ratio = 255 – 100%
Pwm_Init	void Pwm_Init (unsigned long freq)	Инициализирует модуль ШИМ с коэффициентом заполнения 0. Параметр freq задает желаемую частоту ШИМ в герцах

Продолжение таблицы А.5

Функция	Заголовок	Описание
Pwm_Start	void Pwm_Start(void)	Активизирует ШИМ
Pwm_Stop	void Pwm_Stop(void)	Останавливает ШИМ
Usart_Data_Ready	unsigned short Usart_Data_Ready(void)	Возвращает 1, если данные в буфере приемника USART готовы. В противном случае возвращает 0
Usart_Init	void Usart_Init(const unsigned long baud)	Инициализирует модуль USART, устанавливая скорость передачи данных baud (от 100 до 115200)
Usart_Read	unsigned short Usart_Read(void)	Возвращает принятый через USART байт. Если байт не принят, возвращает 0
Usart_Write	void Usart_Write(unsigned short data)	Передает через USART байт data

Таблица А.6 – Функции компилятора для работы с памятью EEPROM и Flash

Функция	Заголовок	Описание
Eeprom_Read	unsigned short Eeprom_Read (unsigned int addr)	Считывает байт по адресу addr памяти EEPROM
Eeprom_Write	void Eeprom_Write(unsigned int addr, unsigned short data)	Записывает байт data по адресу addr памяти EEPROM
Flash_Erase	void Flash_Erase (unsigned addr)	Очищает блок Flash-памяти, начиная с адреса addr
Flash_Erase_64	void Flash_Erase_64 (long addr)	Очищает блок в 64 байта Flash-памяти, начиная с адреса addr
Flash_Erase_1024	void Flash_Erase_1024 (long addr)	Очищает блок в 1024 байта Flash-памяти, начиная с адреса addr
Flash_Read	unsigned Flash_Read (unsigned addr)	Возвращает байт, хранимый во Flash-памяти по адресу addr
Flash_Read_N_Bytes	unsigned Flash_Pead_N_Bytes (long addr, char *data, unsigned int N)	Записывает N байт из Flash-памяти, начиная с адреса addr, в буфер data
Flash_Write	void Flash_Write(unsigned addr, unsigned int *data)	Записывает содержимое буфера data по адресу addr Flash-памяти
Flash_Write_8	void Flash_Write_8 (long addr, char *data)	Записывает 8 байт data по адресу addr Flash-памяти
Flash_Write_16	void Flash_Write_16 (long addr, char *data)	Записывает 16 байт data по адресу addr Flash-памяти
Flash_Write_32	void Flash_Write_32 (long addr, char *data)	Записывает 32 байта data по адресу addr Flash-памяти
Flash_Write_64	void Flash_Write_64 (long addr, char *data)	Записывает 64 байта data по адресу addr Flash-памяти

Таблица А.7 – Функции компилятора для работы с интерфейсом I^2C

Функция	Заголовок	Описание
I2C_Init	void I2C_Init(unsigned long clock)	Инициализирует интерфейс I^2C частотой тактирования clock
I2C_Is_Idle	unsigned short I2C_Is_Idle(void)	Возвращает 1, если шина I^2C свободна. В противном случае возвращает 0
I2C_Rd	unsigned short I2C_Rd (unsigned short ack)	Возвращает байт данных, принятый от ведомого устройства. Если параметр ack = Q, то бит квитирования не выдается
I2C_Repeated_Start	void I2C_Repeated_Start(void)	Создает повторяющееся условие начала передачи
I2C_Start	unsigned short I2C_Start(void)	Проверяет, свободна ли шина I^2C, и создает условие начала передачи. Если ошибок не обнаружено, функция возвращает 0
I2C_Stop	void I2C_Stop(void)	Создает условие завершения передачи
I2C_Wr	unsigned short I2C_Wr (unsigned short data)	Выдает в шину I^2C байт data. Если ошибок не обнаружено, возвращает 0

Таблица А.8 – Функции компилятора для работы с интерфейсом SPI

Функция	Заголовок	Описание
Spi_Init	void Spi_Init(void)	Конфигурирует и инициализирует интерфейс SPI с параметрами по умолчанию
Spi_Init_Advanced	void Spi_Init_Advanced (unsigned short m_s, unsigned short data_sample, unsigned short clock_idle, unsigned short tx_edge)	Конфигурирует и инициализирует интерфейс SPI. Параметр m_s определяет режим работы: - MASTER_OSC_DIV4; - MASTER_OSC_DIV16; - MASTER_OSC_DIV64; - MASTER_TMR2; - SLAVE_SS_ENABLE; - SLAVE_SS_DIS. Параметр data_sample задает момент опроса бит данных: – DATA_SAMPLE_MIDDLE – в середине интервала; – DATA_SAMPLE_END – в конце интервала. Параметр clock_idle определяет состояние ожидания для тактового сигнала: – CLK_IDLE_HIGH; – CLK_IDLE_LOW. Параметр tx_edge задает фронт тактового сигнала, по которому передается бит данных: - LOW_2_HIGH – нарастающий; - HIGH_2_LOW – ниспадающий.

Продолжение таблицы А.8

Функция	Заголовок	Описание
Spi_Read	unsigned short Spi_Read(unsigned short buffer)	Возвращает байт, принятый по интерфейсу SPI
Spi_Write	void Spi_Write (unsigned short data)	Записывает байт в буфер интерфейса SPI и сразу же начинает передачу