ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет

С.В. ДВОЙНИШНИКОВ
К.Ф. ЛЫСАКОВ

ОСНОВЫ ПРОГРАММИРОВАНИЯ
(ЯЗЫК С)

Учебное пособие

Новосибирск
2022

Двойнишников С.В., Лысаков К.Ф. Основы программирования (язык С): Учебное пособие, 2-е изд., переработанное и дополненное / Новосиб. гос. ун-т. Новосибирск, 2022. 154с.

Пособие предназначено для людей впервые приступивших к изучению программирования, как основной источник теоретического и практического материала. Предлагается базовый фундаментальный подход по изучения основ практического программирования в областях физики и математики с использованием языка С.

.

Рецензент
М.Ю. Шадрин

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Программирование — это процесс создания (разработки) программы, который может быть представлен последовательностью следующих шагов:

1. Спецификация (определение, формулирование требований к программе).
2. Разработка алгоритма.
3. Кодирование (запись алгоритма на языке программирования).
4. Отладка.
5. Тестирование.

В большинстве своем различные самоучители и другие печатные издания во главу угла ставят изучение какого-либо языка программирования, на примере которого решают задачи. При этом для людей, впервые встречающихся с программированием, наибольшую сложность представляют именно разработка самого алгоритма решения задачи.

В качестве языка программирования в пособии рассмотрен язык Си.

1.1. Описание алгоритмов решения

Для решения любой задачи необходимо выполнить следующие этапы:

1. Четко определить условия задачи, входные данные и какой результат должен быть получен после решения задачи.
2. Какие дополнительные данные необходимы для решения задачи.
3. Составить блок-схему решения задачи и записать ее в виде удобного описания.
4. Анализ всех возможных проблем и усовершенствование алгоритма.

Рассмотрим, каким образом выполняются перечисленные этапы на примере вычисления корней квадратного уравнения.

Этап первый: постановка задачи

Для определенности будем решать уравнение следующего вида:

$$ax^2 + bx + c = 0$$

Входными данными для нашей задачи являются три коэффициента: a, b и c.

Решение задачи предполагает вычисление возможных корней уравнения. При этом, так как количество корней может быть от 0 до 2, то необходимо в качестве решения указать количество корней и собственно перечислить их.

Этап второй: дополнительные данные

При решении квадратного уравнения необходимо вычислить значение дискриминанта. В нашей задаче дискриминант является промежуточным результатом необходимым для решения задачи.

Этап третий: составление блок-схемы решения

Описание блок-схемы может производиться с помощью различных средств и обозначений. Основной принцип заключается в наглядности шагов исполнения и однозначности переходов при ветвлении.

На рисунке (рис. 1) приведена основная блок-схема, к которой чаще всего приходят студенты при решении квадратного уравнения.

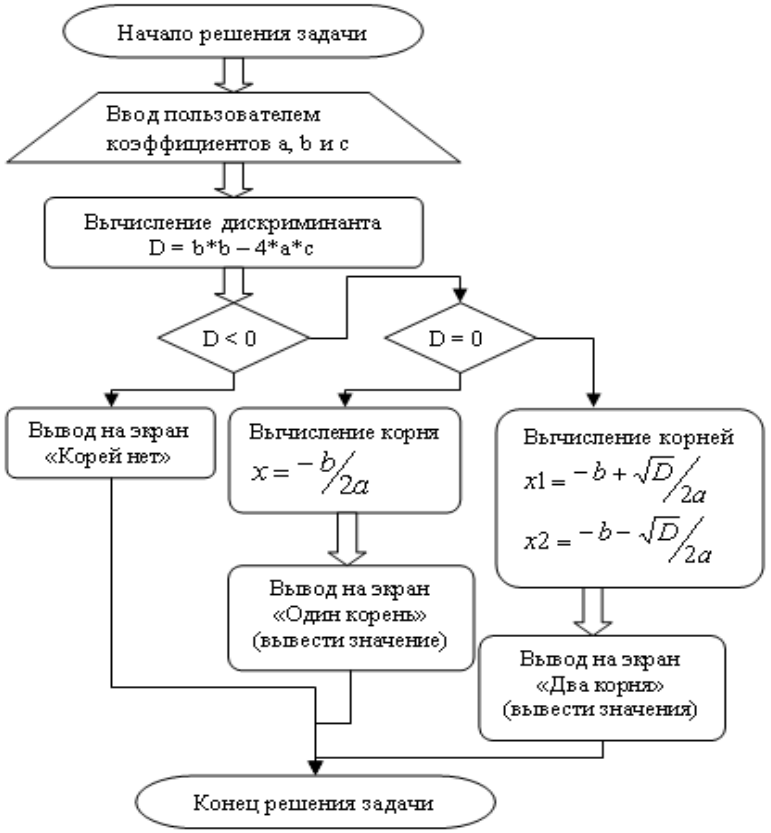


Рис. 1. Блок-схема решения квадратного уравнения

Этап четвертый: корректность и усовершенствования

Данный этап предполагает анализ созданной блок-схемы решения задачи. При этом необходимо ответить на два вопроса:

1. Будет ли схема корректно работать во всем диапазоне входных параметров?
2. Возможны ли оптимизации, которые позволят ускорить процесс выполнения задачи?

При анализе совершаемых действий становится очевидным, что при значении коэффициента **a = 0**, происходит деление на **0!** Данный случай необходимо дополнительно обработать в программе. В качестве решения можно предложить два основных метода:

- Добавить проверку корректности введенных пользователем значений и, при вводе **a=0**, выдавать сообщение об ошибке: «Уравнение не является квадратным!».
- Допустить возможность решения линейных уравнений, по сути, расширив диапазон применения вашей реализации. Для этого необходимо добавить такую проверку до вычисления дискриминанта и идти описанным путем лишь при коэффициенте **a** отличном от 0, иначе добавить еще одну ветвь исполнения.

Что касается оптимизации, то основной ее смысл в удалении лишних, а также в исключении дублирующих действий. На приведенной выше схеме дублированным действием является извлечение квадратного корня из дискриминанта.

1.2. Отладка программной реализации

После создания программной реализации, нередки случаи, когда ее функционирование отличается от запланированного. Для выяснения причин некорректного поведения существуют два основных метода отладки:

- метод отладочной печати,
- метод пошаговой отладки.

1.2.1. Метод отладочной печати

Одним из основных средств отладки является отладочная печать, позволяющая получить данные о ходе и состоянии процесса вычислений. Обычно разрабатываются специальные отладочные методы, вызываемые в

критических точках программы: на входе и выходе программных модулей, на входе и выходе циклов и так далее. Искусство отладки состоит в том, чтобы получить нужную информацию о прячущихся ошибках, проявляющихся, возможно, только в редких ситуациях.

Иногда пошаговая отладка невозможна. Например, программа может быть связана с внешним процессом, который не будет ждать, пока разработчик проверяет значения переменных. Другой весьма вероятный вариант — программа скомпилирована без отладочной информации с оптимизацией. В таких случаях единственным способом узнать, что происходит во время исполнения, остаётся вывод сообщений, по которым можно судить о состоянии программы.

Суть метода заключается в том, что программист вставляет в код программы вывод определенных сообщений, по которым впоследствии можно проанализировать какая именно ветвь программы исполняется, какие получаются промежуточные результаты и сравнить это с предполагаемым ходом выполнения и вычислениями.

1.2.2. Метод пошаговой отладки

Метод пошаговой отладки заключается в том, что программист заставляет компьютер выполнять программу по шагам, инструкцию за инструкцией, а сам следит за состоянием программы на каждом шаге. При поиске ошибки очень удобно видеть, какая строка программы сейчас выполняется, и выполнять программу строка за строкой в соответствии с исходным кодом.

Команды управления отладкой, как правило. собраны в меню Debug. Для удобства разработчика на самые нужные команды назначены клавиши (для Microsoft Visual Studio режим отладки запускается кнопкой F5).

Программа запускается в отладочном режиме командой ***Debug***. При этом за работой программы следит отладчик, переводящий исполнение в

пошаговый режим в двух случаях: при возникновении ошибки, которая в обычном режиме привела бы к аварийному завершению программы, или по достижении точки останова.

Точка останова (англ. ***Breakpoint***), может быть установлена в любой строке программы, содержащей выполняемую инструкцию. Это может быть начало цикла, вызов функции, выражение или даже фигурная скобка, закрывающая блок. При стандартных настройках точка останова устанавливается или снимается нажатием клавиши F9.

После того как отладчик перевёл исполнение в пошаговый режим, можно заняться непосредственно отладкой.

Для того чтобы увидеть значение переменной существующей в текущем контексте достаточно навести указатель мыши на её идентификатор в исходном тексте. Этой же цели служит окно ***Watch***: впишите в левую колонку интересующие вас идентификаторы и их значения будут показаны всё время, пока идентификаторы имеют смысл. Дополнительно можно использовать окна ***Locals*** и ***Autos.*** В них отображаются текущие переменные контекста и локальные переменные.

Команды пошагового исполнения позволят вам проконтролировать работу интересующего вас фрагмента кода:

- ***Step Over*** (F10) выполняет очередную строку и останавливает программу на следующей строке. Если текущая строка содержит вызов функции, то данная команда выполнит этот вызов в обычном режиме, не показывая по шагам подробности внутренней работы вызываемой функции.
- ***Step Into*** (F11) заставляет отладчик войти в функцию, вызов которой находится в очередной строке. Когда интересующий вас фрагмент пройден, можно выйти из пошагового режима командой ***Debug*** (***F5***). Исполнение программы продолжится до следующей точки останова или критической ошибки.

- ***Step Out*** (Shift+F11) заставляет отладчик выйти из текущей функции и остановиться в месте вызова текущей функции на один уровень в стеке выше.

Для выхода из режима отладчика необходимо вызвать команду ***Stop Debugging*** (Shift+F5).

При работе в режиме пошаговой отладки также будут доступны другие окна с отладочной информацией:

- ***Call Stack*** – отображает последовательность вызванных функций (стек вызовов) в программе. К тексту любой функции можно переместиться, сделав двойной щелчок на её имени.
- ***Threads –*** диалог показывает список потоков, созданных вашей программой.
- ***Modules –*** показывает список загруженных модулей. Для каждого модуля выводится диапазон адресов и имя файла

1.3. Среда разработки Microsoft Visual Studio

MSVS (Microsoft Visual Studio) версии 2019 является одной из распространенных систем разработки среди профессиональных разработчиков.

1.3.1. Общие сведения

Для того, чтобы писать программу, вам *необходимы* проект (*project*) и рабочее пространство (*solution*).

Проект — это специальный файл, имя которого имеет расширение *vcproj(vcxproj)*. В проекте содержится информация о том, из каких исходных файлов строится программа. Всё, что перечислено в проекте, будет скомпоновано в один целевой модуль (обычно исполняемый файл — **.exe*, но есть и другие варианты); если вам нужно получить два исполняемых файла, придётся сделать несколько проектов.

Среда разработки следит за изменениями файлов, перечисленных в файле проекта и, при сборке проекта, компилирует заново только файлы, изменённые после предыдущей компиляции.

Компилятор обрабатывает только те файлы, которые перечислены в файле проекта. При этом совершенно всё равно, как называется папка в проекте — *Source files* или *Include files*: инструмент для обработки файла выбирается, исходя из расширения имени файла, а папки вы можете создавать для собственного удобства в любом количестве.

Рабочее пространство — это специальный файл, имеющий расширение *sln*, который описывает зависимости между отдельными проектами, входящими в рабочее пространство: разрабатывая большую программу, состоящую из ряда модулей, удобно собрать несколько проектов в одно пространство и получать последнюю версию всех модулей нажатием одной клавиши!

Visual Studio устроена так, что проект обязательно должен содержаться внутри рабочего пространства. Если у вас уже есть рабочее пространство, то открывать в *Visual Studio* нужно его, а не проект или файл с исходным текстом!

Файл рабочего пространства создаётся автоматически, когда вы создаёт новый проект.

1.3.2. Создание проекта

Запуск среды можно осуществить из меню как на рисунке ниже

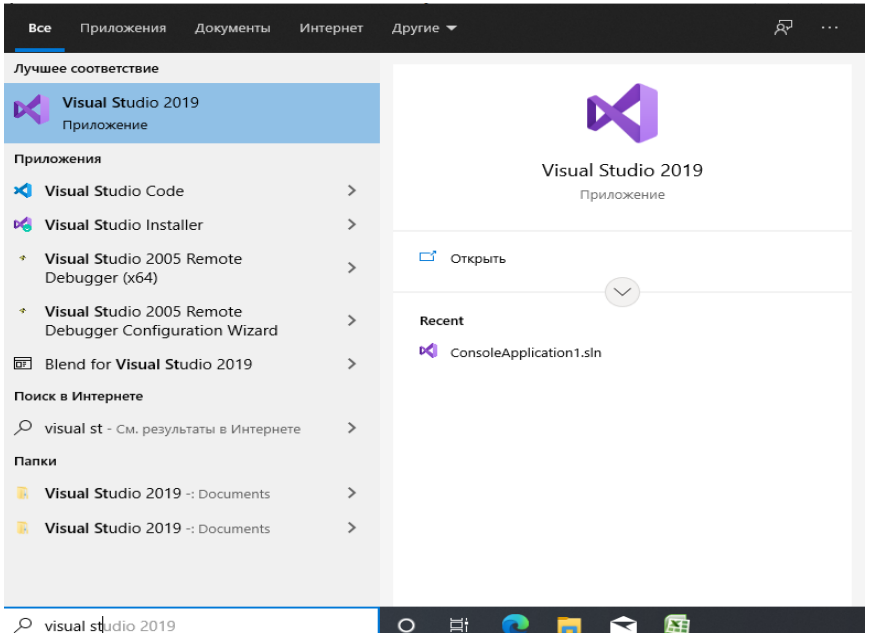


Рис. 2. Запуск среды разработки

Для начала работы необходимо создать проект, выбрав в меню.

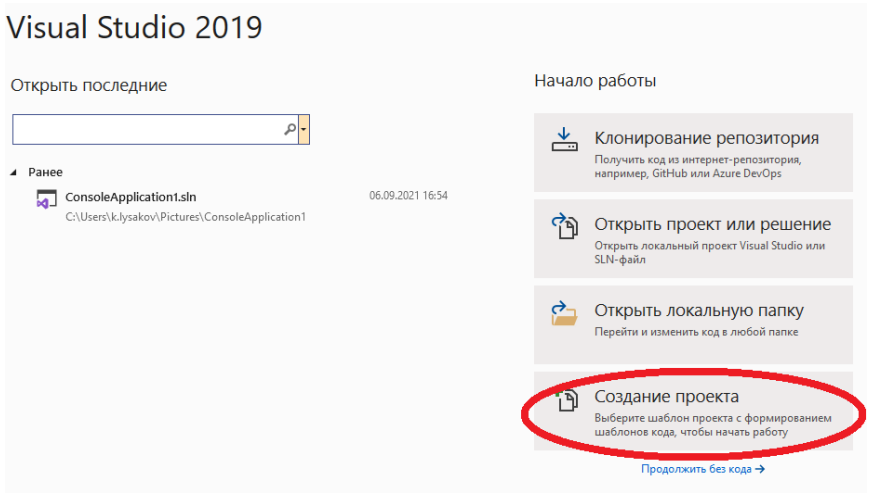


Рис. 3. Создание нового проекта

Далее необходимо выбрать необходимый шаблон. Для большинства учебных заданий идеальным стартом является пустой проект *Empty project*.

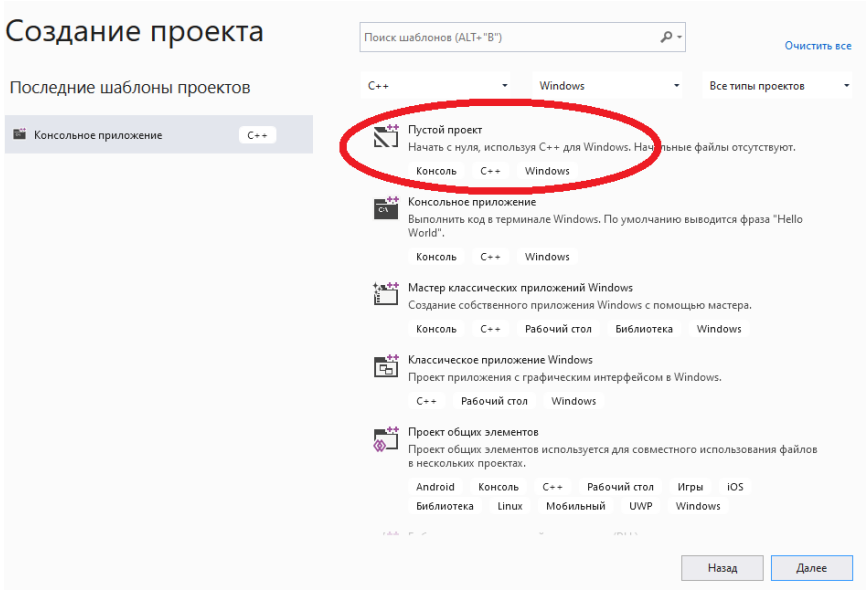


Рис. 4. Выбор пустого проекта

Новому проекту нужно дать имя. Правила для имён проектов такие же, как для имён файлов, но с русскими буквами или пробелами могут возникнуть проблемы. На данном этапе происходит задание имени проекта, его расположения в файловой системе и имя «решения», в которое в будущем можно добавлять другие проекты.

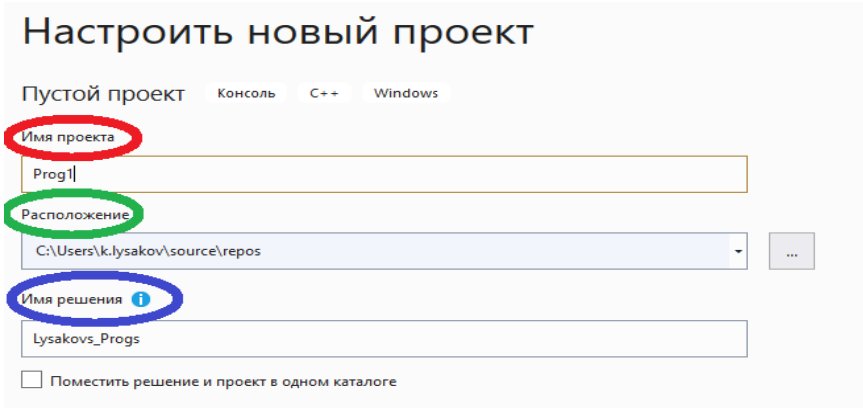


Рис. 5. Задание имен и расположения.

После создания проекта он не содержит никаких исходных текстов; для того чтобы начать программировать, нужно добавить в проект файл исходного кода.

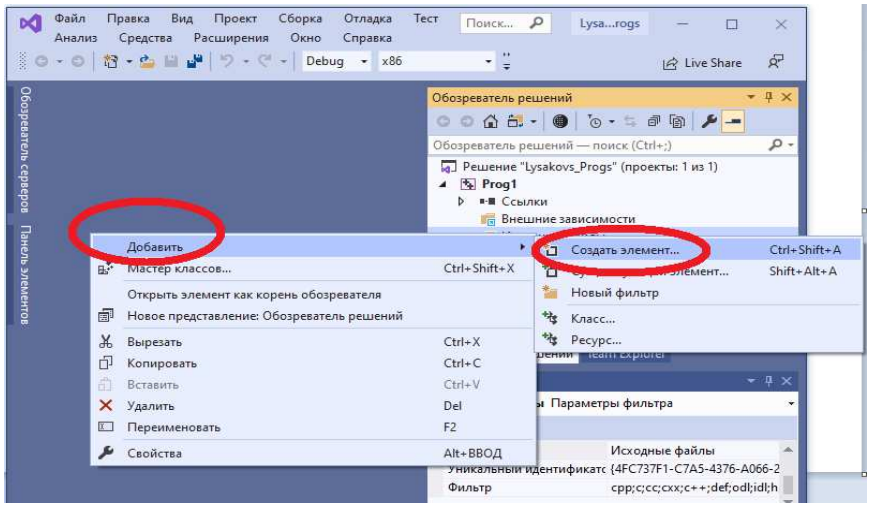


Рис. 6. Добавление файлов в проект

Для добавление файла, содержащего текст программы на языке C/C++ необходимо выбрать в типе файла (левом окне) тип «Код», а в правом тип файла C++. Здесь важно отметить: среда разработки не имеет отдельного

типа файла для разработке на языке C, поэтому при выборе типа C++ необходимо в расширении файла явно указать что программа будет на языке C, установив соответствующее расширение файла. В таком случае будет использован компилятор языка C для работы с вашей программой.

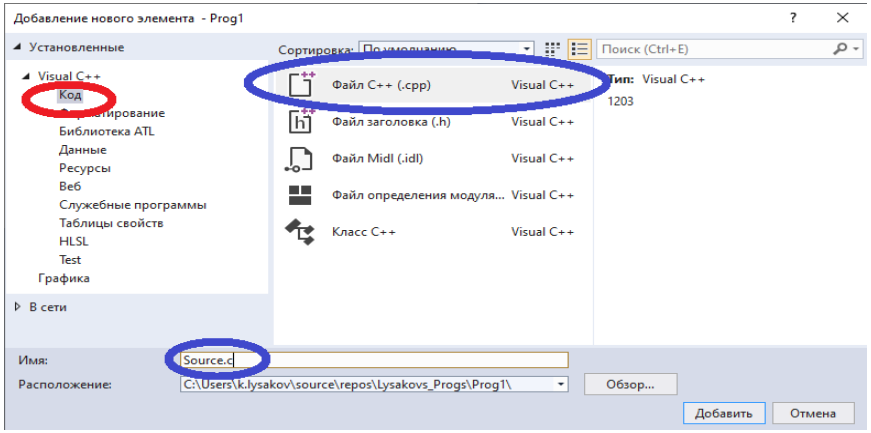


Рис. 7. Указание типа файла для программного кода

После этого в левой части экрана откроется редактор файла, где можно писать программный код.

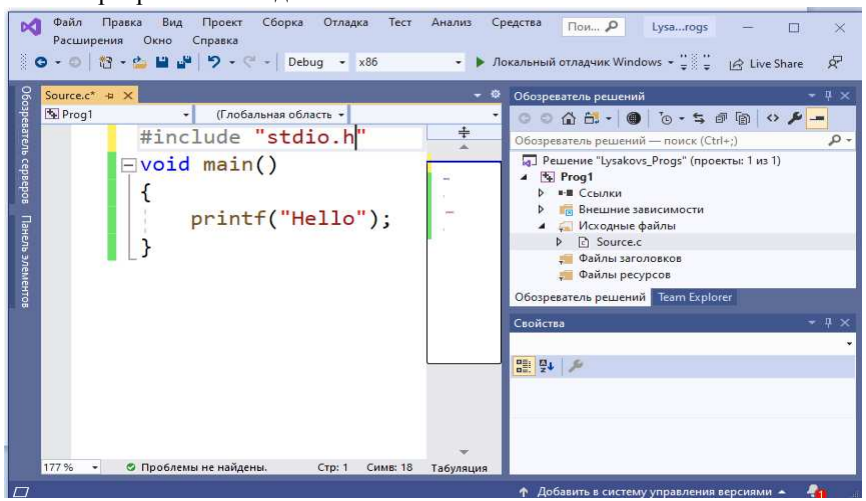


Рис. 8. Вид среды разработки

1.3.3. Сборка проекта

После того, как программист написал некоторое количество исходного кода, у него обычно возникает желание посмотреть, как этот код работает. Для этого нужно скомпилировать и скомпоновать программу.

Компилятор и компоновщик запускаются из меню *Build->Build Solution* (или нажатием соответствующей «горячей клавиши»). Кроме того, можно скомпилировать только тот файл, который вы в данный момент редактируете (*Build->Compile*), однако в этом случае запустить исполнение кода не получится: для этого необходимо скомпилировать и скомпоновать весь проект.

Если всё получилось (в коде нет ошибок), то результатом работы компилятора и компоновщика будет исполняемый файл (с расширением *exe*), который можно будет запустить. Запускать можно прямо из среды разработки как показано на рисунке ниже.

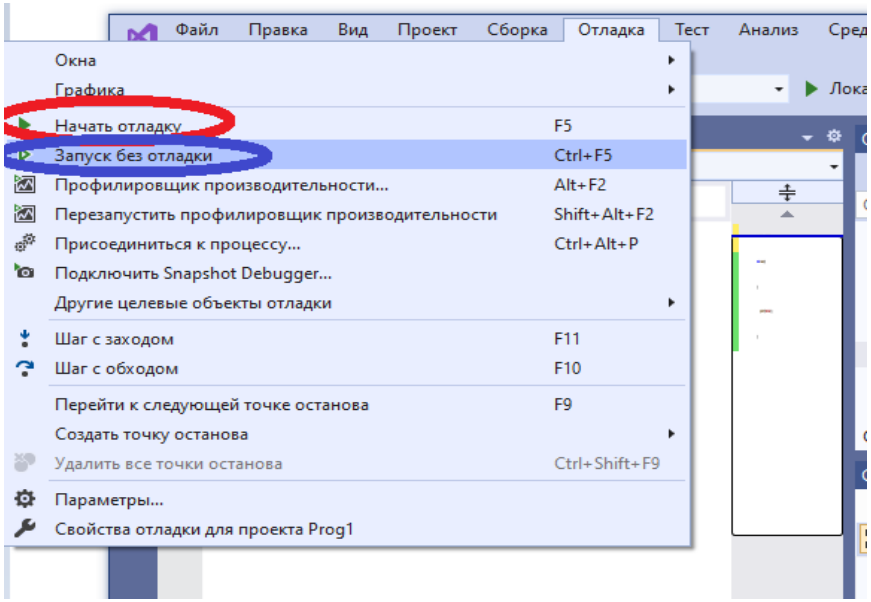


Рис. 9. Запуск программы

В окне «Вывод» (снизу слева) показывается статус сборки проекта. После запуска проекта будет открыта консоль в которой будет происходить общение пользователя с программой.

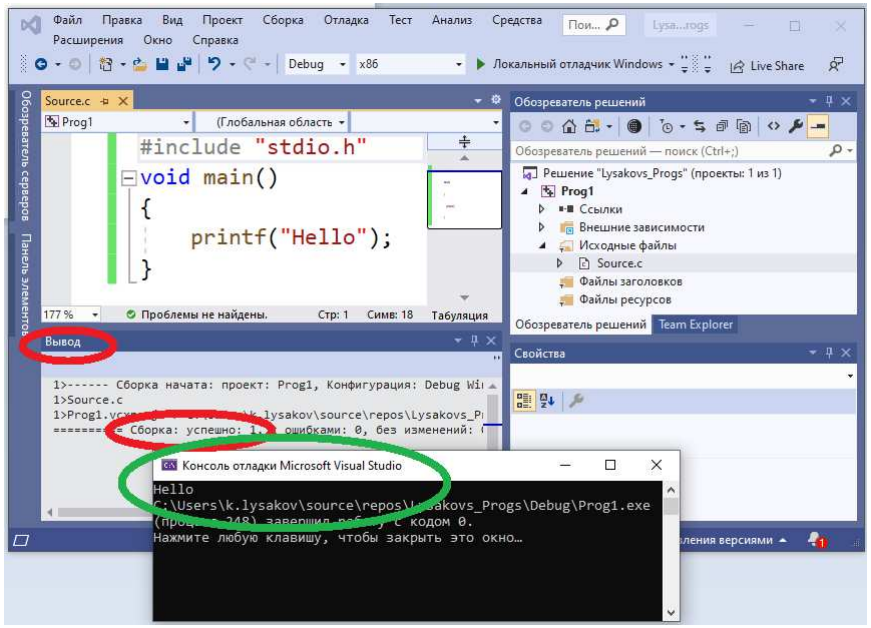


Рис. 10. Запуск программы и консоль

1.3.4. Отладка встроенным отладчиком

Встроенный отладчик необходим тогда, когда программа работает не так как, как предполагал программист. Для отладки может использоваться метод пошагового исполнения. Однако если интересующее место программы находится не в начале, то дойти до него пошагово бывает неудобно, поэтому применяется метод установки специальной метки на строке, дойдя до выполнения которой программа останавливается и управление передается программисту.

Среда разработки MSVS позволяет поставить «точку останова» (breakpoint), которая отмечена красным кружком слева кода. Ее постановка

производится через меню «Поставить точку останова», либо горячей клавишей F9, либо кликнув по колонке слева от строки, где отладчик должен остановить выполнение программы.

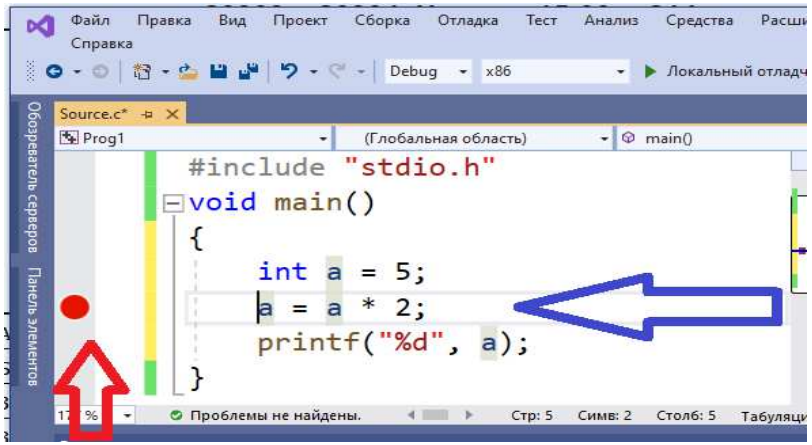


Рис. 11. Точка останова для отладки программ

Для запуска отладчика можно воспользоваться меню Отладка -> Начать отладку, либо нажать горячую клавишу F5.

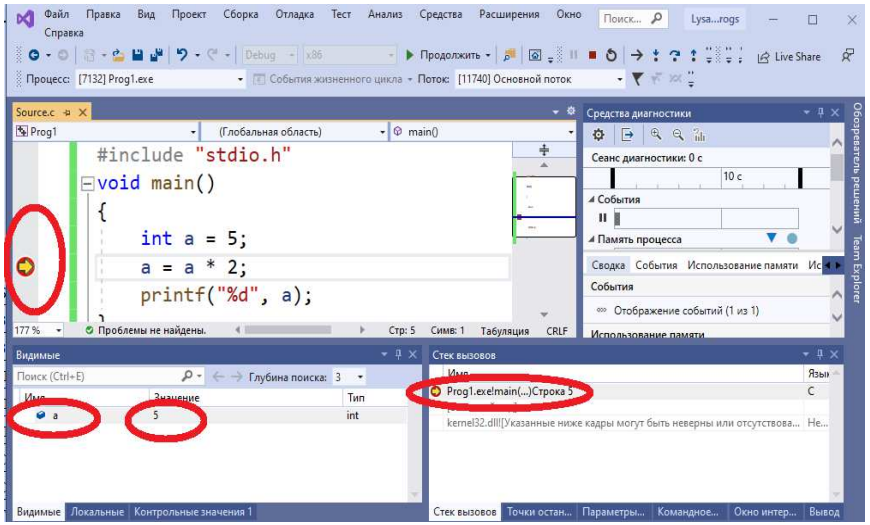


Рис. 12. Вид при пошаговой отладке

После запуска программы, дойдя до точки останова, управление передается программисту. В среде разработки открываются окна видимости переменный и их значений, а слева окна кода появляется стрелка, показывающая на какой строке остановлено исполнение программы.

Обратите внимание, что на приведенном выше скриншоте значение переменной a равно 5. Другими словами, желтая стрелка исполнения показывает какая строка будет исполнена, но ее действие еще не произошло.

Для дальнейшего выполнения программы можно нажать горячую клавишу F5. В этом случае программа будет исполняться до следующей «точки останова», либо до завершения программы. При этом, вы можете в любой момент исполнения программы прервать ее выполнение и отладчик покажет вам текущее место в программе, которое исполнялось в момент прерывания. Для этого нужно нажать клавиши Ctrl+Shift+Break. Кроме того, можно выполнять пошаговую отладку. В этом случае программа будет выполнять один шаг программы и останавливаться. При этом можно переходить на следующий шаг, находясь в текущей функции (горячая клавиша F10), либо заходить внутрь вызываемой функции (горячая клавиша F11), либо выходить из текущей функции на уровень выше (горячая клавиша Shift+F11). Для выхода из режима отладки можно воспользоваться клавишей Shift + F5.

1.4. Среда разработки Visual Studio Code

Visual Studio Code — редактор исходного кода, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки.

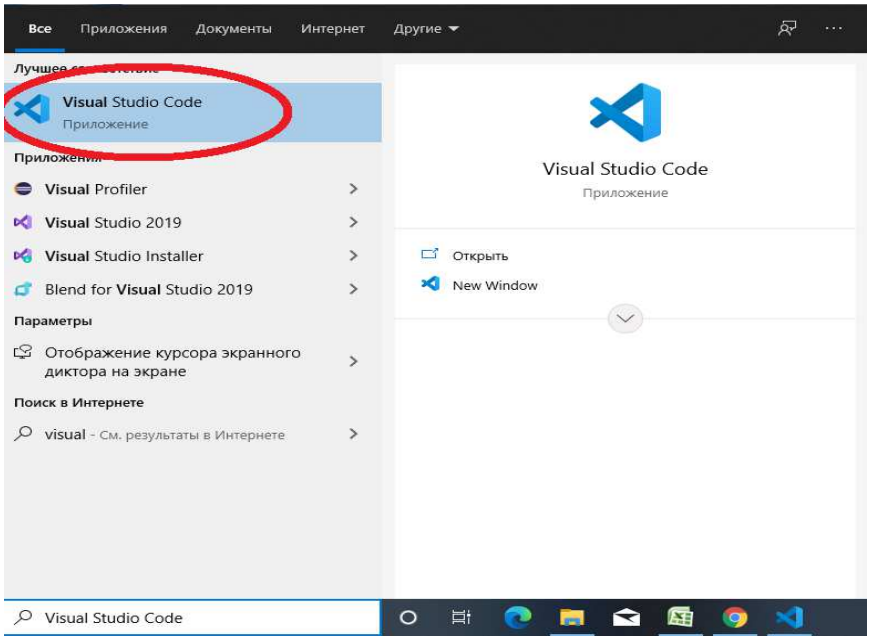


Рис. 13. Запуск Visual Studio Code

1.4.1. Настройка среды

Visual Studio Code - популярный редактор кода, бесплатный и с открытым исходным кодом. Поскольку Visual Studio Code был создан как легкое средство разработки, не требующее много ресурсов компьютера. То перед использованием его необходимо сконфигурировать, установив необходимые компоненты для разработке на нужном языке и запуска проекта:

1. Необходимо добавить нужный язык разработки. Для этого необходимо нажать «Customize» на начальном экране.

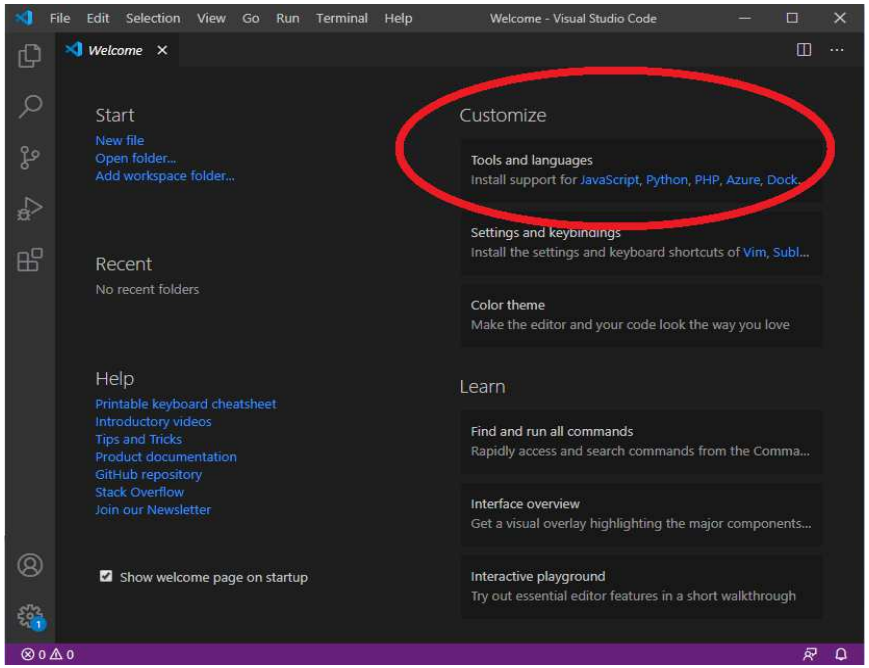


Рис. 14. Меню дополнительных модулей

Далее необходимо найти модуль для поддержки языков C/C++ и установить его, нажав кнопку «Install».

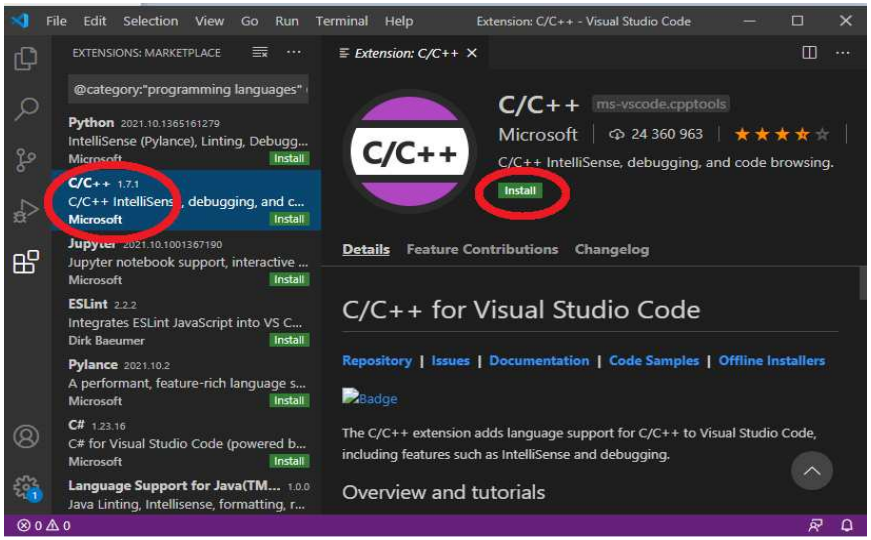


Рис. 15. Установка модуля поддержки языков C/C++

2. Для того, чтобы программа запускалась в среде исполнения необходимо чтобы было установлено расширение Code Runner.

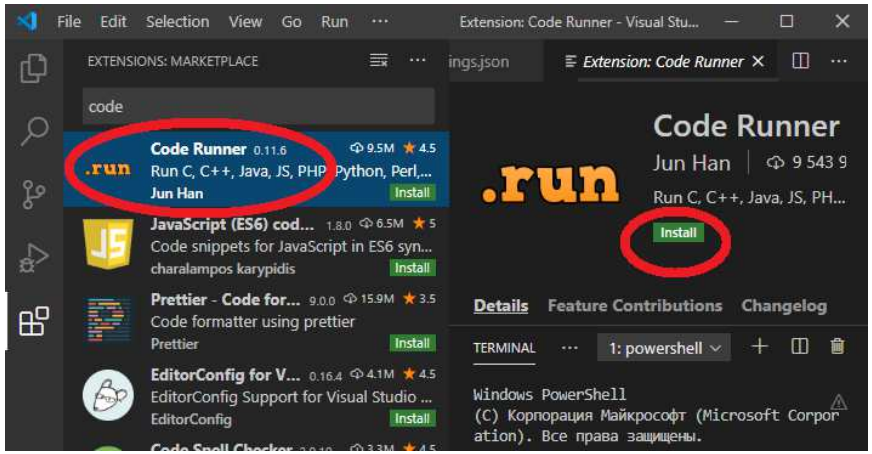


Рис. 16. Установка модуля поддержки языков Code Runner

3. Необходимо войти в настройки среды (*File -> Preferences -> Settings*).

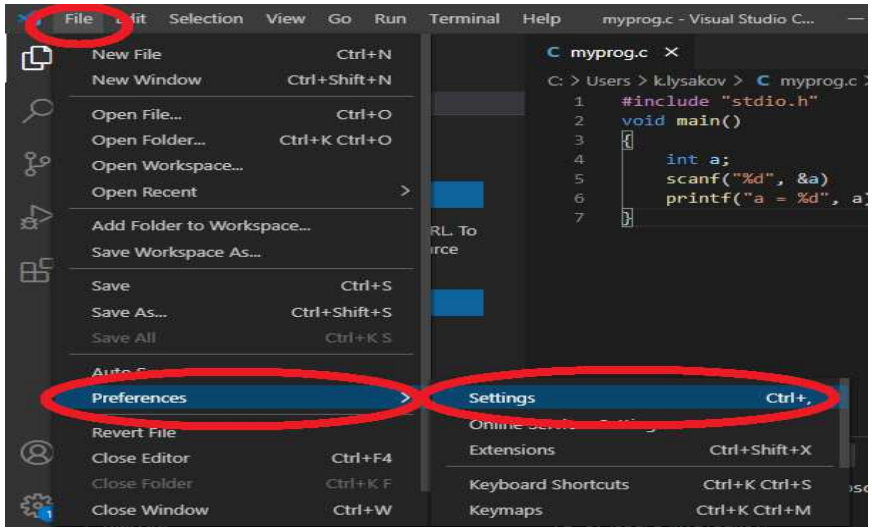


Рис. 17. Настройки среды

Далее ввести в строке поиска «code runner run in terminal» и поставить галочку, которая позволит программе запускаться во встроенном терминале для обеспечения общения программиста с программой.

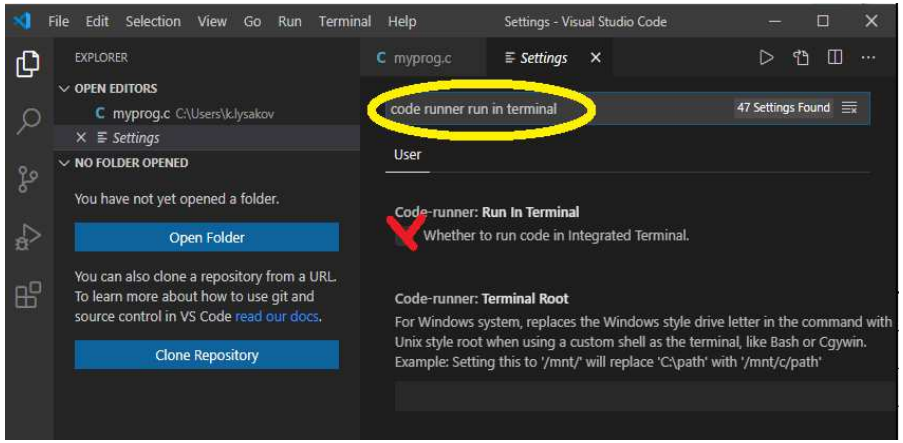


Рис. 18. Галочка запуска во встроенном терминале

1.4.2. Работа в среде

В отличии от среды Microsoft Visual Studio, разработка в среде Visual Studio Code ведется в понятиях файла, а не проекта. Поэтому для начала работы необходимо добавить файл, в котором будет далее писаться код.

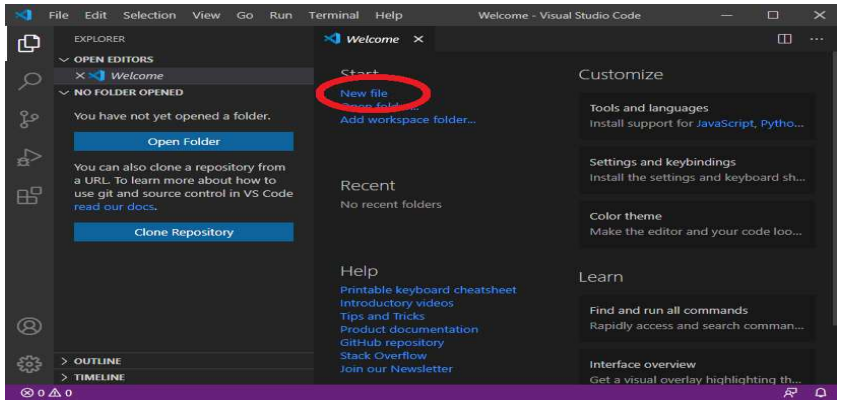


Рис. 19. Добавление файла для написания программного кода

При этом по умолчанию будет создан файл с названием по умолчанию и без расширения. Поэтому далее необходимо сохранить файл как необходимо программисту.

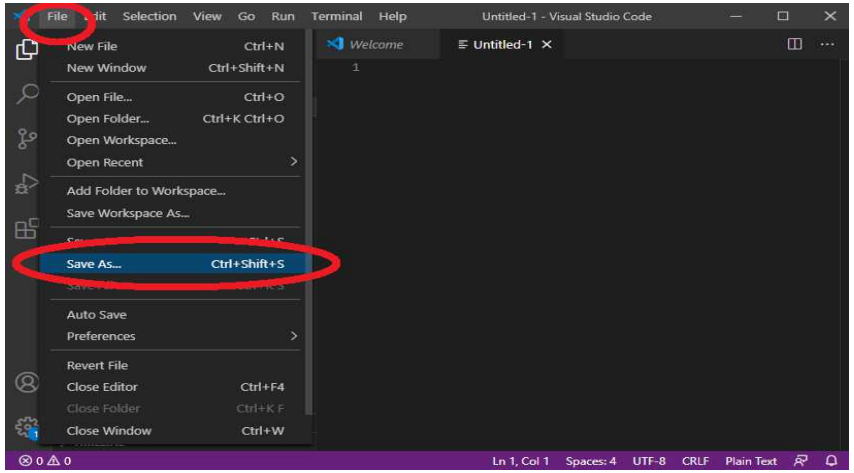


Рис. 20. Сохранение файла

При сохранении необходимо выбрать путь размещения файла с программой а также расширение, соответствующее языку программирования, на котором ведется разработка.

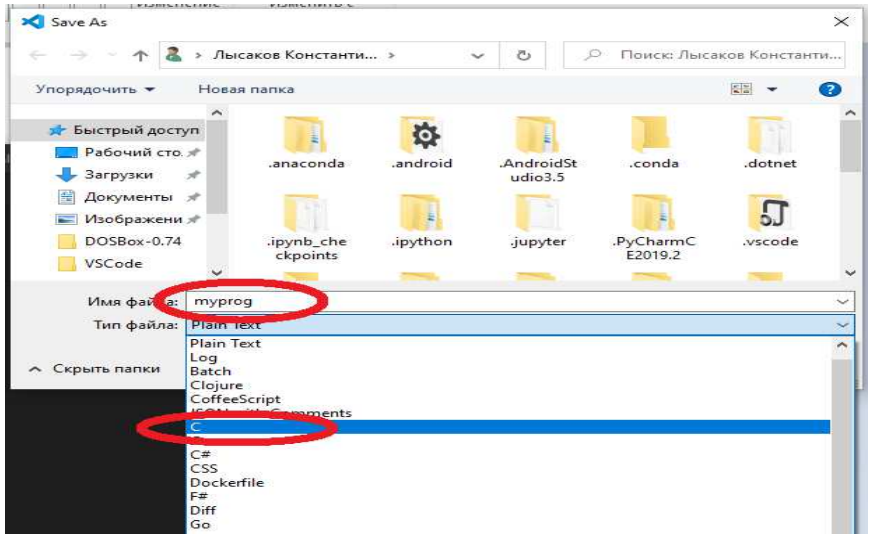


Рис. 21. Указание пути и расширения файла с программой

Далее работа происходит аналогично среде Microsoft Visual Studio.

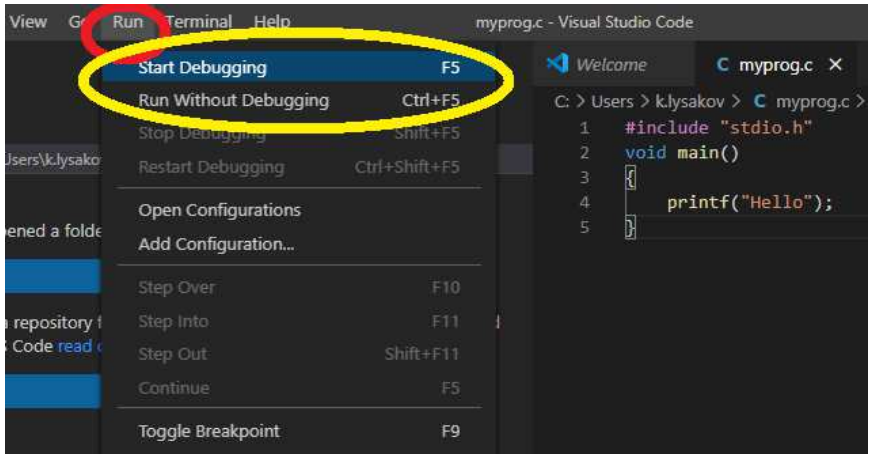


Рис. 22. Запуск проекта

1.5. Среда разработки Code Blocks

Code::Blocks — интегрированная кросс-платформенная среда разработки с отрытым исходным кодом, которая поддерживает использование различных компиляторов, таких как GCC (MingW/GNU GCC), MSVC, Digital Mars, Borland C 5.5 и Open Watcom. По умолчанию Code Blocks использует компилятор MinGW, который поставляется в одном комплекте.

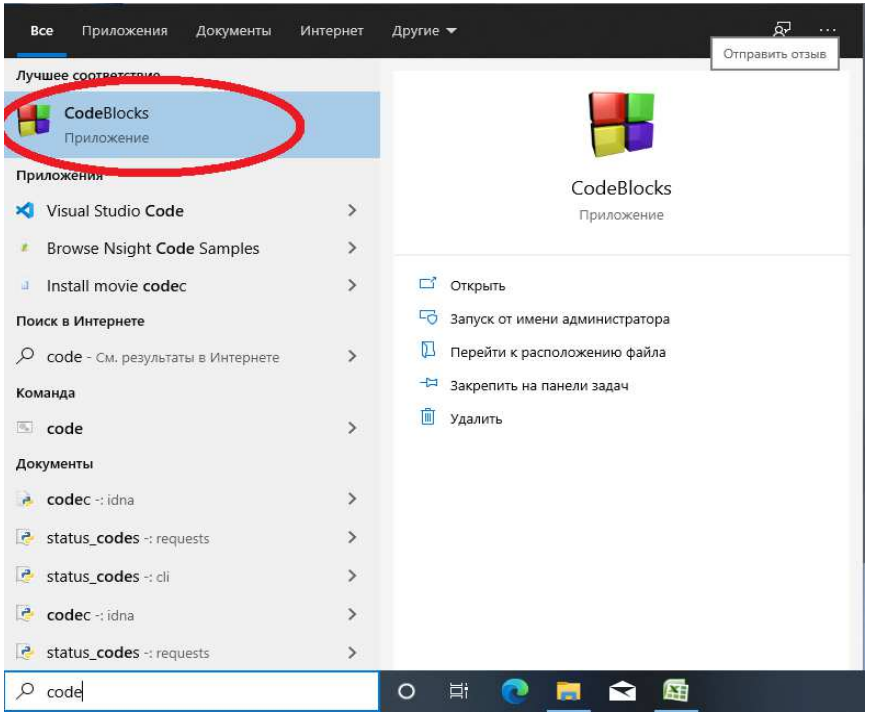


Рис. 22. Запуск среды разработки Code Blocks

1.5.1. Работа в среде

Идеология работы в среде Code::Blocks во многом аналогична среде Microsoft Visual Studio, поскольку разработка ведется в проектах.

Таким образом, для начала работы необходимо создать проект

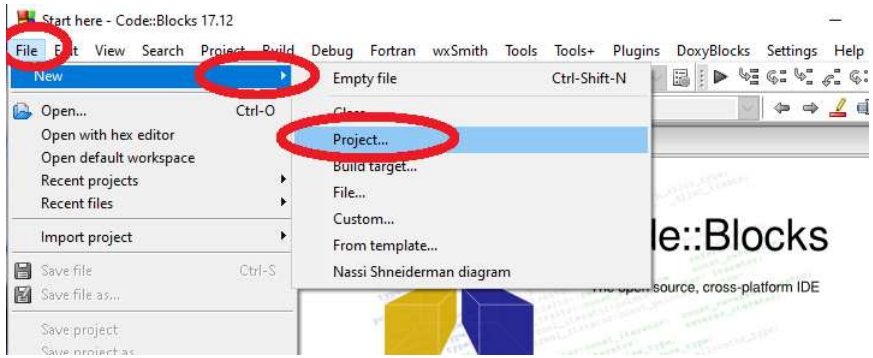


Рис. 23. Добавление нового проекта в среде разработки

В типе проекта и категории необходимо указать «Консольное приложение» и выбрать пустой проект.

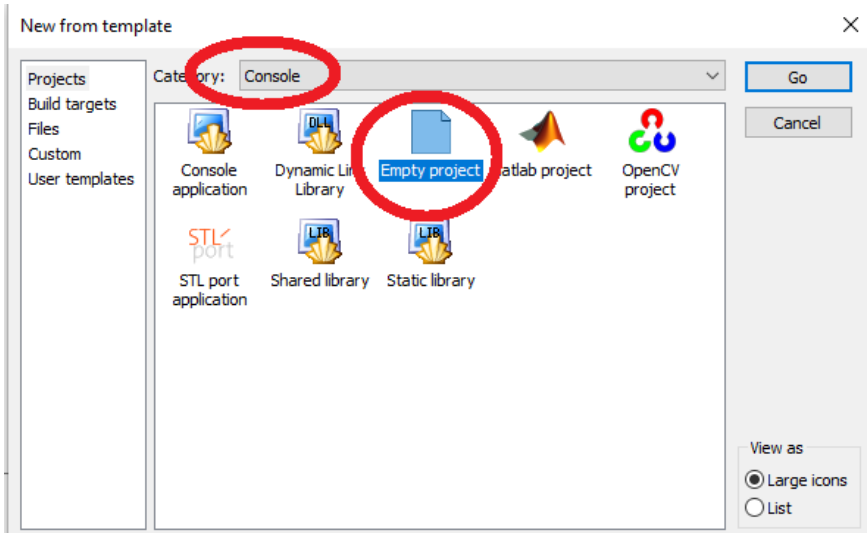


Рис. 24. Выбор типа проекта

Далее потребуется указать имя проекта и его расположение.

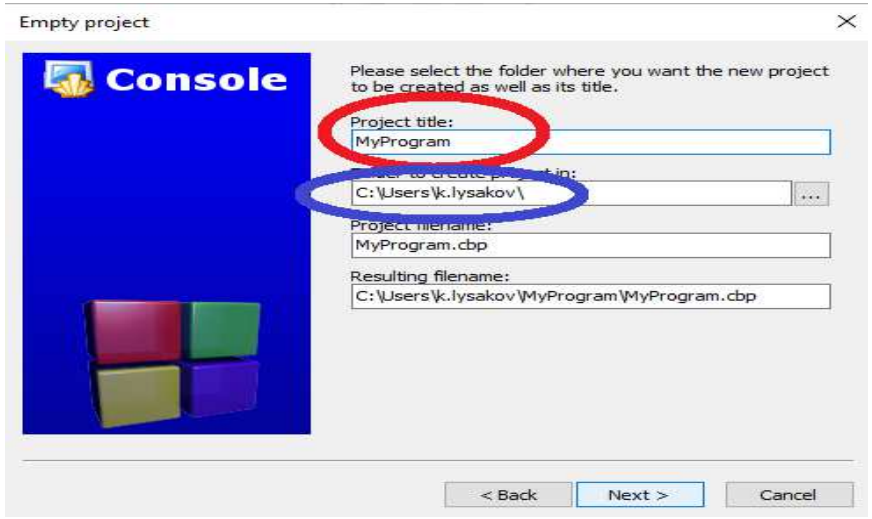


Рис. 25. Указание имени проекта и его расположения

Поскольку изначально среда разработки была нацелена на максимально широкий охват поддерживаемых компиляторов, то при создании проекта необходимо выбрать каким компилятором предпочитает пользоваться программист.

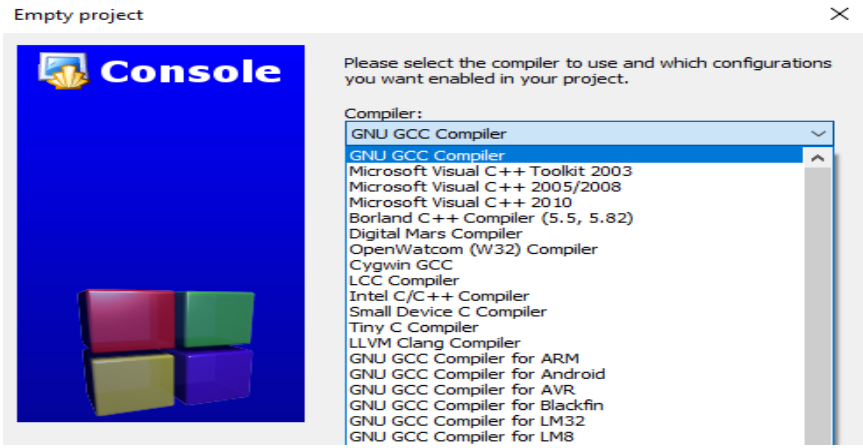


Рис. 26. Выбор компилятора

Далее, поскольку мы создали пустой проект, в него необходимо добавить файл, в котором будет храниться собственно программный код.

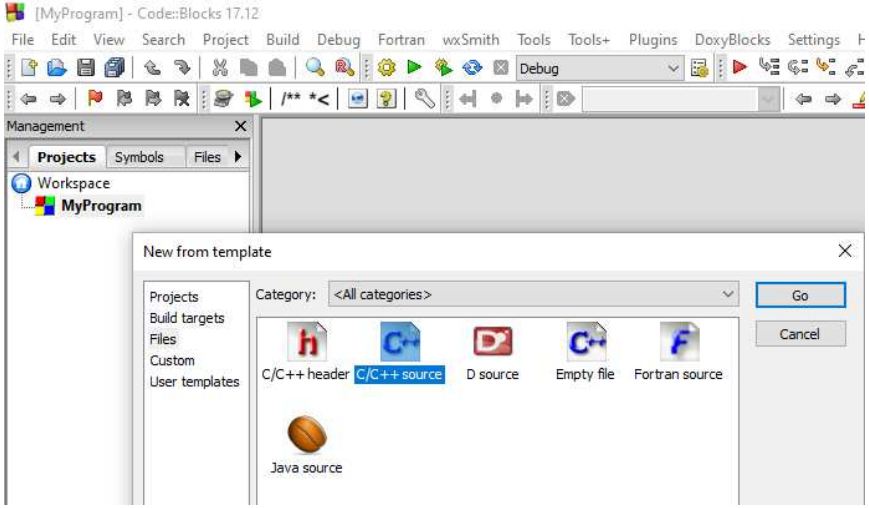


Рис. 27. Выбор типа добавляемого файла

При этом в среде Code::Blocks можно четко указать язык разработки.

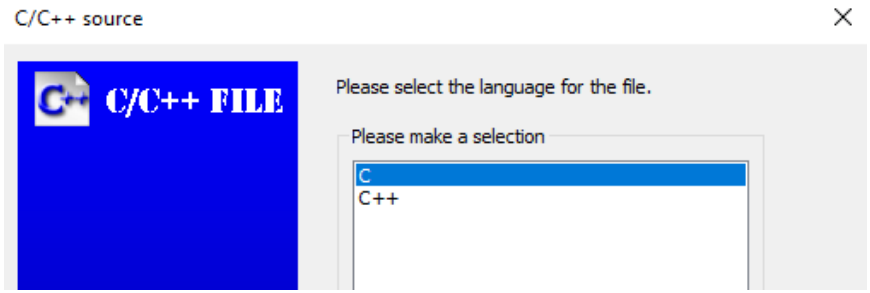


Рис. 28. Выбор языка разработки

На последнем шаге добавления файла необходимо будет указать его имя.

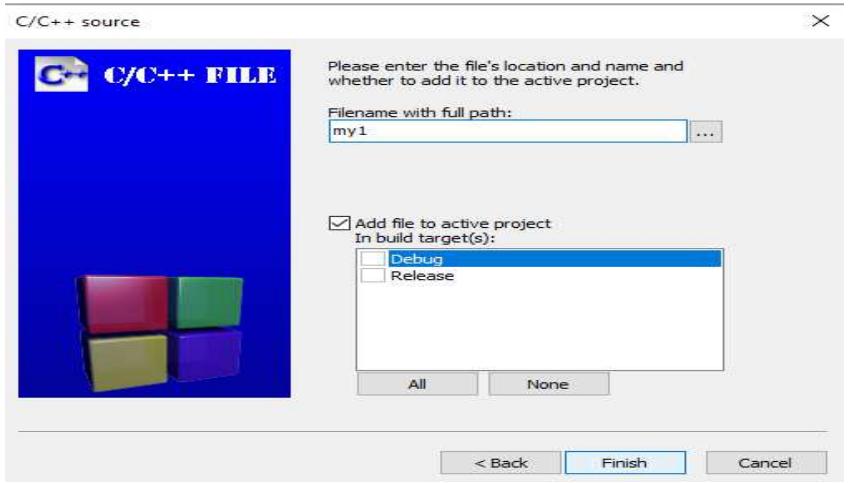


Рис. 29. Указание имени добавляемого файла

Ниже представлен общий вид среды разработки. Для запуска можно нажать горячую клавишу. Исполнение будет происходить в отдельном окне терминала.

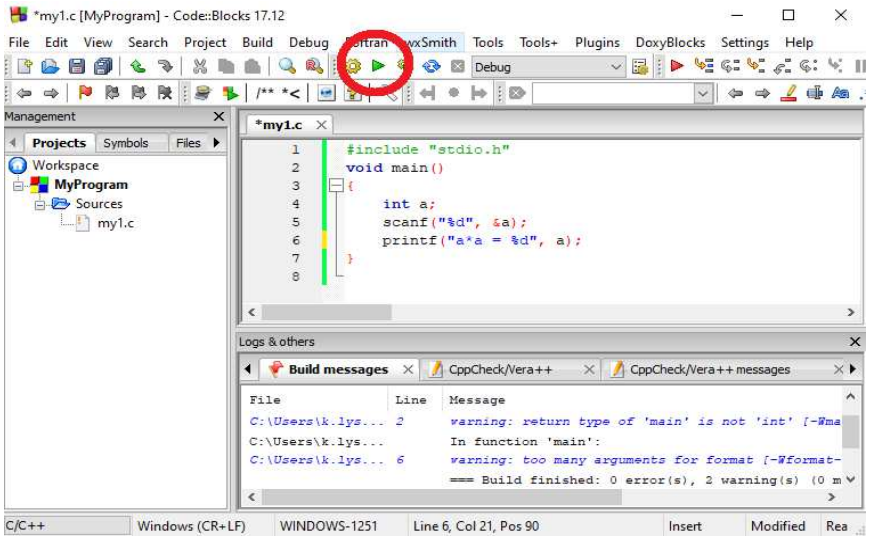


Рис. 30. Вид среды разработки и запуск

1.6. Системы счисления

Система счисления - это совокупность правил и приемов записи чисел с помощью набора цифровых знаков. Количество цифр, необходимых для записи числа в системе, называют основанием системы счисления.

Различают два типа систем счисления:

- позиционные, когда значение каждой цифры числа определяется ее позицией в записи числа;

- непозиционные, когда значение цифры в числе не зависит от ее места в записи числа.

Примером непозиционной системы счисления является римская: числа IX, IV, XV и т.д. Примером позиционной системы счисления является десятичная система, используемая в повседневной жизни.

Десятичная система использует десять цифр – 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9, а также символы “+” и “–” для обозначения знака числа и запятую или точку для разделения целой и дробной частей числа.

В вычислительных машинах используется двоичная система счисления, её основание - число 2. Для записи чисел в этой системе используют только две цифры - 0 и 1. Вопреки распространенному заблуждению, двоичная система счисления была придумана не инженерами-конструкторами ЭВМ, а математиками и философами задолго до появления компьютеров, еще в XVII - XIX веках.

Выбор двоичной системы для применения в вычислительной технике объясняется тем, что электронные элементы - триггеры, из которых состоят микросхемы ЭВМ, могут находиться только в двух рабочих состояниях. С помощью двоичной системы кодирования можно зафиксировать любые данные и знания. Это легко понять, если вспомнить принцип кодирования и передачи информации с помощью азбуки Морзе. Телеграфист, используя только два символа этой азбуки - точки и тире, может передать практически любой текст.

Двоичная система удобна для компьютера, но неудобна для человека: числа получаются длинными и их трудно записывать и запоминать. Конечно, можно перевести число в десятичную систему и записывать в таком виде, а потом, когда понадобится, перевести обратно, но все эти переводы трудоёмки. Поэтому применяются системы счисления, родственные двоичной - восьмеричная и шестнадцатеричная. Для записи чисел в этих системах требуется соответственно 8 и 16 цифр. В 16-теричной первые 10 цифр общие, а дальше используют латинские буквы. Шестнадцатеричная цифра A соответствует десятеричному числу 10, шестнадцатеричная B – десятичному числу 11 и т. д. Использование этих систем объясняется тем, что переход к записи числа в любой из этих систем от его двоичной записи очень прост. Ниже приведена таблица соответствия чисел, записанных в разных системах.

Десятичная	Двоичная	Восьмеричная	Шестнадцатеричная
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Любое целое число в позиционной системе можно записать в форме многочлена:

$$X_S = \{A_n A_{n-1} A_{n-2} \ldots A_2 A_1\} = A_n \cdot S^{n-1} + A_{n-1} \cdot S^{n-2} + \ldots + A_1 \cdot S^0$$

где S - основание системы счисления, A_n - цифры числа, записанного в данной системе счисления, n - количество разрядов числа.

К примеру, число 5472 в десятичной системе счисления запишется в форме многочлена следующим образом:

$$5472 = 5 \cdot 10^3 + 4 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0$$

1.6.1. Перевод из одной системы счисления в другую

Перевод чисел из одной системы счисления в другую составляет важную часть машинной арифметики. Правила перевода целых чисел становится ясным из общей формулы записи числа в произвольной позиционной системе. Пусть число X в исходной системе счисления s имеет вид

$$X_s = \{x_r x_{r-1} \ldots x_2 x_1\}$$

Требуется получить запись числа в системе счисления с основанием h:

$$X_h = \{y_w y_{w-1} \ldots y_2 y_1\} = y_w h^{w-1} + y_{w-1} h^{w-2} + \ldots + y_1 h^0$$

После деления многочлена на h получим:

$$X / h = y_w h^{w-2} + y_{w-1} h^{w-3} + \ldots + y_1 / h$$

Младший разряд X_h равен первому остатку y_1. Следующий разряд y_2 определяется делением частного Q_1 на h:

$$Q_1 = y_w h^{w-2} + y_{w-1} h^{w-3} + \ldots + y_2$$

$$Q_1 / h = y_w h^{w-3} + y_{w-1} h^{w-4} + \ldots + y_2 / h$$

Остальные разряды получаются аналогично до тех пор, пока $Q_{w-1} > 0$.

В результате правило перевода из одной системы счисления в другую можно записать так: ***для перевода целого числа из s-ичной системы счисления в h-ичную необходимо последовательно делить это число и получаемые частные на h (по правилам системы счисления с основанием s) до тех пор, пока частное не станет равным нулю. Старшей цифрой в записи числа с основанием h служит последний остаток, а следующие за ней цифры образуют остатки от предшествующих делений, выписываемые в последовательности, обратной их получению.***

2. БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА С

Данная глава посвящена описанию основных принципов построения программ, включая не только функциональное описание программной реализации, но и стиль написания.

2.1. Переменные

Переменная — это именованная область памяти, в которую могут быть записаны различные значения. Также из этой области памяти может быть извлечено значение переменной, используя ее имя. В каждый момент времени переменная может иметь только одно значение.

Значения, которые может хранить переменная, определяется ее **типом**.

В простейшем виде переменную можно определять следующим образом:

Тип список_имен_переменных;

Тип переменной определяет значения, которые может принимать переменная.

Типы могут быть следующими:

char — целое значение, 8 бит (диапазон от –128 до 127);

int — целое значение, обычно 4 байта (зависит от платформы);

float — вещественные числа;

double— вещественные числа удвоенной точности.

Каждый из целочисленных типов может быть определен, как знаковый **signed,** либо как беззнаковый **unsigned** (для MS VS по умолчанию **signed**).

Пример определения переменных:

```
char my_symbol;
int val, val2;
double Result;
```


После того как переменная создана, ей можно присваивать значения. Это можно сделать как при определении переменной, так и после этого:

Тип имя_переменной = начальное_значение;
имя_переменной = начальное_значение;

Например:

```
int val = 5;
 val = 5;
```

Необходимо помнить, что задавать значения переменной можно только после того, как эта переменная создана. А если значение явно не определено при создании, то по умолчанию оно ничему не присваивается. То есть попытка вывести значения неинициализированной переменной на экран может привести либо к выводу «мусора», либо даже к ошибке выполнения программы (в зависимости от среды выполнения).

2.2. Структура программы

Классической первой программой, которую обычно пишут, является вывод на экран «Hello World!». Ниже приведен код программы, которая это делает.

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("Hello World!");
6 }
```

Разберем подробно структуру программы.

В 1 строке происходит подключение стандартной библиотеки для использования ввода/вывода

Строчка 2 остается пустой для большей наглядности программы

В строке 3 описывается функция void main(). Подробное изучение функций происходи ниже, поэтому пока такое описание необходимо принять за аксиому.

В 4 строке открывается фигурная скобка, которая обозначает начало функции main, а в 6 строке фигурная скобка закрывается, обозначая что функция закончена. Внутри этих скобок собственно и происходит описание функции – тех действий, которые совершает программа.

Функция main является главной функцией программы, так как только она исполняется в программе. То есть при запуске программы, происходит выполнение тех команд, которые написаны в этой функции. В описанном примере это одна команда – вывод на экран текстового сообщения. Внутри функции main (а также всех других функций), каждая строка должна заканчиваться символом **;**, за исключением операторов ветвления и циклов.

Написанный код, как видно, имеет удобное форматирование, что обеспечивает комфортное чтение текста программы. Форматирование производится командой табуляции (кнопка **Tab** на клавиатуре). Но при этом, если написание программы происходит построчно, то при переходе на следующую строчку (кнопка **Enter** на клавиатуре), текст форматируется автоматически.

2.3. Ввод и вывод переменных

При работе программы зачастую необходимо чтобы пользователь сообщал программе различные параметры, а программа, по ходу выполнения, выдавала результат вычислению пользователю.

В языке программирования С для вывода значений переменных на консоль необходимо дополнительно указывать тип данных.

```
int a = 4;
char ch = 'Q';
float q = 0.125;
double pi = 3.141592;
```


Поэтому вывод переменных выглядит следующим образом:

```
printf("%d\n", a);
printf("%c\n ", ch);
printf("%f\n ", q);
printf("%f\n ", pi);
```

При вводе значений переменных с консоли, программист должен априорно быть знакомым с понятиями ссылок и указателей на объекты, которые будут освещены позже. Ниже приводится пример ввода значения переменных различного типа с консоли.

```
int a;
char ch;
float q;
double pi;

scanf("%d", &a);
scanf("%c", &ch);
scanf("%f", &q);
scanf("%lf", &pi);
```

При операциях вывода значений переменных возможны комбинации переменных и текстовых выражений. При операциях ввода такие средства не предусмотрены. Ниже приведен фрагмент кода, запрашивающий у пользователя значение переменной, и выводящий ее квадрат.

```
int a;
int Res;

printf("Input value a = ");
scanf("%d", &a);

Res = a*a;

printf("a*a = %d*%d = %d\n", a, a, Res);
```

2.4. Арифметические и логические операции и их использование

Для вычисления выражений в языке С могут использоваться различные арифметические операции. При этом каждая операция имеет свой ранг, определяющий приоритет ее выполнения. Чем ниже ранг операции, тем больший приоритет имеет операция. Операции одного ранга исполняются согласно правилам ассоциативности либо слева направо (→), либо справа налево (←). В таблице разобраны типовые операции по рангам и тип их ассоциативности:

Ранг	Операции	Ассоциативность
1	() [] –> .	➔
2	! ~ + – ++ -- & * (тип) sizeof()	➔
3	* / % (мультипликативные бинарные)	➔
4	+ – (аддитивные бинарные)	➔
5	<< >> (поразрядного сдвига)	➔
6	< <= >= > (отношения)	➔
7	== != (отношения)	➔
8	& (поразрядная конъюнкция «И»)	➔
9	^ (поразрядное исключающее «ИЛИ»)	➔
10	| (поразрядное дизъюнкция «ИЛИ»)	➔
11	&& (конъюнкция «И»)	➔
12	|| (дизъюнкция «ИЛИ»)	➔
13	?: (условная операция)	➔
14	= *= /= %= += –= &= ^= |= <<= >>=	➔
15	, (операция «запятая» – перечисление)	➔

2.4.1. Выражения и приведение арифметических типов

Каждое выражение состоит из одного или нескольких операндов, символов операций и ограничителей, в качестве которых чаще всего выступают круглые скобки ( ). Назначение любого выражения — формирование некоторого значения. Тип результата определяется исходя из типов значений и правил вычисления операторов. Если значениями

выражения являются целые и вещественные типы, то говорят об арифметических выражениях, в которых допустимы следующие операции:

+ — сложение;

– — вычитание;

* — умножение;

/ — деление;

% — получение остатка от целочисленного деления.

Примеры выражений с двумя операндами:

```
A + b
12.3 – x
3.14169 * Z
e / 8
8 % i
```

Также существуют специфичные унарные операции **++** и **--** для изменения операнда на 1. При этом эти операнды могут применяться как после операнда (a++) - постинкремент, так и до него (--x) - предекремент. При этом результат будет различный:

- Если изначально a было равно 6. То после выполнения z = a++, результатом будет: z = 6; a = 7;
- Если изначально a было равно 6. То после выполнения z = ++a, результатом будет: z = 7; a = 7.

Для операнда -- переменные будут изменяться аналогично.

2.4.2. Отношения и логические выражения

Отношение определяется как пара выражений, между которыми стоит знак операции отношения. Допустимы следующие операции:

==	проверка равенства	<	меньше;
!=	не равно	>=	больше или равно;
>	больше	<=	меньше или равно.

Примеры отношений:

```
a – b > 6.8
(x – 8) * 3 == 15
5 >= 1
```

Логический тип (истина или ложь) в языке С отсутствует. Поэтому принято, что отношение имеет ненулевое значение (обычно 1), если оно истинно, и равно 0, если оно ложно.

Логических операций в языке С три:

! — отрицание — логическое НЕ

&& — конъюнкция — логическое И

|| — дизъюнкция — логическое ИЛИ

Так как значением отношения является целое, то ничего не противоречит применению логических операций к целочисленным значениям. При этом любое положительное ненулевое значение будет восприниматься как истинное. Иначе говоря значением !0 будет 1, а !54 будет 0.

2.4.3. Приведение типов

Особенность языков С и С++ заключается в том, что при вычислении значений выражений, результат перед присваиванием всегда приводится к типу первого операнда оператора присваивания. То есть при делении двух целых значений результат будет также иметь тип целого числа, поскольку каждый из операндов представлен целым типом.

```
int a = 5;
int b = 2;
double res;
res = a / b;     // Результатом будет 2
```

Для получения вещественного типа необходимо, чтобы хотя бы один операнд имел тип вещественного числа. Для этого можно использовать приведение типов. В коде программ достаточно часто используется явное приведение типов, которое не изменяет тип самой переменной.

```
int a = 5;
int b = 2;
double res;

res = (double)a / b;
```

Рекомендуется в случае подобных вычислений приводить к желаемому типу первый операнд, что упрощает чтение и последующую отладку.

2.4.4. Выражения с поразрядными операциями

Такие операции позволяют конструировать выражения, в которых обработка операндов выполняется побитно (в двоичном представлении числа). Возможны следующие операции:

~ — инвертирование битов; // ~170 равно 85

>> — сдвиг последовательности битов вправо; // 100 >> 2 равно 25

<< — сдвиг последовательности битов влево // 5 << 2 равно 20

^ — поразрядное исключающее ИЛИ;

| — поразрядное ИЛИ;

& — поразрядное И;

Операции побитового сдвига часто применяются в программировании, поскольку позволяют значительно ускорять операции умножения или целочисленного деления на 2^n.

```
int a;
printf("Input value a = ");
scanf("%d", &a);
printf("a*16 = %d\n",(a << 4));
printf("a/8 = %d\n",(a >> 3));
```

В данном примере выполнятся умножение переменной a на 2 в 4 степени и деление на 2 в 3 степени.

2.5. Операторы ветвления

В языке С существует два оператора ветвления:

- условный оператор if.
- переключатель switch.

2.5.1. Оператор if

Оператор **if** может использоваться либо для условного выполнения определенного действия, либо для разветвления программы на небольшое количество ветвей.

В случае условного исполнения, оператор имеет следующий вид:

```
if ( условие )
    оператор;
```

При этом, если необходимо исполнить несколько операторов, то их необходимо выделить в ТЕЛО, которое обозначается фигурными скобками.

```
if ( условие )
{   оператор_1;
    оператор_2;
    оператор_3;
```

В случае ветвления исполнения, оператор имеет следующий вид:

```
if ( условие )
    оператор_1;
else
    оператор_2;
```


При этом возможно ветвление и на большее количество ветвей:

```
if ( условие_1 )
    оператор_1;
else
    if(условие_2 )
        оператор_2;
    else
        оператор_3;
```

В качестве условия могут использоваться арифметические выражения, отношения и логические выражения (x > 10 && x < 87).

Приведем фрагмент программы для определения корней уравнения вида

$$ax^2 + bx + c = 0$$

```
D = b*b-4*a*c;
if ( D < 0)
   printf("No roots\n");
else
   if(D == 0)
   {
      x = (double)-b / (2*a);
      printf("1 root: x = %a\n", x);
   }
   else
   {
      sqrt_D = sqrt(D);
      x1 = ( - b + sqrt_D) / (2*a);
      x2 = ( - b - sqrt_D) / (2*a);
      printf("2 root: x1 = %f\tx2 = %f\n",x1, x2);
   }
```

2.5.2. Переключатель switch

Данный оператор используется для организации мультиветвления и выглядит следующим образом:

switch (выражение)

{

case Константа_1: операторы_1;

case Константа_2: операторы_2;

….

default: операторы;

}

При первом совпадении значения выражения с константой происходит выполнение операторов, помеченных данной меткой. Если после их выполнения не предусмотрено никаких операторов перехода, то выполняются также все последующие операторы. То есть по сути, CASE является меткой, обозначающей место выполнения программы после SWITCH.

Оператором перехода может быть **break**, который осуществляет выход из тела (фрагмент кода, обозначенный фигурными скобками).

Переключатели чаще всего используются, когда количество ветвей больше 3, либо необходимо перебрать заданные константы.

Для иллюстрации приведен фрагмент программы, которая выводит на экран название введенной цифры от 0 до 9:

```
int a;
printf("Input value a = ");
scanf("%d", &a);
```



```
switch (a)
  {
    case 1: printf("%d - One\n", a);
      break;
    case 2: printf("%d - Two\n", a);
      break;
    case 3: printf("%d - Three\n", a);
      break;
    case 4:printf("%d - Four\n", a);
      break;
    case 5:printf("%d - Five\n", a);
      break;
    case 6:printf("%d - Six\n", a);
      break;
    case 7:printf("%d - Seven\n", a);
      break;
    case 8:printf("%d - Eight\n", a);
      break;
    case 9:printf("%d - Nine\n", a);
      break;
    case 0:printf("%d - Zero\n", a);
      break;
  default: printf("The value is not from 0 to 9\n");
  }
```

Еще раз отметим, что если бы в программе отсутствовали операторы break, то при вводе, например, цифры 6, на экран бы распечаталось следующее:

```
6 – Six
7 – Seven
8 – Eight
9 – Nine
0 – Zero
The value is not from 0 to 9
```

Очевидно, что программист не планировал такое поведение программы.

2.6. Операторы циклов

В языке С существует три разных способа организации циклов. Ниже мы их разберем с примерами использования.

2.6.1. Цикл for

Оператор **for** является, пожалуй, базовым оператором для организации циклов. Цикл **for** — это параметрический цикл, он предоставляет возможность задавать параметры для выполнения цикла: начальные значение и условия. Форма записи выглядит следующим образом:

for (выражение_1; условие_цикла; выражение_2)

Тело цикла

выражение_1 – определяет действия, выполняемые до начала цикла.

условие_цикла – обычно логическое или арифметическое условие. Пока это условие истинно, выполняется цикл. Проверка происходит перед началом очередной итерации тела цикла.

выражение_2 — это действие, выполняемое в конце каждой итерации цикла.

В качестве примера рассмотрим программу, распечатывающую все числа, которые делятся на 2 до введенного пользователем значения.

```
int i, Max;

printf("Input Max = ");
scanf("%d", &Max);

for(i = 0; i <= Max; i++)
  if(i%2 == 0)
    printf("%d\t", i);
```


Выражения в скобках оператора **for**, не являются обязательными. Ниже приведен полностью идентичный код, выполняющий точно такой же цикл.

```
int i, Max;
printf("Input Max = ");
scanf("%d", &Max);
i = 0;
for(;;)
{
  if(i <= Max)
  {
    if(i%2 == 0)
      printf("%d\t", i);
    i++;
  }
  else
    break;
}
```

Помните, что если программист не указывает внутри оператора for какие-либо выражения, они считаются пустыми и точки с запятой не опускаются! Например, for(; x < 5;) или for(;; counter++)"

2.6.2. Цикл while

Цикл while — это цикл с предусловием. В этом цикле перед каждой итерацией проверяется условие, и только если оно истинно, происходит выполнение тела цикла. Он имеет следующий вид:

while (выражение условия цикла)

Тело цикла

Приведем фрагмент кода, который также распечатывает все числа, которые делятся на 2 до введенного пользователем значения

```
i = 0;
while(i <= Max)
{
   if(i%2 == 0)
      printf("%d\t", i);
   i++;
}
```

Как видно из примера, условие цикла записывается в более наглядном виде, чем при использовании **for**. Но при таком описании довольно часто возникают ошибки связанные с тем, что необходимо явно описывать инкрементацию счетчика цикла.

2.6.3. Цикл do-while

Цикл **do** — это цикл с постусловием. Проверка условия цикла происходит после каждой итерации. Это обеспечивает минимум одну итерацию выполнения вне зависимости от условий.

do

Тело цикла

while (выражение условия цикла);

2.7. Массивы данных

В случае, если программа обрабатывает множество однотипных данных, именовать каждую переменную не только неудобно, но порой и просто невозможно из-за их большого количества. Для обработки однотипных данных применяются ***массивы***, которые позволяют компоновать переменные. Наиболее наглядными примерами применения массивов являются задачи статистической обработки данных. Также можно упомянуть обработку векторов различной размерности.

Для определения массива необходимо указать, какой тип будут иметь его элементы, и количество этих элементов:

```
int arr[10];
```


Здесь определен одномерный массив, состоящий из целых чисел, имеющий 10 элементов. Элементы в массиве нумеруются, начиная с 0, поэтому в приведенном примере массив содержит элементы с номерами от 0 до 9. Обращаться к его элементам следует следующим образом:

arr[i]

Ниже приведен фрагмент кода, в котором пользователь задает 10 значений, а программа выводит их сумму.

```
int arr[10];
int i;
int Sum = 0;

for(i=0; i < 10; i++)
{
   printf("a[%d] = ", i);
   scanf("%d", &arr[i]);
}
for(i=0; i < 10; i++)
   Sum += arr[i];

printf("Summa = %d\n", Sum);
```

Из примера видно, что работа с элементами массивов мало чем отличается от работы с любыми другими переменными.

2.8. Функции

Теперь некоторые пояснения к самой программе. Любая программа на языке С состоит из одной или более "***функций***", указывающих фактические операции, которые должны быть выполнены. По своей сути, в функции перечисляется порядок действий, а также заводятся переменные для хранения данных.

Функция с именем **main** – особенная. Именно выполнение этой функции и происходит при запуске любой программы. Это означает, что

каждая программа должна в каком-то месте содержать функцию с именем **main**. Для выполнения определенных действий функция **main** обычно обращается к другим функциям, часть из которых находится в той же самой программе, а часть - в библиотеках, содержащих ранее написанные функции.

Действия, выполняемые при обращении к функции, задает ее тело, которое выделяется фигурными скобками **{ }**. Структура определения функции имеет вид:

Тип_результата Имя функции (список_аргуметнов);

Реализация функции происходит следующим образом:

Тип_результата Имя_Функции (список_аргументов)
{
...
}

Необходимо помнить, что в отличие от объявления переменных, в списке аргументов (параметров) функции недопустимо перечислять несколько имен переменных одного типа – для каждой переменной должен быть явно указан тип.

Реализация функции всегда должна происходить после ее объявления. В ряде случаев (например, когда вся программа состоит только из одного файла), допускается не делать отдельных объявлений функций.

Помните, что объявление (или реализация, если объявления не существует) должно производится до того места, где функция вызывается! То есть, если создается пользовательская функция, которая будет вызываться из функции **main**, то она должна быть описана до функции **main**.

Для возврата значения используется команда **return** (она может быть выполнена в любом месте кода тела, но выполнение функции после этого

заканчивается). Если функция не возвращает никаких значений, то используется специальный тип **void**.

В качестве примера рассмотрим программу, в которой описана пользовательская функция для вычисления дискриминанта квадратного уравнения, вида

$$ax^2 + bx + c = 0:$$

```
#include <stdio.h>

double Discriminant (int a, int b, int c)
{
  double D;
  D = b*b - 4*a*c;
  return D;
}

void main()
{
  int a, b, c;
  double D;
  a = 4;
  b = 2;
  c = 3;

  D = Discr_Calc(a, b, c);

  printf("D = %f\n", D);
}
```

Функция может возвращать только одно значение. Обычно функции используются для написания более простых для чтения программ. Так программа, которая содержит только функцию **main()**, состоящую из 13 окон теста, очень трудна для понимания, тем более сторонним человеком. Поэтому часто используется негласное соглашение, что текст любой функции не должен превышать по размеру одного экрана — размера, который человек может охватить взглядом. При выборе имен функций программист должен руководствоваться принципом: имя функции

отражает действия, которые она выполняет и дает интуитивное понимание функциональности даже постороннему человеку.

2.9. Локальные и глобальные переменные

Все создаваемые в программе переменные имеют определенную область видимости. Для переменных выделяют две основных области видимости: **локальные** и **глобальные**.

2.9.1. Глобальные переменные

Из названия этого класса переменных становится понятно, что они доступны всем. Под всеми подразумеваются все функции проекта и другие глобальные переменные. В случае, если проект включает несколько файлов, для использования глобальной переменной, объявленной в другом файле, необходимообъявить ее еще раз с префиксом *extern*.

Для того чтобы переменная была глобальная, она должна быть объявлена вне функций. Ниже приведен текст программы, в которой переменная с именем D является глобальной.

```
#include <stdio.h>

double D;

double Discr_Calc(int a, int b, int c)
{
  D = b*b - 4*a*c;
  return D;
}

void main()
{
  int a, b, c;
  a = 4;
  b = 2;
  c = 3;
```



```
  D = Discr_Calc(a, b, c);

  printf("D = %f\n", D);
}
```

В данном примере к глобальной переменной D обращаются две различные функции, для которых эта переменная является общей.

2.9.2. Локальные переменные

Локальные переменные обладают ограничением области видимости – они видны и доступны только внутри того блока, в котором они созданы. Важно помнить, что локальные переменные перестают существовать при выходе из блока, в котором они были объявлены.

Блоками, ограничивающими видимость, могут являться функции, либо набор действий, выделенных в отдельное тело с помощью скобок {}.

Таким образом, если в тесте предыдущей программы описать функцию вычисления дискриминанта следующим образом:

```
void Discriminant (int a, int b, int c)
{
  double D;
  D = b*b - 4*a*c;
}
```

то в ней существует собственная локальная переменная D, которая перестает существовать, как только исполнение программы выходит из этой функции. Другими словами, локальные переменные перекрывают глобальные, в результате чего, выполнение описанной программы с такими изменениями приведет к тому, что на экран всегда будет выводиться НОЛЬ!

Ниже приведен пример еще одной неправильно написанной программы, которая не сможет быть скомпилирована и запущена.

```c
#include <stdio.h>

double Discr_Calc(int a, int b, int c)
{
  D = b*b - 4*a*c;
  return D;
}

void main()
{
  int a, b, c;
  double D;
  a = 4;
  b = 2;
  c = 3;

  D = Discr_Calc(a, b, c);

  printf("D = %f\n", D);
}
```

В приведенной программе ошибка заключается в том, что переменная с именем **D** определена в функции ***main()*** и не существует за ее пределами. Таким образом, в функции ***Discr_Calc()*** происходит обращение к несуществующей переменной **D**.

Переменные, перечисленные в списке параметров функции, также являются локальными для этой функции. Их отличие от описанных в теле функции состоит только в том, что они будут гарантированно инициализированы к моменту вызова функции.

2.10. Динамическая память

Работа с динамической памятью позволяет программисту решать задачи с неизвестным количеством исходных данных, добавляя новые возможности в функционал программ.

Основными понятиями для работы с динамической памятью являются указатели и ссылки. Именно с помощью переменных этих типов осуществляется управление памятью, ее выделение и освобождение.

2.10.1. Указатели и работа с ними

Каждая переменная представляет собой именованный участок памяти ПК, в котором хранится ее значение. При этом, чтобы получить доступ к значению переменной, необходимо обратиться к ней по ее имени.

```c
int a;
a = 1;
```

Для того чтобы получить значение адреса в памяти, по которому находится значение конкретной переменной, необходимо произвести **унарную операцию взятия адреса &.** Выражение **&a** позволяет получить адрес участка памяти, выделенного для переменной a.

```c
#include <stdio.h>

void main()
{
  int a;
  a = 5;
  printf("%p\n", &a);
}
```

Имея возможность определять адрес переменной или другого объекта программы, нужно уметь его сохранять, преобразовывать и передавать. Для этих целей введены переменные типа «**указатель**».

Указатель — это переменная, значение которой является адресом объекта конкретного типа. Для обозначения значения указателя, который никуда не указывает, используется специальная константа **NULL**.

При определении указателя необходимо обозначать, на какой тип данных указывает эта переменная. Для этого используется символ *.

Например, если Вы хотите объявить указатель на переменную типа int, нужно объявить переменную типа int*. Здесь и далее при объявлении указателей будем использовать символ «**p**» в названиях указателей. Это позволит легко отличать в тексте программы указатели от собственно переменных. Примеры определения указателя и присваивания ему значения:

```
int Ch = 14;
int* pCh = NULL;
pCh = &Ch;
```

Чтобы получить значение, которое находится по указанному адресу, применяется операция разыменования – «*****»:

```
int Ch = 14;
int* pCh = &Ch;
printf("%d\n", *pCh);
```

В случае, если программист забудет применить операцию разыменования, функция printf выведет значение адреса, хранящегося в указателе.

2.10.2. Арифметика указателей и массивы

В языке С допустимы только две арифметические операции над указателями: суммирование и вычитание. При этом данные операции обеспечивают передвижение по памяти не по байтам и битам, а на размер элемента типа данных, который указан при создании переменной указателя.

Указатели непосредственно связаны с массивами данных, поскольку при создании массива все его элементы создаются в памяти последовательно, т. е. массив представляет собой единую и неразрывную область памяти. При этом, если обратиться к массиву через его имя, то результатом будет адрес первого элемента массива (имеющего индекс 0).


```
int mas[3] = {1, 2, 3};
int* pVal;
pVal = mas; // тождественно pVal = &mas[0]
printf("%d\n", *pVal);
```

Для указателей определены арифметические операции. Таким образом, к указателям можно добавлять или отнимать целые значения. Так операция «++», примененная к указателю, изменяет адрес, хранящийся в указателе на число байт, соответствующее размеру одной переменной типа указателя. Иначе говоря, если у нас имеется указатель на тип ***int***, занимающий 4 байта, то операции ++ сместит указатель в памяти на 4 байта.

Поэтому для адресации внутри массива бывает удобно использовать указатели. В качестве примера приведем фрагмент кода для инициализации массива:

```
int mas[10], i;
int* pMas;

for(i = 0, pMas = mas; i < 10; i++)
  printf("%d\n", *(pMas+i));
```

До сих пор все массивы данных у нас были строго определенного размера. В некоторых задачах это бывает неудобно. Например, требуется, чтобы пользователь задал некий массив данных, при этом на этапе написания программы, его размер неизвестен. Можно, конечно, заранее выделить под массив 100 МБайт памяти и считать, что этого всегда хватит. Но возможны два варианта:

- массив все-таки окажется недостаточным, поскольку данные занимают 101 МБайт.
- Для конкретной задачи достаточно 10 элементов по 4 КБайт.

В обоих этих случаях необходимый объем памяти становится известен только на момент использования программы. Поэтому было бы намного удобнее, если бы программист имел возможность выделять необходимый объем памяти по требованию пользователя. Такая возможность реализуется с помощью указателей и динамического выделения памяти, из которых основными являются функции **free()** и **malloc()**. Эти функции используются для выделения и освобождения памяти.

Приведем фрагмент кода, в котором происходит выделение необходимого количества памяти для хранения данных, количество которых задает пользователь. А также происходит инициализация значений, поиск максимального элемента и среднего арифметического значения.

```
int* pArr;
int Count;
int i;
int Sum = 0;
double Avg;

printf("Input elements count: ");
scanf("%d", &Count);
pArr = malloc(sizeof(int)*Count);
if (pArr == NULL)
{
  Printf(“Can’t allocate memory!”);
}

for(i = 0; i < Count; i++)
  scanf("%d", (pArr+i));

for(i = 0; i < Count; i++)
  Sum += *(pArr+i);

Avg = (double)Sum/Count;
printf("Sum = %d\n", Sum);
printf("Avg = %f\n", Avg);
free(pArr);
```


В языке программирования С функция **malloc()** возвращает указатель на выделенную память. Вызов **free()** должен происходить для каждого вызова **malloc()**, дабы избежать утечки памяти.

В случае, если программа по каким то причинам не может выделить указанный вами объем памяти, функция malloc() вернет NULL. Поэтому, для обеспечения корректной работы программы необходимо всегда выполнять проверку значения, которое вернула функция malloc().

2.10.3. Передача переменных в функцию

Еще раз напомним про различие локальных и глобальных переменных.

```
#include <stdio.h>
void MyFunc(int a, int b, int c)
{
  a = 11;
  b = 12;
  c = 13;
}
void main()
{
  int a, b, c;
  a = 1; b = 2; c = 3;

  MyFunc(a, b, c);
  printf("%d\t%d\t%d\n", a, b, c);
}
```

В результате исполнения приведенного кода, на экран будет выведено **1 2 3**. Другими словами, переменные в функции ***main()*** не изменят своего значения. В функции ***MyFunc()*** определены собственные локальные переменные с такими же именами, но которые не имеют отношения к переменным в функции ***main()***.

При написании программ часто возникает желание произвести внутри функции манипуляции с переменными таким образом, чтобы в вызываемой функции они также изменили значения. Например, в

программах часто применяется метод инициализации всех переменный внутри отдельного блока (функции). **Одним из способов** решения такой задачи является заведение глобальных переменных следующим образом.

```
#include <stdio.h>
int a, b, c;
void MyFunc()
{
   a = 11;
   b = 12;
   c = 13;
}
void main()
{
   a = 1; b = 2; c = 3;
   MyFunc();
   printf("%d\t%d\t%d\n", a, b, c);
}
```

В этом случае никаких переменных в функцию передавать не надо, так как она имеет непосредственный доступ ко всем глобальным переменным.

Помните, если вы напишите следующим образом:

```
#include <stdio.h>
int a, b, c;
void MyFunc(int a, int b, int c)
{  a = 11;
   b = 12;
   c = 13;
}
void main()
{

   a = 1; b = 2; c = 3;
   MyFunc(a, b, c);
   printf("%d\t%d\t%d\n", a, b, c);
}
```


то в функции будут использоваться уже не глобальные, а локальные переменные. Это вновь приведет к тому, что значения глобальных переменных не изменятся после выполнения функции.

Второй способ заключается в передаче переменных по адресу.

Оператор **&** — это унарный оператор, возвращающий адрес своего операнда. (Напомним, что унарный оператор имеет один операнд). Например, если написать **&count**, то результатом будет адрес переменной **count**. Оператор **&** можно представить, как оператор, возвращающий адрес объекта.

Для хранения адресов используются указатели, а для присвоения значения переменной через указатель на нее, унарный оператор разыменования *****. Таким образом, корректной является запись

```
int* p;
int count;
p = &count;
*p = 5;
printf("%d", count);
```

Приведем фрагмент кода, в котором переменные передаются по адресу в функцию, которая изменяет их значения.

```
void MyFunc(int* pa, int* pb, int* pc)
{  *pa = 11;
   *pb = 12;
   *pc = 13;
}
void main()
{
   int a, b, c;
   a = 1; b = 2; c = 3;

   MyFunc(&a, &b, &c);

   printf("%d\t%d\t%d\n", a, b, c);
}
```

Способ передачи переменных по адресу является более предпочтительным по сравнению с использованием глобальных переменных, поскольку предусматривает более гибкое поведение программы и возможность локализации ошибок при ее отладке: в случае применения глобальных переменных их значения могут быть изменены из любой строки вашей программы.

2.11. Строки

В программе на языке С строки могут определяться как строковые константы, массивы символов, указатель на символьный тип, массивы строк. Разработчик обязательно должен помнить о необходимости выделения памяти для хранения строки.

Любая последовательность символов, заключенная в двойные кавычки "", рассматривается как строковая константа. Необходимая память под строковые константы выделяется в статической памяти автоматически.

Под хранение строки выделяются последовательно идущие ячейки оперативной памяти. Таким образом, строка представляет собой массив символов. Для хранения кода каждого символа строки отводится 1 байт.

Для корректного вывода любая строка должна заканчиваться нуль-символом '\0'. Целочисленное значение нуль-символа равно 0. При объявлении строковой константы нуль-символ добавляется к ней автоматически. Таким образом, последовательность символов, представляющая собой строковую константу, будет размещена в оперативной памяти компьютера, включая нулевой байт. То есть ее фактический размер в памяти будет на 1 байт больше, чем количество символов в строке.

Для помещения в строковую константу некоторых служебных символов используются символьные комбинации. К примеру, если необходимо включить в строку символ двойной кавычки, ему должен предшествовать символ "обратный слеш": '\"'.

При определении массива символов необходимо сообщить компилятору требуемый размер памяти.

```
char m[82];
```

Компилятор также может самостоятельно определить размер массива символов, если инициализация массива задана при объявлении строковой константой:

```
char m2[]="Happy birthday.";
char m3[]={'H','a','p','p','y',' 
','b','i','r','t','h','d','a','y','.','\0'};
```

В этом случае имена m2 и m3 являются указателями на первые элементы массивов:

m2	эквивалентно &m2[0]
m2[0]	эквивалентно 'H'
m2[1]	эквивалентно 'a'
m3	эквивалентно &m3[0]
m3[2]	эквивалентно 'p'

При объявлении массива символов и инициализации его строковой константой можно явно указать размер массива, но указанный размер массива должен быть больше, чем размер инициализирующей строковой константы:

```
char m2[80]="Happy birthday.";
```

Для задания строки можно использовать указатель на символьный тип.

```
char *m4;
```

В этом случае переменной m4 может быть присвоен адрес массива:

```
m4 = m3;
```

*m4 эквивалентно m3[0]='H'
*(m4+1) эквивалентно m3[1]='a'

Следует отметить, что m3 является константой-указателем. Нельзя изменить m3, так как это означало бы изменение положения (адреса) массива в памяти, в отличие от m4.

Для указателя можно использовать операцию инкремент (перемещение на следующий символ): m4++.

Иногда в программах возникает необходимость описание массива символьных строк. В этом случае можно использовать индекс строки для доступа к нескольким разным строкам.

```
char *poet[4] = {"Alexey", "Viktor", "Oleg" , "Petr"};
```

В этом случае poet является массивом, состоящим из четырех указателей на символьные строки. Каждая строка символов представляет собой символьный массив, поэтому имеется четыре указателя на массивы. Указатель poet[0] ссылается на первую из строк:

*poet[0] эквивалентно 'A',
*poet[l] эквивалентно 'V',

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны инициализации каждой строки в массиве. Запятая разделяет соседние последовательности. Кроме того, можно явно задавать размер строк символов, используя описание, подобное такому:

```
char poet[4][23];
```


Разница заключается в том, что такая форма задает «прямоугольный» массив, в котором все строки имеют одинаковую длину.

Для определения свободного массива используется следующее объявление:

```
char *poet[4];
```

Здесь длина каждой строки определяется тем указателем, который эту строку инициализирует. Достоинства свободного массива - не тратится лишняя память. Недостаток – требуется более аккуратная инициализация и удаление.

Большинство операций языка Си, имеющих дело со строками, работает с указателями. Для размещения в оперативной памяти строки символов необходимо выделить блок оперативной памяти под массив и проинициализировать строку.

Для выделения памяти под хранение строки могут использоваться функции динамического выделения памяти. При этом необходимо учитывать требуемый размер строки:

```
char *name;
name = (char*)malloc(10 * sizeof(char));
scanf("%9s", name);
```

Для ввода строки использована функция scanf(), причем введенная строка не может превышать 9 символов. Последний символ будет содержать '\0'. Но из соображений корректности рекомендуется вместо этой функции использовать fgets следующим образом:

```
fgets(указатель на строку,  размер, stdin);
```

2.11.1. Ввод строк

Для ввода строки может использоваться функция scanf(). Однако функция scanf() предназначена скорее для получения слова, а не строки. Если применять формат "%s" для ввода, строка вводится до (но не включая) следующего пустого (пробельного) символа, которым может быть пробел, табуляция или перевод строки.

Для ввода строки, включая пробелы, используется функция

```
char * gets(char *);
```

В качестве аргумента функции передается указатель на строку, в которую осуществляется ввод. Функция просит пользователя ввести строку, которую она помещает в массив, пока пользователь не нажмет Enter.

2.11.2. Вывод строк

Для вывода строк можно воспользоваться функцией printf:

```
printf("%s", str); // str - указатель на строку
```

Для вывода строк также может использоваться функция

```
int puts (char *s);
```

Функция puts печатает строку s и переводит курсор на новую строку (в отличие от printf()). Функция puts() также может использоваться для вывода строковых констант, заключенных в кавычки.

2.11.3. Ввод и вывод символов

Для ввода символов может использоваться функция getchar, которая возвращает значение символа, введенного с клавиатуры:

```
char getchar();
```

Для вывода символов может использоваться функция putchar, которая возвращает значение выводимого символа и выводит на экран символ, переданный в качестве аргумента.

```
char putchar(char);
```

В качестве примера ниже приведен фрагмент программы, которая считает количество повторений символа во введенной строке.

```
int main()
{
   char s[80], sym;
   int count, i;
   printf("Enter string: ");
   gets(s);
   printf("Enter symbol: ");
   sym = getchar();
   count = 0;
   for(i=0; s[i]!='\0'; i++)
   {
      if(s[i]==sym)
         count++;
   }
   printf("In string\n");
   puts(s); // Вывод строки
   printf("symbol ");
   putchar(sym); // Вывод символа
   printf(" presented %d times",count);
   return 0;
}
```

2.12. Битовые операции

Язык C поддерживает все существующие битовые операторы. Поскольку C создавался, чтобы заменить ассемблер, то была необходимость поддержки всех (или по крайней мере большинства) операций, которые может выполнить ассемблер. Битовые операции — это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка C *char* и *int*. Битовые операторы не могут использоваться с *float*, *double*, *long double*, *void* и другими сложными типами. В языке C есть следующие битовые операторы:

Оператор	Действие
&	И
|	ИЛИ
~	Дополнение (НЕ)
^	Исключающее ИЛИ
>>	Сдвиг вправо
<<	Сдвиг влево

Битовые операторы И, ИЛИ, НЕ используют ту же таблицу истинности, что и их логические эквиваленты, за тем исключением, что они работают побитно:

p	Q	p^q	p|q	p&q	~p
0	0	0	0	0	1
0	1	1	1	0	1
1	0	1	1	0	0
1	1	0	1	1	0

2.12.1. Побитовые И, ИЛИ, НЕ, исключающее ИЛИ

В побитовых (bit-wise) операциях значение бита, равное 1, рассматривается как логическая истина, а 0 как ложь. Побитовое И (оператор &) берёт два числа и логически умножает соответствующие биты. Например, если логически умножить 3 на 8, то получим 0

```
char a = 3;
char b = 8;
char c = a & b;
printf("%d", c);
```

Так как в двоичном виде 3 в виде однобайтного целого представляет собой

```
00000011
```

а 8

```
00001000
```

Первый бит переменной с равен логическому произведению первого бита числа a и первого бита числа b. И так для каждого бита.

```
00000011
00001000
↓↓↓↓↓↓↓↓
00000000
```

Аналогично, побитовое произведение чисел 31 и 17 даст 17, так как 31 это 00011111 , а 17 это 00010001:

```
00011111
00010001
↓↓↓↓↓↓↓
00010001
```

Побитовое произведение чисел 35 и 15 равно 3.

```
00100011
00001111
↓↓↓↓↓↓↓
00000011
```

Аналогично работает операция побитового ИЛИ (оператор |), за исключением того, что она логически суммирует соответствующие биты чисел без переноса.

Следующие пример выведет 15, так как 15 это 00001111, а 11 это 00001011.

```
char a = 15;     // 00001111
char b = 11;     // 00001011
char c = a | b; // 00001111
printf("%d", c);
```

Побитовое ИЛИ для чисел 33 и 11 вернёт 43, так как 33 это 00100001, а 11 это 00001011

```
00100001
00001011
↓↓↓↓↓↓↓
00101011
```

Побитовое дополнение (оператор ~) работает не для отдельного бита, а для всего числа целиком. Оператор инверсии меняет ложь на истину, а

истину на ложь, для каждого бита. Например, программа ниже выведет -66, так как 65 это 01000001, а инверсия даст 10111110, что равно -66:

```
char a = 65; // 01000001
char b = ~a; // 10111110
printf("%d", b);
```

Исключающее ИЛИ (оператор ^) применяет побитово операцию XOR Можно рассматривать операцию XOR, как сложение по модулю 2 или операцию «не равно». Применительно к побитовому XOR – это побитовое сложение по модулю 2 или отмеченные единицами различающиеся биты операндов. Пример ниже выведет 89, так как a равно 00001100, a b равно 01010101. В итоге получим 01011001:

```
char a = 12;  // 00001100
char b = 85;  // 01010101
char c = a ^ b; // 01011001
printf("%d", c);
```

Иногда логические операторы && и || путают с операторами & и |. Такие ошибки могут существовать в коде достаточно долго, потому что такой код в ряде случаев будет работать. Например, для чисел 1 и 0. Но так как в языке C истиной является любое ненулевое значение, то побитовое умножение чисел 3 и 4 вернёт 0, хотя логическое умножение должно вернуть истину.

```
int a = 3;
int b = 4;
printf("a & b = %d\n", a & b);  // 0
printf("a && b = %d\n", a && b);// 1
```

2.12.2. Операции побитового сдвига

В языке C имеется 2 операции побитового сдвига – битовый сдвиг влево (оператор <<) и битовый сдвиг вправо (оператор >>). Битовый сдвиг вправо сдвигает биты числа вправо, дописывая слева нули. Битовый сдвиг влево делает противоположное: сдвигает биты влево, дописывая справа нули. Вышедшие за пределы числа биты отбрасываются. Например, сдвиг числа 5 влево на 2 позиции

```
00000101 << 2 == 00010100
```

Сдвиг числа 19 вправо на 3 позиции

```
00010011 >> 3 == 00000010
```

Числа в двоичном виде представляются слева направо, от более значащего бита к менее значащему. Побитовый сдвиг принимает два операнда – число, над которым необходимо произвести сдвиг, и число бит, на которое необходимо произвести сдвиг.

```
int a = 12;
printf("%d << 1 == %d\n", a, a << 1);
printf("%d << 2 == %d\n", a, a << 2);
printf("%d >> 1 == %d\n", a, a >> 1);
printf("%d >> 2 == %d\n", a, a >> 2);
```

Так как сдвиг вправо (>>) дописывает слева нули, то для целых чисел данная операция равносильна целочисленному делению пополам, а сдвиг влево умножению на степень 2. Произвести битовый сдвиг для числа с плавающей точкой без явного приведения типа нельзя. Это вызвано тем, что в языке C не определено представление числа с плавающей точкой.

Особенностью операторов сдвига является то, что они могут по-разному вести себя с числами со знаком и без знака, в зависимости от компилятора. Действительно, отрицательное число обычно содержит один бит знака. Когда мы будем производить сдвиг влево, он может пропасть, число станет положительным. Однако, компилятор может сделать так, что сдвиг останется знакопостоянным и будет проходить по другим правилам.

Побитовые операторы и операторы сдвига не изменяют значения числа, возвращая новое. Они, как и арифметические операторы, могут входить в состав сложного присваивания

```
int a = 10;
int b = 1;
a >>= 3;
a ^= (b << 3);
```

2.13. Работа с файлами

Первоначально язык C был реализован в операционной системе UNIX. Как таковые, ранние версии C (да и многие нынешние) поддерживают набор функций ввода/вывода, совместимый с UNIX. Этот набор иногда называют UNIX-подобной системой ввода/вывода или небуферизованной системой ввода/вывода. Однако когда C был стандартизован, то UNIX-подобные функции в него не вошли — в основном из-за того, что оказались лишними. Кроме того, UNIX-подобная система может оказаться неподходящей для некоторых сред, которые могут поддерживать язык C, но не эту систему ввода/вывода.

В языке C система ввода/вывода реализуется с помощью библиотечных (стандартных) функций. Благодаря этому система ввода/вывода является очень мощной и гибкой. Например, во время работы с файлами данные могут передаваться или в своем внутреннем двоичном представлении или

в текстовом формате, то есть в более удобочитаемом виде. Это облегчает задачу создания файлов в нужном формате.

Библиотека С поддерживает три уровня ввода-вывода: потоковый ввод-вывод, ввод-вывод нижнего уровня и ввод-вывод для консоли и портов.

2.13.1. Потоки

Файловая система в языке С предназначена для работы с самыми разными устройствами, в том числе терминалами, дисководами и накопителями на магнитной ленте. Даже если какое-то устройство сильно отличается от других, буферизованная файловая система все равно представит его в виде логического устройства, которое называется потоком. Все потоки ведут себя похожим образом. И так как они в основном не зависят от физических устройств, то та же функция, которая выполняет запись в дисковый файл, может ту же операцию выполнять и на другом устройстве, например, на консоли. Потоки бывают двух видов: текстовые и двоичные:

- **Текстовый поток** — это последовательность символов. В стандарте С считается, что текстовый поток организован в виде строк, каждая из которых заканчивается символом новой строки. Однако в конце последней строки этот символ не является обязательным. В текстовом потоке по требованию базовой среды могут происходить определенные преобразования символов. Например, символ новой строки может быть заменен парой символов — возврата каретки и перевода строки. Поэтому однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся во внешнем устройстве, может и не быть. Кроме того, количество тех символов, которые пишутся (читаются), и тех, которые хранятся во внешнем устройстве, может также не совпадать из-за возможных преобразований.

- **Двоичный поток** — это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве, причем никакого преобразования символов не происходит. Кроме того, количество тех байтов, которые пишутся (читаются), и тех, которые хранятся на внешнем устройстве, одинаково. Однако в конце двоичного потока может добавляться определяемое приложением количество нулевых байтов. Такие нулевые байты, например, могут использоваться для заполнения свободного места в блоке памяти незначащей информацией, чтобы она в точности заполнила сектор на диске.

2.13.2. Файлы

В языке С файлом может быть все что угодно, начиная с дискового файла и заканчивая терминалом или принтером. Поток связывают с определенным файлом, выполняя операцию открытия. Как только файл открыт, можно проводить обмен информацией между ним и программой.

Но не у всех файлов одинаковые возможности. Например, к дисковому файлу прямой доступ возможен, в то время как к некоторым принтерам — нет. Таким образом, мы пришли к одному важному принципу, относящемуся к системе ввода/вывода языка С: все потоки одинаковы, а файлы — нет.

Если файл может поддерживать запросы на местоположение (указатель текущей позиции), то при открытии такого файла указатель текущей позиции в файле устанавливается в начало. При чтении из файла (или записи в него) каждого символа указатель текущей позиции увеличивается, обеспечивая тем самым продвижение по файлу.

Файл отсоединяется от определенного потока (т.е. разрывается связь между файлом и потоком) с помощью операции закрытия. При закрытии файла, открытого с целью вывода, содержимое (если оно есть) связанного с ним потока записывается на внешнее устройство. Этот процесс, который

обычно называют дозаписью потока, гарантирует, что никакая информация случайно не останется в буфере диска. Если программа завершает работу нормально, т.е. либо main() возвращает управление операционной системе, либо вызывается exit(), то все файлы закрываются автоматически. В случае аварийного завершения работы программы, например, в случае краха или завершения путем вызова abort(), файлы не закрываются.

У каждого потока, связанного с файлом, имеется управляющая структура, содержащая информацию о файле; она имеет тип FILE. В этом блоке управления файлом никогда ничего не меняйте.

Если вы новичок в программировании, то разграничение потоков и файлов может показаться излишним или даже "заумным". Однако надо помнить, что основная цель такого разграничения — это обеспечить единый интерфейс. Для выполнения всех операций ввода/вывода следует использовать только понятия потоков и применять всего лишь одну файловую систему. Ввод или вывод от каждого устройства автоматически преобразуется системой ввода/вывода в легко управляемый поток.

Функции библиотеки ввода-вывода языка С, поддерживающие обмен данными с файлами на уровне потока, позволяют обрабатывать данные различных размеров и форматов, обеспечивая при этом буферизованный ввод и вывод. Таким образом, поток — это файл вместе с предоставляемыми средствами буферизации.

2.13.3. Основы файловой системы

Файловая система языка С состоит из нескольких взаимосвязанных функций. Самые распространенные из них показаны в ниже таблице. Для их работы требуется заголовочный файл <stdio.h>.

Часто используемые функции файловой системы С	
fopen()	Открывает файл
fclose()	Закрывает файл
putc()	Записывает символ в файл
fputc()	То же, что и putc()
getc()	Читает символ из файла
fgetc()	То же, что и getc()
fgets()	Читает строку из файла
fputs()	Записывает строку в файл
fseek()	Устанавливает указатель позиции на определенный байт
ftell()	Возвращает текущее значение указателя в файле
fprintf()	Для файла то же, что printf() для консоли
fscanf()	Для файла то же, что scanf() для консоли
feof()	Возвращает значение true, если достигнут конец файла
ferror()	Возвращает значение true, если произошла ошибка
rewind()	Устанавливает указатель позиции в начало файла
remove()	Стирает файл
fflush()	Дозапись потока в файл

Заголочный файл <stdio.h> предоставляет описания (прототипы) функций ввода/вывода и определяет следующие три типа: size_t, fpos_t и FILE. size_t и fpos_t представляют собой определенные разновидности такого типа, как целое без знака. А о третьем типе, FILE, рассказывается в следующем разделе.

Кроме того, в <stdio.h> определяется несколько макросов. Из них к материалу этой главы относятся NULL, EOF, FOPEN_MAX, SEEK_SET, SEEK_CUR и SEEK_END. Макрос NULL определяет пустой (null) указатель. Макрос EOF, часто определяемый как -1, является значением, возвращаемым тогда, когда функция ввода пытается выполнить чтение после конца файла. FOPEN_MAX определяет целое значение, равное максимальному числу одновременно открытых файлов. Другие макросы используются вместе с fseek() — функцией, выполняющей операции прямого доступа к файлу.

2.13.4. Указатель файла

Указатель файла — это то, что соединяет в единое целое всю систему ввода/вывода языка С. Указатель файла — это указатель на структуру типа FILE. Он указывает на структуру, содержащую различные сведения о файле, например, его имя, статус и указатель текущей позиции в начало файла. В сущности, указатель файла определяет конкретный файл и используется соответствующим потоком при выполнении функций ввода/вывода. Чтобы выполнять в файлах операции чтения и записи, программы должны использовать указатели соответствующих файлов. Чтобы объявить переменную-указатель файла, используйте такого рода оператор:

```
FILE *fp;
```

2.13.5. Открытие файла

Функция fopen() открывает поток и связывает с этим потоком определенный файл. Затем она возвращает указатель этого файла. Чаще всего (а также в оставшейся части этой главы) под файлом подразумевается дисковый файл. Прототип функции fopen() следующий:

FILE *fopen(const char *имя_файла, const char *режим);

где имя_файла — это указатель на строку символов, представляющую собой допустимое имя файла, в которое также может входить спецификация пути к этому файлу. Строка, на которую указывает режим, определяет, каким образом файл будет открыт. Ниже в таблице показано, какие значения строки режим являются допустимыми. Строки, подобные "r+b" могут быть представлены и в виде "rb+".

Допустимые значения режим	
r	Открыть текстовый файл для чтения
w	Создать текстовый файл для записи
a	Добавить в конец текстового файла
rb	Открыть двоичный файл для чтения
wb	Создать двоичный файл для записи
ab	Добавить в конец двоичного файла
r+	Открыть текстовый файл для чтения/записи
w+	Создать текстовый файл для чтения/записи
a+	Добавить в конец текстового файла или создать текстовый файл для чтения/записи
r+b	Открыть двоичный файл для чтения/записи
w+b	Создать двоичный файл для чтения/записи
a+b	Добавить в конец двоичного файла или создать двоичный файл для чтения/записи

Как уже упоминалось, функция fopen() возвращает указатель файла. Никогда не следует изменять значение этого указателя в программе. Если при открытии файла происходит ошибка, то fopen() возвращает пустой (NULL) указатель.

В следующем коде функция fopen() используется для открытия файла по имени TEST для записи.

```
FILE *fp;
fp = fopen("test", "w");
```

Хотя предыдущий код технически правильный, но его обычно пишут немного по-другому:

```
FILE *fp;

if ((fp = fopen("test","w"))==NULL)
  printf("Ошибка при открытии файла.\n");
```

Этот метод помогает при открытии файла обнаружить любую ошибку, например, защиту от записи или полный диск, причем обнаружить еще до того, как программа попытается в этот файл что-либо записать. Вообще говоря, всегда нужно вначале получить подтверждение, что функция - fopen() выполнилась успешно, и лишь затем выполнять с файлом другие операции.

Хотя название большинства файловых режимов объясняет их смысл, однако не помешает сделать некоторые дополнения. Если попытаться открыть файл только для чтения, а он не существует, то работа fopen() завершится отказом. А если попытаться открыть файл в режиме дозаписи, а сам этот файл не существует, то он просто будет создан. Более того, если файл открыт в режиме дозаписи, то все новые данные, которые записываются в него, будут добавляться в конец файла. Содержимое, которое хранилось в нем до открытия (если только оно было), изменено не будет. Далее, если файл открывают для записи, но выясняется, что он не существует, то он будет создан. А если он существует, то содержимое, которое хранилось в нем до открытия, будет утеряно, причем будет создан новый файл. Разница между режимами r+ и w+ состоит в том, что если

файл не существует, то в режиме открытия r+ он создан не будет, а в режиме w+ все произойдет наоборот: файл будет создан! Более того, если файл уже существует, то открытие его в режиме w+ приведет к утрате его содержимого, а в режиме r+ оно останется нетронутым.

Из таблицы видно, что файл можно открыть либо в одном из текстовых, либо в одном из двоичных режимов. В большинстве реализаций в текстовых режимах каждая комбинация кодов возврата каретки ‘\r’ и конца строки ‘\n’ преобразуется при вводе в символ новой строки. При выводе же происходит обратный процесс: символы новой строки преобразуются в комбинацию кодов возврата каретки ‘\r’ и конца строки ‘\n’. В двоичных режимах такие преобразования не выполняются.

Максимальное число одновременно открытых файлов определяется FOPEN_MAX. Это значение не меньше 8, но чему оно точно равняется – это должно быть написано в документации по компилятору.

2.13.6. Закрытие файла

Функция fclose() закрывает поток, который был открыт с помощью вызова fopen(). Функция fclose() записывает в файл все данные, которые еще оставались в дисковом буфере, и проводит, так сказать, официальное закрытие файла на уровне операционной системы. Отказ при закрытии потока влечет всевозможные неприятности, включая потерю данных, испорченные файлы и возможные периодические ошибки в программе. Функция fclose() также освобождает блок управления файлом, связанный с этим потоком, давая возможность использовать этот блок снова. Так как количество одновременно открытых файлов ограничено, то, возможно, придется закрывать один файл, прежде чем открывать другой. Прототип функции fclose() такой:

```
int fclose(FILE * файл);
```

Возвращение нуля означает успешную операцию закрытия. В случае же ошибки возвращается EOF. Чтобы точно узнать, в чем причина этой ошибки, можно использовать стандартную функцию ferror() (о которой вскоре пойдет речь). Обычно отказ при выполнении fclose() происходит только тогда, когда диск был преждевременно удален (стерт) с дисковода или на диске не осталось свободного места.

2.13.7. Запись символа

В системе ввода/вывода языка С определяются две эквивалентные функции, предназначенные для вывода символов: putc() и fputc(). (На самом деле putc() обычно реализуется в виде макроса.) Две идентичные функции имеются просто потому, чтобы сохранять совместимость со старыми версиями С. В этой книге используется putc(), но применение fputc() также вполне возможно.

Функция putc() записывает символы в файл, который с помощью fopen() уже открыт в режиме записи. Прототип этой функции следующий:

```
int putc(int ch, FILE * файл);
```

Указатель файла сообщает putc(), в какой именно файл следует записывать символ. Хотя ch и определяется как int, однако записывается только младший байт.

Если функция putc() выполнилась успешно, то возвращается записанный символ. В противном же случае возвращается EOF.

2.13.8. Чтение символа

Для ввода символа также имеются две эквивалентные функции: getc() и fgetc(). Обе определяются для сохранения совместимости со старыми

версиями С. В этой книге используется getc() (которая обычно реализуется в виде макроса), но если хотите, применяйте fgetc().

Функция getc() читает символы из файла, который с помощью fopen() уже открыт в режиме для чтения. Прототип этой функции следующий:

```
int getc(FILE * файл);
```

Функция getc() возвращает целое значение, но символ находится в младшем байте. Если не произошла ошибка, то старший байт (байты) будет обнулен.

Если достигнут конец файла, то функция getc() возвращает EOF. Поэтому, чтобы прочитать символы до конца текстового файла, можно использовать следующий код:

```
do
{
  ch = getc(fp);
}
while(ch!=EOF);
```

Однако getc() возвращает EOF и в случае ошибки. Для определения того, что же на самом деле произошло, можно использовать ferror().

2.13.9. Использование fopen(), getc(), putc(), и fclose()

Функции fopen(), getc(), putc() и fclose() — это минимальный набор функций для операций с файлами. Следующая программа, представляет собой простой пример, в котором используются только функции putc(), fopen() и fclose(). В этой программе символы считываются с клавиатуры и записываются в дисковый файл до тех пор, пока пользователь не введет знак доллара. Имя файла определено в программа «test.txt».

```
void main()
{
  FILE *fp;  int ch;

  if((fp=fopen("test.txt", "w"))==NULL)
  {
    printf("ERROR\n");
    return;
  }

  do {
     ch = getchar();
     putc(ch, fp);
  }
  while (ch != '$');

  fclose(fp);
}
```

Следующая программа, читает любой текстовый файл и выводит его содержимое на экран.

```
void main()
{
  FILE *fp;  int ch;

  if((fp=fopen("test.txt", "r")) == NULL)
  {
    printf("ERROR.\n");
    return;
  }
  ch = getc(fp);

  while (ch!=EOF)
  {
    putchar(ch);
    ch = getc(fp);
  }
  fclose(fp);
}
```


2.13.10. Использование feof()

Как уже говорилось, если достигнут конец файла, то getc() возвращает EOF. Однако проверка значения, возвращенного getc(), возможно, не является наилучшим способом узнать, достигнут ли конец файла. Во-первых, файловая система языка С может работать как с текстовыми, так и с двоичными файлами. Когда файл открывается для двоичного ввода, то может быть прочитано целое значение, которое, как выяснится при проверке, равняется EOF. В таком случае программа ввода сообщит о том, что достигнут конец файла, чего на самом деле может и не быть. Во-вторых, функция getc() возвращает EOF и в случае отказа, а не только тогда, когда достигнут конец файла. Если использовать только возвращаемое значение getc(), то невозможно определить, что же на самом деле произошло. Для решения этой проблемы в С имеется функция feof(), которая определяет, достигнут ли конец файла. Прототип функции feof() такой:

```
int feof(FILE * файл);
```

Если достигнут конец файла, то feof() возвращает true (истина); в противном же случае эта функция возвращает нуль. Поэтому следующий код будет читать двоичный файл до тех пор, пока не будет достигнут конец файла:

```
while(!feof(fp))
  ch = getc(fp);
```

Ясно, что этот метод можно применять как к двоичным, так и к текстовым файлам.

В следующей программе, которая копирует текстовые или двоичные файлы, имеется пример применения feof(). Файлы открываются в двоичном режиме, а затем feof() проверяет, не достигнут ли конец файла.

```
void main()
{
  FILE *in, *out;  int ch;

  if((in=fopen("in.txt", "rb"))==NULL)
  {
    printf("ERROR\n");
    return;
  }

  if((out=fopen("out.txt", "wb")) == NULL)
  {
    printf("ERROR\n");
    return;
  }

  while(!feof(in))
  {
    ch = getc(in);
    if(!feof(in)) putc(ch, out);
  }

  fclose(in);
  fclose(out);
}
```

2.13.11. Ввод / вывод строк: fputs() и fgets()

Кроме getc() и putc(), в языке С также поддерживаются родственные им функции fgets() и fputs(). Первая из них читает строки символов из файла на диске, а вторая записывает строки такого же типа в файл, тоже находящийся на диске. Эти функции работают почти как putc() и getc(), но читают и записывают не один символ, а целую строку. Прототипы функций fgets() и fputs() следующие:

```
int fputs(const char *cmp, FILE * файл);
char *fgets(char *cmp, int длина, FILE * файл);
```


Функция fputs() пишет в определенный поток строку, на которую указывает cmp (более подробно строки рассмотрены далее). В случае ошибки эта функция возвращает EOF.

Функция fgets() читает из определенного потока строку, и делает это до тех пор, пока не будет прочитан символ новой строки или количество прочитанных символов не станет равным (***длина-1***). Если был прочитан разделитель строк, он записывается в строку, чем функция fgets() отличается от функции gets(). Полученная в результате строка будет оканчиваться символом конца строки (***'0'***). При успешном завершении работы функция возвращает cmp, а в случае ошибки — пустой указатель (NULL).

В следующей программе показано использование функции fputs(). Она читает строки с клавиатуры и записывает их в файл, который называется TEST. Для завершения программы вводится пустая строка. Так как функция gets() не записывает разделитель строк, то его приходится специально вставлять перед каждой строкой, записываемой в файл для того, чтобы файл было легче читать:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void main(void)
{
  char str[80]; FILE *fp;
  if((fp = fopen("test.txt", "w"))==NULL)
  { printf("Error\n");
    return;
  }
  do {
    printf("Input string\n");
    gets(str);
    strcat(str, "\n");
    fputs(str, fp);
  } while(*str!='\n');
  fclose(fp);
}
```

2.13.12. Функции fread() и fwrite()

Для чтения и записи данных, тип которых может занимать более 1 байта, в файловой системе языка C имеется две функции: fread() и fwrite(). Эти функции позволяют читать и записывать блоки данных любого типа. Их прототипы следующие:

```
size_t fread(void *буфер, size_t колич_байт,
             size_t счетчик, FILE * файл);

size_t fwrite(const void *буфер, size_t колич_байт,
              size_t счетчик, FILE * файл);
```

Для fread() буфер — это указатель на область памяти, в которую будут прочитаны данные из файла. А для fwrite() буфер — это указатель на данные, которые будут записаны в файл. Значение счетчик определяет, сколько считывается или записывается элементов данных, причем длина каждого элемента в байтах равна колич_байт. (Вспомните, что тип size_t определяется как одна из разновидностей целого типа без знака.) И, наконец, файл — это указатель файла, то есть на уже открытый поток.

Функция fread() возвращает количество прочитанных элементов. Если программа достигла конца файла или произошла ошибка, то возвращаемое значение может быть меньше, чем счетчик. А функция fwrite() возвращает количество записанных элементов. Если ошибка не произошла, то возвращаемый результат будет равен значению счетчик.

Как только файл открыт для работы с двоичными данными, fread() и fwrite() соответственно могут читать и записывать информацию любого типа. Например, следующая программа записывает в дисковый файл данные типов double, int и long, а затем читает эти данные из того же файла. Обратите внимание, как в этой программе при определении длины каждого типа данных используется функция sizeof().


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

void main(void)
{
  FILE *fp;
  double d = 12.23;
  int i = 101;
  long l = 123023L;

  if((fp=fopen("test", "wb+"))==NULL) {
    printf("Error");
    return;
  }
  fwrite(&d, sizeof(double), 1, fp);
  fwrite(&i, sizeof(int), 1, fp);
  fwrite(&l, sizeof(long), 1, fp);

  rewind(fp);

  fread(&d, sizeof(double), 1, fp);
  fread(&i, sizeof(int), 1, fp);
  fread(&l, sizeof(long), 1, fp);

  printf("%f %d %ld", d, i, l);

  fclose(fp);
}
```

Как видно из этой программы, в качестве буфера можно использовать (и часто именно так и делают) просто память, в которой размещена переменная. В этой простой программе значения, возвращаемые функциями fread() и fwrite(), игнорируются. Однако на практике эти значения необходимо проверять, чтобы обнаружить ошибки.

Одним из самых полезных применений функций fread() и fwrite() является чтение и запись данных пользовательских типов, особенно структур. Например, если определена структура (о структурах более подробно рассказано ниже):

```
struct struct_type
{
   float balance;
   char name[80];
} cust;
```

то следующий оператор записывает содержимое cust в файл, на который указывает fp:

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

2.13.13. Функции fprintf() и fscanf()

Кроме основных функций ввода/вывода, о которых шла речь, в системе ввода/вывода языка С также имеются функции fprintf() и fscanf(). Эти две функции, за исключением того, что предназначены для работы с файлами, ведут себя точно так же, как и printf() и scanf(). Прототипы функций fprintf() и fscanf() следующие:

```
int fprintf(FILE *файл, const char *строка, ...);
int fscanf(FILE *файл, const char *строка, ...);
```

Операции ввода/вывода функции fprintf() и fscanf() выполняют операции ввода\вывода с тем файлом, который указан в параметрах этих функций.

В качестве примера предлагается рассмотреть следующую программу, которая читает с клавиатуры строку и целое значение, а затем записывает их в файл на диске; имя этого файла — **test**. После этого программа читает этот файл и выводит информацию на экран. После запуска программы проверьте, каким получится файл **test**. Как вы и увидите, в нем будет вполне удобочитаемый текст.


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

void main(void)
{
  FILE *fp;
  char s[80];
  int t;
  if((fp=fopen("test", "w")) == NULL)
  {
    printf("Error\n");
    return;
  }
  printf("Введите строку и число: ");

  /* читать с клавиатуры */
  fscanf(stdin, "%s%d", s, &t);

   /* писать в файл */
  fprintf(fp, "%s %d", s, t);
  fclose(fp);

  if((fp=fopen("test","r")) == NULL)
  {
    printf("Error\n");
    return;
  }

  fscanf(fp, "%s%d", s, &t);
  fprintf(stdout, "%s %d", s, t);
  fclose(fp);
}
```

Предупреждение: хотя читать разносортные данные из файлов на дисках и писать их в файлы, расположенные также на дисках, часто легче всего именно с помощью функций fprintf() и fscanf(), но это не всегда самый эффективный способ выполнения операций чтения и записи. Поскольку данные в формате ASCII записываются так, как они должны

появиться на экране (а не в двоичном виде), то каждый вызов этих функций сопряжен с определенными накладными расходами. Поэтому, если надо заботиться о размере файла или скорости, то, скорее всего, придется использовать fread() и fwrite().

3. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственного усложнения программного обеспечения. В 70-е годы XX века объёмы и сложность программ достигли такого уровня, что «интуитивная» (неструктурированная, или «рефлекторная») разработка программ, которая была нормой в более раннее время, перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать, поэтому потребовалась какая-то систематизация процесса разработки и структуры программ.

Наиболее сильной критике со стороны разработчиков структурного подхода к программированию подвергся оператор **GOTO** (оператор безусловного перехода), имевшийся тогда почти во всех языках программирования. Неправильное и необдуманное использование произвольных переходов в тексте программы приводит к получению запутанных, плохо структурированных программ (т.н. спагетти-кода), по тексту которых практически невозможно понять порядок исполнения и взаимозависимость фрагментов.

Перечислим некоторые достоинства структурного программирования:

1. Структурное программирование позволяет значительно сократить число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы и, что ещё важнее, облегчает понимание её другими разработчиками.
2. В структурированных программах логически связанные операторы находятся визуально ближе, а слабо связанные — дальше, что позволяет обходиться без блок-схем и других графических форм изображения алгоритмов (по сути, сама программа является собственной блок-схемой).

3. Сильно упрощается процесс тестирования и отладки структурированных программ.

Следование принципам структурного программирования сделало тексты программ, даже довольно крупных, нормально читаемыми. Серьёзно облегчилось понимание программ, появилась возможность разработки программ в нормальном промышленном режиме, когда программу может без особых затруднений понять не только её автор, но и другие программисты. Это позволило разрабатывать достаточно крупные для того времени программные комплексы силами коллективов разработчиков, и сопровождать эти комплексы в течение многих лет, даже в условиях неизбежных изменений в составе персонала.

3.1. Методология

Структурное программирование — методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков, предложенная в 70-х годах XX века Э. Дейкстрой, а разработана и дополнена Н. Виртом.

В соответствии с данной методологией

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:
    a. ***последовательное исполнение*** — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
    b. ***ветвление*** — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
    c. ***цикл*** — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

2. Повторяющиеся фрагменты программы, либо представляющие собой логически целостные вычислительные блоки, оформляются в виде функций (подпрограмм).
3. Разработка программы ведётся пошагово, методом «сверху вниз».

3.2. Создание программ методом «сверху вниз»

Для иллюстрации разберем пример: «Программа для сортировки массива данных». При этом исходный массив данных находится в файле (имя задается пользователем), а результат необходимо представить на экране компьютера (в консоли).

3.2.1. Определение данных, результата и характеристик

Как уже говорилось выше, при решении задачи необходимо в первую очередь определиться с корректной постановкой задачи. В данном случае, постановка может быть сформулирована следующими пунктами:

1. Тип данных – целые числа. Диапазон значений позволяет использовать тип int.
2. Количество данных изначально неизвестно. Признак окончания данных в файле – специальное значение ***990099.***
3. Имя файла с исходными данными задается с консоли пользователем. Файл изменению не подлежит.
4. Алгоритм сортировки – обычный пузырек (рассмотрен ниже). Направление сортировки задается пользователем.
5. Программа работает один раз, после чего завершается.

3.2.2. Блок-схема основных действий

При решении любой задачи первоочередным делом является осознание основных шагов, описывающих ее решение. Результат может быть

представлен, например, в виде блок-схемы. Ниже представлена блок схема решения поставленной задачи.

3.2.3. Реализация основных действий

После того, как основные шаги определены, необходимо описать эти действия на языке программирования. В случае языка С это означает реализовать основную функцию main().

Согласно принципам структурного программирования, на данном этапе не нужно задумываться о всех деталях реализации программы и всех используемых алгоритмах.

Реализацию этого этапа часто удобно производить в 2 этапа:

1. Описание порядка действий.
2. Создание необходимых переменных.

Описанные в блок-схеме действия могут быть реализованы либо вызовом существующей стандартной функции, либо вызовом пользовательской функции, которая будет реализована позднее.

```
  scanf("%s", FileName);
  pMas = GetMasFromFile(FileName, &size);

  scanf("%d", &SortType);
  // 1 - sort up, -1 - sort down
  SortMas(pMas, size, SortType);

  PrintMas(pMas, size);
```

После этого, необходимо еще раз проверить соответствие описанных действий блок-схеме, корректность имен переменных и функций. Затем необходимо дополнить код объявлением переменных, результатом чего станет готовая функция main().

```
void main()
{
  char FileName[20];
  int SortType, size;
  int* pMas = NULL;

  scanf("%s", FileName);
  pMas = GetMasFromFile(FileName, &size);

  scanf("%d", &SortType);
  // 1 - sort up, -1 - sort down
  SortMas(pMas, size, SortType);

  PrintMas(pMas, size);
}
```

3.2.4. Создание каркаса программы

Каркас программы представляет собой логическую заготовку, реализующую заданный функционал верхнего уровня без деталей реализации пользовательских функций (подпрограмм).

После того, как функция main() реализована, необходимо проверить корректность синтаксиса и отсутствие ошибок типизации. При наличии нереализованных функций компилятор не сможет корректно отработать и будет выдавать ошибки. Реализованные объявления функций в тексте программы необходимы для корректной работы компилятора.

Для нашей программы объявления выглядят следующим образом:

```
int* GetArrFromFile(char*, int);
void SortArr(int*, int);
void PrintArr(int*, int);
```

Свидетельством корректности написанного кода (функция main() и объявления функций) является корректная работа компилятора и отсутствие ошибок.

После этого этапа происходит написание ***заглушек*** функций – работающих подпрограмм, не реализующих функционал программы. Ниже приведет полный текст каркаса требуемой программы.

```
include <stdio.h>
int* GetArrFromFile(char*, int);
void SortArr(int*, int);
void PrintArr(int*, int);

int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  *pSize = 0;
  return pArr;
}
```



```
void SortArr(int* pArr, int size, int SortType)
{
}

void PrintArr(int* pArr, int size)
{
}

void main()
{
  char FileName[20];
  int SortType, size;
  int* pArr = NULL;

  scanf("%s", FileName);
  pArr = GetArrFromFile(FileName, &size);

  scanf("%d", &SortType);
  // 1 - sort up, -1 - sort down
  SortArr(pArr, size, SortType);

  PrintArr(pArr, size);
}
```

Признаками корректности каркаса программы являются:

1. Отсутствие ошибок при компиляции и линковке проекта
2. Успешный запуск программы
3. Соответствие действий программы с исходной блок-схемой (запросы имени и типа сортировки)
4. Отсутствие ошибок при завершении программы.

Только при корректности всех пунктов можно приступать к дальнейшей разработке программы и реализации логического функционала.

3.2.5. Реализация подпрограмм распечатки и сортировки

Реализация подпрограмм осуществляется двумя способами:

1. В случае если подпрограмма достаточно простая, то делается ее прямая последовательная реализация.
2. В случае если подпрограмма является трудоемкой и реализует какой-то алгоритм, то ее реализация должна делаться аналогичными для всей программы шагами.

Для нашей программы, функция распечатки массива является достаточно простой, поэтому реализуется напрямую:

```
void PrintArr(int* pArr, int size)
{
  int i;
  for(i = 0; i < size; i++)
    printf("%d\t", pArr[i]));
}
```

Функция сортировки тоже может быть реализована напрямую, однако согласно принципам структурного программирования логичным является выделение двух дополнительных подпрограмм, используемых при сортировке пузырьком, но при этом обладающих достаточной долей самостоятельности: функция сравнения элементов массива и функция перестановки элементов массива. В таком случае, функция сортировки пузырьком будет выглядеть следующим образом:

```
void SortArr(int* pArr, int size, int SortType)
{
  // 1 - sort up, -1 - sort down
  int n, i;
  for(n = 0; n < size; n++)
    for(i = 0; i < size-1; i++)
      if(CompareElements(pArr, i, i+1) == SortType)
        SwapElements(pArr, i, i+1);
}
```


А функции сравнения и перестановки так:

```
int CompareElements(int* pArr, int i, int j)
{
  if(pArr[i] > pArr[j])
    return 1;
  // Return -1 - if [i] < [j]
  if(pArr[i] < pArr[j])
    return -1;
  // Return 0 - if [i] = [j]
  return 0;
}

void SwapElements(int* pArr, int i, int j)
{
  int temp;
  temp = pArr[i];
  pArr[i] = pArr[j];
  pArr[j] = temp;
 }
```

Как видите, сопровождение кода комментариями позволяет существенно упростить его понимание в случае наличия неявной логики – значение переменной, определяющей направление сортировки.

3.2.6. Реализация подпрограммы чтения массива из файла

Поскольку часто чтение данных из файлов представляет собой проблему для обучающихся, существует рекомендация вначале отладить логику на примере чтения данных из консоли.

Если бы количество значений в массиве было известно заранее (например задан в тексте программы), то процесс выглядел бы следующим образом:

```
int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  int i, size = 5;

  pArr = malloc(size * sizeof(int));
  for(i = 0; i < size; i++)
    scanf("%d", (pArr+i));

  *pSize = size;
  return pArr;
}
```

Чтение данных из файла происходит аналогичным образом чтению данных из консоли:

```
int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  int i, size;
  FILE* pFile;
  size = 5;

  pArr = malloc(size * sizeof(int));

  pFile = fopen(pFileName, "r");
  for(i = 0; i < size; i++)
    fscanf(pFile, "%d", (pArr+i));
  fclose(pFile);

  *pSize = size;
  return pArr;
}
```

В данной программе большую сложность представляет чтение неизвестного количества данных из файлов. Возможно два пути решения: введение ограничения на максимальный размер данных, либо реализация считывания произвольного количества данных.

Чтение данных при ограничении максимального количества

Если условием задачи не является точное соответствие размера массива количеству данных, то допустима следующая реализация:

```
int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  int MaxSize, RealCount, temp;
  FILE* pFile;
  MaxSize = 100;

  pArr = malloc(MaxSize * sizeof(int));

  pFile = fopen(pFileName, "r");

for(RealCount = 0; RealCount < MaxSize;RealCount++)
  {
    fscanf(pFile, "%d", &temp);
    if (temp == 990099)
      break;
    *(pArr+RealCount) = temp;
  }
  fclose(pFile);

  *pSize = RealCount;
  return pArr;
  }
```

Если же условием является точное соответствие размера массива количеству данных, то необходимо сначала читать данные в некоторый временный буфер, после чего производить выделение нужного количества памяти и копировать данные:

```
int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  int RealCount, temp, i;
  int TempArr[100];
```

```
  FILE* pFile;

  pFile = fopen(pFileName, "r");
  for(RealCount = 0; RealCount < 100; RealCount++)
  {
    fscanf(pFile, "%d", &temp);
    if (temp == 990099)
      break;
    TempArr [RealCount] = temp;
  }
  fclose(pFile);

  *pSize = RealCount;
  pArr = malloc(RealCount * sizeof(int));
  for(i = 0; i < RealCount; i++)
    pArr[i] = TempArr[i];

  return pArr;
}
```

Чтение любого количества данных

Данный режим может реализовываться двумя основными способами, в зависимости от задачи:

- Чтение данных порциями. При превышении заданного размера выделение дополнительной памяти и продолжение работы. Реализация данного способа дается в качестве самостоятельной работы.
- При возможности позиционирования или нескольких чтениях входных данных реализуется двухпроходной алгоритм: сначала считается количество, затем выделяется память и происходит собственно чтение. Реализация этого способа приведена ниже.

```
int* GetArrFromFile(char* pFileName, int* pSize)
{
  int* pArr = NULL;
  int RealCount, temp;
```



```
  int TempArr [100];
  FILE* pFile;

  pFile = fopen(pFileName, "r");
  for(RealCount = 0; ; RealCount++)
  {
    fscanf(pFile, "%d", &temp);
    if (temp == 990099)
      break;
  }
  fclose(pFile);

  *pSize = RealCount;
  pArr = malloc(RealCount * sizeof(int));

  pFile = fopen(pFileName, "r");
  for(RealCount = 0; RealCount < 100; RealCount++)
    fscanf(pFile, "%d", (pArr+RealCount));
  fclose(pFile);

  return pArr;
}
```

3.2.7. Возможные ошибки и пути решения

При разработке программы описанным методом «сверху-вниз» возможно возникновение ошибок на различных стадиях. Пути их решения зависят от квалификации программиста и уровня осознания методологии структурного программирования.

Рассмотрим пример: взяв за основу корректную блок-схему, мы бы реализовали следующий каркас программы:

```
#include <stdio.h>

int* GetArrFromFile(char*);
void SortArr(int*);
void PrintArr(int*);
```

```
int* GetArrFromFile(char* pFileName)
{
  int* pArr = NULL;
  return pArr;
}
void SortArr(int* pArr, int SortType)
{
}
void PrintArr (int* pArr)
{
}
void main()
{
  char FileName[20];
  int SortType;
  int* pArr = NULL;

  scanf("%s", FileName);
  pArr = GetArrFromFile(FileName);

  scanf("%d", &SortType);
  // 1 - sort up, -1 - sort down
  SortArr (pArr, SortType);

  PrintArr (pArr);
  }
```

Данный код полностью описывает заданную блок-схему, успешно компилируется, собирается и запускается.

Проблема начинается тогда, когда происходит переход к следующей итерации: реализации подпрограмм. Именно по этой причине, ранее была реализация наиболее простых подпрограмм, начиная с функции распечатки массива.

```
void PrintArr(int* pArr)
{  int i;
  for(i = 0; i < ?????; i++)
    printf("%d\t", pArr[i]);
}
```


При реализации функции, заданным в объявлении способом, возникает вопрос: откуда взять размер массива? И именно в этот момент происходит осознание того, что не все параметры задачи были учтены при формировании требований. Единственным правильным решением в случае возникновения ошибок проектирования (параметры подпрограмм) является переделка каркаса программы с внесением необходимых корректив.

3.2.8. Сопровождение

Описанная последовательность действий при разработке программ гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна.

При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры и на каком уровне нужно внести изменения, и они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

4. СТРУКТУРЫ ДАННЫХ

4.1. Объединения

Объединениями называют сложный тип данных, позволяющий размещать в одном и том же месте оперативной памяти данные различных типов.

Размер оперативной памяти, требуемый для хранения объединений, определяется размером памяти, необходимым для размещения данных того типа, который требует максимального количества байт. Когда используется элемент меньшей длины, чем наиболее длинный элемент объединения, то этот элемент использует только часть отведенной памяти. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Общая форма объявления объединения

```
union ИмяОбъединения
{
  тип ИмяОбъекта1;
  тип ИмяОбъекта2;
  . . .
  тип ИмяОбъектаn;
};
```

Объединения применяют:

- для инициализации объекта, если в каждый момент времени только один из многих объектов является активным;

- для интерпретации представления одного типа данных в виде другого типа.

Пример программы, которая меняет 2 байта местами во введенном числе


```
union
{
   unsigned char p[2];
   unsigned int t;
} type;

int main()
{
  char temp;
  printf("Enter number: ");
  scanf("%d", &type.t);
  printf("%d = %x hex.\n",type.t, type.t);
 // Замена байтов
  temp = type.p[0];
  type.p[0] = type.p[1];
  type.p[1] = temp;
  printf("Bytes are changed.\n");
  printf("%d = %x hex\n",type.t, type.t);
  return 0;
}
```

4.2. Структуры

Структура — объединение нескольких объектов, возможно, различного типа под одним именем. Это имя является типом структуры. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.

Структуры позволяют трактовать группу связанных между собой объектов не как множество отдельных элементов, а как единое целое. Структура представляет собой сложный тип данных, составленный из простых типов. Общая форма объявления структуры:

```
struct тип_структуры
{
  тип ИмяЭлемента1;
  тип ИмяЭлемента2;
  . . .
  тип ИмяЭлементаn;
};
```

После закрывающей фигурной скобки } в объявлении структуры обязательно ставится точка с запятой. Пример объявления структуры:

```
struct date
{
  int day;      // 4 байта
  char *month; // 4 байта
  int year;     // 4 байта
};
```

Элементы, из которых состоит структура, называются полями. Поля структуры располагаются в памяти в том порядке, в котором они объявлены. В указанном примере структура date занимает в памяти 12 байт (не всегда точно 12 байт, может зависеть от компилятора и платформы). Кроме того, указатель month при инициализации будет указывать на начало текстовой строки с названием месяца, размещенной в памяти. При объявлении структур их разрешается вкладывать одну в другую, как в следующем примере.

```
struct persone
{
  char lastname[20];   // фамилия
  char firstname[20]; // имя
  struct date bd;      // дата рождения
};
```

4.2.1. Инициализация полей структуры

Инициализация полей структуры может осуществляться двумя способами:

- инициализация полей переменной-структуры при её объявлении;
- инициализация полей переменной-структуры после её объявления, например, с использованием функций ввода-вывода (printf() и scanf()).

В первом способе инициализация осуществляется по следующей схеме:

```
struct ИмяСтруктуры ИмяПеременной=
  {ЗначениеЭлемента1,
  ЗначениеЭлемента_2, . . . ,
  ЗначениеЭлементаn};
```

Пример

```
struct date bd={22,"февраля", 1998};
```

Имя элемента структуры является составным. Для обращения к полю структуры нужно указать имя структуры и имя самого поля. Они разделяются точкой: ИмяПеременной.ИмяПоляСтруктуры

Ниже приведен пример инициализации переменной типа `persone` с использованием ввода-вывода.

```
struct persone p;
printf("Enter name: ");
scanf("%s",p.firstname);
printf("Enter surname: ");
scanf("%s",p.lastname);
printf("Enter birthday\nDay: ");
scanf("%d",&p.bd.day);
printf("Month: ");
scanf("%s",p.bd.month);
printf("Year: ");
scanf("%d",&p.bd.year);
printf("\nYou entered: %s%s, Birthday %d%s%d",
     p.firstname, p.lastname,
     p.bd.day, p.bd.month, p.bd.year);
```

.

4.2.2. Массивы структур

Работа с массивами структур аналогична работе со статическими массивами других типов данных.

Пример программы, использующей массив структур

```
struct book
{
  char title [15];
  char author [15];
  int value;
};

int main()
{
  struct book libry[3];
  int i;
  // считываем данные с клавиатуры
  for(i=0;i<3;i++)
  {
    printf("Enter title of %d book:",i+1);
    gets(libry[i].title);
    printf("Enter author of %d book: ",i+1);
    gets(libry[i].author);
    printf("Enter value of %d book: ",i+1);
    scanf("%d",&libry[i].value);
    getchar();
  }
  // выводим результат на экран
  for(i=0;i<3;i++)
  {
    printf("\n %d. %s ", i+1,libry[i].author);
    printf("%s
%d",libry[i].title,libry[i].value);
  }
  return 0;
}
```

4.2.3. Указатели на структуры и динамическая память

Доступ к элементам структуры или объединения можно осуществить с помощью указателей. Для этого необходимо инициализировать указатель адресом структуры или объединения.

Для организации работы с массивом можно использовать указатель p или имя массива:

выражение->идентификатор

(*выражение).идентификатор

выражение — указатель на структуру или объединение;

идентификатор — поле структуры или объединения;

Динамически выделять память под массив структур необходимо в том случае, если заранее неизвестен размер массива. Для определения размера структуры в байтах используется операция sizeof(имя структуры).

Пример программы, использующей динамический массив структур.

```
struct book
{char title [15];
 char author [15];
 int value;
};
int main()
{ struct book *lib;
  int i;
  lib = malloc(3*sizeof(struct book));
  for(i=0;i<3;i++)
  {
    printf("Enter title of %d book:",i+1);
    gets((lib+i)->title);
    printf("Enter author of %d book: ",i+1);
    gets((lib+i)->author);
    printf("Enter value of %d book: ",i+1);
    scanf("%d",&(lib+i)->value);
    getchar();
  }
  for(i=0;i<3;i++)
  { printf("\n %d. %s ", i+1, (lib+i)->author);
    printf("%s%d", (lib+i)->title, (lib+i)->value);
  }
  free(lib);
  return 0;
}
```

Результат выполнения аналогичен предыдущему примеру.

4.2.4. Использование typedef

Синтаксис языка С предусматривает использование спецификатора typedef для определения новых имен типов данных (тегов). Спецификатор typedef можно использовать как со стандартными типами данных, так и со структурами, объединениями и перечислениями.

Для объявления нового типа, формат которого будет совпадать с типом int можно использовать конструкцию:

```
typedef int MyInt;
```

Теперь можно использовать переменные типа MyInt:

```
MyInt i = 0;
printf("MyInt=%d", i);
```

Аналогично, спецификатор typedef можно использовать со структурами:

```
struct MyStruct
{
   int i;
   float v;
};
typedef struct MyStruct MyStruct;

void main()
{
   MyStruct m;
   m.i = 10;
   m.v = 0.3;
}
```

Более того, объявление структуры MyStruct и создание тега MyStruct можно объединить:


```
typedef struct MyStruct
{
   int i;
   float v;
} MyStruct;
```

В результате далее можно не писать слово struct при использовании созданной структуры.

4.3. Списки данных

4.3.1. Методы организации и хранения списков

Список – конечная последовательность однотипных элементов (узлов), возможно, с повторениями. Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться.

При работе со списками на практике чаще всего приходится выполнять следующие операции:

- найти элемент с заданным свойством;
- определить первый элемент в списке;
- вставить дополнительный элемент до или после указанного узла;
- исключить определенный элемент из списка;
- расставить узлы списка в определенном порядке.

В языке программирования С нет какой-либо стандартной структуры данных для представления списка. Более того, не существует такой реализации списка, в которой бы все указанные операции над ним выполнялись в одинаковой степени эффективно. Поэтому при работе со списками важным является представление используемых в программе списков таким образом, чтобы была обеспечена максимальная эффективность по времени выполнения программы и по объему требуемой памяти.

Основное отличие списков от массивов заключается в методе хранения данных в памяти: в случае массива данные хранятся в памяти последовательно и для доступа к следующему элементу достаточно изменить адрес соответствующего указателя на размер элемента массива. В списках используют принцип связанного хранения данных: в качестве элементов хранения используются структуры, связанные по одной из компонент в цепочку, на начало которой (первую структуру) указывает указатель Head. Таким образом, структура, образующая элемент хранения, должна кроме соответствующего элемента списка содержать и указатель на соседний элемент хранения.

Описание структуры и указателя в этом случае может иметь вид:

```
struct DL // структура элемента хранения
{
   float  val;     // элемент списка
   struct DL *n ; // указатель на элемент хранения
};
DL *p;               // указатель текущего элемента
DL *dl;              // указатель на начало списка
```

4.3.2. Односвязный список

В простом односвязном списке каждый элемент представляет собой структуру nd, состоящую из двух элементов:

val - предназначен для хранения элемента списка,

n - для указателя на структуру, содержащую следующий элемент списка.

На первый элемент списка указывает указатель dl:

```
struct ND
{
   float val;
   struct ND * n;
};
ND * dl;
```


Для реализации операций могут использоваться следующие примеры программ:

1) печать значения i-го элемента

```
ND * r = dl;
int j = 1;
if (r==NULL)
{
   printf("\nEmpty");
}
else
{
   while(r!=NULL)
   {
      printf("\n %d",r->val);
      r=r->n;
   }
}
```

2) печать обоих соседей узла (элемента), определяемого указателем p

```
ND * r = p;
if ((r->n)==NULL)
{
   printf("\n No right neighbor");
}
else
{
   printf("\n  right neighbor %f",  r->n->val);
}
if (dl==p)
{
   printf("\n No left neighbor");
}
else
{
   r=dl;
   while( r->n!=p )
   {r=r->n;}
   printf("\n left neighbor %f", r->val);
}
```

3) удаление элемента, следующего за узлом, на который указывает p

```
ND * r = p->n;
if (r==NULL) printf("\n  Nothing to delete");
else
{
   p->n=r->n;
   free(r);
}
```

4) вставка нового узла со значением new_val за элементом, определенным указателем p

```
ND * r = (ND*) malloc(sizeof(ND));
r->n=p->n;
r->val=new_val;
p->n=r;
```

4.3.3. Двусвязный список

Связанное хранение линейного списка называется списком с двумя связями или двусвязным списком, если каждый элемент хранения имеет два компонента указателя (ссылки на предыдущий и последующий элементы линейного списка).

В программе двусвязный список можно реализовать с помощью структуры ndd:

```
typedef struct ndd
{
   float val;      /* значение элемента */
   struct ndd * n; /* указатель на след. элемент */
   struct ndd * m; /* указатель на пред. элемент */
} NDD;
NDD * dl, * p, * r;
```

Вставка нового узла со значением new за элементом, определяемым указателем p, осуществляется при помощи операторов:


```
void add(int new_val)
{
   r=malloc(NDD);
   r->val=new_val;
   r->n=p->n;
   (p->n)->m=r;
   p->n=r;
}
```

Удаление элемента, следующего за узлом, на который указывает p

```
void rem()
{
   r=p->n;
   p->n=(p->n)->n;
   free(r);
}
```

4.3.4. Стеки и очереди

В зависимости от метода доступа к элементам линейного списка различают разновидности линейных списков называемые стеком, очередью и двусторонней очередью.

Стек - это конечная последовательность некоторых однотипных элементов - скалярных переменных, массивов, структур или объединений, среди которых могут быть и одинаковые. Стек представляет динамическую структуру данных; ее количество элементов заранее не указывается и в процессе работы, как правило изменяется. Если в стеке элементов нет, то он называется пустым.

Типичные операциями над стеком:

- проверка стека на пустоту S=< >,

- добавление нового элемента Sn+1 в конец стека - преобразование < S1,...,Sn> в < S1,...,Sn+1>;

- изъятие последнего элемента из стека - преобразование < S1,...,Sn-1,Sn> в < S1,...,Sn-1>;

- доступ к последнему элементу Sn, если стек не пуст.

Таким образом, операции добавления и удаления элемента, а также доступа к элементу выполняются только в конце списка. Стек можно представить как стопку книг на столе, где добавление или взятие новой книги возможно только сверху.

Очередь - это линейный список, где элементы удаляются из начала списка, а добавляются в конец списка (как обыкновенная очередь в магазине).

Двусторонняя очередь - это линейный список, у которого операции добавления и удаления элементов и доступа к элементам возможны как в начале, так и в конце списка. Такую очередь можно представить как последовательность книг стоящих на полке, так что доступ к ним возможен с обоих концов.

Реализация стеков и очередей в программе может быть выполнена в виде последовательного или связанного хранения.

ЗАДАЧИ

1. Программа решения квадратного уравнения

Описание

Напишите программу, которая вычисляет корни квадратного уравнения. Поведение программы должно зависеть от варианта решения:

- уравнение имеет 2 вещественных корня;
- уравнение имеет 1 вещественный корень;
- уравнение не имеет решений;
- любое вещественное значение будет корнем данного уравнения.

Коэффициенты задаются пользователем с клавиатуры (в консоли) в виде целых значений. Корни уравнения ищем в виде вещественных чисел.

Основные требования к реализации

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь. После расчета корней квадратного уравнения спрашивает у пользователя, хочет ли он решить еще одно уравнение.

Теория

1. Повторите разделы теоретической части:
   a. Типы данных: целочисленные и вещественные. Спецификаторы для ввода-вывода
   b. Оператор ветвления IF-ELSE
   c. Использование IF в качестве условного оператора
   d. Оператор цикла FOR.
   e. Разбор блок схемы работы цикла FOR (в каких местах стоят поля)
   f. Циклы WHILE и DO-WHILE как частные случаи for (разбор блок-схем)

g. Правила форматирования кода, использование {}

2. Особенности сравнения вещественных чисел в языке С

Выполнять сравнение на равенство вещественных чисел в языке С в общем случае некорректно. Это связано с особенностями представления чисел с плавающей точкой в памяти. Рекомендуется избегать сравнения на равенство чисел с плавающей точкой. Используйте сравнение на больше или меньше. Если же вам необходимо выполнить сравнение двух вещественных чисел – используйте следующий подход: выберите значение погрешности eps, которую вы допускаете в своей программе при определении равны числа или нет. Тогда числа f1 и f2 будут считаться равными, если будет выполняться условие

(abs(f1-f2)<eps)

Здесь abs возвращает модуль выражения (f1-f2).

3. Особенности оператора деления в языке С: (int)/(int), (float)/(int)

При делении целочисленных результат - целое, даже если вы указываете в качестве результата число с плавающей точкой. Например, float a = 5/2; здесь a равно 2. Для решения данной проблемы можно использовать приведение типов: (float) a / b .

4. Правила именования переменных

Язык С позволяет давать переменным практически любые имена. Тем не менее, существует ряд правил, которых рекомендуется придерживаться для того, чтобы программный код был более прост для понимания:

– названия переменных не должны быть слишком длинными

– названия переменных должны характеризовать свойства или характер значения, который хранит эта переменная в контексте логики всей программы

– переменных, состоящих из одного символа, следует избегать, за исключением временных(одноразовых) переменных. Общие имена для временных переменных: i, j, k, m. Общие имена для числовых переменных: n. Общие имена для символьных переменных: c, d.

– рекомендуется не использовать в качестве имени переменной «e». С переменной e часто возникает сложно отлавливаемых ошибок, связанных с тем, что e – показатель степени в числах с плавающей точкой.

5. Подключение математической библиотеки и использование функции извлечения квадратного корня

Для вычисления квадратного корня можно использовать стандартную функцию sqrt . Для этого вам необходимо подключить библиотеку math . h следующим образом: # include < math . h > .

6. Основные ошибки компиляции: variable not defined, отсутствие символа «;»

При написании первых программ очень, наиболее часто возникающие ошибки компиляции:

variable not defined… - переменная используется, но не была объявлена. Необходимо в начале функции сделать объявление всех используемых переменных.

отсутствие символа «;» - в конце каждого вызова необходимо ставить символ “;”. Исключение составляют строки, после которых идет открывающая скобка (объявление функции, оператор ветвления или цикла)

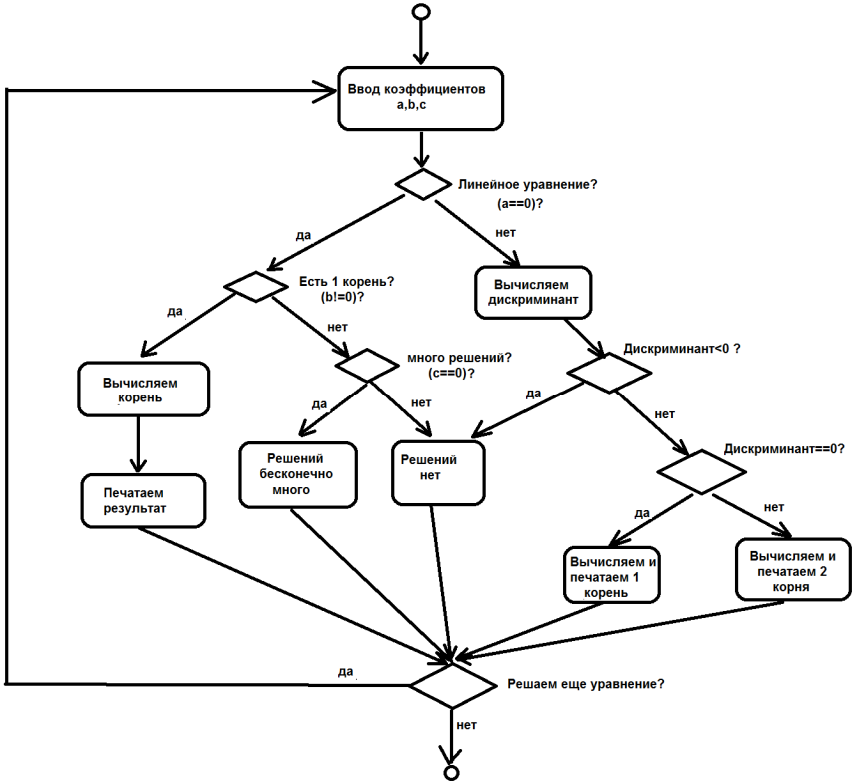
7. Ошибка линковки: не подключенная библиотека math.h

Ошибка встречается в случае неправильной настройки линковщика в вашем проекте.

8. Ошибка по ходу исполнения – деление на ноль.

В программе необходимо исключить операцию деления на 0. В противном случае вы получите некорректное значение переменой. Для этого рекомендуется всегда проверять значение знаменателя при делении.

9. Блок-схема работы программы



Вопросы

1. Измените программу так, чтобы пользователь задавал коэффициенты уравнения в виде вещественных чисел. Какие потенциальные проблемы могут возникнуть при проверке, является ли уравнение линейным?
2. Что изменится в программе с точки зрения основных блоков схемы, если искать корни в виде комплексных чисел?

2. Печать всех простых чисел не превышающих N

Условие

Вывести на экран все простые числа меньше N, которое введено пользователем.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какой параметр должен ввести пользователь. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить поиск простых чисел для другого N.

Теория

1. Повторите разделы теоретической части:
   a. Типы данных: целочисленные и вещественные. Спецификаторы для ввода-вывода
   b. Оператор ветвления IF-ELSE
   c. Использование IF в качестве условного оператора
   d. Оператор цикла FOR.
   e. Разбор блок схемы работы цикла FOR (в каких местах стоят поля)
   f. Циклы WHILE и DO-WHILE как частные случаи for (разбор блок-схем)
2. Вложенные циклы

В языке С нет особого подхода к вложенным циклам. Как и в других языках программирования, в С рассматриваются внутренние и внешние

циклы. Их называют вложенными. Если один цикл находится внутри другого цикла, то первый цикл называют внутренним, а второй – внешним.

В реализации данной программы необходимо использовать вложенные циклы: внешний цикл будет проходить по всем целым числам от 2 до веденного пользователем, а внутренний будет считать количество делителей у каждого числа внешнего цикла.

3. Определение, является ли число m делителем числа n

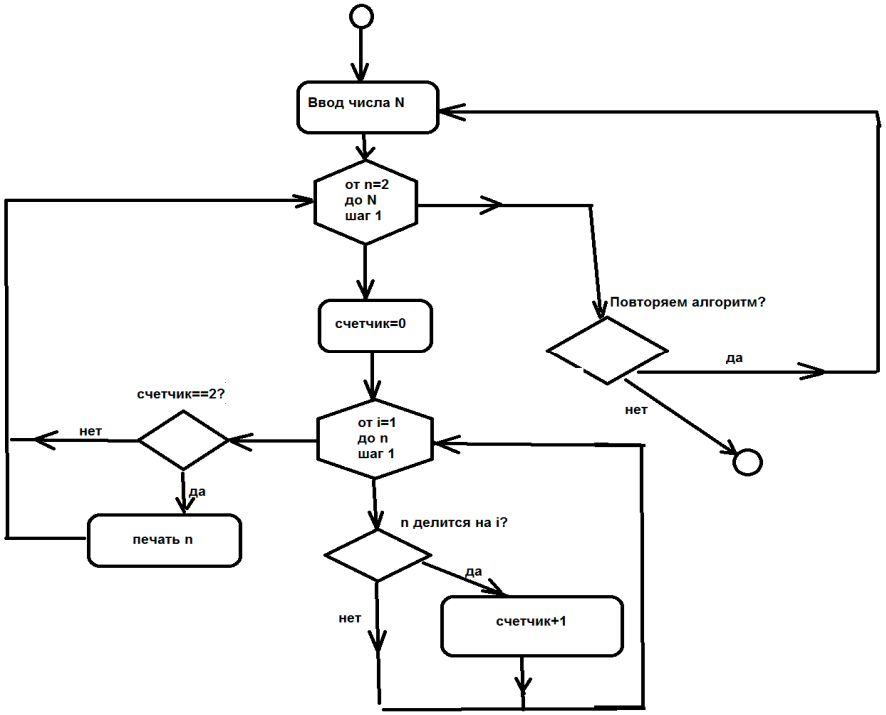
Для определения является ли число m делителем числа n, достаточно проверить остаток от деления n на m. Для этого можно использовать оператор %.

Условие, что m является делителем n: (n%m==0)

4. Алгоритм проверки является ли число простым.

Программа должна определять, является ли число n простым. Для этого необходимо найти все делители от 1 до n. Если количество делителей равно 2, значит n – простое число. Для этого необходимо завести переменную (count), которая будет хранить количество делителей числа n и реализовать цикл, который будет проходить все значения от 1 до n. на каждом шаге цикла программа должна проверять, является ли текущая переменная цикла делителем числа n. Если она является делителем, то счетчик делителей увеличивается на 1.

5. Блок-схема работы программы



Вопросы

1. Предложите оптимизацию программы в части определения является ли число простым (уменьшение шагов цикла)

2. Реализуйте проверку делимости двумя способами: через проверку остатка от деления и через приведение типов.

3. Реализуйте циклы через оператор for и while

3. Вычислить число π с заданной точностью (кол-во знаков после запятой), используя ряд Грегори

Описание

Посчитать число Пи с заданной пользователем точностью.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какой параметр должен ввести пользователь. Далее выполняется расчет числа Пи, используя конечное количество членов ряда Грегори и выводит результат пользователю в виде числа с плавающей точкой с заданным пользователем количеством значащих цифр после запятой. (если пользователь ввел рассчитать число пи до 3-го знака, то должен увидеть на экране результат в виде «3.141» или «3.142»). После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет числа Пи с другим значением точности.

Программа должна проверять вводимые пользователем значения на корректность: необходимо, чтобы число было больше 0 и меньше разумного значения, определяемое размерностью переменных, в которых выполняются вычисления. В случае неверного ввода – программа должны сообщить пользователю об ошибке и попросить ввести число еще раз.

Теория

1. Формула для вычисления числа Пи

В 1671 году Джеймс Грегори установил

$$\theta = tg\,\theta - \frac{1}{3}tg^3\theta + \frac{1}{5}tg^5\theta - \frac{1}{7}tg^7\theta + \frac{1}{9}tg^9\theta \pm \dots$$

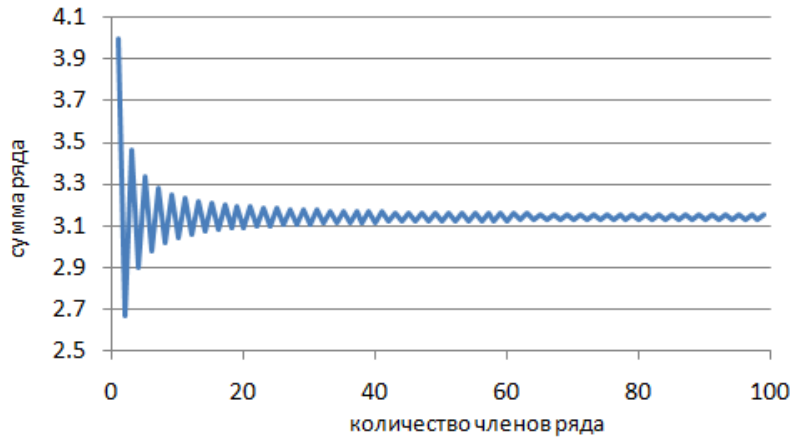
Опираясь на данное выражение Лейбниц получил:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \pm \dots$$

Или, если умножить данный ряд на 4, получим:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} \pm \dots$$

С увеличением количества членом ряда в вычислении, будем получать значения, все более приближающиеся к значению числа Пи:



2. Вывод на экран заданного количества знаков переменной после запятой

При выводе результата используйте функцию printf в следующей виде

printf(“%.nf”, val);

n – число знаков после запятой, которые необходимо вывести пользователю, val – переменная, значение которой необходимо вывести на

экран. К примеру, если хотите вывести 3 знака после запятой, используйте printf(“%.3f”, val);

3. Комментарии к написанию программы

Программа должна состоять из следующих основных шагов, которые должны быть «обернуты» циклом, позволяющим повторить вычислении числа Пи с другой точностью:

- ввод пользователем количества знаков после запятой;
- расчет значения «порога» до которого необходимо выполнять расчет числа Пи;
- вычисление числа Пи с заданной точностью;
- вывод результата в нужном формате;

Вопросы

- Предложите, как сделать вывод нужного количества знаков после запятой без использования формата ”%.nf” функции printf
- докажите корректность вашего критерия остановки вычисления числа Пи
- какая зависимость между количеством вычисляемых знаков после запятой в числе Пи и числом членов ряда, которые должны участвовать в расчете?
- если вы потребуете достаточно высокую точность (порядка 9 знаков и больше), программа начнет выдавать неправильное значение либо войдет в бесконечный цикл (в зависимости от реализации). Объясните, что в первую очередь ограничивает количество знаков, которые можно задать.

4. Программа для решения уравнения вида F(x) = 0 методом Ньютона

Описание

Вычислить корень уравнения методом Ньютона или методом касательных.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь: начальное приближение и точность, с которой надо найти решение. Далее выполняется вычисление корня введенного уравнения с указанной точностью. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет с другими параметрами.

Функция, которую будет решать программа, задает преподаватель.
Программа может содержать 4 функции main, solve, func, d_func.
Функция main выполняет ввод\вывод параметров, информации и результат работы программа на консоль. Функция Solve выполняет поиск решения. Имеет вид:
float solve(float x0, float accuracy)
Функция и ее производная задаются в тексте программы в виде отдельных функций

float func(float x)
float d_func(float x)

Теория

1. Повторите теоретические разделы
   a. Локальные и глобальные переменные. Пересечение имен.
   b. Объявление и реализация собственных функций. Параметры функции – локальные переменные.
   c. Особенности применения переменных типа double и float в вычислительных задачах.
2. Метод Ньютона

Метод Ньютона позволяет выполнить численный поиск решения уравнения вида

$$f(x)=0$$

на отрезке [a,b]. Метод работает при условии непрерывности и гладкости функции f на отрезке [a,b]. При наличии хорошего приближения x_k к корню x функции f для уточнения или более точного вычисления корня уравнения применяют следующее линейное уравнение, которое верно в малой окрестности корня:

$$f(x_k) + f'(x_k)(x - x_k) = 0$$

Решение этого уравнения принимают за очередное приближение к искомому корню уравнения:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Геометрическая интерпретация метода выглядит следующим образом:

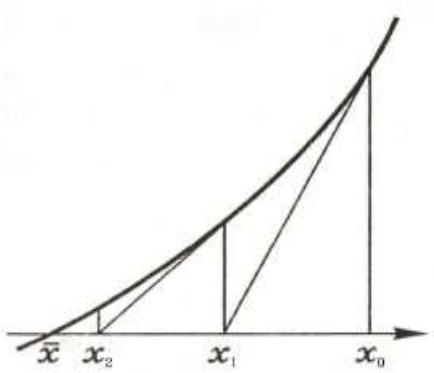


График функции заменяют касательной к нему в точке $(x_k, f(x_k))$ и за очередное приближение x_{k+1} принимают абсциссу ее точки пересечения с осью OX. Используя эти интерпретации легко получить расчетные формулы метода Ньютона.

Вопросы

- докажите, что ваше решение найдено с заданной точностью

- предложите решение методом Ньютона без функции d_func. Какие ограничения у такого алгоритма? Чем он лучше или хуже классического метода Ньютона?
- Проверьте полученный результат с помощью открытых ресурсов в Интернете.
- Придумайте, как сообщить функции solve или main информацию о том, что расчет функции или ее производной невозможен (например, потребовалось взять корень из отрицательного числа)

Приложение

Примеры функций для поиска корня

1. $x - \sin x = 0.25;$
2. $x^3 = e^x - 1;$
3. $\sqrt{x} - \cos x = 0;$
4. $x^2 + 1 = \arccos x;$
5. $\lg x - \frac{7}{2x+6} = 0;$
6. $\operatorname{tg}(0.5x + 0.2) = x^2;$
7. $3x - \cos x - 1 = 0;$
8. $x + \lg x = 0.5;$
9. $x^2 = \arcsin(x - 0.2);$
10. $x^2 + 4\sin x = 2;$
11. $\operatorname{ctg} x - x^2 = 0;$
12. $\operatorname{tg} x = \cos x - 0.1;$
13. $x \ln(x+1) - 0.3 = 0;$
14. $x^2 - \sin 10x = 0;$
15. $\operatorname{ctg} x = x;$
16. $\operatorname{tg} 3x + 0.4 = x^2;$
17. $x^2 + 1 = \operatorname{tg} x;$
18. $x^2 - 1 = \ln x;$
19. $0.5^x + 1 = (x-2)^2;$
20. $(x+3)\cos x = 1;$
21. $x^2 \cos 2x = -1;$
22. $\cos(x + 0.3) = x^2;$
23. $2^x(x-1)^2 = 2;$
24. $x \ln(x+1) = 0.5.$

5. Вычисление интеграла функции F(x) методом трапеций

Описание

Вычислите интеграл заданной функции F(x) методом трапеций.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь: пределы интегрирования, число разбиений. Далее выполняется вычисление интеграла. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет с другими параметрами.

Функция F(x) задается преподавателем. Пределы интегрирования a, b и число разбиений N интервала интегрирования задается с клавиатуры.

Теория

Для вычисления интеграла

$$I = \int_a^b f(x)dx$$

методом трапеций интервал (a, b) разбивается на N одинаковых отрезков размером $h = (b - a)/N$. В пределах каждого отрезка функция считается линейной, поэтому ее интеграл равен площади трапеции

$$S_i = h \cdot (f(x_i) + f(x_{i+1}))/2, \quad x_i = a + i \cdot h, \quad i = 0, \dots, N$$

Просуммировав площади всех трапеций, получаем численное значение интеграл :

Для вычисления значения функции f(x) определите в программе отдельную функцию. Вычисление суммы реализуйте в основной программе.

Вопросы

1. Проверьте полученный результат с помощью открытых ресурсов в Интернете.
2. Напишите программу так, чтобы значение функции в каждой точке x_i вычислялось только один раз.
3. *Оцените точность вычисленного значения. Как быстро точность улучшается при увеличении числа разбиений?

Указание. Для оценки точности можно воспользоваться неравенством:

$$\Delta I \le \frac{h^2(b-a)}{12} \max_{x\in[a,b]} |f''(x)|$$

$$I = \int_a^b f(x)dx \approx \frac{b\ a}{N} \sum_{i=0}^{N} \frac{f(x_i) + f(x_{i+1})}{2},$$

$$x_i = a + i \cdot \frac{b-a}{N}$$

Вторую производную вычислите аналитически и определите в программе отдельную функцию для вычисления ее значения. В то же цикле, в котором вычисляется основная сумма (1), вычислите и максимальное абсолютное значение второй производной.

6. Программа по вычислению максимума, минимума, среднего значения, среднеквадратичного отклонения во введенном статическом массиве

Описание

Программа для вычисления максимального, минимального, среднего значения и среднеквадратичного отклонения во введенном статическом массиве.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь: количество элементов в массиве и значения элементов. Далее выполняется вычисление параметров введенного массива. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет с другими параметрами.

Максимальный размер массива задается #define и сообщается пользователю на старте программы.

Хранение массива необходимо выполнять в виде глобальной переменной.

Все вычислительные операции с массивом необходимо делать в отдельных функциях

Не используйте указатели в программе, используйте оператор []

Теория

Формула для вычисления среднеквадратичного отклонения

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}.$$

Вопросы

- сколько раз ваша программа проходит по массиву?

- Какое наименьшее число раз необходимо пройти, если бы была поставлена задача минимизировать этот параметр?

- Реализуйте программу так, чтобы среднее значение и среднеквадратичное отклонение вычислялись в едином цикле – одном проходе через массив. Тогда вам придется реализовать функцию, которая вернет два значения. Как это сделать?

7. Программа по вычислению максимума, минимума, среднего значения, среднеквадратичного отклонения во введенном динамическом массиве

Описание

Программа для вычисления максимального, минимального, среднего значения и среднеквадратичного отклонения во введенном динамическом массиве.

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь: количество элементов в массиве и значения элементов. Далее выполняется вычисление параметров введенного массива. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет с другими параметрами.

Размер массива задается пользователем

Целочисленные элементы массива задаются пользователем с клавиатуры

Выделение и освобождение памяти в функции main()

Все вычислительные операции с массивом необходимо делать в отдельных функциях

Работа с элементами через указатели, не используйте оператор доступа к элементу массива []

8. Сортировка введенного динамического массива

Описание

Реализовать алгоритм «пузырек»сортировки динамического массива

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, какие параметры должен ввести пользователь: количество элементов в массиве, диапазон значений элементов, направление сортировки. Далее выполняется заполнение массива случайными элементами, вывод несортированного массива, затем сортировка и вывод отсортированного массива. После вывода результатов программа спрашивает у пользователя, хочет ли он повторить расчет с другими параметрами.

Реализация единой функции сортировки sort_array() в соответствии с математической моделью, за счет выделения функций compare() и replace()

Реализуйте в программе несколько функций сравнения, для различных алгоритмов сортировки (по убыванию, по возрастанию, по абсолютному значению), и передайте указатель на соответствующую функцию сравнения в единую функцию сортировки

Теория

1. Передача указателей на функции

В языке С можно передавать указатель на функцию в качестве параметра в другую функцию. Рассмотрим пример

```
double f1(double x)
{
 return x+5.0;
}

void function(double f(double))
{
```



```
 printf("%f", f(0));
}

int main()
{
 function(sin);
 function(cos);
 function(f1);
 return 0;
}
```

В этом примере в функцию function передается указатель на функцию «double f(double)». При вызове функции function в качестве параметра передается название функции, которая имеет такую же сигнатуру, как и f в объявлении функции function. В данном примере передавались математические функции sin, cos из библиотеки math.h и созданная пользователем функция f1.

Аналогично можно передавать в функции и другие функции с самыми разнообразными определениями.

Вопросы

- Оцените вычислительную сложность алгоритма сортировки
- Предложите варианты оптимизации алгоритма сортировки

9. Обработка текста введенного пользователем

Описание

Программа обработки текста, хранящегося в памяти в статическом массиве

Основные требования к реализации:

Программа имеет «дружественный интерфейс». Вначале программа печатает информацию о себе, затем сообщает, что должен ввести

пользователь. После вывода результатов программа спрашивает у пользователя, хочет ли он ввести другую строку.

Текст хранится в статическом массиве заданного размера, определенного макросом #define.

Необходимо реализовать функцию коррекции текста, которая поддерживает следующие правила:

1) большая буква только в начале предложения или в начале текста,
2) удаление нескольких пробелов, следующих подряд,
3) удаление пробелов перед точкой и запятой,
4) добавление пробела после знаков препинания.

При работе функции коррекции необходимо свести использование дополнительной памяти до постоянного размера, не зависимо от длины введенной строки.

Программа может состоять из таких функций:

main – выполняет ввод/вывод данных пользователю и вызов функции коррекции

correct_string – выполняет коррекцию введенной строки. Использует функции replace_symb, delete_symb, insert_symb

replace_symb – заменяет один символ в строке другим

delete_symb – удаляет символ из заданной позиции строки

insert_symb – вставляет символ в заданную позицию строки

Вопросы

1. Символьный тип char. Таблица ASCII. Ввод / вывод (%c)
2. Строка как массив символов. ‘\0’. (спецификатор %s)
3. Предложите, как можно оптимизировать функцию корректировки

10. Реализация функции ввода текста произвольного размера: char* GetTextFromConsole()

Описание

Реализуйте функцию ввода строки произвольного размера.

Основные требования к реализации:

Функция должна в и итоге выделить ровно столько памяти, сколько символов ввел пользователь. Функцию realloc использовать нельзя. Выделение памяти нужно организовать блочно: выделяется массив длиной К символов. Если пользователь вводит К-й символ, то функция выделяет новый массив, который на К символов длиннее, копирует в него введенные данные и освобождает старый массив. В конечном итоге будет массив длиной n*K символов, в котором заполнены не все элементы. Программа должна выделить массив нужной длины, скопировать в него введенные данные, освободить «длинный массив» и вернуть из функции массив нужной длины.

Для определения конца строки используйте символ #.

Программа должна поддерживать работу функций корректировки строки из предыдущего задания.

Программа должна уметь считывать строку не только из консоли, но и из файла (способ ввода выбирает пользователь)

Программа может содержать эти функции:

console_input - выполняет «посимвольный» ввод данных с консоли. в случае нехватки памяти вызывает функцию reallocate_buffer. После ввода строки, вызывает функцию trim_buffer для «отрезки» лишней выделенной памяти

file_input - выполняет «посимвольный» ввод данных из файла. в случае нехватки памяти вызывает функцию reallocate_buffer. После ввода строки, вызывает функцию trim_buffer для «отрезки» лишней выделенной памяти

main – выводит информацию о программе, запрашивает у пользователя способ ввода и вызывает соответствующую функцию. После выполняет корректировку строки, выводит ее на экран и освобождает память.

reallocate_buffer – выделяет буфер под вводимые данные большего размера, чем текущий, копирует туда строку и освобождает использованный, заполненный буфер

trim_buffer – выделяет буфер под введенные данные ровно того размера, какой нужен и освобождает данные в использованном до этого массиве

Вопросы

Предложите оптимизацию функции, перевыделяющей память

11. Работа с матрицами 3x3

Описание

Программа, выполняющая арифметические операции с матрицами размера 3x3

Основные требования к реализации:

Программа имеет дружественный интерфейс. Матрицы хранятся в виде структур с полем в виде двумерного статического массива. Исходная матрица заполняется целыми числами.

Необходимо поддержать возможность чтения матрицы из консоли и из файла, который введет пользователь

Программа поддерживает операции: сумма, произведение, определитель и обратная матрица

Результат вычисления сохраняется в файл (по желанию пользователя)

Теория

Определитель матрицы

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \cdot a_{22} \cdot a_{33} + a_{12} \cdot a_{23} \cdot a_{31} + a_{13} \cdot a_{21} \cdot a_{32} -$$

$$- a_{13} \cdot a_{22} \cdot a_{31} - a_{11} \cdot a_{23} \cdot a_{32} - a_{12} \cdot a_{21} \cdot a_{33}$$

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} \cdot a_{22} - a_{12} \cdot a_{21}$$

Обратная матрица

$$A^{-1} = \frac{1}{|A|} \begin{bmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} \\ \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{bmatrix}$$

Умножение матриц

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \cdot \begin{vmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{vmatrix} =$$

$$= \begin{vmatrix} a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31} & a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32} & \cdot b_{13} + a_{12} \cdot b_{23} + a_{13} \cdot b_{33} \\ a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31} & a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32} & \cdot b_{13} + a_{22} \cdot b_{23} + a_{23} \cdot b_{33} \\ a_{31} \cdot b_{11} + a_{32} \cdot b_{21} + a_{33} \cdot b_{31} & a_{31} \cdot b_{12} + a_{32} \cdot b_{22} + a_{33} \cdot b_{32} & \cdot b_{13} + a_{32} \cdot b_{23} + a_{33} \cdot b_{33} \end{vmatrix}$$

Сложение матриц

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{pmatrix}$$

Вопросы

Докажите, что обратная матрица в вашей программе вычисляется верно

12. Работа с матрицами произвольного размера

Описание

Программа, выполняющая арифметические операции с матрицами размера NxM

Основные требования к реализации:

Программа имеет дружественный интерфейс. Матрицы хранятся в виде структур с полем в виде динамического массива. Исходная матрица заполняется целыми числами. Входные данные проверяются на корректность. Все результаты вычислений возвращаются в корректном виде (деление на 0 исключить).

Необходимо поддержать возможность чтения матрицы из файла, который введет пользователь, либо заполняться случайными числами

Программа поддерживает операции: сумма, произведение, определитель и обратная матрица

Результат вычисления сохраняется в файл (по желанию пользователя)

Определитель вычисляется рекурсивным алгоритмом

Программа имеет следующие обязательные функции

1. Функция init_matrix задает размер и выделяет память.
2. Функция print_matrix выводит матрицу на экран в табличном виде
3. Функция sum_matrix суммирует матрицы и возвращает результат.
4. Функция mult_matrix вычисляет произведение матриц
5. Функция det_matrix вычисляет детерминант
6. Функция inv_matrix вычисляет обратную матрицу
7. Функция free_matrix задает размер и выделяет память.

Теория

Вычисление обратной матрицы

$$A^{-1} = \frac{\mathrm{adj}(A)}{\det(A)}$$

adj(A)=C* – присоединенная матрица – составленная из алгебраических дополнений для соответствующих элементов транспонированной матрицы.

$$C^* = \begin{pmatrix} A_{11} & A_{21} & \cdots & A_{n1} \\ A_{12} & A_{22} & \cdots & A_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ A_{1n} & A_{2n} & \cdots & A_{nn} \end{pmatrix}$$

Aij – алгебраическое дополнение к элементам исходной матрицы

$$A_{ij} = (-1)^{i+j} M_{ij},$$

Mij – дополнительный минор, получающиися их исходной матрицы путем вычеркивания i-й строки и j-го столбца

Вычисление определителя матрицы

$$\Delta = \sum_{j=1}^{n} (-1)^{1+j} a_{1j} \bar{M}_j^1,$$

Вопросы

- Докажите, что обратная матрица в вашей программе вычисляется верно
- Опишите по шагам алгоритм вычисления обратной матрицы размером 4x4

13. Телефонная книга

Описание

Написать программу, реализующую односвязный список с функциями добавления, удаления и перестановкой

Основные требования к реализации:

1. Список из структур данных CPeople
2. Поля данных: имя, фамилия, номер телефона, дата рождения
3. Функции добавления элемента, удаления (по имени-фамилии), сортировки (по любому полю)
4. Собственная реализация функции сравнения строк

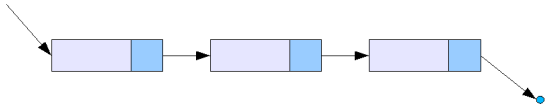
5. Реализация «пользовательского меню» (действие выбирает пользователь с консоли)

6. Функция сохранения списка в файл и загрузки из файла (с добавлением либо заменой по выбору пользователя)

7. Корректное поведение программы в случае неверных действий пользователя (удаление элемента из пустого списка и т.д.)

Теория

Односвязный список —структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.



Вопросы

Нарисуйте блок-схему алгоритмов добавления, удаления, поиска и сортировки данных в списке

14. «Бродилка» в случайном лабиринте (*)

Описание

Написать игру – бродилку в случайно сгенерированном лабиринте

Основные требования к реализации:

1. Размер поля соответствует размеру консоли
2. Края поля выделены символами «|» и “-”

3. Пользователь задает процент заполнения препятствиями
4. Корректность заполнения при 1%, 99% и 100%
5. Контроль проходимости лабиринта
6. Карта препятствий хранится в отдельной структуре Pole (дин. массив данных)
7. Контроль невозможности наступить на препятствия и выйти за границу

Теория

Разделы теоретической части

1. Текстовая графика (библиотека CONLIB): позиционирование, перерисовка, управление цветом.
2. Интерактивность: реагирования на действия пользователя.

Заполнение заданным количеством препятствий в случайных местах с помощью генератора случайных чисел.

Алгоритм заполнения:

Формируем цикл, в котором будут заполняться кирпичи в лабиринте. Причина остановки цикла: счетчик поставленных в лабиринте кирпичей равен планируемому количеству. На каждом шаге цикла:

С помощью генератора случайных чисел выбираете случайные координаты X,Y, куда хотите поставить «кирпич» в лабиринте. Если в этом месте уже есть кирпич, то проходим цикл заново. Если кирпича нет – ставим кирпич, увеличиваем счетчик кирпичей на 1, идем цикл заново.

Данный алгоритм не предусматривает варианта, когда пользователь хочет поставить кирпичей больше, чем количество свободных клеток. Для обработки такого случая необходима дополнительная проверка.

Алгоритм контроля проходимости по правым поворотам

Наиболее наглядный способ контроля проходимости лабиринта – обход по правой стороне. Если в результате вы придете обратно к входу – выход недоступен. Если придете к выходу – значит доступен.

Алгоритм контроля проходимости по заполнению лабиринта

Помечаем место входа в лабиринт флажком. Далее пробегаем по всему лабиринту и ставим в окрестности каждого флажка еще флажки. Данную процедуру повторяем до тех пор, пока пробежав по всему лабиринту, не поставим ни одного нового флажка. После этого проверяем, стоит ли флажок на позиции выхода из лабиринта. Если стоит – значит лабиринт проходим.

Вопросы

- Оцените сложность алгоритма заполнения лабиринта, реализованного вами
- Предложите варианты оптимизации алгоритма заполнения лабиринта
- Плюсы и минусы реализованного вами алгоритма контроля проходимости лабиринта. Его вычислительная сложность
- Предложите, как можно оптимизировать заполнение лабиринта

15. Игра «собери яблоки» (*)

Описание

Игра «собери яблоки» (“змейка”)

Основные требования к реализации:

1. Змейка ходит по полю и не останавливается
2. 2 типа препятствий: съедобные яблоки и стены (процент задает пользователь)
3. Интерактивность: Змейка ходит с постоянной скоростью (задает пользователь). Пользователь может менять направление ее движения
4. Задача игрока – собрать все яблоки на поле
5. Сохранение/восстановление карты и состояния игры
6. Игра интерактивная. «Змейка» увеличивает размер пропорционально съеденным яблокам.
7. Пользователю нельзя врезаться в стены и в себя

Учебное издание

Двойнишников Сергей Владимирович
Лысаков Константин Федорович

ОСНОВЫ ПРОГРАММИРОВАНИЯ
(ЯЗЫК С)

УЧЕБНОЕ ПОСОБИЕ