Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Воронежский государственный лесотехнический
университет имени Г.Ф. Морозова»

Программирование и алгоритмизация в задачах автоматизации и управления

Лабораторный практикум

для направления подготовки бакалавра 15.03.04
Автоматизация технологических процессов и производств
Профиль – Автоматизация и управление в технологических системах
для очной и заочной форм обучения

Воронеж 2018

УДК 004.432

Программирование и алгоритмизация в задачах автоматизации и управления : лабораторный практикум для направления подготовки бакалавра 15.03.04 Автоматизация технологических процессов и производств (профиль – Автоматизация и управление в технологических системах) для очной и заочной форм обучения / А. В. Стариков; М-во науки и высшего образования РФ, ФГБОУ ВО «ВГЛТУ». – Воронеж, 2019. – 67 с.

Печатается по решению редакционно-издательского совета ВГЛТУ

Рецензент: заведующий кафедрой электротехники и автоматики ФГБОУ ВО «Воронежский государственный аграрный университет имени императора Петра I», доктор технических наук, профессор Афоничев Д.Н.

Введение

Деятельность специалиста в практически любой области основывается на умении ставить задачи, разрабатывать алгоритмы, получать решения, производить анализ полученных данных и делать правильные выводы. Следовательно, подготовка студентов к будущей профессиональной деятельности должна включать в себя изучение основ алгоритмизации и программирования, приобретения навыков владения персональным компьютером, применения его для решения научных и производственных задач.

Создание нетривиальной компьютерной программы представляет собой сложный, многоэтапный и трудоёмкий процесс, включающий последовательность действий от постановки задачи до получения решения. В общем случае, эта последовательность состоит из следующих этапов:

1. ***Общая формулировка задачи.*** На этом этапе задача формулируется в содержательных терминах, определяются входные и выходные данные задачи.

2. Математическая формулировка задачи. На этом этапе определяются математические величины, которые будут описывать задачу, а также математические связи между ними, т.е. составляется математическая модель. Необходимо отметить, что неверно составленная математическая модель обречёт на неудачу всю дальнейшую работу над проектом, поэтому этот этап является крайне важным.

3. ***Выбор или разработка метода решения.*** Исходя как из субъективных причин (например, знание тех или иных математических методов), так и объективных (имеющиеся ресурсы), из большого количества математических методов выбирается тот, который целесообразно использовать для решения поставленной задачи. Если подходящего метода решения не существует, то потребуется его разработать, что представляет собой весьма сложную задачу.

4. Разработка алгоритма решения. На этом этапе должна чётко прослеживаться связь с предыдущими этапами. В ходе его выполнения разрабатывается эффективный алгоритм, т.е. такой алгоритм, реализация которого потребует наименьшего количества ресурсов компьютера.

5. Составление и отладка программы. На этом этапе применяются основные правила записи и преобразования команд, записанных на естественном языке, на язык машинных кодов.

6. ***Тестирование программы.*** На этом этапе происходит подтверждение или опровержение правильности работы алгоритма. Для этого, как правило, решаются задачи с такими исходными данными, для которых известно достоверное решение, либо применяются косвенные свидетельства правильности полученного решения.

7. Решение поставленной задачи и представление результатов. На данном этапе осуществляется удобный и наглядный вывод результатов.

При решении конкретных задач некоторые из этих этапов могут исключаться самой постановкой задачи. Для каждого из этапов создания и использования программы существуют определённые приёмы обеспечения качества

программы. Большую роль в создании программных продуктов высокого качества играет глубина и тщательность проработки схемы алгоритма.

Кроме того, нужно иметь в виду, что разработка программы это не просто последовательно выполняющиеся этапы, приводящие к желаемому результату. Ряд перечисленных выше этапов может повторяться в случае, если на более поздних этапах будут обнаружены ошибки, допущенные ранее. Например, если при тестировании программы будет обнаружена ошибка, которая была допущена при разработке алгоритма, то придётся повторить этапы разработки (корректировки) алгоритма, составления (корректировки) и отладки программы, тестирования программы. Если же ошибка была сделана при составлении математической модели на этапе математической формулировки задачи, то придётся повторить и более ранние этапы процесса решения задачи.

После того, как разработан алгоритм решения задачи, на основании которого составлена программа на некотором алгоритмическом языке, её необходимо перевести в исполняемую программу для вычислительной машины (процессора) или промежуточной исполняемой среды (языка интерпретируемого типа).

Этапами подготовки исполняемой программы являются:

1. Трансляция (компиляция) программы – это преобразование (перевод) исходного текста программы в объектный код, который представляет собой набор машинных инструкций (команд) без включения кода стандартных программ функций (например, математических функций, функций ввода-вывода и других). При этом выполняется проверка правильности синтаксиса исходной программы. Если в программе обнаружены синтаксические ошибки, то объектный код не создаётся. Если ошибок не обнаружено, то формируется объектный модуль, который передаётся на следующий этап. Трансляция (компиляция) программы выполняется с помощью специальной программы –транслятора (компилятора), входящей в состав среды программирования.
2. Компоновка (редактирование связей) – это сборка объектных модулей программы, модулей ввода-вывода и компонентов стандартной библиотеки объектных модулей в один модуль, который называется выполняемым файлом или загрузочным модулем.
3. Выполнение загрузочного (исполняемого) модуля. На этом этапе получается решение поставленной задачи. При этом возможно прерывание решения (аварийный останов, например, при делении на нуль), зацикливание или неправильные результаты. Причинами этого могут быть как ошибки в исходных данных, так и ошибки в алгоритме. В последнем случае, потребуется вернуться к начальным этапам разработки алгоритма.

Данный лабораторный практикум предназначен для того, чтобы помочь студентам в приобретении навыков разработки алгоритмов вычислительных задач, овладении приёмами программирования на алгоритмическом языке Си.

Лабораторная работа №1. Базовые алгоритмические конструкции: линейная, разветвляющаяся, циклическая. Комбинация базовых алгоритмических конструкций (6 часов)

Краткие теоретические сведения

Алгоритмизация – это раздел информатики, в котором изучаются методы и приёмы построения алгоритмов, а также их свойства. Понятие алгоритма относится к числу основных понятий современной вычислительной математики и информатики. Алгоритмизация задачи – это процесс разработки алгоритма решения задачи.

Понятие алгоритма

Для решения задачи некоторым исполнителем (например, человеком, роботом или вычислительной машиной) ему необходимо указать последовательность действий, выполнение которых приведёт к достижению требуемой цели – получению результата решения задачи. Другими словами, исполнителю должен быть указан алгоритм решения задачи, представленный на понятном ему языке.

Умение выполнять определённые команды является свойством, характеризующим любого исполнителя. Совокупность всех команд, которые данный исполнитель может выполнять, называется системой команд исполнителя. Соответственно алгоритм описывается в командах определённого исполнителя, который будет его реализовывать. Объекты, над которыми исполнитель может совершать действия, образуют среду исполнителя.

Алгоритм – это описанная на некотором языке точная и конечная система правил, определяющая содержание и порядок действий над некоторыми объектами, строгое выполнение которых даёт решение поставленной задачи.

Можно дать и более короткое определение. ***Алгоритм*** – это строго определённая последовательность действий, необходимых для решения данной задачи.

В общем случае можно сказать, что алгоритм описывает процесс преобразования исходных данных в результат (выходные данные), так как для решения любой задачи необходимо:

1) ввести исходные данные;
2) преобразовать исходные данные в результат;
3) вывести результат.

Как следует из определения, разработка алгоритма решения задачи означает, прежде всего, разбиение процесса решения задачи на последовательно выполняемые этапы. При этом результаты выполнения предыдущих этапов могут использоваться при выполнении последующих. Содержание каждого этапа и порядок выполнения этапов должны быть чётко указаны. Отдельный этап алгоритма должен быть простым и понятным без пояснений, либо представлять собой другую, более простую задачу, алгоритм решения которой известен (раз-

работан заранее). Поэтому описание алгоритма решения задачи выполняется в соответствии со следующими правилами:

1. Определяются исходные данные задачи.
2. Процесс решения задачи разбивается на этапы, понятные и однозначные для исполнителя.
3. Указывается порядок, в котором выполняются этапы, а также признак завершения процесса.
4. Определяется, что является результатом решения задачи.

Свойства алгоритма

Свойства алгоритма – это набор свойств, отличающих алгоритм от любых предписаний и обеспечивающих его автоматическое исполнение. Алгоритм обладает следующими основными свойствами:

Дискретность (прерывистость) – свойство алгоритма, характеризующее его структуру: каждый алгоритм состоит из отдельных законченных действий, т.е. делится на шаги.

Массовость – применимость алгоритма ко всем задачам рассматриваемого типа, при любых исходных данных.

Детерминированность (определённость) – свойство алгоритма, указывающее на то, что каждый шаг алгоритма должен быть строго определён и не допускать различных толкований; также строго должен быть определён порядок выполнения отдельных шагов.

Результативность – свойство, состоящее в том, что любой алгоритм должен завершаться за конечное (пусть даже очень большое) число шагов.

Формальность – свойство алгоритма, указывающее на то, что любой исполнитель, способный воспринимать и выполнять инструкции алгоритма, действует формально, т.е. отвлекается от содержания поставленной задачи и лишь строго выполняет инструкции. Другими словами, механически выполняя все указанные в алгоритме этапы в требуемом порядке, исполнитель может всегда правильно решить задачу.

На этапе разработки алгоритма решения задачи рекомендуется придерживаться следующих правил его составления:

1. Алгоритм должен быть максимально прост и понятен.
2. Алгоритм должен состоять из мелких шагов.
3. Сложная задача должна разбиваться на достаточно простые, легко воспринимаемые части (блоки).
4. Логика алгоритма должна опираться на минимальное число достаточно простых базовых управляющих структур.

В итоге процесс разработки алгоритма должен быть направлен на получение чёткой структуры алгоритмических конструкций.

Сущность алгоритмизации вычислительного процесса проявляется в следующих действиях, отражающих его свойства:

1) выделение законченных частей вычислительного процесса;
2) формальная запись каждого из них;
3) назначение определённого порядка выполнения выделенных частей;

4) проверки правильности выбранного алгоритма по реализации заданного метода вычислений.

Способы описания алгоритмов

Для строгого задания различных структур данных и алгоритмов их обработки нужно иметь такую систему формальных обозначений и правил их использования, чтобы всякое действие трактовалось точно и однозначно. Соответствующие системы правил называют языками описаний алгоритмов.

В настоящее время используются следующие языки описания:

- словесная запись;
- псевдокод;
- схемы алгоритмов;
- алгоритмические языки.

При разработке сложных проектов программисты используют специальные программные средства для автоматизированного проектирования алгоритмов и программ.

Ниже рассмотрены различные способы описания алгоритма решения несложной задачи.

Задача. Составить алгоритм и написать программу для решения квадратного уравнения вида $ax^2 + bx + c = 0$. Программа должна проверять правильность исходных данных и в случае, если коэффициент *a* при второй степени неизвестного равен нулю, выводить соответствующее сообщение.

Словесная запись алгоритма. Данный способ основан на использовании общепринятых средств общения между людьми и содержит набор фраз, который не допускает лишних слов, повторений и неоднозначностей. Действия, предусмотренные алгоритмом, нумеруются, что даёт возможность на них ссылаться. Допускается использование математической символики.

Словесная запись алгоритма решения квадратного уравнения:

1. Ввести значения коэффициентов *a, b* и свободного члена *c*.
2. Если величина *a*=0, то вывести сообщение «Уравнение не является квадратным». Перейти к пункту 8.
3. Определить величину дискриминанта *d* по следующей формуле:
$$d = b^2 - 4ac\,.$$
4. Если величина *d*<0, то вывести сообщение «Уравнение не имеет действительных корней». Перейти к пункту 8.
5. Если величина *d*=0, то вычислить значения корней уравнения по следующей формуле:
$$x_1 = x_2 = \frac{-b}{2a}\,.$$
Перейти к пункту 7.
6. Вычислить значения корней уравнения по следующим формулам:
$$x_1 = \frac{-b + \sqrt{d}}{2a};\; x_2 = \frac{-b - \sqrt{d}}{2a}\,.$$

7. Вывести значения x_1 и x_2.
8. Конец алгоритма.

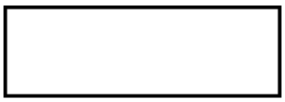
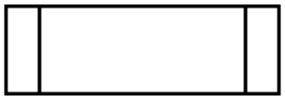
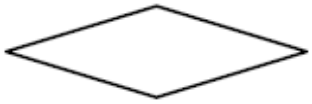
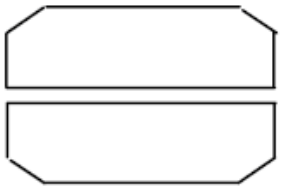
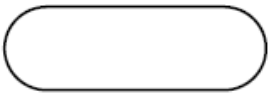
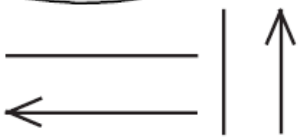
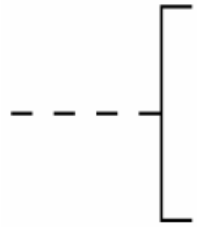
Псевдокод. Это формализованный язык записи структурированных алгоритмов, основанный на представлении предписаний, задаваемых с помощью ограниченного набора типовых синтаксических конструкций. Набор конструкций состоит из смеси конструкций алгоритмического языка высокого уровня и фраз родного языка исполнителя. Как правило, стандартов на псевдокод не существует. Псевдокод используется для облегчения разработки программ. Так же как и программа, псевдокод имеет все достоинства структурированной записи, поэтому алгоритм, написанный на псевдокоде, достаточно легко преобразовать в программный код.

Запись алгоритма с использованием псевдокода:

```
алг Решение_квадратного_уравнения (арг вещ a, b, c, рез вещ x1, x2)
    начало
        вещ d
            ввод        a, b, c
            если a = 0 то вывод “Уравнение не является квадратным”
                иначе
                    d = b*b-4*a*c
                    если d < 0 то вывод “Уравнение не имеет действительных корней”
                        иначе
                            если d = 0 то x1= x2=-b/(2*a)
                                иначе
                                    x1=(-b+√d)/(2*a)
                                    x2=(-b-√d)/(2*a)
                            конец если
                            вывод x1, x2
                    конец если
            конец если
    конец
```

Схемы алгоритмов. Данный способ является наиболее распространённым и обеспечивает графическое представление метода решения задачи, в котором используются символы, отображающие операции (действия) и данные.

В схеме алгоритма каждому типу действий (например, ввод исходных данных, вычисление значений выражений, проверка условий и т.д.) соответствует определённая геометрическая фигура, представляющая символ действия. Символы действия соединяют линиями переходов, которые определяют очерёдность выполнения действий. Форма символов и правила составления схем установлены Единой системой программной документации (ЕСПД) ГОСТ 19701-90. Наиболее часто употребляемые символы действий указанного стандарта приведены ниже.

Название символа	Обозначение	Пояснение
Процесс		Выполнение определённой операции или группы операций
Предопределённый процесс		Вычисления по подпрограмме, стандартной подпрограмме
Решение		Проверка условий. Переключение вычислительного процесса на одну из ветвей алгоритма после вычисления условий, определённых внутри символа
Граница цикла		Символ состоит из двух частей и отображает начало и конец цикла. Обе части имеют один и тот же идентификатор. Условия инициализации, приращения, завершения и т.д. помещаются внутри символа в начале или конце – в зависимости от расположения операции, проверяющей условие
Подготовка		Модификация команды или группы команд на некоторую последующую функцию (например, модификация параметров цикла). Используется для циклов с параметром.
Терминатор		Начало и конец программы, вход и вывод из подпрограммы
Данные		Символ отображает ввод данных из внешней среды или вывод результатов (носитель данных не определён)
Документ		Символ отображает данные, представленные на носителе в удобочитаемой форме. Используется как символ печати результатов
Линия		Символ отображает поток данных или управления. При необходимости могут быть добавлены стрелки-указатели
Соединитель		Символ отображает выход в часть схемы и вход из другой части схемы. Используется для обрыва линии и продолжения её в другом месте
Комментарий		Используется для добавления описательных комментариев или пояснительных записей в целях объяснений (примечаний). Пунктирная линия связана с соответствующим символом или может обводить группы символов. Текст помещается около ограничивающей фигуры

Символы ***Данные*** и ***Документ*** используются для обозначения операций ввода-вывода.

Символ ***Процесс*** применяется для обозначения выполнения одной или нескольких операций, приводящих к изменению значения, формы или размещения информации. Представление операций достаточно свободно. Например, для обозначения вычислений можно использовать математические выражения или текст.

Символ ***Решение*** используется для обозначения переходов управления по условию. Имеет один вход и ряд выходов, при этом только один из них может быть активизирован после вычисления условий, определённых внутри этого символа. Результаты вычисления могут быть записаны рядом с линиями, отображающими эти пути.

Символ ***Граница цикла*** используется для организации циклических вычислительных процессов. Параметр цикла записывается в обеих частях символа. Начальное значение, граничное условие и правило изменения параметра цикла указывается в начале или в конце в зависимости от расположения операции проверяющей условие.

Символ ***Подготовка*** используется, как правило, для организации цикла с параметром. Внутри символа записывается параметр цикла, указывается его начальное значение, граничное условие и правила изменения значения параметра. Символ размещается в начале циклической конструкции.

Линии переходов используются для обозначения порядка выполнения действий.

Соединители используются в том случае, когда схема алгоритма разделяется на автономные части, особенно если она не умещается на одном месте, или когда необходимо избежать излишних пересечений линий переходов.

Комментарий позволяет включать в схемы алгоритмов пояснения к функциональным блокам. Частое использование комментариев нежелательно, так как это усложняет схему, делает её менее наглядной.

Алгоритм начинается и заканчивается символами ***Начало*** и ***Конец***.

Основные правила применения символов и выполнения схем алгоритмов:

1. Символы в схеме должны быть расположены равномерно. Нужно придерживаться разумной длины соединений и минимального числа длинных линий.
2. Символы должны быть по возможности одного размера и предпочтительно горизонтальной ориентации.
3. Внутри символа помещается минимальное количество текста, необходимое для понимания функции символа. Для записи используется естественный язык с элементами математической символики. Если объем текста превышает размер символа, то нужно использовать символ ***Комментарий***.
4. Потоки данных и потоки управления в схемах показываются линиями. Направление потока слева направо и сверху вниз считается стандартным. Если поток имеет направление, отличное от стандартного, стрелки должны указывать это направление.

5. Линии в схемах должны подходить к символу либо слева, либо сверху, а исходить либо справа, либо снизу. Линии должны быть направлены к центру символа.
6. Каждый символ имеет один вход и один выход. Исключением является символ ***Решение***, который имеет один вход и несколько выходов. При этом каждый выход должен сопровождаться значениями условий, чтобы указать логический путь, который он представляет.

Схема алгоритма является самым наглядным способом представления алгоритма, при этом нет никаких ограничений на степень его детализации. На рисунке 1.1 показана схема алгоритма решения квадратного уравнения.

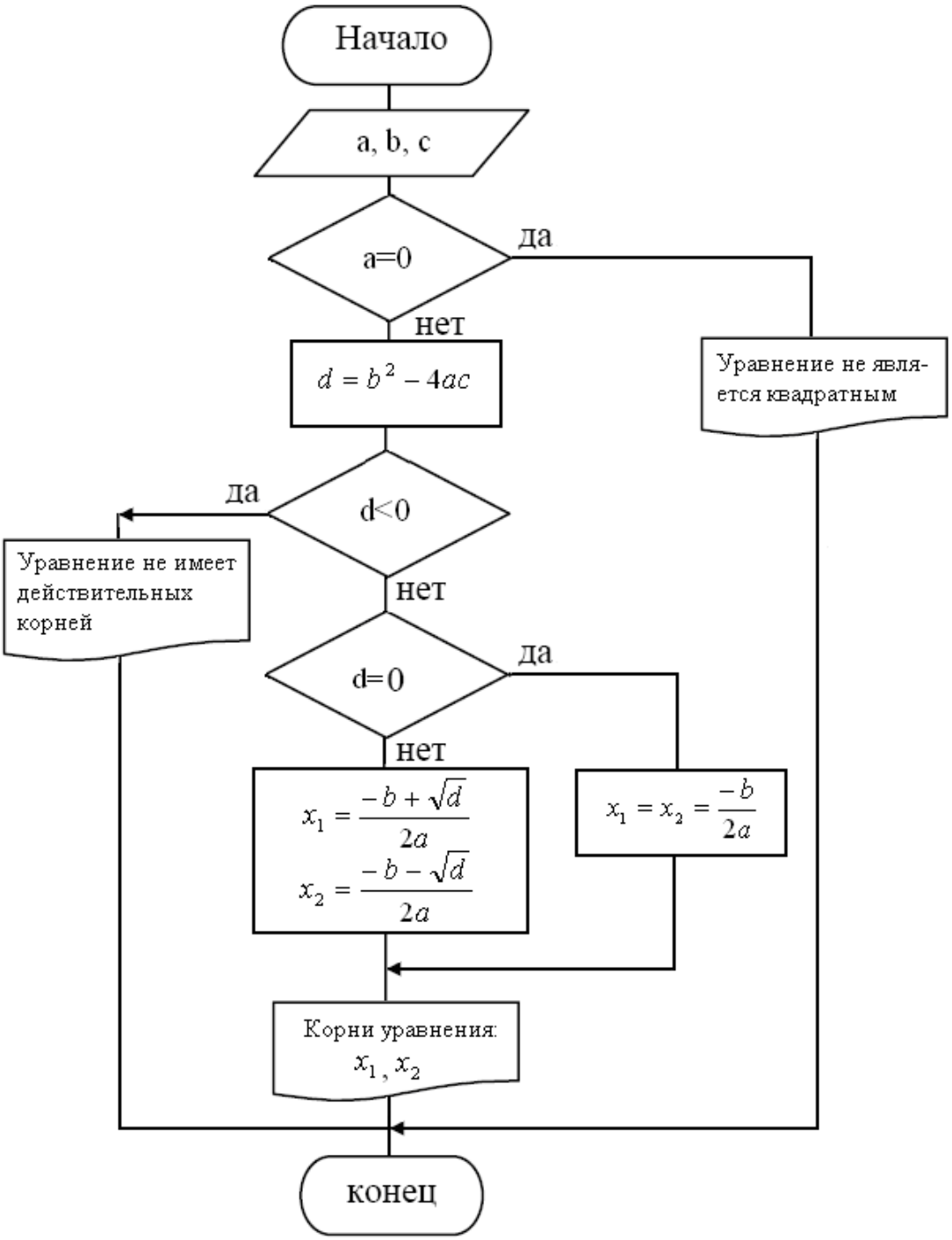


Рисунок 1.1 – Схема алгоритма решения квадратного уравнения

Базовые алгоритмические конструкции

Алгоритмы представляют собой определённые структуры, которые состоят из отдельных базовых (основных) элементов, представляющих собой элементарные шаги алгоритма.

В зависимости от особенностей построения алгоритмов различают следующие их виды (конструкции):

- линейные;
- разветвляющиеся;
- циклические.

Разнообразие алгоритмов определяется тем, что любой алгоритм состоит из фрагментов, каждый из которых, в свою очередь, представляет собой алгоритм одного из указанных видов. Поэтому важно знать структуру каждого из алгоритмов и принципы их составления.

Линейный алгоритм представляет собой конструкцию, в которой все этапы решения задачи выполняются только один раз и строго последовательно. Другими словами, линейный (последовательный) алгоритм выполняется в естественном порядке его написания и не содержит разветвлений и повторений.

Примерами линейных алгоритмов являются: алгоритм отпирания дверей – достать ключ, вставить в замочную скважину, открыть замок; алгоритм заваривания чая – достать чайник, насыпать в него чай, залить кипятком, настоять 5-10 минут.

Структурная схема линейного алгоритма показана на рисунке 1.2, где прямоугольник обозначает одну или несколько операций, необходимых для выполнения обработки данных.

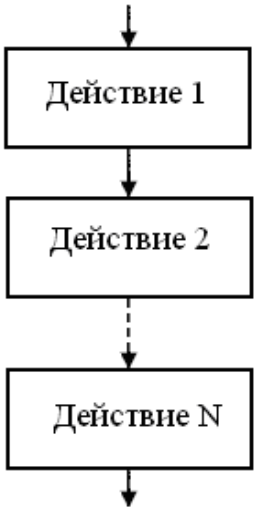


Рисунок 1.2 – Линейная алгоритмическая конструкция

При разработке линейного алгоритма необходимо учитывать, что:

- данный вид алгоритма является простейшим;
- он чаще всего используется для реализации простых вычислений по формулам;
- действия в нём выполняются последовательно, одно за другим.

Задача 1.1. Составить схему алгоритма вычисления площади треугольника по известным длинам трёх его сторон (по формуле Герона).

Входные данные: *a*, *b*, *c* – длины сторон треугольника.

Выходные данные: *S* – площадь треугольника.

Формула Герона для вычисления площади треугольника *S* имеет следующий вид:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

где $p = (a+b+c)/2$ – полупериметр треугольника.

Словесное описание алгоритма:

1. Начало.
2. Ввести значения переменных *a*, *b*, *c*.
3. Вычислить значение полупериметра: $p = (a+b+c)/2$.
4. Вычислить значение площади: $S = \sqrt{p(p-a)(p-b)(p-c)}$.
5. Вывести значение *S*.
6. Конец.

Структурная схема алгоритма вычисления площади треугольника по формуле Герона показана на рисунке 1.3.

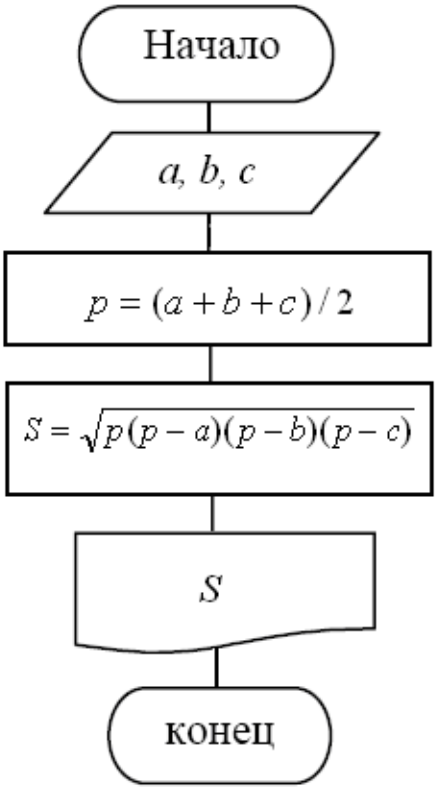


Рисунок 1.3 – Схема линейного алгоритма для вычисления площади треугольника по формуле Герона

Задача 1.2. Составить словесное описание и разработать схему алгоритма вычисления объёма и массы цилиндрического тела по заданным для него радиусу, высоте и плотности материала.

Входные данные: R – радиус основания цилиндра, h – высота цилиндра, ρ – плотность материала, из которого изготовлен цилиндр.

Выходные данные: V – объём тела, m – масса тела.

Формулы, использующиеся для вычислений:

$V = \pi R^2 h$ – объём цилиндра;

$m = \rho V$ – масса тела.

Разветвляющийся алгоритм представляет собой конструкцию, в которой обеспечивается выбор между двумя альтернативами в зависимости от значения входных данных. Другими словами, разветвляющийся алгоритм – это алгоритм, в котором в зависимости от условия выполняется либо одна, либо другая последовательность действий. Если в разветвляющейся алгоритмической конструкции присутствуют оба действия, то говорят о полной альтернативе (рисунок 1.4).

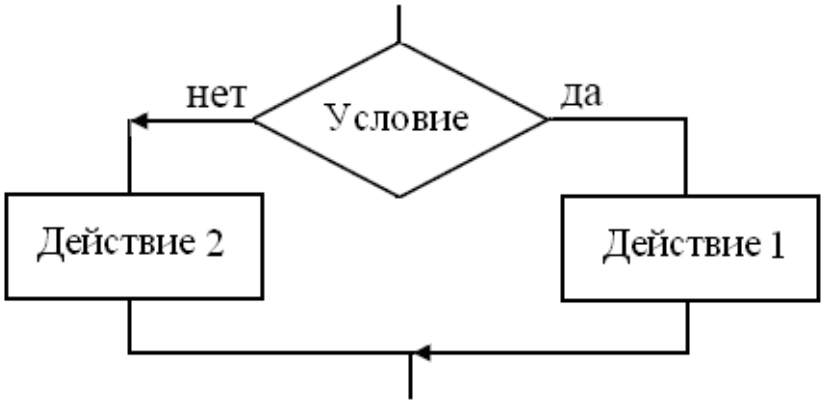


Рисунок 1.4 – Разветвляющаяся алгоритмическая конструкция с полной альтернативой

Словесное описание разветвляющейся алгоритмической конструкции с полной альтернативой:

***Если* Условие *справедливо, то выполняется* Действие 1, *иначе выполняется* Действие 2.**

Если одно из действий в разветвляющейся алгоритмической конструкции отсутствует, то говорят о неполной альтернативе (рисунок 1.5).

Словесное описание разветвляющейся алгоритмической конструкции с полной альтернативой:

***Если* Условие *справедливо, то выполняется* Действие**

Условие – это логическое выражение, результатом которого может быть одно из двух значений: **истина** (да) и **ложь** (нет). Если **Условие** верно (истинно), то выполняется **Действие 1**, показанное на рисунке 1.4 (**Действие** – на рисунке 1.5); в противном случае (**Условие** ложно) – выполняется **Действие 2**, на рисунке 1.4 (ничего не выполняется – на рисунке 1.5).

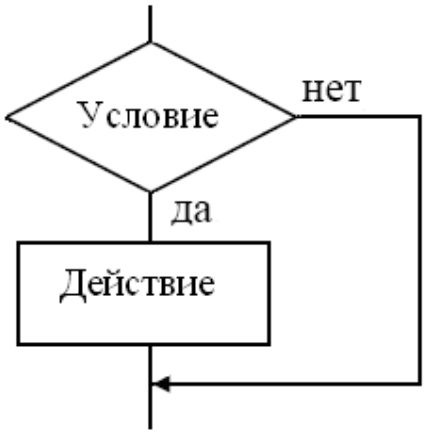


Рисунок 1.5 – Разветвляющаяся алгоритмическая конструкция с неполной альтернативой

Разветвляющаяся алгоритмическая конструкция с неполной альтернативой использована в алгоритме решения квадратного уравнения (см. выше рисунок 1.1).

Циклический алгоритм представляет собой конструкцию, в которой предусмотрено неоднократное выполнение одной и той же последовательности действий при различных значениях входящих в них величин. Совокупность повторяющихся действий алгоритма называется ***циклом***. Группа повторяющихся действий на каждом шаге (итерации) цикла называется ***телом цикла***.

Циклический алгоритм включает в себя следующие действия:

- задание начального значения параметра (переменной) цикла перед началом цикла;
- изменение параметра (переменной) цикла перед каждым новым повторением тела цикла;
- проверка условия окончания (выхода из цикла) или повторения цикла;
- переход к началу цикла, если цикл не закончен, или выход из цикла, если условие выхода выполнено.

Существует несколько видов циклических алгоритмических конструкций, с помощью которых можно организовать циклы. Их можно классифицировать следующим образом.

По месту расположения условий проверки повторения или окончания цикла различают ***циклы с предусловием*** и ***циклы с постусловием***.

В цикле с предусловием (с предварительным условием) проверка условия выхода из цикла стоит перед телом цикла. Условие записывается в виде логического выражения. Операторы цикла (тело цикла) выполняются, пока условие ***истинно***. Если при входе в цикл условие ***ложно*** (не выполняется), то осуществится выход из цикла. В этом случае цикл не выполнится ни одного раза.

В цикле с постусловием (с последующим условием) проверка условия выхода стоит после тела цикла. Операторы тела цикла будут выполняться до тех пор, пока не станет возможным (истинным) условие выхода из цикла. Данный цикл выполняется хотя бы один раз.

В инженерных расчётах наиболее часто используются циклы с постусловием. Общий вид вычислительной схемы цикла с постусловием показан на рисунке 1.6.



Рисунок 1.6 – Типовая схема циклического алгоритма с постусловием

По способу контроля окончания цикла различают циклы с неизвестным числом повторений и циклы с известным числом повторений.

Циклы с неизвестным числом повторений (итераций). Например, требуется вычислить сумму сходящегося ряда

$$S = \sum_{n=1}^{\infty} \frac{n}{n^2 + A}.$$

Условие окончания вычислений

$$\left| \frac{n}{n^2 + A} \right| \le EPS,$$

где *EPS* – очень малая величина, которая определяет точность решения задачи.

Данный тип циклов характерен для математических задач. Схема алгоритма для решения данной задачи представлена на рисунке 1.7.

В инженерной практике при использовании численных методов часто применяется разновидность циклов с неизвестным числом повторений, которые называются ***итерационными***. В итерационных циклах может отсутствовать переменная цикла. В итерационных циклах используется зависимость

$$X_n = F(X_{n-1}),$$

где F – функция;

X_{n-1}, X_n – параметры.

Цикл заканчивается при выполнении условия, связанного с проверкой значения изменяющейся в цикле величины. Как правило, это условие имеет вид $\left| X_n - X_{n-1} \right| \le EPS.$

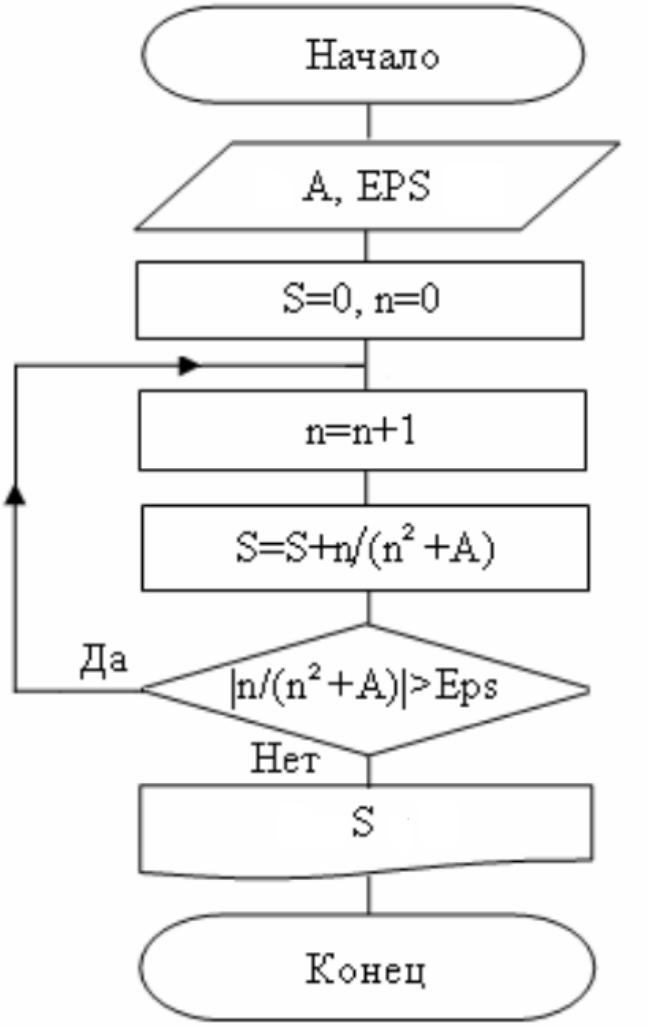


Рисунок 1.7 – Схема циклического алгоритма с неизвестным числом повторений

Циклы с известным числом повторений (итераций) представляются ***арифметическим циклом*** (циклом с параметром). Это цикл предполагает, что число итераций известно заранее, и предписывает выполнять тело цикла для всех значений некоторой переменной (параметра цикла) в заданном диапазоне. Переменная цикла изменяется от заданного начального значения до заданного конечного значения, получая при каждом выполнении цикла постоянное приращение, которое называется шагом параметра цикла. Типовая схема арифметического цикла приведена на рисунке 1.8.

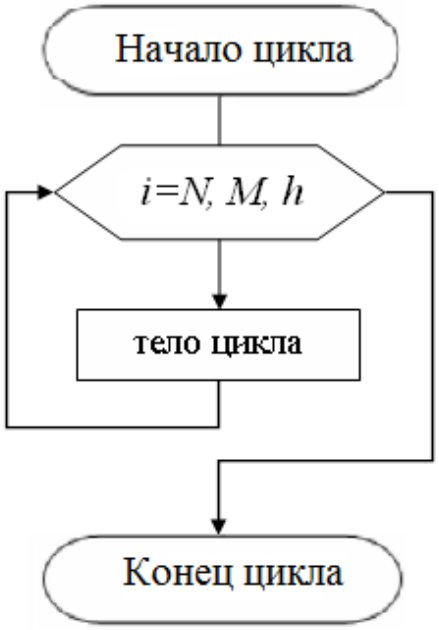


Рисунок 1.8 – Типовая схема арифметического цикла (цикла с параметром)

Здесь использованы следующие обозначения:

i – параметр цикла (счётчик количества повторений);

N – начальное значение параметра цикла;

M – конечное значение параметра цикла;

h – шаг, с которым изменяется параметр цикла.

Количество повторений тела цикла однозначно определяется правилом изменения параметра, которое задаётся с помощью начального и конечного значений параметра и шагом его изменения. На первом шаге цикла значение параметра равно N, на втором – $N+h$, на третьем – $N+2h$ и т.д. На последнем шаге цикла значение параметра не больше M, но такое, что дальнейшее его изменение приведёт к значению большему, чем M.

Задача 1.3. Составить алгоритм вычисления суммы первых 100 элементов ряда

$$S=\sum_{n=1}^{\infty}\frac{n}{n^2+A}.$$

Схема алгоритма решения данной задачи показана на рисунке 1.9. На рисунке используются два варианта символов (см. выше обозначения языка схем алгоритмов).

Алгоритмы решения сложных задач могут включать различные комбинации перечисленных выше алгоритмических конструкций. Например, часто используются так называемые вложенные циклы, когда внутри одного цикла могут находиться один или несколько других циклов (см. ниже задачу1.3). Охватывающий цикл называется внешним, а вложенные в него циклы – внутренними (вложенными), при этом область действия внутреннего цикла должна полностью находиться в области внешнего цикла, т.е. циклы не должны пересекаться. Для этого параметр (переменная цикла) каждого вложенного цикла

должен иметь своё имя. Правила организации как внешнего, так и внутреннего циклов такие же, как и правила организации простого цикла. Параметры внешнего и внутреннего циклов изменяются не одновременно.

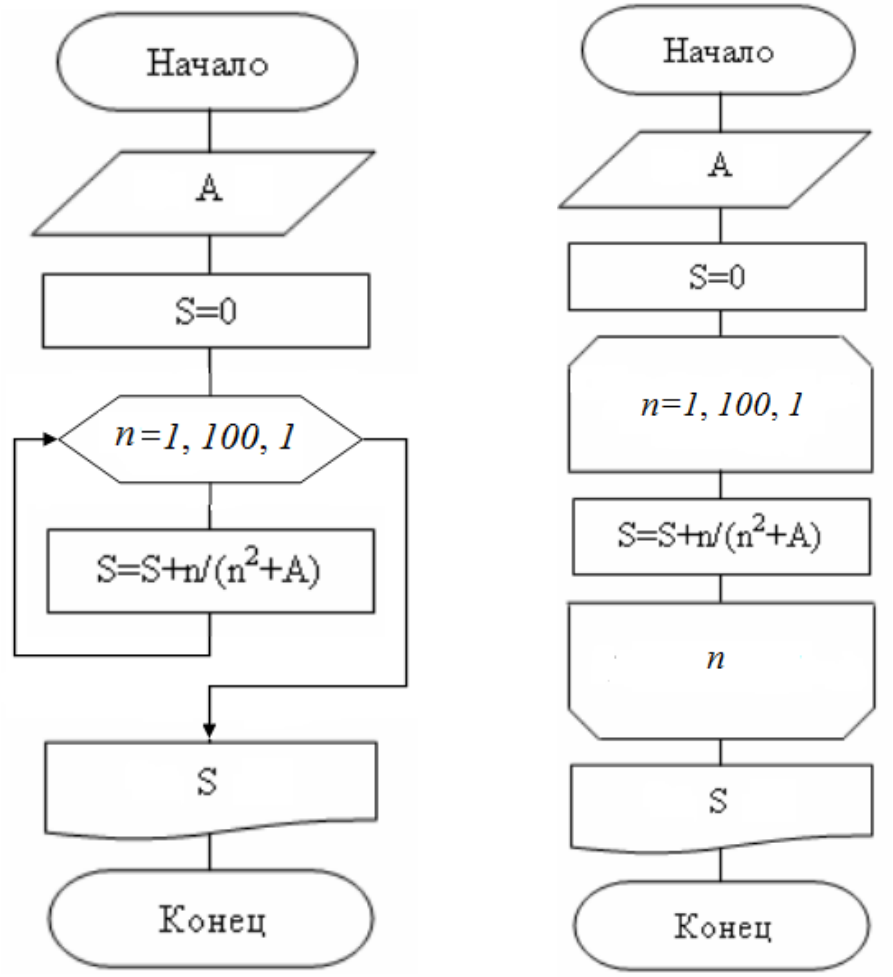


Рисунок 1.9 – Структурная схема алгоритма суммирования первых 100 элементов ряда (2 варианта графических символов для обозначения цикла с параметром)

При работе с циклами необходимо придерживаться условий их правильного использования. Начальное, конечное значения и шаг должны иметь тот же тип, что и параметр цикла. Если начальное и конечное значения равны, то цикл выполняется один раз. Внутри цикла не рекомендуется изменять параметр цикла и его конечное значение, так как эти значения устанавливаются в самом начале работы цикла. Цикл заканчивается, когда параметр цикла принимает конечное значение. При организации цикла следует особое внимание уделить правильному оформлению изменения параметра цикла, поскольку ошибка на этом этапе может привести к «зацикливанию» вычислительного процесса.

Задача 1.3. Вычислить сумму положительных элементов каждой строки матрицы $A(10\times8)$.

Для вычисления суммы положительных элементов одной строки матрицы необходимо организовать цикл с целью перебора всех элементов строки, поэтому параметром этого цикла следует выбрать номер столбца j. Перед циклом

нужно задать начальное значение суммы $S = 0$. После окончания цикла результат необходимо вывести на печать.

Если действия повторить во внешнем цикле, изменяя индекс строки *i*, то будут вычислены все 10 сумм. Схема алгоритма решения задачи приведена на рисунке 1.10. В рассмотренной задаче внешний цикл обязательно должен быть по *i* (индексу строки), так как в противном случае были бы вычислены суммы положительных элементов каждого столбца. Часто встречаются задачи, в которых не имеет значения, по какому параметру организовать внешний и внутренний цикл. Все приемы программирования, изложенные выше, можно использовать и при организации вложенных циклов.

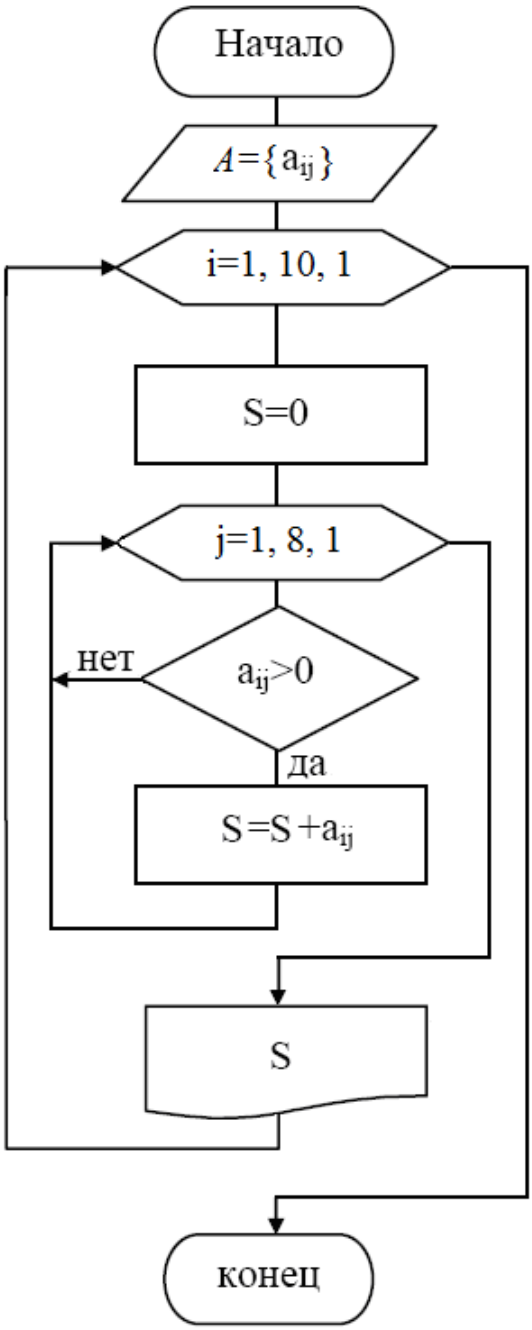


Рисунок 1.10 – Схема алгоритма решения задачи, в которой используются вложенные циклы

Вопросы для самоконтроля

1. Назовите основные виды структур алгоритмов.
2. Из каких символов состоит схема алгоритма линейной структуры? Какие символы она не может содержать?
3. Какой порядок выполнения действий называется естественным?
4. Что общего и в чём разница между полной и неполной альтернативой для разветвлённого алгоритма?
5. Что даёт для алгоритма применение циклических структур?
6. Что называется параметром цикла?
7. Что называется телом цикла?
8. Может ли тело цикла не выполняться ни одного раза? В каких случаях?
9. Чем отличается цикл с предусловием от цикла с постусловием?
10. Нарисовать общую структурную схему цикла с предусловием.
11. Назовите типы циклов по способу контроля окончания цикла.
12. Какие типы циклов характерны для численных методов?
13. Можно ли в качестве параметра цикла во внешнем и внутреннем цикле использовать одну и ту же переменную?

Практические задания

1. Составить структурные схемы алгоритмов для следующих разветвляющихся вычислительных процессов:

№ п/п	Вычислить выражение
1	$Z = \max(A, \min(B, C, D))$
2	$Z = \min(A + 1, \max(B, C + D))$
3	$Y = \min(A, \max(B + C, D + 1))$
4	$Z = \min(A, \max(B, C + D))$
5	$Z = \min(A, B) + \max(B, C)$
6	$Z = \max(A + 1, \min(B, C + D))$
7	$Y = \max(A, \min(B, C - 1))$
8	$Z = \min(A, \max(B - D, C))$
9	$Z = \max(A, \min(B, C + 1, D))$
10	$Y = \min(A+1, \min(B - 1, C + D))$
11	$Z = \max(A, \max(B + C, D + 1))$
12	$Z = \min(A^2, \max(B, C, D + 1))$
13	$Y = \min(A, B^2) + 2.5\max(B, C + 1)$
14	$Z = \max(A^3 + 1, \min(B, C + D))$
15	$Z = \max(A, B + 1) - 0.5\min(C, D)$
16	$Y = \min(A + C, \min(B - 4, C + 2, D))$
17	$Z = \max(A + 1, B) + 1.5\min(C, D - 0.5))$
18	$Y = \min(A, \min(B - 4, C, D + 1))$
19	$Z = \max(A + B^3, \min(B, C, D))$
20	$Z = \min(A^5 + 1, \min(B, C + D))$

№ п/п	Вычислить выражение
21	Z = max(A, min(B – C, D – 1))
22	Z = min(A + D, min(B^3 + 2, C))
23	Y = min(A, B) – 0.1max(B, C + 1)
24	Z = max(A + 1, min(B, C + D))
25	Y = max(A, min (B, C – 1, D))
26	Z = max(A + D, max(B – A, C))
27	Y = min(A, B, C) + max(B, C)
28	Z = max(A, min (B, C – 1), D)
29	Z = min(A, B + 0.6) + 0.5max(C, D)
30	Z = min(A + D, max(B^3 + 2, C))

2. Составить структурные схемы алгоритмов для следующих разветвляющихся вычислительных процессов (сложное выражение):

№ п/п	Вычислить сложное выражение
1	$Z=\begin{cases}0.25\min(A,B)+\lvert C\rvert, & \text{если } A+B+C\ge 0\\ 0.1(1-A^2)+\dfrac{B}{C+1}, & \text{если } A+B+C<0,\ A\ge 0\\ \min(A,B,C), & \text{если } A+B+C<0,\ A<0\end{cases}$
2	$Z=\begin{cases}A+\max(A+B,C), & \text{если } A+B+C\ge 0\\ B^2+\min(A,B,C), & \text{если } A+B+C<0\end{cases}$
3	$Z=\begin{cases}A+\min(A,B,C), & \text{если } A^2-B^2+C\ge 0\\ 0.2(A+B)^2+e^{0.1C}, & \text{если } A^2-B^2+C<0,\ B\ge 0\\ \max(B,C)+A^2, & \text{если } A^2-B^2+C<0,\ B<0\end{cases}$
4	$Z=\begin{cases}\min(x,y)+R^2, & \text{если } x+y+R^2\ge 0\\ x^3+\max(y,R), & \text{если } x+y+R^2<0,\ R\ge 1\\ 0.1(x-y)^3+e^R, & \text{если } x+y+R^2<0,\ R<1\end{cases}$
5	$Z=\begin{cases}0.5\min(X,Y)+X^{\sin(Y^2+1)}, & \text{если } X+Y\ge 0\\ \sqrt{\max(X,0.5Y)+Y^2}, & \text{если } X+Y<0\end{cases}$

№ п/п	Вычислить сложное выражение
6	$Z=\begin{cases}\max(A,B,C+1)+0.5(C+1)^2, & \text{если} \quad A^2+B+C\ge 0\\ \min(B,C)+0.5\max(A,C)\ , & \text{если} \quad A^2+B+C<0\end{cases}$
7	$Z=\begin{cases}\max(A,B)+C^2, & \text{если} \quad A+2.5B\ge C^2\\ A^3+\min(B,C), & \text{если} \quad A+2.5B<C^2,\ C\ge 0\\ 0.1(A-B)^2+\dfrac{1}{\lvert C+1\rvert}, & \text{если} \quad A+2.5B<C^2,\ C<0\end{cases}$
8	$Z=\begin{cases}\max(A,B+C)+C^2, & \text{если} \quad A+B\ge C\\ A^4+0.5\min(A,B,C), & \text{если} \quad A+B<C\end{cases}$
9	$Z=\begin{cases}\max(A,B)+\min(B,C^2), & \text{если} \quad A^2-B\ge C\\ A^3+\min(B,C), & \text{если} \quad A^2-B<C\end{cases}$
10	$Z=\begin{cases}0.25\min(A,B)+\lvert C\rvert, & \text{если} \quad A+B+C\ge 0\\ 0.1(1-A^2\ )+\dfrac{B}{C+1}, & \text{если} \quad A+B+C<0,\ A\ge 0\\ \min(A,B,C), & \text{если} \quad A+B+C<0,\ A<0\end{cases}$
11	$Z=\begin{cases}A^2+2.5\max(A+1,B), & \text{если} \quad A+B+C\ge 0\\ B+\min(A,C), & \text{если} \quad A+B+C<0\end{cases}$
12	$Z=\begin{cases}A+\min(A,B,C), & \text{если} \quad A+B+C\ge 0\\ 0.3(1-A^2)+Ln(A+C), & \text{если} \quad A+B+C<0,\ A\ge 0\\ \max(A,B,C), & \text{если} \quad A+B+C<0,\ A<0\end{cases}$
13	$Z=\begin{cases}2.5\min(x,y,R)+R^2\ , & \text{если} \quad x+y+R\ge 0\\ x^3+\max(y,R)\ , & \text{если} \quad x+y+R<0\end{cases}$

14	$Z=\begin{cases}0.5\max(A,B,C)+A, & \text{если } A+B+C\ge 0\\ 2.1(1-A^2)+\cos\left(\dfrac{B}{C+1}\right), & \text{если } A+B+C<0,\ A\ge 0\\ A^2+\min(B,C), & \text{если } A+B+C<0,\ A<0\end{cases}$
15	$Z=\begin{cases}\max(A,B,C+1)+\sqrt{C+1}, & \text{если } A^2+B+C\ge 0\\ \min(A,B-1,C)+A^3, & \text{если } A^2+B+C<0\end{cases}$
16	$Z=\begin{cases}\max(A,B)+0.4\min(B,C^2), & \text{если } A^2+B^2\ge C\\ \min(A,B,C), & \text{если } A^2+B^2<C\end{cases}$
17	$Z=\begin{cases}\min(A,B)+e^C, & \text{если } A+1.5B\ge C^2\\ A^3+\max(B,C), & \text{если } A+1.5B<C^2,\ C\ge 0\\ 3.1(A-B)^3+\sin^2(C+1), & \text{если } A+1.5B<C^2,\ C<0\end{cases}$
18	$Z=\begin{cases}\max(A,B)+0.2\min(B,C), & \text{если } A^2+B-C\ge 1\\ A^2+\min(A+1,B-1,C), & \text{если } A^2+B-C<1\end{cases}$
19	$Z=\begin{cases}\max(A^2-1,B+C)+C^2, & \text{если } A+B\ge C\\ A^4+\min(B-2,C^2), & \text{если } A+B<C\end{cases}$
20	$Z=\begin{cases}\max(A-1,B,C+1)+C^2, & \text{если } A+B^2\ge C\\ 0.1\sin(A)+\min(B,A-C), & \text{если } A+B^2<C\end{cases}$
21	$Z=\begin{cases}0.5\min(A,B-C)+\lvert C\rvert, & \text{если } A+B+C\ge 0\\ 2.1(1-A)+\dfrac{B}{\lvert C+1\rvert}, & \text{если } A+B+C<0,\ A\ge 0\\ \min(A,B,C-1), & \text{если } A+B+C<0,\ A<0\end{cases}$

22	$Z=\begin{cases}A^2+\max(A+B,A-C), & \text{если } A+B+C\ge 0\\ 0.25(C-A)^3+\min(A,B,C), & \text{если } A+B+C<0\end{cases}$
23	$Z=\begin{cases}1.5\max(A,B+C)-\lvert C\rvert, & \text{если } A+B+C\ge 0\\ 2.2(1-A)+\dfrac{B}{\lvert C+1\rvert}, & \text{если } A+B+C<0,\ A\ge 0\\ \max(B-C,C)+A^2, & \text{если } A+B+C<0,\ A<0\end{cases}$
24	$Y=\begin{cases}\min(x,y,R)+R^2, & \text{если } x+y+R\ge 0,\\ \max(x,y)+\min(y-R,R), & \text{если } x+y+R<0\end{cases}$
25	$Z=\begin{cases}1.5\min(A,B+C)-\lvert C-1\rvert, & \text{если } A-B+C\ge 0\\ 2(B-A^2)+\dfrac{B}{\lvert C+1\rvert}, & \text{если } A-B+C<0,\ A\ge 0\\ \max(B+C,C)+A^2, & \text{если } A-B+C<0,\ A<0\end{cases}$
26	$Z=\begin{cases}\max(A,B-C)+\min(A,C), & \text{если } A+B-C\ge 0\\ \min(A,B,C-1)+\sqrt{A^3}, & \text{если } A+B-C<0\end{cases}$
27	$Z=\begin{cases}\max(A,B)+0.2\min(B,C^2), & \text{если } A+B^2\ge C\\ A^3+\min(A,B+1,C-1), & \text{если } A+B^2<C\end{cases}$
28	$Z=\begin{cases}\max(A^2-B,B+C)+\ln(C), & \text{если } A+B\ge C^2\\ A^3+\min(A,B,C), & \text{если } A+B<C^2\end{cases}$
29	$Z=\begin{cases}A^2+\max(A,B,C+1), & \text{если } A^2-B+C\ge 0\\ 2(A+B)^2+e^{0.1C}, & \text{если } A^2-B+C<0,\ B\ge 0\\ \min(B,C)+A^2, & \text{если } A^2-B+C<0,\ B<0\end{cases}$

30	$Z = \begin{cases} C^2 + \max(A, B+1, C+1), & \text{если} \quad A^2 - B + C \geq 0 \\ \ln(A+B) + \cos(C+1), & \text{если} \quad A^2 - B + C < 0, \ B \geq 0 \\ \max(B, C) + \min(A, B), & \text{если} \quad A^2 - B + C < 0, \ B < 0 \end{cases}$

3. Составить структурные схемы для следующих циклических вычислительных процессов:

№ п/п	Условие задачи
1	Вычислить $Z = 0.2X - Y^2$, где X – минимальный элемент массива Ai, i = 1, …, 25, Y – максимальный элемент массива Bj, j = 1, …, 30.
2	Даны массивы Ai, i = 1, … , 15 и Dj, j = 1, …, 20. Найти минимальные элементы этих массивов A_{min} и D_{min}. Определить, какой элемент меньше.
3	Дан массив Xi, i = 1, …, 15. Найти среднее арифметическое значение положительных и отрицательных элементов массива R1 и R2. Определить, какой из найденных значений больше по абсолютной величине.
4	Дан массив Ai, i = 1, 2, …, 15. Найти сумму положительных элементов и сумму отрицательных элементов массива S1 и S2. Определить, что больше по абсолютной величине – S1 или S2.
5	Даны массивы Xi, i = 1, …, 10 и Yj, j = 1, 2, …, 15. Найти максимальные элементы массивов X_{max} и Y_{max}. Определить, какой элемент больше – X_{max} или Y_{max} – и на сколько.
6	Дан массив Xi, i = 1, 2, …, 15. Найти произведения положительных и отрицательных элементов массива P1 и P2. Определить, что больше по абсолютной величине – P1 или P2.
7	Дан массив Xi, i = 1, …, 15. Найти кол-во положительных и отрицательных элементов массива K1 и K2. Определить, каких элементов больше – положительных или отрицательных – и на сколько?
8	Дан массив Xi, i = 1, 2, …, 15. Найти номера минимального X_{min} и максимального X_{max} элементов массива. Определить, какой из элементов лежит ближе к началу массива.
9	Даны массивы Xi, i = 1, …, 10 и Yj, j = 1, 2, …, 15. Найти номера максимальных элементов массивов X_{max} и Y_{max}. Определить, какой из элементов лежит ближе к началу массиву?
10	Вычислить $Z = 0.2X - Y^2$, где X – максимальный элемент массива Ai, i = 1, 25, Y – минимальный элемент массива Bj, j = 1, 30.
11	Даны массивы Ai, i = 1, … , 13 и Dj, j = 1, …, 15. Найти максимальные элементы этих массивов A_{max} и D_{max}. Определить, какой элемент больше.
12	Дан массив Bi, i = 1, …, 15. Найти среднее арифметическое значение положительных и отрицательных элементов массива R1 и R2. Определить, какой из найденных элементов меньше по абсолютной величине.
13	Дан массив Ai, i = 1, 2, …, 15. Найти сумму элементов, стоящих на четных местах массива, S1 и сумму элементов, стоящих на нечетных местах, S2. Определить, что больше по абсолютной величине – S1 или S2.

14	Даны массивы Xi, i = 1, …, 10 и Yj, j = 1, 2, …, 15. Найти номера максимальных элементов массивов X_{max} и Y_{max}. Определить, какой из элементов лежит дальше от начала массива.
15	Дан массив Bi, i = 1, …, 27. Найти произведение элементов, стоящих на четных местах массива, P1 и произведение элементов, стоящих на нечетных местах, P2. Определить, что меньше по абсолютной величине – P1 или P2.
16	Дан массив Di, i = 1, …, 14. Найти кол-во положительных и отрицательных элементов массива K1 и K2. Определить, каких элементов больше, положительных или отрицательных и на сколько?
17	Дан массив Xi, I = 1, 2, …, 15. Найти произведения положительных и отрицательных элементов массива P1 и P2. Определить, что меньше по абсолютной величине – P1 или P2.
18	Даны массивы Xi, i = 1, …, 10 и Yj, j = 1, 2, …, 15. Найти максимальные элементы массивов X_{max} и Y_{max}. Определить, какой из X_{max} или Y_{max} больше и на сколько.
19	Вычислить $Z = 0.3X - Y^3$, где X – максимальный элемент массива Ai, i = 1, …, 25, Y – минимальный элемент массива Bj, j = 1, …, 30.
20	Даны массивы Bi, i = 1, … , 15 и Cj, j = 1, …, 20. Найти минимальные элементы этих массивов B_{min} и C_{min}. Определить, какой элемент меньше.
21	Даны массивы Ai, i = 1, …, 15 и Cj, j = 1, …, 22. Найти среднее арифметическое значение положительных элементов массива Ra и Rc. Определить, какое из найденных значений меньше по абсолютной величине.
22	Дан массив Ci, i = 1, 2, …, 15. Найти сумму положительных элементов и сумму отрицательных элементов массива S1 и S2. Определить, что меньше по абсолютной величине – S1 или S2.
23	Даны массивы Xi, i = 1, …, 10 и Yj, j = 1, 2, …, 15. Найти номера максимальных элементов массивов X_{max} и Y_{max}. Определить, какой из X_{max} или Y_{max} лежит дальше от начала массива.
24	Даны массивы Ai, i = 1, …, 20 и Cj, j = 1, …, 25. Найти количество положительных элементов в каждом массиве. Определить в каком массиве больше положительных элементов.
25	Дан массив Ai, i = 1, …, 30. Найти кол-во положительных и отрицательных элементов массива K1 и K2. Определить, каких элементов больше – положительных или отрицательных – и на сколько?
26	Дан массив Yi, i = 1, …, 25. Найти произведения положительных и отрицательных элементов массива P1 или P2. Определить, что меньше по абсолютной величине – P1 или P2.
27	Даны массивы Xi, i = 1, …, 15 и Yj, j = 1, …, 12. Найти максимальные элементы массивов X_{max} и Y_{max}. Определить, какой элемент меньше – X_{max} или Y_{max} – и на сколько.
28	Дан массив Ai, i = 1, …, 18. Найти сумму положительных и произведение отрицательных элементов массива S и P. Определить, какое значение больше – S или P – по абсолютной величине и на сколько.

29	Даны массивы Ai, i = 1, …, 30 и Cj, j = 1, …, 25. Найти количество отрицательных элементов в каждом массиве. Определить в каком массиве больше отрицательных элементов.
30	Дан массив Bi, i = 1, …, 18. Найти сумму отрицательных и произведение положительных элементов массива S и P. Определить, какое значение больше – S или P – по абсолютной величине и на сколько.

Лабораторная работа №2. Структура программы на языке Си. Простые операции и операторы языка Си. Ввод-вывод в программах на языке Си (6 часов)

Краткие теоретические сведения

Структура программы. Программа на языке Си строится из отдельных блоков, называемых функциями. В свою очередь, каждая функция составлена из предложений, среди которых могут директивы препроцессора, адресованные компилятору, объявления (определения) различных объектов программы, операторы, являющиеся инструкциями для компьютера.

В любой программе на языке Си имеется основная (главная) функция с именем `main()`, которая имеет следующую структуру:

```
int main()
{
  ...
  return 0;
}
```

Алфавит языка. Для записи предложений программы на языке Си применяются определённые символы, составляющие алфавит языка.

Алфавит языка – совокупность допустимых в языке символов или групп символов, рассматриваемых как единое целое.

Для записи операторов используются следующие символы: латинские прописные и строчные буквы, цифры и специальные знаки. К специальным знакам относятся: точка (.), запятая (,), двоеточие (:), точка с запятой (;) и некоторые другие. Совокупность символов, используемых в языке, называется алфавитом языка.

Для записи комментариев, символьных и строковых констант (значений) могут использоваться буквы национальных алфавитов, например, русского алфавита.

Различают *видимые* (печатные) и *управляющие* символы. Первые могут быть отображены на экране монитора либо выведены на принтер. Вторые вызывают определённые действия в компьютере, например, подача звукового сигнала – код 7_{10}, возврат курсора на один шаг – код 8_{10}, горизонтальная табуляция – код 9_{10}, перевод курсора на новую строку – код 10_{10}, перемещение курсора в начало строки – код 13_{10} и т.д. Такие управляющие символы имеют десятичные коды от 0 до 31, 127.

Идентификаторы. Важным понятием языка является идентификатор, который используется в качестве имени функции, переменной, константы и других. Идентификаторы должны выбираться с учётом следующих правил:

1. Они должны начинаться с буквы латинского алфавита (a,...,z, A,...,Z) или с символа подчеркивания (_).
2. В них могут использоваться буквы латинского алфавита, символ подчеркивания и цифры (0, ..., 9). Использование других символов в идентификаторах запрещено.
3. В языке Си буквы нижнего регистра (a, ..., z), применяемые в идентификаторах, отличаются от букв верхнего регистра (A, ..., Z). Это означает, что следующие идентификаторы считаются различными: name, NaMe, NAME и т.д.
4. Идентификаторы могут иметь любую длину, но воспринимается и используется для различения объектов (функций, переменных, констант и т.д.) только часть символов. Их число меняется для разных систем программирования, но в соответствии со стандартом ANSI C не превышает 32 (в языке C++ это ограничение снято).
5. Идентификаторы для новых объектов не должны совпадать с ключевыми словами языка и именами стандартных функций из библиотеки.

Комментарии. В программах на языке Си важная роль отводится комментариям. Они позволяют пояснить выполняемые действия в программе, а также повышают наглядность и удобство чтения программ. Комментарии обрамляются следующими парами символов: /* и */. Их можно записывать в любом месте программы. Пример комментария: /* это комментарий */

В языке C++ введена еще одна форма записи комментариев. Все, что находится после знака // до конца текущей строки, будет также рассматриваться как комментарий. Отметим, что компилятор языка Си, встроенный в систему программирования Borland C++, позволяет использовать данный комментарий и в программах на Си.

Комментарии, пробелы, символы табуляции и перехода на новую строку, представленные в программах на Си, компилятором игнорируются. Это позволяет записывать различные выражения в хорошо читаемом виде. Кроме того, строки программы можно начинать с любой позиции, что даёт возможность выделять в тексте группы операторов.

Типы данных

Программы оперируют с различными данными, которые могут быть простыми и структурированными. *Простые данные* – это целые и вещественные числа, символы и указатели (адреса объектов в памяти). Целые числа не имеют, а вещественные имеют дробную часть. *Структурированные данные* – это массивы и структуры; они будут рассмотрены ниже.

В языке различают понятия «тип данных» и «модификатор типа». Тип данных – это, например, целый, а модификатор – со знаком или без знака. Целое со знаком будет иметь как положительные, так и отрицательные значения, а

целое без знака – только положительные значения. В языке Си можно выделить пять базовых типов, которые задаются следующими ключевыми словами:

- *char* – символьный;
- *int* – целый;
- *float* – вещественный;
- *double* – вещественный двойной точности;
- *void* – не имеющий значения.

Краткая характеристика базовых типов:

1. Переменная типа *char* имеет размер 1 байт, её значениями являются различные символы из кодовой таблицы, например: 'z', ':', 'ц' (при записи в программе символы заключаются в одинарные кавычки).
2. Размер переменной типа *int* в стандарте языка Си не определён. В большинстве систем программирования размер переменной типа *int* соответствует размеру целого машинного слова. Например, в компиляторах для 16-разрядных процессоров переменная типа *int* имеет размер 2 байта. В этом случае знаковые значения этой переменной могут лежать в диапазоне от -32768 до 32767.
3. Ключевое слово *float* позволяет определить переменные вещественного типа. Их значения имеют дробную часть, отделяемую точкой, например: -5.6, 3.14 и т.п. Вещественные числа могут быть записаны также в форме с плавающей точкой, например: -1.09e+4. Число перед символом «e» называется мантиссой, а после «e» – порядком. Переменная типа *float* занимает в памяти 32 бита. Она может принимать значения в диапазоне от 3.4e-38 до 3.4e+38.
4. Ключевое слово *double* позволяет определить вещественную переменную двойной точности. Она занимает в памяти в два раза больше места, чем переменная типа *float* (т.е. её размер 64 бита). Переменная типа *double* может принимать значения в диапазоне от 1.7e-308 до 1.7e+308.
5. Ключевое слово *void* (не имеющий значения) используется для нейтрализации значения объекта, например, для объявления функции, не возвращающей никаких значений.

Объект некоторого базового типа может быть модифицирован. С этой целью используются специальные ключевые слова, называемые модификаторами. В стандарте ANSI языка Си имеются следующие модификаторы типа:

- *unsigned* – беззнаковый (без знака);
- *signed* – знаковый (со знаком);
- *short* – короткий;
- *long* – длинный.

Модификаторы записываются перед спецификаторами типа, например: *unsigned char*. Если после модификатора опущен спецификатор, то компи-

лятор предполагает, что этим спецификатором является *int*. Таким образом, следующие строки:

```
long a;
long int a;
```

являются идентичными и определяют переменную `a` с типом длинное целое.

Таблица 2.1 иллюстрирует возможные сочетания модификаторов (*`unsigned`*, *`signed`*, *`short`*, *`long`*) со спецификаторами (*`char`*, *`int`*, *`float`* и *`double`*), а также показывает размер и диапазон значений объекта (для 16-разрядных компиляторов).

Таблица 2.1

Тип	Размер в байтах (битах)	Диапазон значений
`char`	1 (8)	от -128 до 127
`unsigned char`	1 (8)	от 0 до 255
`signed char`	1 (8)	от -128 до 127
`int`	2 (16)	от -32768 до 32767
`unsigned int`	2 (16)	от 0 до 65535
`signed int`	2 (16)	от -32768 до 32767
`short int`	2 (16)	от -32768 до 32767
`unsigned short int`	2 (16)	от 0 до 65535
`signed short int`	2 (16)	от -32768 до 32767
`long int`	4 (32)	от -2147483648 до 2147483647
`unsigned long int`	4 (32)	от 0 до 4294967295
`signed long int`	4 (32)	от -2147483648 до 2147483647
`float`	4 (32)	от 3.4E-38 до 3.4E+38
`double`	8 (64)	от 1.7E-308 до 1.7E+308
`long double`	10 (80)	от 3.4E-4932 до 3.4E+4932

Переменные и константы

Все переменные до их использования должны быть определены (объявлены). При этом задаётся тип, а затем идёт список из одной или более переменных этого типа, разделённых запятыми. Например:

```
int a, b, c;
char x, y;
```

В языке различают понятия объявления переменной и её определения. Объявление устанавливает свойства объекта: его тип (например, целый), размер

(например, 4 байта) и т.д. Определение наряду с этим вызывает выделение памяти (в приведённом примере дано определение переменных).

Переменные можно разделять по строкам произвольным образом, например:

```
float a;
float b;
```

Переменные в языке Си могут быть инициализированы при их определении:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r*123;
```

Выясним теперь, где в тексте программы определяются данные. В языке Си возможны глобальные и локальные объекты. Первые определяются вне функций и, следовательно, доступны для любой из них. Локальные объекты по отношению к функциям являются внутренними. Они начинают существовать, при входе в функцию и уничтожаются после выхода из нее. Ниже показана структура программы на Си и возможные места в программе, где определяются глобальные и локальные объекты.

```
int a;      /* Определение глобальной переменной */
int function (int b, char c); /* Объявление функции
                          (т.е. описание ее заголовка)*/
void main (void)
{                  //Тело программы
   int d, e;       //Определение локальных переменных
   float f;        //Определение локальной переменной
   ...
}
int function (int b, char c) /* Определение функции и
                 формальных параметров (по существу -
                 локальных переменных) b и c */
{                //Тело функции
   char g;       //Определение локальной переменной
   ...
}
```

Выполнение программы всегда начинается с вызова основной функции `main()`, которая содержит тело программы. Тело программы, подобно телу любой другой функции, заключено в фигурные скобки – открывающую и закрывающую.

В языке Си все определения должны следовать перед операторами, составляющими тело функции. В языке C++ это ограничение снято и определения могут находиться в любом месте программы. Если они сделаны в функции, то соответствующие объекты будут локальными, а если вне функций, то глобальными.

Наряду с переменными в языке существуют следующие виды констант:

- *вещественные,* например 123.456, 5.61e-4. Они могут снабжаться суффиксом F (или f), например 123.456F, 5.61e-4f;
- *целые,* например 125;
- *короткие целые,* в конце записи которых добавляется буква (суффикс) H (или h), например 275h, 344H;
- *длинные целые,* в конце записи которых добавляется буква (суффикс) L (или l), например 361327L;
- *беззнаковые,* в конце записи которых добавляется буква U (или u), например 62125U;
- *восьмеричные,* в которых перед первой значащей цифрой записывается нуль (0), например 071;
- *шестнадцатеричные,* в которых перед первой значащей цифрой записывается пара символов нуль-икс (0x), например 0x5F;
- *символьные* – единственный символ, заключенный в одинарные кавычки, например 'O', '2', '.' и т.п. Символы, не имеющие графического представления, можно записывать, используя специальные комбинации, например \n (код 10), \0 (код 0). Эти комбинации представляются двумя символами, хотя фактически это один символ. Так же можно представить любой двоичный образ одного байта: '\NNN', где NNN – от одной до трёх восьмеричных цифр. Допускается и шестнадцатеричное задание кодов символов, которое представляется в виде: '\x2B', '\x36' и т.п.;
- *строковые* – последовательность из нуля символов и более, заключенная в двойные кавычки, например: "Это строковая константа". Кавычки не входят в строку, а лишь ограничивают её. Строка представляет собой массив из перечисленных элементов, в конце которого помещается байт с символом '\0'. Таким образом, число байтов, необходимых для хранения строки, на единицу превышает число символов между двойными кавычками;
- *константное выражение,* состоящее из одних констант, которое вычисляется во время трансляции (например: `a=60+301`);
- типа `long double`, в конце записи которых добавляется буква L (или l), например: 1234567.89L.

Операции языка Си

Любое выражение языка состоит из операндов (переменных, констант и др.), соединённых знаками операций. Знак операции – это символ или группа символов, которые сообщают компилятору о необходимости выполнения определённых арифметических, логических или других действий.

Операции выполняются в строгой последовательности в соответствии с их приоритетами (старшинством). В таблице 2.2 перечислены различные операции языка Си. Порядок выполнения операций может регулироваться с помощью круглых скобок.

Таблица 2

Знак операции	Назначение операции
( )	Вызов функции
[ ]	Выделение элемента массива
.	Выделение элемента записи
->	Выделение элемента записи
!	Логическое отрицание
~	Поразрядное отрицание
-	Изменение знака
++	Увеличение на единицу
--	Уменьшение на единицу
&	Взятие адреса
*	Обращение по адресу
(тип)	Преобразование типа (например, (float) a)
sizeof( )	Определение размера в байтах
*	Умножение
/	Деление
%	Определение остатка от целочисленного деления
+	Сложение
-	Вычитание
<<	Сдвиг влево
>>	Сдвиг вправо
<	Меньше, чем
<=	Меньше или равно
>	Больше, чем
>=	Больше или равно
= =	Равно
!=	Не равно
&	Поразрядное логическое "И"
^	Поразрядное исключающее "ИЛИ"
|	Поразрядное логическое "ИЛИ"
&&	Логическое "И"

||	Логическое "ИЛИ"
?:	Условная (тернарная) операция
=	Присваивание
+=, - =, *=, /=, %=, <<=, >>=, &=, |=, ^=	Составные операции присваивания (например, a *= b (т.е. a = a * b) и т.д.)
,	Операция запятая

Для исключения путаницы в понятиях «операция» и «оператор», отметим, что оператор – это наименьшая исполняемая единица программы. Различают операторы выражения, действие которых состоит в вычислении заданных выражений (например: `a = sin(b)+c; j++;`), операторы объявления, составные операторы, пустые операторы, операторы метки, цикла и т.д. Для обозначения конца оператора в языке Си используется точка с запятой. Что касается составного оператора (или блока), представляющего собой набор логически связанных операторов, помещённых между открывающей и закрывающей фигурными скобками (так называемыми «операторными» скобками – {}), то за ним точка с запятой не ставится. Отметим, что блок отличается от составного оператора наличием определений в теле блока.

Охарактеризуем основные операции языка Си. Сначала рассмотрим одну из них – операцию присваивания (=). Выражение вида

```
x = y;
```

присваивает переменной `x` значение переменной `y`. Операцию «=» разрешается использовать многократно в одном выражении, например:

```
x = y = z = 100;
```

Различают *унарные* и *бинарные операции*. Первые выполняются над одним операндом, вторые – над двумя. Начнем их рассмотрение с операций, отнесенных к первой из следующих традиционных групп:

1. Арифметические операции.
2. Логические операции и операции отношения.
3. Операции с битами.

Арифметические операции задаются следующими символами (см. табл. 2): +, -, *, /, %. Последнюю из них нельзя применять к переменным вещественного типа. Например:

```
a = b + c;
x = y - z;
r = t * v;
s = k / l;
p = q % w;
```

Логические операции отношения задаются следующими символами (см. табл. 2): && ("И"), || ("ИЛИ"), ! ("НЕ"), >, >=, <, <= , = = (равно), != (не равно).

Традиционно эти операции должны давать одно из двух значений: истину или ложь. В языке Си принято следующее правило: истина - это любое ненулевое значение; ложь - это нулевое значение. Выражения, использующие логические операции и операции отношения, возвращают 0 для ложного значения и 1 для истинного. Ниже приводится таблица истинности для логических операций.

Таблица 3

x	y	x&&y	x||y	!x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Битовые операции можно применять к переменным, имеющим типы `int`, `char`, а также их вариантам (например, *`long int`*). Их нельзя применять к переменным типов *`float`*, *`double`*, *`void`* (или более сложных типов). Эти операции задаются следующими символами: ~ (поразрядное отрицание), << (сдвиг влево), >> (сдвиг вправо), & (поразрядное «И»), ^ (поразрядное исключающее «ИЛИ»), | (поразрядное «ИЛИ»).

Например, если *a* = 0000 1111 и *b* = 1000 1000, то результат выполнения следующих операций будет следующим:

```
    ~a = 1111 0000,
a << 1 = 0001 1110,
a >> 1 = 0000 0111,
a & b  = 0000 1000,
a ^ b  = 1000 0111,
a | b  = 1000 1111.
```

В языке Си предусмотрены операции инкремента (++) и декремента (--), которые предназначены для увеличения и уменьшения на единицу значения операнда. Операции ++ и -- можно записывать как перед операндом, так и после него. В первом случае (`++n` или `--n`) значение операнда (`n`) изменяется перед его использованием в соответствующем выражении, а во втором (`n++` или `n--`) – после его использования. Рассмотрим две следующие строки программы:

```
a = b + c++;
a1 = b1 +  ++c1;
```

Предположим, что `b = b1 = 2`, `c = c1 = 4`. Тогда после выполнения операций: `a = 6`, `b = 2`, `c = 5`, `a1 = 7`, `b1 = 2`, `c1 = 5`.
Достаточно широко используются также выражения с тернарной (трёхоперандной) или условной операцией `?:`. Например, в следующем выражении:

```
y = x ? a : b;
```

`y = a`, если `x` не равно нулю (т.е. истинно), и `y = b`, если `x` равно нулю (ложно). Следующее выражение:

```
y = (a>b) ? a : b;
```

позволяет присвоить переменной `y` значение большей переменной (`a` или `b`), т.е. `y = max(a, b)`.

Ещё одним отличием языка Си является то, что выражение вида `a=a+5;` можно записать в другой форме: `a += 5;`. Вместо знака + можно использовать и символы других бинарных операций (см. табл. 2).

Преобразование типов

Если в выражении появляются операнды различных типов, то они преобразуются к некоторому общему типу, при этом к каждому арифметическому операнду применяется такая последовательность правил:

1. Если один из операндов в выражении имеет тип *`long double`*, то остальные тоже преобразуются к типу *`long double`*.
2. В противном случае, если один из операндов в выражении имеет тип *`double`*, то остальные тоже преобразуются к типу *`double`*.
3. В противном случае, если один из операндов в выражении имеет тип *`float`*, то остальные тоже преобразуются к типу *`float`*.
4. В противном случае, если один из операндов в выражении имеет тип *`unsigned long`*, то остальные тоже преобразуются к типу *`unsigned long`*.
5. В противном случае, если один из операндов в выражении имеет тип *`long`*, то остальные тоже преобразуются к типу *`long`*.
6. В противном случае, если один из операндов в выражении имеет тип *`unsigned`*, то остальные тоже преобразуются к типу *`unsigned`*.
7. В противном случае все операнды преобразуются к типу *`int`*. При этом тип *`char`* преобразуется в *`int`* со знаком; тип *`unsigned char`* в *`int`*, у которого старший байт всегда нулевой; тип *`signed char`* в *`int`*, у которого в знаковый разряд передается знак из *`char`*; тип *`short`* в *`int`* (знаковый или беззнаковый).

Предположим, что вычислено значение некоторого выражения в правой части оператора присваивания. В левой части оператора присваивания записана некоторая переменная, причем ее тип отличается от типа результата в правой части. Здесь правила преобразования очень простые: значение справа от оператора присваивания преобразуется к типу переменной слева от оператора присваивания. Если размер результата в правой части больше размера операнда в левой части, то старшая часть этого результата будет потеряна.

В языке Си можно явно указать тип любого выражения. Для этого используется операция преобразования («приведения») типа. Она применяется следующим образом:

```
(тип) выражение
```

(в качестве элемента *тип* можно указать любой допустимый в языке Си тип данных).

Рассмотрим следующий пример:

```
int a = 30000;
float b;
...
b = (float) a * 12;
```

(переменная a целого типа явно преобразована к типу *float*; если этого не сделать, то результат будет потерян, так как a * 12 > 32767).

Преобразование типа также может использоваться для преобразования типов аргументов при вызове функций.

Ввод-вывод данных

Операции ввода/вывода в языке Си организованы посредством библиотечных функций (причём их довольно много).

Простейший механизм ввода – это чтение одного символа из стандартного входного потока (с клавиатуры) с помощью функции *getchar()*, имеющей следующий прототип (т.е. описание заголовка):

```
int getchar(void);
```

Здесь определён тип единственного аргумента (*void*) и тип возвращаемого функцией значения (*int*).

Оператор следующего вида:

```
x = getchar();
```

присваивает переменной *x* очередной вводимый символ. Переменная *x* должна иметь символьный или целый тип.

Аналогично, простейший механизм вывода – функция *putchar(x)*, которая выдаёт значение переменной *x* в стандартный выходной поток (на экран монитора). Функция *putchar()* имеет следующий прототип:

```
int putchar(int);
```

Объявления функций *getchar()* и *putchar()* сделаны в заголовочном файле *stdio.h*, содержащем описания заголовков библиотечных функций стандартного ввода/вывода. Чтобы библиотечные функции стали доступны программе, к ней необходимо подключить данный файл. Подключение осуществляется с помощью следующей директивы #*include* препроцессора языка Си:

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

помещаемой в начало программы (подробнее см. ниже).

Следует отметить, что для функции *getchar()* после нажатия клавиши с вводимым символом необходимо нажать клавишу <Enter>. Иногда это создаёт

определённые неудобства. Функции *getch*() и *getche*() устраняют их. Они имеют следующие прототипы:

```
int getch(void);
int getche(void);
```

Обе эти функции вводят символ сразу же после нажатия соответствующей клавиши (здесь не требуется дополнительно нажимать клавишу <Enter>). Отличие между ними заключается в том, что *getche*() отображает вводимый символ на экране монитора, а *getch*() – нет. Прототипы этих функций содержатся в файле *conio.h* (консольный ввод/вывод). Для их использования файл *conio.h* также следует подключить к программе с помощью директивы #*include*.

Форматированный вывод данных

Функция *printf*() (прототип содержится в файле *stdio.h*) обеспечивает форматированный вывод. Её можно записать в следующем формальном виде:

```
printf("управляющая строка",аргумент_1,аргумент_2,...);
```

Управляющая строка содержит компоненты трёх типов: обычные символы, которые просто копируются в стандартный выходной поток (выводятся на экран монитора); спецификации преобразования, каждая из которых вызывает вывод на экран очередного аргумента из последующего списка; управляющие символьные константы.

Каждая спецификация преобразования начинается со знака процента % и заканчивается некоторым символом, задающим преобразование. Между знаком % и символом преобразования могут встречаться другие знаки в соответствии со следующим форматом:

```
%[признаки][ширина_поля][точность][F|N|h|l|L] c_n
```

Все параметры в квадратных скобках не являются обязательными.

На месте параметра `c_n` (символ преобразования) могут быть записаны:

c – значением аргумента является символ;

d или **i** – значением аргумента является десятичное целое число;

e – значением аргумента является вещественное десятичное число в экспоненциальной форме вида 1.23e+2;

E – значением аргумента является вещественное десятичное число в экспоненциальной форме вида 1.23E+2;

f – значением аргумента является вещественное десятичное число с плавающей точкой;

g (или **G**) – используется, как e или f, и исключает вывод незначащих нулей;

o – значением аргумента является восьмеричное целое число;

s – значением аргумента является строка символов (символы строки выводятся до тех пор, пока не встретится символ конца строки или же не будет, выведено число символов, заданное точностью);

u – значением аргумента является беззнаковое целое число;

x – значением аргумента является шестнадцатеричное целое число с цифрами 0,..., 9, a, b, c, d, e, f;

X – значением аргумента является шестнадцатеричное целое число с цифрами 0,..., 9, A, B, C, O, E, F;

p – значением аргумента является указатель;

n – применяется в операциях форматирования. Аргумент, соответствующий этому символу спецификации, должен быть указателем на целое. В него возвращается номер позиции строки (отображаемой на экране), в которой записана спецификация `%n`.

Необязательные параметры в спецификации преобразования:

- признак минус (-) указывает, что преобразованный параметр должен быть выровнен влево в своем поле;
- признак плюс (+) требует вывода результата со знаком;
- строка цифр, задающая минимальный размер поля (ширина поля). Здесь может так же использоваться символ *, который тоже позволяет задать минимальную ширину поля и точность представления выводимого числа;
- точка (.), отделяющая размер поля от последующей строки цифр;
- строка цифр, задающая максимальное число выводимых символов, или же количество цифр, выводимых справа от десятичной точки в значениях типов *`float`* или *`double`* (точность);
- символ F, определяющий указатель типа *`far`*;
- символ N, определяющий указатель типа *`near`*;
- символ h, определяющий аргумент типа *`short int`* (используется вместе с символами преобразования d, i, o, u, x, X);
- символ l, указывающий, что соответствующий аргумент имеет тип *`long`* (в случае символов преобразования d, i, o, u, x, X) или double (в случае символов преобразования e, E, f, g, G);
- символ L, указывающий, что соответствующий аргумент имеет тип *`long double`* (используется вместе с символами преобразований e, E, f, g, G);
- символ #, который может встречаться перед символами преобразования g, f, e и перед символом x. В первом случае всегда будет выводиться десятичная точка, а во втором – префикс 0x перед соответствующим шестнадцатеричным числом.

Если после знака % записан не символ преобразования, то он выводится на экран. Таким образом, строка %% приводит к выводу на экран знака %.

Функция *`printf`*`()` использует управляющую строку, чтобы определить, сколько всего аргументов и каковы их типы. Аргументами могут быть пе-

ременные, константы, выражения, вызовы функций; главное, чтобы их значения соответствовали заданной спецификации.

При наличии ошибок, например, в числе аргументов или типе преобразования, результаты будут неверными.

Среди управляющих символьных констант наиболее часто используются следующие:

\a – для кратковременной подачи звукового сигнала;

\b – для перевода курсора влево на одну позицию;

\f – для подачи формата;

\n – для перехода на новую строку;

\r – для возврата каретки;

\t – горизонтальная табуляция;

\v – вертикальная табуляция;

\\ – вывод символа \;

\' – вывод символа ' ;

\" – вывод символа ";

\? – вывод символа ?.

Например, в результате следующего вызова функции:

```
printf("\tComputer\n%d\n", i);
```

сначала выполняется горизонтальная табуляция (`\t`), т.е. курсор сместится от края экрана, затем на экран будет выведено слово `Computer`, после этого курсор переместится в начало следующей строки (`\n`), затем будет выведено целое число `i` по формату `%d` (десятичное целое), и, окончательно, курсор перейдет в начало новой строки (`\n`).

Напечатать строку символов можно и так:

```
printf("Это строка символов");
```

Форматированный ввод данных

Функция `scanf()` (прототип содержится в файле `stdio.h`) обеспечивает форматированный ввод. Ее можно записать в следующем формальном виде:

```
scanf("управляющая строка",аргумент_1,аргумент_2,...);
```

Аргументы `scanf()` должны быть указателями на соответствующие значения. Для этого перед именем переменной записывается символ &. Назначение указателей будет рассмотрено далее.

Управляющая строка содержит спецификации преобразования и используется для установления количества и типов аргументов. В неё могут включаться:

- пробелы, символы табуляции и перехода на новую строку (все они игнорируются);

- спецификации преобразования, состоящие из знака %, возможно, символа * (запрещение присваивания), возможно, числа, задающего максимальный размер поля, и самого символа преобразования;
- обычные символы, кроме знака процента % (считается, что они должны совпадать с очередными неизвестными символами во входном потоке).

Рассмотрим символы преобразования функции *scanf()* (указываются после символа %):

c – на входе ожидается появление одиночного символа;

d или **i** – на входе ожидается десятичное целое число и аргумент является указателем на переменную типа *int*;

D или **l** – на входе ожидается десятичное целое число и аргумент является указателем на переменную типа *long*;

e или **E** – на входе ожидается вещественное число с плавающей точкой;

f – на входе ожидается вещественное число с плавающей точкой;

g или **G** – на входе ожидается вещественное число с плавающей точкой;

o – на входе ожидается восьмеричное целое число и аргумент является указателем на переменную типа *int*;

O – на входе ожидается восьмеричное целое число и аргумент является указателем на переменную типа *long*;

s – на входе ожидается появление строки символов;

x – на входе ожидается шестнадцатеричное целое число и аргумент является указателем на переменную типа *int*;

X – на входе ожидается шестнадцатеричное целое число и аргумент является указателем на переменную типа *long*;

p – на входе ожидается появление указателя в виде шестнадцатеричного числа;

n – применяется в операциях форматирования. Аргумент, соответствующий этому символу спецификации, должен быть указателем на целое. В него возвращается номер позиции (после ввода), в которой записана спецификация %n;

u – на входе ожидается беззнаковое целое число и аргумент является указателем на переменную типа *unsigned int*;

U – на входе ожидается беззнаковое целое число и аргумент является указателем на переменную типа *unsigned long*;

[] – сканирует входную строку для получения символов.

Перед некоторыми символами преобразования могут записываться следующие модификаторы:

F – изменяет указатель, заданный по умолчанию, на указатель типа *far*;

N – изменяет указатель, заданный по умолчанию, на указатель типа *near*;

h – преобразует аргумент к типу *short int* (может записываться перед символами d, i, o, u, x);

l – преобразует аргумент к типу *long int* (может записываться перед символами d, i, o, u, x);

L – преобразует аргумент к типу *long double* (может записываться перед символами e, f, g).

Ввести целое число (*int* a;), символ (*char* b;) и вещественное число (*float* t;) можно так:

```
scanf("%d", &a);
scanf("%c", &b);
scanf("%d%c%f",&a, &b, &t);
```

Практические задания

1. Написать программы в соответствии с алгоритмами решения задач, разработанными в лабораторной работе №1, в которых используются описанные выше средства форматированного ввода-вывода для ввода исходных данных и вывода результатов (вычислительную часть выполнить в упрощённом варианте).

2. Реализовать написанные программы в инструментальной среде программирования, и выполнить пробный прогон программы.

Лабораторная работа №3. Программирование линейных и разветвляющихся алгоритмов на языке Си. Условные операторы языка Си (6 часов)

Для организации условных и безусловных переходов в программе на языке Си используются операторы: *if – else*, *switch* и *goto*. Первый из них записывается следующим образом:

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

Если *условие* в скобках принимает истинное значение, выполняется *оператор_1*, если ложное – *оператор_2*. Если необходимо выполнить несколько операторов, то они заключаются в фигурные скобки. В операторе *if* слово *else* может отсутствовать.

В операторе *if – else* непосредственно после ключевых слов *if* и *else* должны следовать другие операторы. Если хотя бы один из них является оператором *if*, его называют вложенным. Согласно принятому в языке Си соглашению слово *else* всегда относится к ближайшему предшествующему ему *if*.

Оператор *switch* позволяет выбрать одну из нескольких альтернатив. Он записывается в следующем формальном виде:

```
switch (выражение)
{
   case константа_1:  операторы_1;
                      break;
   case константа_2:  операторы_2;
```

```
                        break;
    ...                 ...
    default:            операторы_default;
}
```

Здесь вычисляется значение целого выражения, заключённого в скобки (его иногда называют селектором), и оно сравнивается со всеми константами (константными выражениями). Все константы должны быть различными. При совпадении выполнится соответствующий вариант операторов (один или несколько операторов). Вариант с ключевым словом *default* реализуется, если ни один другой не подошел (вариант *default* может и отсутствовать). Если *default* отсутствует, а совпадения с константами отсутствуют, то ни один вариант не выполняется.

Для прекращения последующих проверок после успешного выбора некоторого варианта используется оператор *break*, обеспечивающий немедленный выход из переключателя *switch*. Допускаются вложенные конструкции *switch*.

Рассмотрим правила выполнения безусловного перехода, который можно представить в следующей форме:

```
goto метка;
```

Метка – это любой идентификатор, после которого поставлено двоеточие. Оператор *goto* указывает на то, что выполнение программы необходимо продолжить, начиная с оператора, перед которым записана метка. Метку можно поставить перед любым оператором в той функции, где находится соответствующий ей оператор *goto*. Она не требует объявления.

Ниже представлен текст программы на языке Си, в котором используются условные операторы и вызовы функций для форматированного ввода-вывода данных.

```
// Решение квадратного уравнения
#include <stdio.h>
#include <conio.h>
#include <math.h>
void main()
{
   float a, b, c;      // коэффициенты уравнения
   float x1, x2;       // корни уравнения
   float d;            // дискриминант
   printf("\nРешение квадратного уравнения\n");
   printf("Введите в одной строке значения коэффициентов\n");
   scanf("%f%f%f", &a, &b, &c);              // ввод коэффициентов
   if (a==0)
       printf("Уравнение не является квадратным\n");
   else
       {
          d=b*b-4*a*c;    // дискриминант
```

```
            if (d<0)
                printf("Уравнение не имеет действительных корней\n");
            else
               {
                   if (d==0)
                      x1=x2=-b/(2*a);
                   else
                      {
                        x1=(-b+sqrt(d))/(2*a);
                         x2=(-b-sqrt(d))/(2*a);
                       }
                    printf("Корни уравнения: x1=%3.2f x2=%3.2f \n", x1, x2);
                }
            }
        printf("\nДля завершения нажмите любую клавишу");
        while(!kbhit());
}
```

Практические задания

1. Добавить в программы, разработанные в лабораторной работе №2, вычислительную часть, использующую условные операторы.

2. Реализовать написанные программы в инструментальной среде программирования, и выполнить прогон программы.

Лабораторная работа №4. Программирование циклических алгоритмов на языке Си. Операторы цикла языка Си. Использование массивов в программах на языке Си (6 часов)

Циклы организуются, чтобы выполнить некоторый оператор или группу операторов определённое число раз. В языке Си имеются три оператора цикла: *for*, *while* и *do-while*.

Первый из них, называемый оператором цикла с параметром, формально записывается, в следующем виде:

```
for (выражение_1; выражение_2; выражение_3) тело_цикла
```

Тело цикла составляет либо один оператор, либо несколько операторов, заключённых в фигурные скобки { ... } (после блока точка с запятой не ставится). В выражениях 1, 2, 3 фигурирует специальная переменная, называемая управляющей переменной (параметром). По её значению устанавливается необходимость повторения цикла или выхода из него.

Выражение_1 присваивает начальное значение управляющей переменной, выражение_3 изменяет его на каждом шаге, а выражение_2 проверяет, не достигло ли оно граничного значения, устанавливающего необходимость выхода из цикла.

Примеры операторов цикла с параметром:

```
for (i = 1; i < 10; i++)
{
  ...
}

for (ch = 'a'; ch != 'p';) scanf ("%c", &ch);
          /* Цикл будет выполняться до тех пор, пока
          с клавиатуры не будет введен символ 'p' */
```

Любое из трех выражений в цикле *for* может отсутствовать, однако точка с запятой должна оставаться. Таким образом, `for ( ; ; ) {...}` – это бесконечный цикл, из которого можно выйти лишь другими способами.

В языке Си принято следующее правило. Любое выражение с операцией присваивания, заключённое в круглые скобки, имеет значение, равное присваиваемому. Например, выражение (a=7+2) имеет значение 9. После этого можно записать другое выражение, например: ((a=7+2)<10), которое в данном случае будет всегда давать истинное значение. Следующая конструкция:

```
((ch = getch( )) == 'i')
```

позволяет вводить значение переменной *ch* и давать истинный результат только тогда, когда введенным значением является буква 'i'. В скобках можно записывать и несколько формул, составляющих сложное выражение. Для этих целей используется операция запятая. Формулы будут вычисляться слева направо, и все выражение примет значение последней вычисленной формулы. Например, если имеются две переменные типа *char*, то выражение

```
z = (x = y, y = getch( ));
```

определяет следующие действия: значение переменной *y* присваивается переменной *x*; вводится символ с клавиатуры и присваивается переменной *y*; *z* получает значение переменной *y*. Скобки здесь необходимы, поскольку операция запятая имеет более низкий приоритет, чем операция присваивания, записанная после переменной *z*. Операция запятая находит широкое применение для построения выражений цикла *for* и позволяет параллельно изменять значения нескольких управляющих переменных.

Допускаются вложенные конструкции, т.е. в теле некоторого цикла могут встречаться другие операторы *for*.

Оператор *while* формально записывается в таком виде:

```
while (выражение) тело_цикла
```

Выражение в скобках может принимать ненулевое (истинное) или нулевое (ложное) значение. Если оно истинно, то выполняется тело цикла и выражение вычисляется снова. Если выражение ложно, то цикл *while* заканчивается.

Оператор *do-while* формально записывается следующим образом:

```
do {тело_цикла} while (выражение);
```

Основным отличием между циклами *while* и *do-while* является то, что тело в цикле *do-while* выполняется, по крайней мере, один раз. Тело цикла будет выполняться до тех пор, пока выражение в скобках не примет ложное значение. Если оно ложно при входе в цикл, то его тело выполняется ровно один раз.

Допускается вложенность одних циклов в другие, т.е. в теле любого цикла могут появляться операторы *for*, *while* и *do-while*.

В теле цикла могут использоваться операторы *break* и *continue*. Оператор *break* обеспечивает немедленный выход из цикла, оператор *continue* вызывает прекращение очередной и начало следующей итерации.

Массивы

Массив состоит из элементов одного и того же типа. Ко всему массиву целиком можно обращаться по имени. Кроме того, можно выбирать любой элемент массива. Для этого необходимо задать индекс, который указывает на его относительную позицию. Число элементов массива назначается при его определении и в дальнейшем не изменяется. Если массив объявлен, то к любому его элементу можно обратиться следующим образом: указать имя массива и индекс элемента в квадратных скобках. Массивы определяются так же, как и переменные:

```
int a[100];
char b[20];
float d[50];
```

В первой строке объявлен массив *a* из 100 элементов целого типа: *a*[0], *a*[1], ..., *a*[99] (индексация всегда начинается с нуля). Во второй строке элементы массива *b* имеют тип *char*, а в третьей строке элементы массива *d* – тип *float*.

Двумерный массив представляется как одномерный, элементами которого так же являются массивы. Например, определение `char a[10][20];` задаёт такой массив. По аналогии можно установить и большее число измерений. Элементы двумерного массива хранятся по строкам, т.е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется самый правый индекс. Например, обращение к девятому элементу пятой строки запишется так: *a*[5][9].

Пусть задан следующий массив:

```
int a[2][3];
```

Тогда элементы массива a будут размещаться в памяти следующим образом: *a*[0][0], *a*[0][1], *a*[0][2], *a*[1][0], *a*[1][1], *a*[1][2].

Имя массива – это константа, которая содержит адрес его первого элемента (в данном примере *a* содержит адрес элемента *a*[0][0]). Предположим, что `a = 1000`. Тогда адрес элемента *a*[0][1] будет равен 1002 (элемент типа *int* занимает в памяти 2 байта), адрес следующего элемента *a*[0][2] – 1004 и т.д. Что же произойдёт, если выбрать элемент, для которого не выделена па-

мять? К сожалению, компилятор не отслеживает данной ситуации. В результате возникнет ошибка и программа будет работать неправильно.

В языке Си существует сильная взаимосвязь между указателями и массивами. Любое действие, которое достигается индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет работать быстрее.

Определение

```
int a[5];
```

задаёт массив из пяти элементов a[0], a[1], a[2], a[3], a[4]. Если объект *y определен как

```
int *y;
```

то оператор y = &a[0]; присваивает переменной y адрес элемента a[0]. Если переменная y указывает на очередной элемент массива a, то y+1 указывает на следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта (для типа *int* – 2 байта, *long* – 4 байта, *double* – 8 байт и т.д.).

Так как само имя массива есть адрес его нулевого элемента, то оператор y = &a[0]; можно записать и в другом виде: y = a. Тогда элемент a[1] можно представить как *(a+1). С другой стороны, если y – указатель на массив a, то следующие две записи: a[i] и *(y+i) – эквивалентны.

Между именем массива и соответствующим указателем есть одно важное различие. Указатель – это переменная и y = a; или y++; - допустимые операции. Имя же массива – константа, поэтому конструкции вида a = y; a++; использовать нельзя, так как значение константы постоянно и не может быть изменено.

Переменные с адресами могут образовывать некоторую иерархическую структуру (могут быть многоуровневыми) типа указатель на указатель (т.е. значение указателя является адресом другого указателя), указатель на указатель на указатель и т.д. Если указатели адресуют элементы одного массива, то их можно сравнивать (отношения вида <, >, = =, != и другие работают правильно). В то же время нельзя сравнивать либо использовать в арифметических операциях указатели на разные массивы (соответствующие выражения не приводят к ошибкам при компиляции, но в большинстве случаев не имеют смысла). Любой адрес можно проверить на равенство или неравенство с константой NULL. Указатели на элементы одного массива можно также вычитать. Тогда результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым объектами.

Язык Си позволяет инициализировать массив при его определении. Для этого используется следующая форма:

```
тип имя_массива[...] ... [...] = {список значений};
```

Примеры:

```
int a[5] = {0, 1, 2, 3, 4};
char ch[3] = {'d', 'e', '9'};
```

```
int b[2][3] = {1, 2, 3, 4, 5, 6};
```

В последнем случае: b[0][0] = 1, b[0][1] = 2, b[0][2] = 3, b[1][0] = 4, b[1][1] = 5, b[1][2] = 6.

В языке допускаются массивы указателей, которые определяются, например, следующим образом: char *m[5];. Здесь m[5] – массив, содержащий адреса элементов типа char.

Строки символов

Язык Си не поддерживает отдельный строковый тип данных, но он позволяет определить строки двумя различными способами. В первом используется массив символов, а во втором – указатель на первый символ массива.

Определение `char a[10];` указывает компилятору на необходимость резервирования места для 10 символов. При этом константа `a` содержит адрес ячейки памяти, в которой помещено значение первого из десяти объектов типа *`char`*. Процедуры, связанные с занесением конкретной строки в массив a, копируют её по одному символу в область памяти, на которую указывает константа *a*, до тех пор, пока не будет скопирован нулевой символ, оканчивающий строку.

Когда выполняется функция типа `printf("%s",a)`, ей передаётся значение *a*, т.е. адрес первого символа, на который указывает *a*. Если первый символ – нулевой, то работа функции `printf()` заканчивается, а если нет, то она выводит его на экран, прибавляет к адресу единицу и снова начинает проверку на нулевой символ. Такая обработка позволяет снять ограничения на длину строки (конечно, в пределах объявленной размерности): строка может иметь любую длину, но в пределах доступной памяти.

Инициализировать строку при таком способе определения можно следующим образом:

```
char array[7] = "Строка";
char s[ ] = {'С', 'т', 'р', 'о', 'к', 'а', '\0'};
```

(при определении массива с одновременной инициализацией пределы изменения индекса можно не указывать).

Второй способ определения строки – это использование указателя на символ. Определение `char *b;` задает переменную *b*, которая может содержать адрес некоторого объекта. Однако в данном случае компилятор не резервирует место для хранения символов и не инициализирует переменную *b* конкретным значением. Когда компилятор встречает оператор вида `b ="IBM PC";`, он производит следующие действия. Во-первых, как и в предыдущем случае, он создаёт в каком-либо месте объектного модуля строку "IBM PC", за которой следует нулевой символ ('\0'). Во-вторых, он присваивает значение начального адреса этой строки (адрес символа 'I') переменной *b*. Функция `printf("%s",b)` работает так же, как и в предыдущем случае, осуществляя вывод символов до тех пор, пока не встретится заключительный нуль.

Массив указателей можно инициализировать, т.е. назначать его элементам конкретные адреса некоторых заданных строк при определении.
Для ввода и вывода строк символов помимо scanf( ) и printf() могут использоваться функции gets( ) и puts( ) (их прототипы находятся в файле stdio.h).

Если string – массив символов, то ввести строку с клавиатуры можно так:

```
gets(string);
```

(ввод оканчивается нажатием клавиши <Enter>). Вывести строку на экран можно следующим образом:

```
puts(string);
```

Отметим также, что для работы со строками существует специальная библиотека функций, прототипы которых находятся в файле `string.h`.

Наиболее часто используются функции `strcpy()`, `strcat()`, `strlen()` и `strcmp()`.

Если `string1` и `string2` – массивы символов, то вызов функции `strcpy()` имеет вид:

```
strcpy(string1, string2);
```

Эта функция служит для копирования содержимого строки `string2` в строку `string1`. Массив `string1` должен быть достаточно большим, чтобы в него поместилась строка `string2`. Так как компилятор не отслеживает этой ситуации, то недостаток места приведет к потере данных.

Вызов функции `strcat()` имеет вид:

```
strcat(string1, string2);
```

Эта функция присоединяет строку *`string2`* к строке *`string1`* и помещает её в массив, где находилась строка *`string1`*, при этом строка *`string2`* не изменяется. Нулевой байт, который завершал первую строку, заменяется первым байтом второй строки.

Функция `strlen()` возвращает длину строки, при этом завершающий нулевой байт не учитывается. Если *`a`* – целое, то вызов функции имеет вид:

```
a = strlen(string);
```

Функция `strcmp()` сравнивает две строки и возвращает 0, если они равны.

Практические задания

1. Написать программы в соответствии с алгоритмами решения задач, разработанными в лабораторной работе №1, в которых используются описанные выше операторы цикла.

2. Реализовать написанные программы в инструментальной среде программирования, и выполнить пробный прогон программы.

Лабораторная работа №5. Пользовательские функции в языке Си. Механизмы передачи параметров (6 часов)

Общие сведения

Программы на языке Си обычно состоят из большого числа отдельных функций (подпрограмм). Как правило, эти функции имеют небольшие размеры и могут находиться как в одном, так и в нескольких файлах. Все функции являются глобальными. В языке запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные.

В общем случае функции в языке Си необходимо объявлять. Объявление функции (т.е. описание заголовка) должно предшествовать её использованию, а определение функции (т.е. её полное описание) может быть помещено как после тела программы (т.е. функции `main()`), так и до него. Если функция определена до тела программы, а также до её вызовов из определений других функций, то объявление может отсутствовать. Как уже отмечалось, описание заголовка функции обычно называют прототипом функции.

Функция объявляется следующим образом:

```
тип имя_функции(тип имя_параметра_1,тип имя_параметра_2,
                ...);
```

Тип функции определяет тип значения, которое возвращает функция. Если тип не указан, то предполагается, что функция возвращает целое значение (`int`).

При объявлении функции для каждого ее параметра можно указать только его тип (например: тип функция (int, float, ...), а можно дать и его имя (например: тип функция (int a, float b, ...)).

В языке Си разрешается создавать функции с переменным числом параметров. Тогда при задании прототипа вместо последнего из них указывается многоточие.

Определение функции имеет следующий вид:

```
тип имя_функции(тип имя_параметра_1, тип имя_параметра_2,...)
{
    тело функции
}
```

Передача значения из вызванной функции в вызывающую происходит с помощью оператора возврата *`return`*, который записывается следующим образом:

```
return выражение;
```

Таких операторов в подпрограмме может быть несколько, и тогда они фиксируют соответствующие точки выхода. Например:

```
int f(int a, int b)
{
    if (a > b) { printf("max = %d\n", a); return a; }
    printf("max = %d\n", b); return b;
```

```
}
```

Вызвать данную функцию можно любым из следующих способов:

```
c = f(15, 5);
c = f(d, g);
f(d, g);
```

Вызвавшая функция может, при необходимости, игнорировать возвращаемое значение. После слова return можно ничего не записывать; в этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции и в случае выхода "по концу" (последняя закрывающая фигурная скобка).

В языке Си аргументы функции передаются по значению, т.е. вызванная функция получает свою временную копию каждого аргумента, а не его адрес. Это означает, что вызванная функция не может изменить значение переменной вызвавшей её программы. Однако это легко сделать, если передавать в функцию не переменные, а их адреса. Например:

```
void swap(int *a, int *b)
{
    int *tmp = *a;

    *a = *b;
    *b = *tmp;
}
```

Вызов `swap(&b, &c)` (здесь подпрограмме передаются адреса переменных `b` и `c`) приведёт к тому, что значения переменных `b` и `c` поменяются местами.

Если же в качестве аргумента функции используется имя массива, то передаётся только адрес начала массива, а сами элементы не копируются. Функция может изменять элементы массива, сдвигаясь (индексированием) от его начала.

Рассмотрим, как функции можно передать массив в виде параметра. Здесь возможны три варианта:

1. Параметр задаётся как массив (например, `int m[100];`).
2. Параметр задаётся как массив без указания его размерности (например, `int m[];`).
3. Параметр задаётся как указатель (например, `int *m;`). Этот вариант используется наиболее часто.

Независимо от выбранного варианта вызванной функции передаётся указатель на начало массива. Сами же элементы массива не копируются.

Если некоторые переменные, константы, массивы, структуры объявлены как глобальные, то их не надо включать в список параметров вызванной функции.

Классы памяти

В языке Си различают четыре основных класса памяти: внешнюю (глобальную), автоматическую (локальную), статическую и регистровую память.

Внешние (глобальные) переменные определены вне функций и, следовательно, доступны для любой из них. Они могут быть определены только один раз. Выше уже говорилось, что сами функции всегда глобальные. Язык не позволяет определять одни функции внутри других. Область действия внешней переменной простирается от точки во входном файле, где она объявлена, до конца файла. Если на внешнюю переменную нужно ссылаться до ее определения или она определена в другом входном файле, то в подпрограмме или файле она должна быть объявлена как *extern*.

Например:

```
extern int a; /* Объявление a; память под переменную
                 не резервируется */
```

Автоматические переменные по отношению к функциям являются внутренними или локальными. Они начинают существовать при входе в функцию и уничтожаются при выходе из нее (для них можно использовать ключевое слово *auto*). Однако оно практически не используется, так как при отсутствии ключевого слова переменные по умолчанию принадлежат к классу *auto*.

Статические переменные объявляются с помощью ключевого слова *static*. Они могут быть внутренними (локальными) или внешними (глобальными). Внутренние статические переменные, как и автоматические, локальны по отношению к отдельной функции. Однако они продолжают существовать, а не возникают и не уничтожаются при каждом ее вызове. Другими словами, они являются собственной постоянной памятью для функции. Внешние статические переменные доступны внутри оставшейся части файла после того, как они в нем объявлены, однако в других файлах они неизвестны. Это, в частности, позволяет скрыть данные одного файла от другого файла.

Регистровые переменные относятся к последнему классу. Ключевое слово *register* говорит о том, что переменная, о которой идет речь, будет интенсивно использоваться. Если возможно, значения таких переменных помещаются во внутренние регистры микропроцессора, что может позволить сгенерировать более быструю и короткую программу. Для регистровых переменных нельзя взять адрес; они могут быть только автоматическими с допустимыми типами *int* или *char*.

Таким образом, можно выделить четыре модификатора класса памяти: *extern*, *auto*, *static*, *register*. Они используются в следующей общей форме:

```
модификатор_класса_памяти тип список_переменных;
```

Выше уже говорилось об инициализации, т.е. о присвоении различным объектам начальных значений. Если явная инициализация отсутствует, гаран-

тируется, что внешние и статические переменные будут иметь значение нуль, а автоматические и регистровые – неопределённое значение.

Указатели на функции

В языке Си сама функция не может быть значением переменной, но можно определить указатель на функцию. С ним уже можно обращаться как с переменной: передавать его другим функциям, помещать в массивы и т.п.

Код функции в персональном компьютере занимает физическую память. В этой памяти есть точка входа, которая используется для того, чтобы войти в функцию и запустить ее на выполнение. Указатель на функцию как раз и адресует эту точку входа. Это уже будет обычная переменная и с ней можно делать все, что можно делать с переменной.

Через указатель можно войти в функцию, т.е. запустить ее на выполнение. Объявление вида:

```
int (*f)();
```

говорит о том, что f – это указатель на функцию, возвращающую целое значение. Первая пара скобок необходима, без них `int *f();` означало бы, что `f` – функция, возвращающая указатель на целое значение. После объявления указателя на функцию в программе можно использовать объекты: `*f` – сама функция; f – указатель на функцию. Для любой функции её имя (без скобок и аргументов) является указателем на эту функцию.

Аргументы функции main( )

В программы на языке Си можно передавать некоторые аргументы. Когда в начале вычислений производится обращение к функции `main()`, ей передаются три параметра. Первый из них определяет число командных аргументов при обращении к программе. Второй представляет собой массив указателей на символьные строки, содержащие эти аргументы (в одной строке – один аргумент). Третий тоже является массивом указателей на символьные строки, он используется для доступа к параметрам операционной системы (к переменным окружения).

Любая такая строка представляется в следующем виде:

```
переменная = значение\0
```

Последнюю строку можно найти по двум заключительным нулям.

Назовём аргументы функции `main( )` соответственно: `argc`, `argv` и `env` (возможны и любые другие имена). Тогда допустимы следующие описания:

```
main( )
main(int argc)
main(int argc, char *argv[ ] )
main(int argc, char *argv[ ], char *env[ ] )
```

Предположим, что на диске `C:` есть некоторая программа `prog.exe`. Обратимся к ней следующим образом:

```
C:\>prog.exe file1 file2 file3 <Enter>
```

Тогда `argv[0]` – это указатель на строку `C:\prog.exe`, `argv[1]` – на строку `file1` и т.д. На первый фактический аргумент указывает `argv[1]`, а на последний – `argv[3]`. Если `argc=1`, то после имени программы в командной строке параметров нет. В данном примере `argc=4`.

Рекурсия

Рекурсией называется такой способ вызова, при котором функция обращается к самой себе. Важным моментом при составлении рекурсивной программы является организация выхода. Здесь легко допустить ошибку, заключающуюся в том, что функция будет последовательно вызывать саму себя бесконечно долго. Поэтому рекурсивный процесс должен шаг за шагом так упрощать задачу, чтобы, в конце концов, для неё появилось не рекурсивное решение. Использование рекурсии не всегда желательно, так как это может привести к переполнению стека.

Библиотечные функции

В системах программирования подпрограммы для решения часто встречающихся задач объединяются в библиотеки. К числу таких задач относятся: вычисление математических функций, ввод/вывод данных, обработка строк, взаимодействие со средствами операционной системы и другие Использование библиотечных подпрограмм избавляет пользователя от необходимости разработки соответствующих средств и предоставляет ему дополнительный сервис. Включенные в библиотеки функции поставляются вместе с системой программирования. Их объявления даны в файлах `*.h` (это так называемые включаемые или заголовочные файлы). Поэтому, как уже упоминалось выше, в начале программы с библиотечными функциями должны быть строки вида:

```
#include <включаемый_файл_типа_h>
```

Например:

```
#include <conio.h>
```

Существуют также средства для расширения и создания новых библиотек с программами пользователя.

Практические задания

1. Написать и использовать в программах пользовательские функции при решения задач, представленных в лабораторной работе №1 (например, для отыскания максимального или минимального из заданных значений).

2. Реализовать написанные программы, использующие пользовательские функции, в инструментальной среде программирования, и выполнить пробный прогон программы.

Лабораторная работа №6. Использование структур данных и файлов в программах на языке Си (6 часов)

Структуры

Структура – это объединение одного или нескольких объектов (переменных, массивов, указателей, других структур и т.д.). Как и массив, она представляет собой совокупность данных. Отличием является то, что к её элементам необходимо обращаться по имени и что различные элементы структуры не обязательно должны принадлежать одному типу.

Объявление структуры осуществляется с помощью ключевого слова *struct*, за которым идет её тип и далее список элементов, заключенных в фигурные скобки:

```
struct тип { тип элемента_1 имя элемента_1;
                ...
             тип элемента_n имя элемента_n;
           };
```

Именем элемента может быть любой идентификатор. Как и выше, в одной строке можно записывать через запятую несколько идентификаторов одного типа.

Рассмотрим пример:

```
sruct date { int day;
             int month;
             int year;
           };
```

Вслед за фигурной скобкой, заканчивающей список элементов, могут записываться переменные данного типа, например:

```
struct date {...} a, b, c;
```

(при этом выделяется соответствующая память). Описание без последующего списка не выделяет никакой памяти; оно просто задает форму структуры. Введенное имя типа позже можно использовать для объявления структуры, например:

```
struct date days;
```

Теперь переменная `days` имеет тип `date`.

При необходимости структуры можно инициализировать, помещая вслед за описанием список начальных значений элементов.

Разрешается вкладывать структуры друг в друга, например:

```
struct man { char name[20], fam[20];
             struct date bd;
             int age;
           };
```

Определённый выше тип `data` включает три элемента: `day`, `month`, `year`, содержащий целые значения (`int`). Структура `man` включает элементы name, `fam`, `bd` и `age`. Первые два – `name[20]` и `fam[20]` – это символьные массивы из 20 элементов каждый. Переменная `bd` представлена составным элементом (вложенной структурой) типа `data`. Элемент `age` содержит значения целого типа `int`).

Теперь можно определить переменные, значения которых принадлежат введённому типу:

```
struct man man_100[100];
```

Здесь определён массив `man_100`, состоящий из 100 структур типа `man`.

Чтобы обратиться к отдельному элементу структуры, необходимо указать его имя, поставить точку и сразу же за ней записать имя нужного элемента, например:

```
man_100[j].age = 32;
man_100[j].bd.day = 12;
man_100[j].bd.month = 2
man_100[j].bd.year = 1987;
```

При работе со структурами необходимо помнить, что тип элемента определяется соответствующей строкой описания в фигурных скобках. Например, массив man_100 имеет тип man, year является целым числом и т.п. Поскольку каждый элемент структуры относится к определённому типу, его имя может появиться везде, где разрешено использование значений этого типа. Допускаются конструкции вида man_100[i]=man_[j]; где man_100[i] и man_100[j] – объекты, соответствующие единому описанию структуры. Другими словами, разрешается присваивать одну структуру другой по их именам.

Унарная операция & позволяет взять адрес структуры. Предположим, что определена переменная day:

```
struct date {int d, m, y;} day;
```

Здесь day – это структура типа date, включающая три элемента: d, m, y. Другое определение

```
struct date *db;
```

устанавливает тот факт, что db – это указатель на структуру типа date.

Запишем следующее выражение:

```
db = &day;
```

В этом случае для выбора элементов d, m, y структуры необходимо использовать конструкции:

```
(*db).d; (*db).m; (*db).y;
```

Действительно, db – это адрес структуры, *db – сама структура. Круглые скобки здесь необходимы, так как точка имеет более высокий, чем звездочка, приоритет. Для аналогичных целей в языке Си предусмотрена специальная опера-

ция ->. Эта операция выбирает элемент структуры и позволяет представить рассмотренные выше конструкции в более простом виде:

```
db -> d; db -> m; db -> y;
```

Оператор typedef

Рассмотрим следующее описание структуры:

```
struct data {int d, m, y;};
```

Здесь фактически вводится новый тип данных – data. Теперь его можно использовать для объявления конкретных экземпляров структуры, например:

```
struct data a, b, c;
```

В язык Си введено специальное средство, позволяющее назначать имена типам данных (переименовывать). Таким средством является оператор typedef. Он записывается в следующем виде:

```
typedef тип имя;
```

Здесь «тип» – любой разрешенный тип данных и «имя» – любой разрешенный идентификатор.

Рассмотрим следующий пример:

```
typedef int INTEGER;
```

После этого можно сделать объявление:

```
INTEGER a, b;
```

Оно будет выполнять то же самое, что и привычное объявление int a,b;. Другими словами, INTEGER можно использовать как синоним ключевого слова int.

Битовые поля

Особую разновидность структур представляют собой битовые поля. Битовое поле – это последовательность соседних битов внутри одного, целого значения. Оно может иметь тип `signed int` или `unsigned int` и занимать от 1 до 16 битов. Поля размещаются в машинном слове в направлении от младших к старшим разрядам. Например, структура:

```
struct prim { int a:2;
              unsigned b:3;
              int c:5;
              int d:1;
              unsigned d:5; } i, j;
```

обеспечивает размещение данных в двух байтах (в одном слове). Если бы последнее поле было задано так: unsigned d:6, то оно размещалось бы не в первом слове, а в разрядах 0-5 второго слова.

В полях типа `signed` крайний левый бит является знаковым.

Поля используются для упаковки значений нескольких переменных в одно машинное слово с целью экономии памяти. Они не могут быть массивами и не имеют адресов, поэтому к ним нельзя применять унарную операцию &.

Объединение (union)

Объединение – это некоторая переменная, которая может хранить (в разное время) объекты различного типа и размера. В результате появляется возможность работы в одной и той же области памяти с данными различного вида. Для описания объединения используется ключевое слово `union`, а соответствующий синтаксис аналогичен структурам.

Пусть задано определение:

```
union r {int ir; float fr; char cr;} z;
```

Здесь ir имеет размер 2 байта, fr – 4 байта, cr – 1 байт. Размер переменной z будет равен размеру самого большого из трех приведенных типов (т.е. 4 байтам). В один и тот же момент времени z может иметь значение только одной из переменных ir, fr или cr.

Перечислимый тип данных

Перечислимый тип данных предназначен для описания объектов из некоторого заданного множества. Он задается ключевым словом `enum`.

Рассмотрим следующий пример:

```
enum seasons (spring, summer, autumn, winter);
```

Здесь введен новый тип данных seasons. Теперь можно определить переменные этого типа:

```
enum seasons a, b, c;
```

Каждая из них (a, b, c) может принимать одно из четырех значений: spring, summer, autumn и winter. Эти переменные можно было определить сразу при описании типа:

```
enum seasons (spring, summer, autumn, winter) a, b, c;
```

Рассмотрим еще один пример:

```
enum days {mon, tues, wed, thur, fri, sat, sun} my_week;
```

Имена, занесенные в days (также как и в seasons в предыдущем примере), представляют собой константы целого типа. Первая из них (mon) автоматически устанавливается в нуль, и каждая следующая имеет значение на единицу больше, чем предыдущая (tues=1, wed=2 и т.д.).

Можно присвоить константам определенные значения целого типа (именам, не имеющим их, будут, как и раньше, назначены значения предыдущих констант, увеличенные на единицу). Например:

```
enum days (man=5, tues=8, wed=10, thur, fri, sat, sun}
     my_week;
```

После этого mon=5, tues=8,wed=10, thur=11, fri=12, sat=13, sun=14.

Тип enum можно использовать для задания констант true=1 и false=0, например:

```
enum t_f (false, true) a, b;
```

Файлы

В общем смысле файлом называют способ хранения информации на физическом устройстве.

В языке Си отсутствуют операторы для работы с файлами. Все необходимые действия выполняются с помощью функций, включенных в стандартную библиотеку. Они позволяют работать с различными устройствами, такими, как диски, принтер, коммуникационные каналы и т.д. Эти устройства сильно отличаются друг от друга. Однако файловая система преобразует их в единое *абстрактное логическое устройство*, называемое *потоком*.

В Си существует два типа потоков: текстовые (text) и двоичные (binary).

Текстовый поток – это последовательность символов. При передаче символов из потока на экран, часть из них не выводится (например, символ возврата каретки, перевода строки).

Двоичный поток – это последовательность байтов, которые однозначно соответствуют тому, что находится на внешнем устройстве.

Прежде чем читать или записывать информацию в файл, он должен быть открыт и тем самым связан с потоком. Это можно сделать с помощью библиотечной функции `fopen()`. Она берет внешнее представление файла (например, c:\my_prog.txt) и связывает его с внутренним логическим именем, которое используется далее в программе. Логическое имя – это указатель на требуемый файл. Его необходимо определить; делается это, например, так:

```
FILE *fp;
```

Здесь FILE – имя типа, описанное в стандартном заголовочном файле `stdio.h`, `fp` – указатель на файл. Обращение к функции `fopen()` в программе осуществляется выражением:

```
fp = fopen(спецификация файла,"способ использования файла");
```

Спецификация файла (т.е. имя файла и путь к нему) может, например, иметь вид: "c:\\my_prog.txt" – для файла my_prog.txt на диске c:.

Способ использования файла задается следующими символами:

r – открыть существующий файл для чтения;

w – создать новый файл для записи (если файл с указанным именем существует, то он будет переписан);

a – дополнить файл (открыть существующий файл для записи информации, начиная с конца файла, или создать файл, если он не существует);

r+ – открыть существующий файл для чтения и записи;

w+ – создать новый файл для чтения и записи;

a+ – дополнить или создать файл с возможностью чтения и записи;

rb – открыть двоичный файл для чтения;

wb – создать двоичный файл для записи;

ab – дополнить двоичный файл;

r+b – открыть двоичный файл для чтения и записи;
w+b – создать двоичный файл для чтения и записи;
a+b – дополнить двоичный файл с предоставлением возможности чтения и записи;
rt – открыть текстовой файл для чтения;
wt – создать текстовый файл для записи;
at – дополнить текстовый файл;
r+t – открыть текстовой файл для чтения и записи;
w+t – создать текстовый файл для чтения и записи;
a+t – дополнить текстовый файл с предоставлением возможности записи и чтения.

Если режим t или b не задан (например, r, w или a), то он определяется значением глобальной переменной _fmode. Если fmode=0_BINARY, то файлы открываются в двоичном режиме, а если _fmode=0_TEXT – в текстовом режиме. Константы 0_BINARY и 0_TEXT определены в файле fcntl.h.

Строки вида r+b можно записывать и в другой форме: rb+.

Если в результате обращения к функции `fopen()` возникает ошибка, то она возвращает константу NULL.

Рекомендуется использовать следующий способ открытия файла:

```
if ((fp = fopen("c:\\my_prog.txt", "rt")) == NULL)
{
    puts("Открыть файл не удалось\n");
    exit(1);
}
```

После окончания работы с файлом он должен быть закрыт. Это делается с помощью библиотечной функции `fclose()`. Она имеет следующий прототип:

```
int fclose(FILE *fp);
```

При успешном завершении операции функция fclose( ) возвращает значение нуль. Любое другое значение свидетельствует об ошибке.

Рассмотрим другие библиотечные функции, используемые для работы с файлами (все они описаны в файле `stdio.h`):

1. Функция putc( ) записывает символ в файл и имеет следующий прототип:

```
int putc(int c, FILE *fp);
```

Здесь fp – указатель на файл, возвращенный функцией fopen( ), c – символ для записи (переменная c имеет тип int, но используется только младший байт). При успешном завершении putc( ) возвращает записанный символ, в противном случае возвращается константа EOF. Она определена в файле stdio.h и имеет значение -1.

2. Функция `getc()` читает символ из файла и имеет следующий прототип:

```
int getc(FILE *fp);
```

Здесь fp – указатель на файл, возвращенный функцией `fopen()`. Эта функция возвращает прочитанный символ. Соответствующее значение имеет тип int, но старший байт равен нулю. Если достигнут конец файла, то `getc()` возвращает значение EOF.

3. Функция `feof()` определяет конец файла при чтении двоичных данных и имеет следующий прототип:

```
int feof(FILE *fp);
```

Здесь fp – указатель на файл, возвращенный функцией `fopen()`. При достижении конца файла возвращается ненулевое значение, в противном случае возвращается 0.

4. Функция `fputs()` записывает строку символов в файл. Она отличается от функции `puts()` только тем, что в качестве второго параметра должен быть записан указатель на переменную файлового типа.

Например:

```
fputs("Example", fp);
```

При возникновении ошибки возвращается значение EOF.

5. Функция `fgets()` читает строку символов из файла. Она отличается от функции `gets()` тем, что в качестве второго параметра должно быть указано максимальное число вводимых символов плюс единица, а в качестве третьего – указатель на переменную файлового типа. Строка считывается целиком, если ее длина не превышает указанного числа символов, в противном случае функция возвращает только заданное число символов.

Рассмотрим пример:

```
fgets(string, n, fp);
```

Функция возвращает указатель на строку string при успешном завершении и константу NULL в случае ошибки либо достижения конца файла.

6. Функция `fprintf()` выполняет те же действия, что и функция `printf()`, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа.

Например:

```
fprintf(fp, "%x",a);
```

7. Функция `fscanf()` выполняет те же действия, что и функция `scanf()`, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа.

Например:

```
fscanf(fp, "%x", &a);
```

При достижении конца файла возвращается значение EOF.

8. Функция `fseek()` позволяет выполнять чтение и запись с произвольным доступом и имеет следующий прототип:

```
int fseek(FILE *fp, long count, int access);
```

Здесь fp – указатель на файл, возвращенный функцией fopen( ), count – номер байта относительно заданной начальной позиции, начиная с которого будет выполняться операция, access – способ задания начальной позиции.

Переменная access может принимать следующие значения:

0 – начальная позиция задана в начале файла;

1 – начальная позиция считается текущей;

2 – начальная позиция задана в конце файла.

При успешном завершении возвращается нуль, при ошибке – ненулевое значение.

9. Функция `ferror()` позволяет проверить правильность выполнения последней операции при работе с файлами. Имеет следующий прототип:

```
int ferror(FILE *fp);
```

В случае ошибки возвращается ненулевое значение, в противном случае возвращается нуль.

10. Функция `remove()` удаляет файл и имеет следующий прототип:

```
int remove(char *file_name);
```

Здесь file_name – указатель на строку со спецификацией файла. При успешном завершении возвращается нуль, в противном случае возвращается ненулевое значение.

11. Функция `rewind()` устанавливает указатель текущей позиции в начало файла и имеет следующий прототип:

```
void rewind(FILE *fp);
```

12. Функция `fread()` предназначена для чтения блоков данных из потока. Имеет прототип:

```
unsigned fread(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она читает n элементов данных, длиной size байт каждый, из заданного входного потока fp в блок, на который указывает указатель ptr. Общее число прочитанных байтов равно произведению n*size. При успешном завершении функция fread( ) возвращает число прочитанных элементов данных, при ошибке – 0.

13. Функция `fwrite()` предназначена для записи в файл блоков данных. Имеет прототип:

```
unsigned fwrite(void *ptr,unsigned size,unsigned n,FILE *fp);
```

Она добавляет n элементов данных, длиной size байт каждый, в заданный выходной файл fp. Данные записываются с позиции, на которую указывает указатель ptr. При успешном завершении операции функция fwrite( ) возвращает число записанных элементов данных, при ошибке – неверное число элементов данных.

В языке Си имеются пять стандартных файлов со следующими логическими именами:

stdin – для ввода данных из стандартного входного потока (по умолчанию – с клавиатуры);

stdout – для вывода данных в стандартный выходной поток (по умолчанию – на экран дисплея);

stderr – файл для вывода сообщений об ошибках (всегда связан с экраном дисплея);

stdprn – для вывода данных на принтер;

stdaus – для ввода и вывода данных в коммуникационный канал.

В языке Си имеется также система низкоуровневого ввода/вывода (без буферизации и форматирования данных), соответствующая стандарту системы UNIX. Прототипы составляющих ее функций находятся в файле io.h. К этим функциям относятся:

open( ) – открыть файл;

close( ) – закрыть файл;

read( ) – читать данные;

write( ) – записать данные;

lseek( ) – поиск определенного байта в файле;

unlink( ) – уничтожить файл.

Динамическое распределение памяти. Функции malloc() и free()

В языке Си принято следующая схема распределения памяти (рисунок 6.1):

СТЕК	Верхние адреса
СВОБОДНАЯ ПАМЯТЬ	
РАЗДЕЛ ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ И КОНСТАНТ	
КОД ПРОГРАММЫ	Нижние адреса

Рисунок 6.1 – Схема распределения памяти

Для глобальных переменных отводится фиксированное место в памяти на все время работы программы. Локальные переменные хранятся в стеке. Между ними находится область памяти для динамического распределения.

Функции malloc( ) и free( ) используются для динамического распределения свободной памяти. Функция malloc( ) выделяет память, функция free( ) освобождает ее. Прототипы этих функций хранятся в заголовочном файле stdlib.h и имеют вид:

```
void *malloc(size_t size);
void *free(void *p);
```

Функция malloc( ) возвращает указатель типа void; для правильного использования значение функции надо преобразовать к указателю на соответствующий тип. При успешном выполнении функция возвращает указатель на первый байт свободной памяти размера size. Если достаточного количества памяти нет, возвращается значение 0. Чтобы определить количество байтов, необходимых для переменной, используют операцию sizeof( ).

Пример использования этих функций:

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

void main(void)
{
   int *p, i;
   p = (int *) malloc(100 * sizeof(int)); /* Выделение памяти
                                          для 100 целых чисел */
   if (!p)
   {
       printf("Недостаточно памяти\n");
       exit(1);
   }
   for (i = 0; i < 100; ++i) *(p+i) = i; /* Использование па-
                                            мяти */
   for (i = 0; i < 100; ++i) printf("%d", *(p++) );
   free(p);                              /* Освобождение па-
                                            мяти */
}
```

Перед использованием указателя, возвращаемого malloc( ), необходимо убедиться, что памяти достаточно (указатель не нулевой).

Перечень использованных источников

1. Логинов, В. И. Основы алгоритмизации [Текст] : учеб.-метод. пособие для студ. оч. и заоч. обуч. технич. специальностей / В. И. Логинов, Л. Н. Шемагина. – Н. Новгород : Изд-во ФГОУ ВПО «ВГАВТ», 2010. – 81 с.

2. Бочарова, Т. А. Основы алгоритмизации [Текст] : учеб. пособие / Т. А. Бочарова, Н. О. Бегункова. – Хабаровск : Изд-во Тихоокеан. гос. ун-та, 2011. – 64 с.

3. Основы алгоритмизации инженерных задач [Текст] : учебное пособие / В. Н. Суриков, А. С. Кудрявцев, Г. А. Петров, Е. В. Хардиков. – ГОУВПО СПбГТУ РП. СПб., 2008. – 158 с.

4. Козенко, С. Л. Алгоритмизации инженерных задач [Текст] : методич. указания / С. Л. Козенко. – СПб. : ГОУ ВПО «СПбГУАП», 2005. – 46 с.

5. Белов, М. П. Основы алгоритмизации в информационных системах [Текст] : учеб. пособие / М. П. Белов. – СПб. : СЗТУ, 2003. – 85 с.

6. Петрова, М. В. Алгоритмизация и программирование задач [Текст] : методич. указания / М. В. Петрова. – Ульяновск: УлГТУ, 2003. – 32 с.

Введение……. 3

Лабораторная работа №1. Базовые алгоритмические конструкции: линейная, разветвляющаяся, циклическая. Комбинация базовых алгоритмических конструкций…….. 5

Лабораторная работа №2. Структура программы на языке Си. Простые операции и операторы языка Си. Ввод-вывод в программах на языке Си……..... 28

Лабораторная работа №3. Программирование линейных и разветвляющихся алгоритмов на языке Си. Условные операторы языка Си…… 43

Лабораторная работа №4. Программирование циклических алгоритмов на языке Си. Операторы цикла языка Си. Использование массивов в программах на языке Си……... 45

Лабораторная работа №5. Пользовательские функции в языке Си. Механизмы передачи параметров…….. 51

Лабораторная работа №6. Использование структур данных и файлов в программах на языке Си……. 56

Перечень использованных источников…….. 65

Стариков Александр Вениаминович

Программирование и алгоритмизация в задачах автоматизации и управления

Лабораторный практикум для
направления подготовки бакалавра 15.03.04
Автоматизация технологических процессов и производств
Профиль – Автоматизация и управление в технологических системах
для очной и заочной форм обучения

Редактор С.Ю. Крохотина

Подписано в печать Формат бумаги Заказ
Объем п.л. Усл. п.л. Уч-изд. л. Тираж
ФГБОУ ВО «Воронежский государственный лесотехнический университет имени Г.Ф. Морозова»