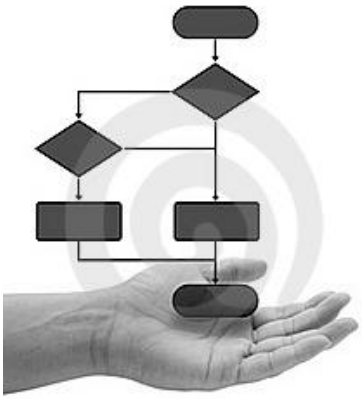
МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

КАРАГАНДИНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

И.В. Солодовникова



АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И ПРОГРАММИРОВАНИЕ

Караганда 2014

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

КАРАГАНДИНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

И.В. Солодовникова

АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И ПРОГРАММИРОВАНИЕ

Утверждено ученым советом университета
в качестве учебного пособия

Караганда 2014

УДК 681.142.2
ББК 32.973.26-018.1я7
С20

Рекомендовано редакционно-издательским советом университета

Рецензенты:
А.М. Омаров, к.ф-м.н., заведующий кафедрой ПМиИ КарГУ им. Е.А.Букетова;
М.М. Коккоз, к.п.н., заведующий кафедрой ИТБ КарГТУ;
М.К. Баймульдин, член редакционно-издательского совета КарГТУ.

Солодовникова И.В.

С20 Алгоритмы, структуры данных и программирование: Учебное пособие/ И.В.Солодовникова; Карагандинский государственный технический университет. – Караганда: Изд-во КарГТУ, 2014. – 93 с.

ISBN

В учебном пособии рассмотрены основные понятия алгоритмизации и программирования. Пособие знакомит с основополагающими понятиями: оператор, переменная, функция, тип данных и т.д. Учит применять основные операторы языка программирования высокого уровня: условие, различные виды циклов, выбор, строить блок-схемы алгоритмов и производить по ним разработку программ. В пособии рассматриваются основные формы представления данных: строки, структуры (пользовательские типы данных), массивы (одномерные и многомерные).

Учебное пособие ориентировано на студентов специальностей 5В070300 «Информационные системы» и 5В070400 «Вычислительная техника и программное обеспечение» а также может быть рекомендовано студентам других специальностей при изучении вопросов, связанных с разработкой программного обеспечения.

УДК 681.142.2
ББК 32.973.26-018.1я7

ISBN

Оглавление

Предисловие

Любой человек ежедневно встречается с множеством повседневных и профессиональных задач. Для решения многих из них существуют определенные правила (инструкции, предписания), объясняющие, как решать определенную задачу. В процессе решения можно применять готовые правила или формулировать собственные. Чем точнее и понятнее описаны правила решения задач, тем быстрее человек овладеет ими и будет эффективнее их применять. Решение многих задач человек передает техническим устройствам – ПК, автоматам, роботам и т. д. Их применение предъявляет очень строгие требования к точности описания правил и последовательности выполнения действий. Поэтому разрабатываются специальные алгоритмы для четкого и строгого описания различных правил.

Алгоритмизация - это раздел информатики, изучающий методы и приемы построений алгоритма, а также их свойства. Она является основным, базовым компонентом компьютерной грамотности в современном компьютерном мире. В настоящее время применение средств вычислительной техники как инструмента для решения инженерных задач требует углубленных знаний в различных областях человеческой деятельности. Часто на практике необходимо преобразовать исходную задачу с учетом дискретного характера машинных вычислений и представить процесс ее решения на ЭВМ в виде последовательности шагов. Такой подход должен выработать у будущего специалиста «алгоритмическое мышление», на основе которого дальнейший процесс разработки программ не вызывает затруднений. Для достижения положительных результатов важную роль играет умение разрабатывать оптимальный алгоритм решения поставленной задачи.

Основу деятельности специалиста практически любой области составляет умение ставить задачи, разрабатывать алгоритмы, получать решения, производить анализ полученных данных и делать выводы. Поэтому в своей будущей профессиональной деятельности студенты должны уметь грамотно применять персональный компьютер (ПК) для решения научных и производственных задач.

В данном пособии рассмотрены основные структуры алгоритмов и типовые приемы алгоритмизации. Приведены примеры и задания, охватывающие основные типы вычислительных процессов.

Изложение базируется на современных принципах синтеза алгоритмов с использованием концепции структурного программирования. Сформулированы этапы подготовки задач для программирования, способы и основные принципы алгоритмизации при решении инженерных задач. Материал расположен именно в том порядке, в котором большинство начинающих пользователей изучает самостоятельно основы алгоритмизации.

Цель пособия – научить студентов анализировать и ставить задачи, разрабатывать алгоритмы их решения.

Все главы завершаются контрольными вопросами и упражнениями, предназначенными для закрепления изученного материала.

1 Основы алгоритмизации

1.1 Основные понятия и определения

Алгоритмизация является основным, базовым компонентом компьютерной грамотности в динамично развивающемся современном компьютерном мире. Для достижения положительных результатов важную роль играет умение разрабатывать оптимальный алгоритм решения поставленной задачи, что требует от исполнителя наличия определённых навыков алгоритмизации и системного анализа. А также знание математики, общей физики, механики и других инженерных дисциплин [1].

Алгоритмизация – это общая последовательность действий, которые необходимо выполнить для построения алгоритма решения задачи, в том числе – выделение конкретных шагов алгоритмического процесса, определение вида формальной записи для каждого шага и установление определённого порядка выполнения каждого шага.

Процесс подготовки решения задачи и её непосредственная реализация на компьютере происходит за некоторое количество самостоятельных 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. Неправильная или плохая модель обречет на неудачу весь дальнейший проект, поэтому этот этап является крайне важным.

3. **Выбор метода решения**. Исходя как из субъективных причин (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.

4. **Составление алгоритма решения.** На этом этапе должна четко прослеживаться связь с предыдущим. В ходе него разрабатывается эффективный алгоритм, т. е. такой, 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**. 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.

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

На этапе разработки алгоритма 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 четкой структуры алгоритмических конструкций.

Порядок действий считается алгоритмом в том случае, если он обладает определенными свойствами [2].

1.2 Свойства алгоритма

Значение слова «алгоритм» является синонимом таких понятий, как «набор инструкций», «последовательность действий», «метод». Однако, в отличие от них, алгоритм характеризуется определенными свойствами. Свойства алгоритма – это набор свойств, отличающих алгоритм от любых предписаний и обеспечивающих его автоматическое исполнение. Характерными свойствами алгоритма являются дискретность, определенность, массовость, результативность**.**

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

Определенность (детерминированность) – означает, что предписанные алгоритмом действия должны быть определены точно и однозначно, исключая какие – либо произвольные (случайные) толкования. Детерминированность обеспечивает одинаковость результата,

полученного при выполнении алгоритма несколько раз, если исходные данные сохраняют свое значение.

Массовость – алгоритм должен обеспечивать выполнение не одной конкретной задачи, а быть пригодным для реализации класса задач, то есть алгоритм можно применить к любому ряду исходных данных.

Результативность – это требование обеспечивает конечность применения указаний, то есть результат должен быть получен за конечное число шагов, либо за конечное число шагов должно быть получено указание на неприменимость данного алгоритма к решаемой задачи.

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

− выделение законченных частей вычислительного процесса;

− формальная запись каждого из них;

− назначение определенного порядка выполнения выделенных частей;

− проверка правильности выбранного алгоритма по реализации заданного метода вычислений.

1.3 Способы записи алгоритмов

На любой стадии существования алгоритмы представляют с помощью конкретных изобразительных средств, состав и правила употребления которых образуют конкретные способы или формы записи. В настоящее время используются следующие способы записи алгоритмов – словесно-формульный, псевдокод, графический.

Эти способы могут быть использованы совместно, дополняя друг друга, чтобы алгоритм был более понятным. Чаще всего используется графический способ задания, который дополняется словесным описанием.

Словесно-формульный 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.

Пример 1. Алгоритм нахождения наибольшего общего делителя (НОД) двух натуральных чисел M и N:

1) задать два числа;
2) если N=0, то B:=M, выход.
3) вычислить остаток R от деления M на N;
4) заменить M:= N, N:=R, перейти на шаг 2.

Пусть M=203 и N=91:

Остаток от деления 203 на 91 R=21 M=91 N=21
Остаток от деления 91 на 21 R=7 M=21 N=7
Остаток от деления 21 на 7 R=0 M=7 N=0 **– выход.**
НОД =7.

Описанный алгоритм применим к любым числам и должен приводить к решению поставленной задачи.

Пример 2. Вычисление корней квадратного уравнения $ax^2+bx+c=0$ в области действительных чисел. Математической моделью этой задачи является известная формула корней квадратного уравнения:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

На основании этой формулы запишем алгоритм:

1. Задать значения a, b, c.
2. Вычислить дискриминант d = b2 – 4ac.
3. Сравнить дискриминант с нулем, если он больше нуля, то вычислить корни по формуле и перейти к п. 4, иначе сообщить: «В области действительных чисел уравнение решений не имеет».
4. Записать результат: «Корни уравнения x1 и x2».

Действия (пункты) алгоритма выполняются в естественной последовательности. Вид действий не формализован. Такой вид записи алгоритма имеет серьёзный недостаток – отсутствие наглядности. Поэтому этот способ записи алгоритма не имеет широкого распространения.

Псевдокод – это язык записи структурированных алгоритмов. Основан на формализованном представлении предписаний, задаваемых с помощью ограниченного набора типовых синтаксических конструкций. Набор конструкций состоит из смеси 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.

Пример псевдокода для нахождения наибольшего элемента из двух натуральных чисел (A и B). Обозначим наибольший элемент – Z.

```
Начало
    Ввод A, B
        Если A≥B, то Z=A
            Иначе Z=B
        Конец Если
    Вывод Z
Конец
```

Основными достоинствами псевдокода являются:

– близость к языкам программирования;

– возможность разобраться в самом длинном и сложном алгоритме, поэтому псевдокод используется чаще всего для документирования программ.

Графический способ описания алгоритма иначе называют блок-схемой. В блок-схемах используются геометрические фигуры, каждая из которых изображает какую-либо операцию или действие, а также этап процесса решения задачи. Каждая фигура называется блоком. Порядок выполнения этапов показывается стрелками, соединяющими блоки. Блоки необходимо размещать сверху вниз или слева направо в порядке их выполнения. Форма символов и правила составления схем установлены Единой Системой Программной Документации (ЕСПД) ГОСТ 19701-90 [3].

Описание алгоритмов с помощью схем – один из наиболее наглядных и компактных способов. Этот способ имеет ряд преимуществ перед остальными:

− наглядное отображение базовых конструкций алгоритма;

− концентрация внимания на структуре алгоритма;

− преобразование алгоритма методом укрупнения (сведения к единому блоку) или детализации (разбиения на ряд блоков);

− быстрая проверка разработанного алгоритма.

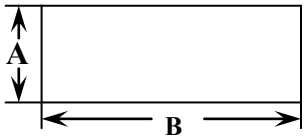
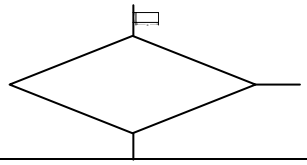
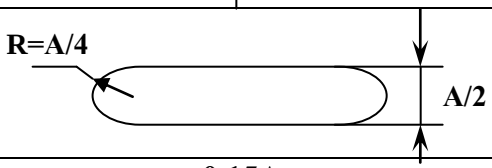
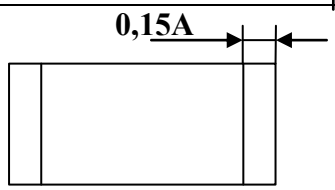
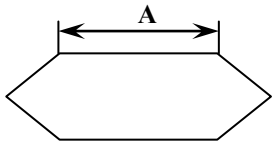
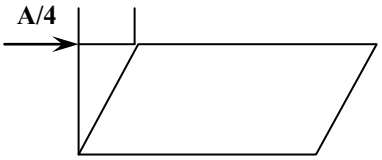
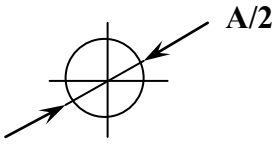
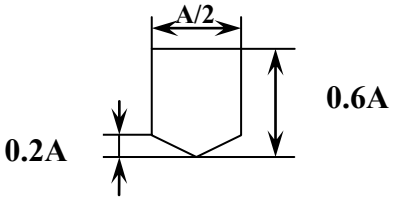
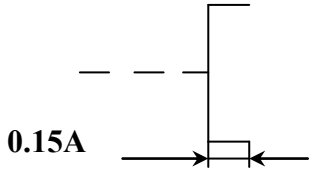
В таблице 1 приведены наиболее часто употребляемые блоки, размеры которых A : B=1 : 2, кратные 5.

Процесс. Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения). Внутри фигуры записывают непосредственно сами операции. В этом блоке знак «=» означает не математическое равенство, а операцию присваивания.

Решение. Отображает решение или функцию переключательного типа с одним входом и двумя или более альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента. Вход в элемент обозначается линией, входящей обычно в верхнюю вершину элемента. Если выходов два или три, то обычно каждый выход обозначается линией, выходящей из оставшихся вершин (боковых и нижней). Если выходов больше трех, то их следует показывать одной линией, выходящей из вершины (чаще нижней) элемента, которая затем разветвляется. Соответствующие результаты вычислений могут записываться рядом с линиями, отображающими эти пути.

Терминатор (пуск-остановка). Элемент отображает вход из внешней среды или выход из нее (наиболее частое применение − начало и конец алгоритма). Внутри фигуры записывается соответствующее действие − начало/конец.

Таблица 1- Средства графического изображения алгоритмов

Наименование	**Символ**	**Выполняемые действия**
Процесс		Выполнение определенной операции или группы операций, преобразование данных.
Решение		Изменение направления продолжения процесса путем проверки некоторого условия.
Терминатор (пуск-остановка)		Обозначает начало и конец алгоритма и прерывание процесса.
Предопределенный процесс		Включение ранее созданных или отдельно сформированных алгоритмов или программ.
Модификация		Операция повторения (изменения) хода выполнения группы действий – начало цикла.
Данные (ввод-вывод)		Ввод – вывод и преобразование данных, носитель которых не определен, в форму, пригодную для обработки на ЭВМ.
Соединитель		Разрыв алгоритма (соединительные блоки) указание связи прерываемых линий потока.
Межстраничный соединитель		Используется для связи прерванных линий потока при переносе блок-схемы на другую страницу.
Комментарий		Используется для вставки примечаний по ходу действия программы. Текст помещается около ограничивающей фигуры.

Предопределенный процесс. Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы (в подпрограмме, модуле). Внутри символа записывается название процесса и передаваемые в него данные.

Модификация. Условия цикла и приращения записываются внутри символа цикла − в зависимости от типа организации цикла. Для изображения итерационного цикла на блок-схеме вместо данного символа используют символ решения, указывая в нем условие, а одну из линий выхода замыкают выше в блок-схеме (перед операциями цикла).

Данные (ввод-вывод). Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод). Данный символ не определяет типа носителя данных.

Соединитель. Используется для обрыва линии и продолжения ее в другом месте (пример: разделение блок-схемы, не помещающейся на листе). Соответствующие соединительные символы должны иметь одно (при том уникальное) обозначение.

Межстраничный соединитель. Используется для связи прерванных линий потока при переносе блок-схемы на другую страницу. Внутри блока указывается номер страницы и номер блока.

Линии переходов используются для обозначения порядка выполнения действий.

Комментарий. Позволяет включать в схемы алгоритмов пояснения к функциональным блокам. Частое использование комментариев нежелательно, т. к. это усложняет (загромождает) схему, делает ее 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 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 детализации.

Вопросы для самопроверки

1. Что такое алгоритм?
2. Какими свойствами должен обладать алгоритм?
3. Какие способы записи алгоритма вы знаете? Приведите примеры.
4. Из каких символов состоит схема алгоритма линейной структуры? Какие символы она не может содержать?
5. Какой порядок выполнения действий называется естественным?

2 Типовые структуры алгоритмов

Алгоритмы можно представлять как некоторые структуры, состоящие из отдельных базовых (т. е. основных) элементов. Эти элементарные шаги объединяются в алгоритмические конструкции.

В зависимости от особенностей своего построения алгоритмы можно разделить на следующие группы:

1) линейные (последовательные);
2) разветвляющиеся;
3) циклические.

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

Для решения любой задачи могут быть построены несколько алгоритмов, приводящих к получению результата ее решения. Из всех возможных алгоритмов следует выбирать наилучший по разным критериям: по точности решения задачи, временным затратам, количеству этапов в алгоритме, их простоте и т. д.

2.1 Линейный алгоритм

Наиболее простой организацией обладает алгоритм линейной

структуры, который обеспечивает получение результата путем однократного выполнения последовательности действий независимо от значений исходных данных или промежуточных результатов. Линейный алгоритм не содержит логических условий и имеет одну ветвь вычислений.

Линейный алгоритм применяется при вычислении арифметического выражения, если в нем используются только простейшие алгебраические действия. Он включает последовательное выполнение следующих этапов:

- ввод исходных данных в память ЭВМ;
- вычисление искомых величин по формулам;
- вывод результатов из памяти ЭВМ на информационный носитель.

Задача 1. Составить алгоритм вычисления объема, массы и площади основания цилиндрического тела, если известны его плотность и геометрические размеры: радиус основания и высота (рис.1).

Входные данные: R (радиус основания цилиндра), h (высота цилиндра), ρ (плотность материала).

Выходные данные: m (масса), V (объем), S (площадь основания).

Потоки данных и управления в данной схеме совпадают со стандартными, поэтому стрелки не используются.

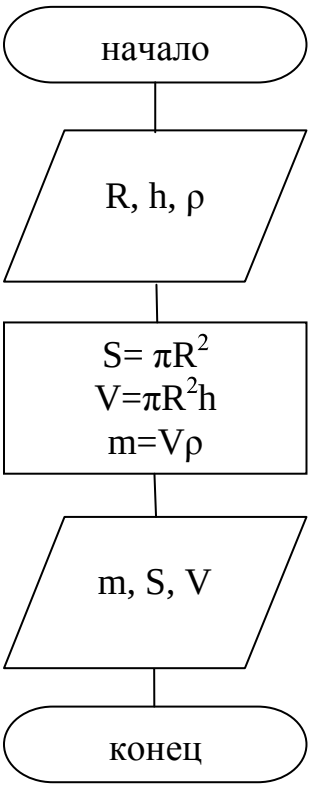


Рисунок 1- Схема линейного алгоритма для задачи 1

Задача 2. Дано действительное число a, за четыре операции **умножения** получить a^3 и a^{10} (рис. 2).

Входные данные: a (исходное число).

Выходные данные: a3 (число a в третьей степени) a10 (число a в десятой степени).

Промежуточные данные: a2 (число a во второй степени) a5 (число a в пятой степени).

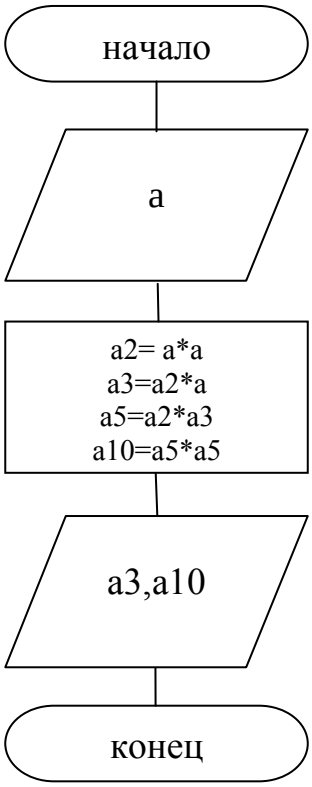


Рисунок 2- Схема линейного алгоритма для задачи 2

В большинстве инженерных задач вычислительный процесс зависит от выполнения некоторых условий, и естественный порядок выполнения алгоритма нарушается, т.е. имеет место или разветвлённый, или циклический вычислительный алгоритм.

2.2 Разветвлённый алгоритм

Разветвленный (разветвляющийся) алгоритм описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки выполнения некоторого логического условия. Результатами проверки являются: «истина» (да), если условие выполняется, и «ложь» (нет), при невыполнении условия.

Разветвлённый алгоритм содержит блок проверки условия **Решение**, и в зависимости от результата проверки выполняется то или иное

действие. Если присутствуют оба действия, то говорят о полной альтернативе (рис. 3).

Полное ветвление

Предполагает выполнение действий для обеих веток в алгоритме:

Если [условие], **то** [действие 1], **иначе** [действие 2]

Условие – это логическое выражение, которое может принимать два значения – «ДА» (истина) или «Нет» (ложь). Если условие верно, действие выполняется, в противном случае – действие не выполняется.

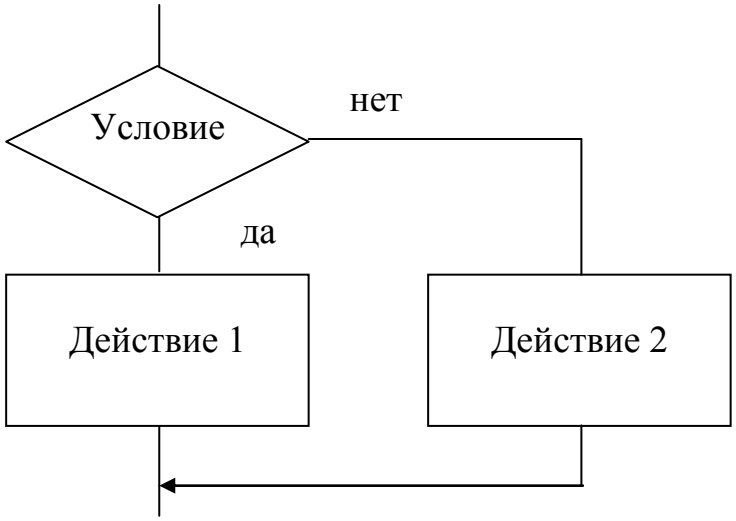


Рисунок 3 - Полная альтернатива

Неполное ветвление

Предполагает выполнение действий только на одной ветви алгоритма (вторая отсутствует):

Если [условие], **то** [действие]

Структура такого алгоритма представлена на рис.4.

При движении по каждой ветви может встретиться следующий логический блок, который образует еще две ветви и так далее. Все ветви, в конце концов, сходятся.

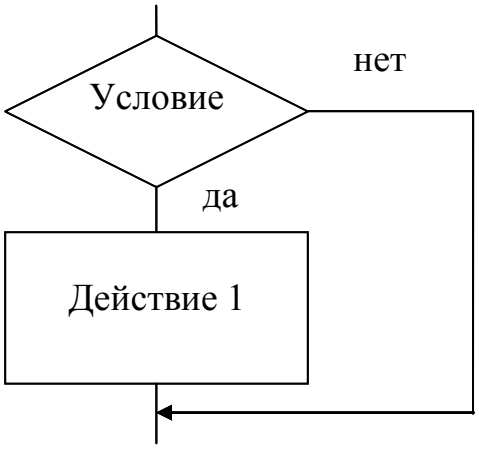


Рисунок 4 - Неполная альтернатива

Задача 1. Составить схему алгоритма вычисления значения y:

$$y = \begin{cases} a + x, \textit{если } a \le b \\ b - 2x, \textit{в остальных случаях} \end{cases}$$

Входные данные: a, b, x.

Выходные данные: y (рис. 5).

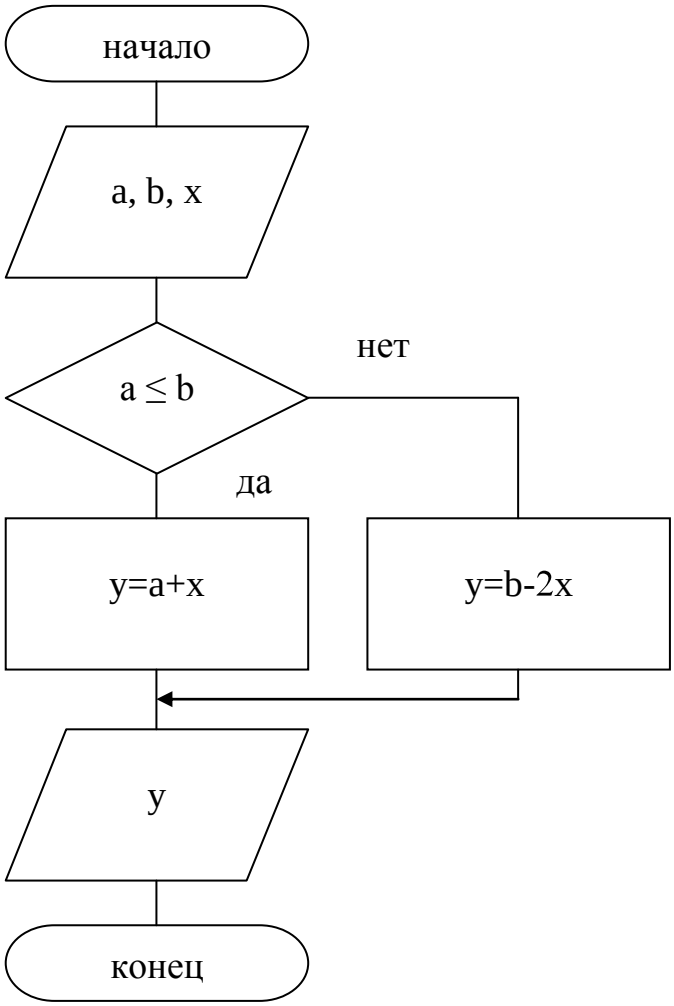


Рисунок 5 – Блок-схема алгоритма вычисления значения y

Задача 2. Составить схему алгоритма нахождения максимального значения среди трех величин: a, b, c.

Входные данные: a, b, c.

Выходные данные: max (рис. 6).

Кроме того, возможны и другие способы решения приведенной задачи, например, при помощи сравнения значений элементов каждой пары или на основе предположений с последующими проверками.

Студенту предлагается самостоятельно составить схему алгоритма решения и провести его оценку.

Аналогично решается задача поиска минимального значения среди нескольких величин.

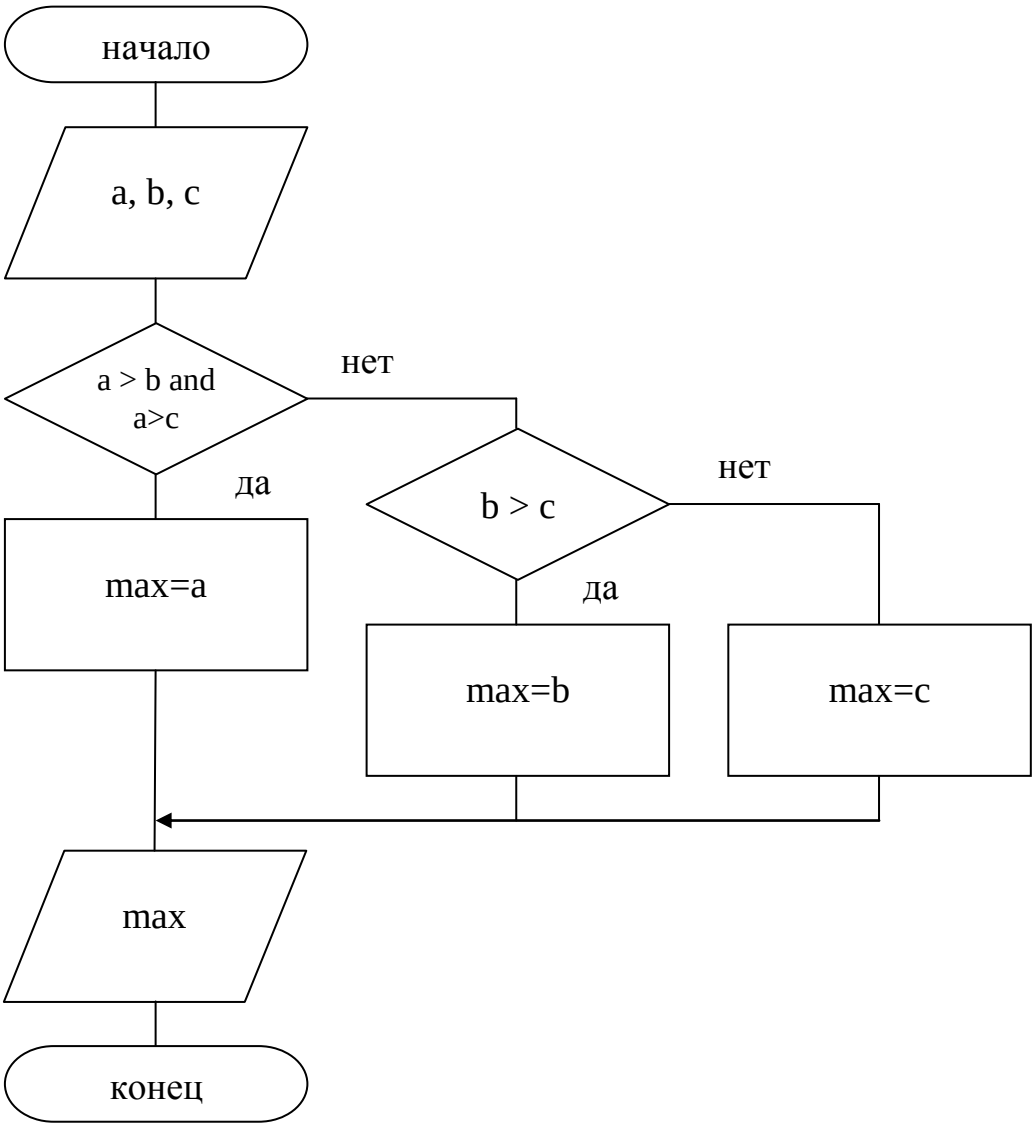


Рисунок 6 – Блок-схема алгоритма нахождения максимального значения

Для программирования ветвящихся алгоритмов применяются условный оператор и оператор выбора.

2.3 Циклический алгоритм

Часто при решении задач приходится проверять одно условие для нескольких значений или повторять некоторые действия несколько раз. Алгоритмы, отдельные действия которых многократно повторяются, называются алгоритмами циклической структуры. Она позволяет существенно сократить объем алгоритма, представить его компактно за счет организации повторений большого числа одинаковых вычислений над разными данными для получения необходимого результата.

Циклической называется конструкция, в которой некая, идущая подряд группа действий (шагов), может выполняться несколько раз в зависимости от входных данных или условия задачи.

Совокупность повторяющихся действий алгоритма называют **циклом,** группа повторяющихся действий на каждом шагу цикла называется **телом цикла.**

Циклический алгоритм включает в себя:

1. Подготовку цикла – действия, связанные с заданием исходных данных, используемых в цикле;

2. Тело цикла – повторяющиеся действия для вычисления искомых величин, а также подготовка значений, необходимых для повторного выполнения действий в теле цикла;

3. Условия продолжения цикла – действия, определяющие необходимость дальнейшего выполнения тела цикла.

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

− по месту расположения условий проверки повторения или окончания цикла можно выделить **циклы с предусловием и постусловием;**

− по способу контроля окончания цикла можно выделить **цикл с неизвестным числом итераций** (количество повторений цикла неизвестно) и цикл с известным числом итераций (**цикл с параметром**).

Цикл с предусловием предполагает, что число итераций заранее не определено и зависит от входных данных задачи. В данной циклической структуре сначала проверяется значение условия перед выполнением очередного шага цикла:

Пока <условие> **выполнять** [действие],

<условие> — некоторое проверяемое логическое условие;

[действие] — тело цикла (последовательность команд, действий).

Условие записывается в виде логического выражения, в нем определяется необходимость дальнейшего выполнения повторяющихся действий.

Оператор цикла с предусловием выполняется следующим образом:

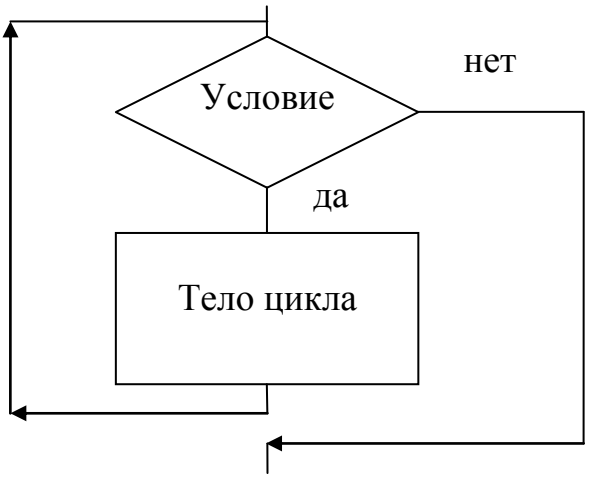
− сначала проверяется условие продолжения цикла;

− если это условие истинно, то выполняется тело цикла;

− затем снова проверяется условие продолжения цикла и т. д.;

− если условие продолжения цикла ложно, то происходит выход из цикла.

Особенностью цикла с предусловием является то, что **если изначально условие ложно, то тело цикла не выполнится ни разу** (рис. 7).



Псевдокод:
начало цикла (нц)
пока
<условие> истинно
выполнять:
тело цикла
(последовательность действий)
конец цикла (кц)

Рисунок 7- Цикл с предусловием

Цикл с постусловием, так же, как и в цикле с предусловием, число итераций заранее не определено и зависит от входных данных задачи. Но, в отличие от него, значение условия проверяется после выполнения очередного шага цикла:

Выполнять [действие] **до тех пор пока** <условие>,

<условие> — некоторое проверяемое логическое условие;

[действие] — тело цикла (последовательность команд, действий).

Условие записывается в виде логического выражения, в нем определяется необходимость дальнейшего выполнения повторяющихся действий.

Оператор цикла с постусловием выполняется следующим образом:

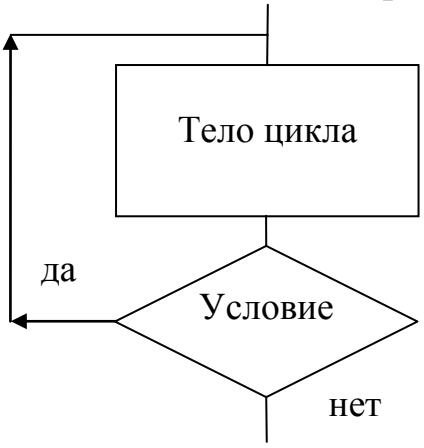
− сначала выполняется тело цикла;

− затем проверяется условие продолжения цикла;

− если это условие истинно, то снова выполняется тело цикла, затем снова проверяется условие и т. д.;

− если условие продолжения цикла ложно, то происходит выход из цикла.

Особенностью цикла с постусловием является то, что **цикл выполнится хотя бы один раз до проверки условия** (рис. 8).



Псевдокод
начало цикла (нц)
выполнять
тело цикла
(последовательность действий)
до тех пор пока истинно <условие>
конец цикла (кц)

Рисунок 8 - Цикл с постусловием

Необходимо отметить, что цикл с постусловием может выполняться по другому сценарию в зависимости от языка программирования, например, в Object Pascal, если условие истинно, то происходит выход из цикла, иначе цикл повторяется.

Задача 1. Дано целое число $m > 1$. Получить наибольшее целое k, при котором $4^k < m$. Составить алгоритм решения задачи (рис. 9).

Входные данные: m (исходное число).

Выходные данные: k (искомое значение).

Промежуточные данные: p (переменная для хранения степени числа 4).

В качестве результата решения задачи производится вывод значения $k-1$, т.к. по условию задачи значение 4^k, которое хранится в P, должно быть меньше m, а выход из цикла происходит при получении значения $P>m$, т.е. предыдущее значение - это искомое значение.

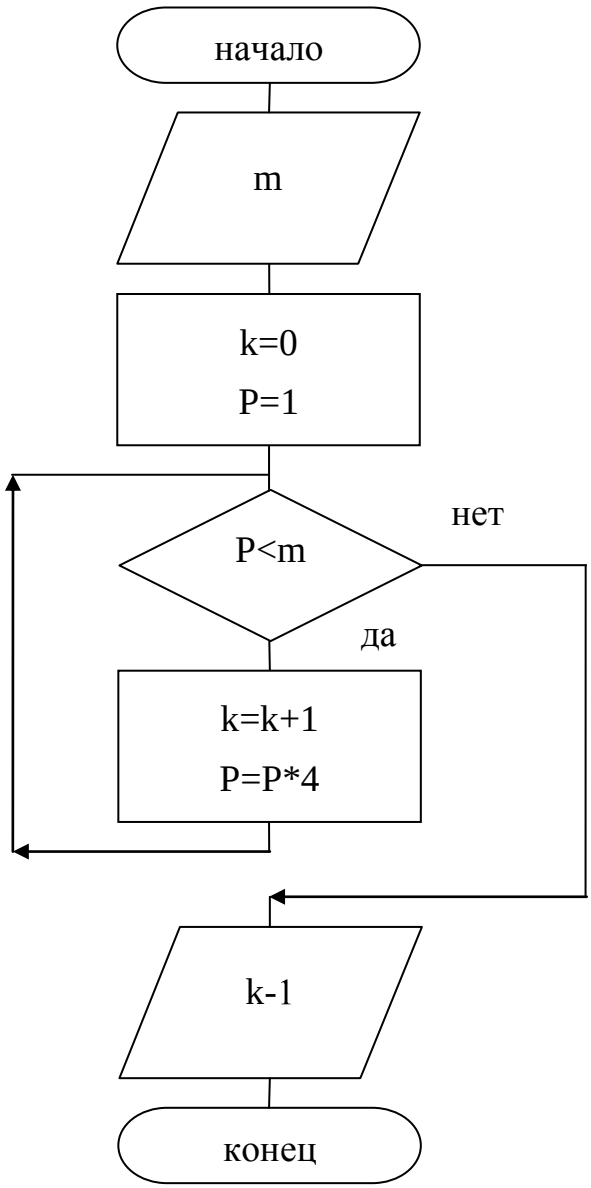


Рисунок 9 – Блок-схема алгоритма нахождения k

Задача 2. Дано натуральное число *n*. Получить наименьшее число вида 2^r, превосходящее *n*. Составить алгоритм решения задачи (рис. 10).

Входные данные: n (исходное число).

Выходные данные: p (искомое значение).

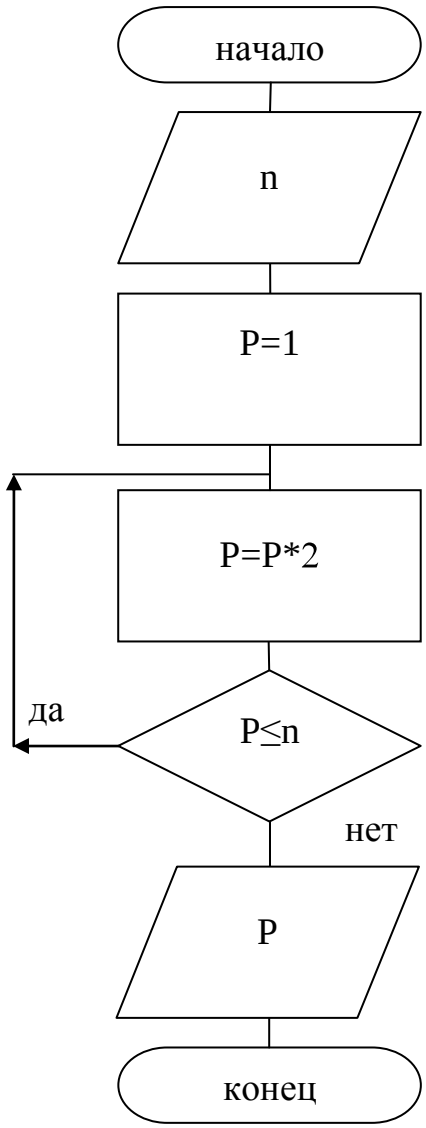


Рисунок 10 – Блок-схема алгоритма нахождения числа вида 2^r

Цикл с параметром предполагает, что число итераций заранее известно. Предписывает выполнять тело цикла для всех значений некоторой переменной (параметра цикла) в заданном диапазоне:

Для всех [параметр цикла] **повторять** [действие]

[параметр цикла] − счетчик количества повторений;

[действие] — тело цикла (последовательность команд, действий).

Количество повторений однозначно определяется правилом изменения параметра, которое задается 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 приведет к значению большему, чем M.

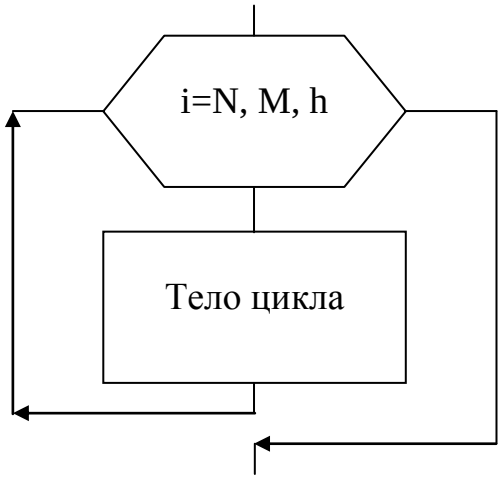
i − параметр цикла (является счетчиком количества повторений);

N − начальное значение параметра цикла;

M − конечное значение параметра цикла;

h − шаг, с которым изменяется параметр цикла, если шаг равен 1 или -1, то его можно не указывать в символе **Модификация**.

Структура цикла с параметром представлена на рис. 11.



Псевдокод
начало цикла (нц)
для
i от M до N с шагом h
повторять:
тело цикла
(последовательность действий)
конец цикла (кц)

Рисунок 11 – Цикл с параметром

При разработке циклического алгоритма необходимо учитывать:

− что использование циклов позволяет существенно сократить схему алгоритма;

− при организации цикла следует особое внимание уделить правильному оформлению изменения параметра цикла, потому что ошибка на этом этапе может привести к «зацикливанию» вычислений;

− внутри цикла параметром не рекомендуется изменять параметр цикла и его конечное значение, т.к. эти значения устанавливаются в самом начале работы цикла;

− количество повторений в цикле зависит от входных данных или условий задачи;

− для завершения в теле цикла должны быть инструкции, выполнение которых влияет на завершение цикла.

Алгоритмы решения сложных задач могут включать все перечисленные структуры. Например, внутри одного цикла могут находиться один или несколько других циклов. 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.

Задача 3 Дано натуральное число *n*. Вычислить *n!* Составить алгоритм решения задачи (рис. 12).

Входные данные: n (исходное число).

Выходные данные: p (искомое значение *n!=1*2*3*…*n*).

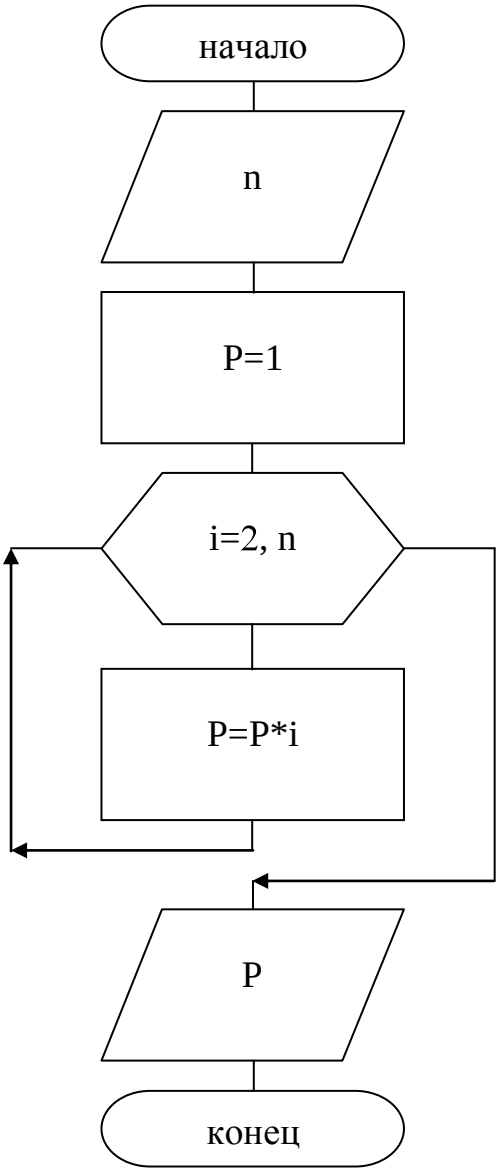


Рисунок 12 – Блок-схема алгоритма нахождения *n!*

На практике часто встречаются задачи, алгоритм решения которых распадается на части, фрагменты и каждый фрагмент представляет собой алгоритм одного из трех описанных типов. Алгоритм, который содержит несколько структур одновременно, называется комбинированным.

В теории алгоритмов доказано, что любой, сколь угодно сложный алгоритм может быть составлен из трех основных алгоритмических

структур: линейной, ветвления и цикла. Алгоритмы решения сложных задач могут включать все перечисленные структуры либо комбинации некоторых из них.

Вопросы для самопроверки

1. Какие типы алгоритмов бывают? Подберите пример алгоритма для каждого типа.
2. Что общего и в чём разница между полной и неполной альтернативой для разветвлённого алгоритма?
3. Назовите типы циклов по способу контроля окончания цикла.
4. Может ли тело цикла не выполняться ни одного раза? В каких случаях?
5. Можно ли в качестве параметра цикла во внешнем и внутреннем цикле использовать одну и ту же переменную?

3 Понятие программы и языка программирования

Программирование – процесс и искусство создания компьютерных программ с помощью языков программирования.

Язык программирования — формальная знаковая система для описания программы работы компьютера в форме, пригодной для трансляции и исполнения на компьютере. Язык программирования – это, прежде всего, инструмент деятельности, и на него в первую очередь оказывает влияние класс решаемых на нем задач. Хотя бы один язык нужно знать досконально, чтобы представлять многообразие имеющихся средств и иметь возможность сравнивать с другими.

Язык C++ наиболее полно представляет парадигмы современного программирования и является фундаментом, на котором оно строится. С его помощью можно писать процедурно-ориентированные программы и создавать библиотеки функций. Он поддерживает объектно-ориентированное программирование и позволяет разрабатывать библиотеки классов. Механизм шаблонов языка C++ и его стандартная библиотека дают возможность создавать программы, применяя методы обобщенного программирования. В программе на C++ можно динамически управлять памятью, использовать адресную арифметику, обращаться к отдельным разрядам двоичного представления данных и т.д.

Язык программирования C++ был разработан Бьерном Страуструпом, сотрудником AT&T Bell Laboratories. Непосредственным предшественником C++ является C with Classes, созданный тем же автором в 1980 году. Язык C with Classes, в свою очередь, был создан под сильным влиянием C и Simula. В определенном смысле C++ можно назвать улучшенным C, тем C, который обеспечивает контроль типов, перегрузку

функций и ряд других удобств. Но главное в том, что C++ добавляет к C объектную ориентированность.

В 1998 г. вышел международный стандарт языка ISO/IEC 14882. С 2003 г. действует вторая редакция этого стандарта ISO/IEC/ANSI /ITI [4].

Изучение программирования нельзя начать, не ответив на вопросы: а что же такое алгоритм, программа, данные, язык программирования.

Определение программы дано в формуле: «Программа = данные + алгоритм». В ней данные и алгоритм являются двумя взаимозависимыми элементами:

– данные (синтаксически) являются аналогом существительных (объектов, над которыми производятся действия), набор операций – аналогом глаголов (выполняемых действий); программа в целом аналогична предложению, описывающему последовательность действий над заданными предметами с целью получения результата;

– если данные в какой-то мере обладают свойствами пространства (объем, протяженность), то алгоритм – свойствами времени (эффективность, быстродействие); эффективность программ может быть принципиально повышена за счет использования дополнительных структур данных в памяти.

Программа – описание на языке программирования структур данных и алгоритма решения задачи, автоматически переводимое, при помощи специальной программы-транслятора (компилятора или интерпретатора), на язык команд компьютера для последующего выполнения.

Можно сказать, что компьютерная программа – один из способов реализации понятия алгоритма, а язык программирования – средство описания алгоритмов. Компьютерная программа, в отличие от абстрактного алгоритма, имеет данные – собственные элементы, над которыми она совершает действия, и которые являются ее составной частью. Алгоритмическая компонента программы – описание последовательности выполняемых действий – обычно состоит из операторов, задающих эту последовательность действий. Программа базируется на наборе операций над данными (арифметические операции, присваивание, проверка значения переменной и т.п.), соответствующем системе команд процессора, на котором она выполняется.

Структура данных – вид представления данных в программе, описание точки зрения пользователя на представление данных. Выбор подходящего представления данных – один из основных вопросов при проектировании программы. При этом под представлением данных понимается их описание на языке программирования в виде констант и переменных разной структуры. При решении задачи на компьютере, анализе исходных данных программы и ее результата, необходимо выбирать экономичный алгоритм решения, который и определит представление исходных, промежуточных и конечных данных.

Неправильное представление данных может сделать программу ненадежной, неэкономичной, сложной и даже вообще неадекватной задаче [5].

Язык программирования содержит в себе компоненты, предназначенные для описания соответствующих частей программы:

– средства описания данных, позволяющие программисту определять различные формы представления данных (типы данных) и переменные разных типов;

– набор операций над основными типами данных (включая ввод-вывод), а также средства записи выражений;

– набор операторов, определяющих различные варианты порядка выполнения выражений в программе (последовательность, условие, повторение, блок);

– средства разбиения программы на независимые части – модули (функции), взаимодействующие между собой через программные интерфейсы.

3.1 Лексические основы языка C++

Алфавит языка C++ включает:

- строчные и прописные буквы латинского алфавита: A..Z и a..z;
- арабские цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- шестнадцатеричные цифры: 0..9, a..f или A..F;
- специальные символы, например «+», «-», «*», «/», «=», «<», «&», «;» и т. д.;
- служебные слова.

Служебные слова зарезервированы в языке для специального применения, т.е. их нельзя использовать в качестве идентификаторов. Они применяются в конструкциях языка в качестве стандартных описателей или ключевых слов. Стандарт ANSI языка C предусматривает следующий список служебных слов:

```
auto      break     case      char      continue
default   do        double    else      enum
extern    float     for       goto      if
int       long      register  return    short
signed    sizeof    static    struct    switch
typedef   union     unsigned  void      while
```

В разных реализациях есть дополнительные служебные слова, например, компиляторы фирмы Borland используют также: `asm cdecl far pascal const volatile.` Язык C++ добавляет еще несколько ключевых слов, например: `catch class friend inline new operator private`.

Из символов алфавита строятся конструкции – слова и предложения языка. Простейшей конструкцией является идентификатор.

Идентификатор – последовательность букв латинского алфавита, десятичных цифр и символов подчеркивания, начинающаяся не с цифры, например: `ABC abc Abc ABc AbC _a MY_Primer_1 Prim_123.`

Прописные и строчные буквы в идентификаторах различаются, т. е. идентификаторы `ABC abc Abc ABc AbC` с точки зрения компилятора языков Си и C++ различны. Идентификаторы используют для обозначения имен переменных, констант, типов, подпрограмм и т. д. На длину различаемой части идентификатора конкретные реализации накладывают ограничения. Так, компилятор Visual C++ 2008 компании Microsoft различает 2048-х первых символов любого идентификатора.

3.2 Типы данных в языке C++

В любом языке программирования каждая константа, переменная, результат вычисления выражения или функции должны иметь определенный тип данных.

Тип данных – это множество допустимых значений, которые может принимать тот или иной объект, а также множество допустимых операций, которые применимы к нему. В современном понимании тип также зависит от внутреннего представления информации.

Таким образом, данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

– внутреннее представление данных в памяти компьютера;

– объем памяти, выделяемый под данные;

– множество (диапазон) значений, которые могут принимать величины этого типа;

– операции и функции, которые можно применять к данным этого типа.

Язык программирования C++ поддерживает следующие типы данных:

– *Базовые типы* (целочисленный, вещественный, символьный, логический, тип `void`). Базовые типы предопределены стандартом языка, указываются зарезервированными ключевыми словами и характеризуются одним значением. Их не надо определять и их нельзя разложить на более простые составляющие без потери сущности данных. Базовые типы объектов создают основу для построения более сложных типов.

– *Производные типы* (массивы, перечисления, указатели, ссылки, структуры, объединения, классы). Производные типы задаются пользователем, и переменные этих типов создаются как с использованием базовых типов, так и типов классов.

– *Типы класса*. Экземпляры этих типов называются объектами.

Существует четыре спецификатора типа данных, уточняющих внутреннее представление и диапазон базовых типов:

`short` (короткий) `long` (длинный)	длина
`signed` (знаковый) `unsigned` (беззнаковый)	знак (модификатор)

Целочисленный (целый) тип данных. Переменные данного типа применяются для хранения целых чисел.

От количества отводимой под объект памяти зависит множество допустимых значений, которые может принимать объект:

– `short int` – занимает 2 байта, следовательно, имеет диапазон от –32 768 до +32 767;

– `int` – занимает 4 байта, следовательно, имеет диапазон от – 2 147 483 648 до +2 147 483 647;

– `long int` – занимает 4 байта, следовательно, имеет диапазон от – 2 147 483 648 до +2 147 483 647;

– `long long int` – занимает 8 байтов, следовательно, имеет диапазон от –9 223 372 036 854 775 808 до +9 223 372 036 854 775 807.

Модификаторы `signed` и `unsigned` также влияют на множество допустимых значений, которые может принимать объект:

– `unsigned short int` – занимает 2 байта, следовательно, имеет диапазон от 0 до 65 536;

– `unsigned int` – занимает 4 байта, следовательно, имеет диапазон от 0 до 4 294 967 295;

– `unsigned long int` – занимает 4 байта, следовательно, имеет диапазон от 0 до 4 294 967 295;

– `unsigned long long int` – занимает 8 байтов, следовательно, имеет диапазон от 0 до 18 446 744 073 709 551 615 [7].

Например:

```
unsigned int b;
signed int a;
int c;
unsigned d;
signed f;
```

Приведем несколько правил, касающихся записи целочисленных значений в исходном тексте программ.

Нельзя пользоваться десятичной точкой. Значения 26 и 26.0 одинаковы, но 26.0 не является значением типа `int`.

Нельзя пользоваться запятыми в качестве разделителей тысяч. Например, число 23,897 следует записывать как 23897.

Целые значения не должны начинаться с незначащего нуля. Он

применяется для обозначения восьмеричных или шестнадцатеричных чисел, так что компилятор будет рассматривать значение 011 как число 9 в восьмеричной системе счисления.

На практике рекомендуется использовать основной целый тип, то есть тип `int`. Данные основного целого типа практически всегда обрабатываются быстрее, чем данные других целых типов. Короткий тип `short` подойдет для хранения больших массивов чисел с целью экономии памяти при условии, что значения элементов не выходят за предельные границы для этих типов. Длинные типы необходимы в ситуации, когда не достаточно типа `int`.

Вещественный (данные с плавающей точкой) тип данных (типы `float` и `double`). Для хранения вещественных чисел применяются типы данных `float` (с одинарной точностью) и `double` (с двойной точностью). Смысл знаков "+" и "-" для вещественных типов совпадает с целыми. Последние незначащие нули справа от десятичной точки игнорируются. Поэтому варианты записи +523.5, 523.5 и 523.500 представляют одно и то же значение.

Для представления вещественных чисел используются два формата:

– с фиксированной точкой

```
[знак][целая часть].[дробная часть]
```

Например: –8.13; .168 (аналогично 0.168); 183. (аналогично 183.0).

– с плавающей точкой (экспоненциальной форме)

```
мантисса E/e порядок
```

Например: 5.235e+02 ($5.235 \cdot 10^2 = 523.5$);

–3.4E-03 ($-3.4 \cdot 10^{-3} = -0.0034$)

В большинстве случаев используется тип `double`, он обеспечивает более высокую точность, чем тип `float`. Максимальную точность и наибольший диапазон чисел достигается с помощью типа `long double`.

Величина с модификатором типа `float` занимает 4 байта. Из них 1 бит отводится для знака, 8 бит для избыточной экспоненты и 23 бита для мантиссы. Отметим, что старший бит мантиссы всегда равен 1, поэтому он не заполняется, в связи с этим диапазон модулей значений переменной с плавающей точкой приблизительно равен от 3.14E–38 до 3.14E+38.

Величина типа `double` занимает 8 байтов в памяти. Ее формат аналогичен формату `float`. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы. С учетом опущенного старшего бита мантиссы диапазон модулей значений переменной с двойной точностью равен от 1.7E–308 до 1.7E+308.

Величина типа `long double` аналогична типу `double`.

Например:

```
float a, b;
double x, y;
```

```
long double z;
```

Символьный тип данных (тип `char`). В стандарте C++ нет типа данных, который можно было бы считать действительно символьным. Для представления символьной информации есть два типа данных, пригодных для этой цели, – это типы `char` и `wchar_t.`

Переменная типа `char` рассчитана на хранение только одного символа (например, буквы или пробела). В памяти компьютера символы хранятся в виде целых чисел. Соответствие между символами и их кодами определяется таблицей кодировки, которая зависит от компьютера и операционной системы. Почти во всех таблицах кодировки есть прописные и строчные буквы латинского алфавита, цифры 0, ... , 9 и некоторые специальные символы. Самой распространенной таблицей кодировки является таблица символов ASCII (American Standard Code for Information Interchange – Американский стандартный код для обмена информацией).

Так как в памяти компьютера символы хранятся в виде целых чисел, то тип `char` на самом деле является подмножеством типа `int.`

Под величину символьного типа отводится 1 байт.

Тип `char` может использоваться со спецификаторами `signed` и `unsigned`. В данных типа `signed char` можно хранить значения в диапазоне от –128 до 127. При использовании типа `unsigned char` значения могут находиться в диапазоне от 0 до 255. Для кодировки используется код ASCII. Символы с кодами от 0 до 31 относятся к служебным и имеют самостоятельное значение только в операторах ввода-вывода.

Величины типа `char` также применяются для хранения чисел из указанных диапазонов.

Тип `wchar_t` предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например в кодировке Unicode. Размер типа `wchar_t` равен 2 байтам. Если в программе необходимо использовать строковые константы типа `wchar_t`, то их записывают с префиксом `L`, например, `L"Слово".`

Например:

```
char c='c';
char a,b;
char r[]={'A','B','C','D','E','F','\0'};
char s[] = ”ABCDEF”;
```

Логический (булевый) тип данных (тип `bool`). В языке C++ используется двоичная логика (истина, ложь). Лжи соответствует нулевое значение, истине – единица. Величины данного типа могут также принимать значения `true` и `false`.

Внутренняя форма представления значения `false` соответствует 0,

любое другое значение интерпретируется как `true`. В некоторых компиляторах языка С++ нет данного типа, в этом случае используют тип `int`, который при истинных значениях выдает 1, а при ложных – 0. Под данные логического типа отводится 1 байт.

Перечисляемый тип (тип `enum`). Данный тип определяется как набор идентификаторов, являющихся обычными именованными целыми константами, которым приписаны уникальные и удобные для использования обозначения. Таким образом, перечисления представляют собой упорядоченные наборы целых значений. Они имеют своеобразный синтаксис и достаточно специфическую область использования.

Переменная, которая может принимать значение из некоторого списка определенных констант, называется переменной перечисляемого типа или перечислением. Данная переменная может принимать значение только из именованных констант списка. Именованные константы списка имеют тип `int`. Следовательно, память, соответствующая переменной перечисления, – это память, необходимая для размещения значения типа `int`.

Например:

```
enum year {w inter, spring, summer, autumn};
enum week {Sunday, Monday, Tuesday, Wednesday,
 Thursday, Friday, Saturday};
```

Тип `void`. Множество значений этого типа пусто. Тип `void` имеет три назначения:

– указание о невозвращении функцией значения;

– указание о неполучении параметров функцией;

– создание нетипизированных указателей.

Тип `void` в основном используется для определения функций, которые не возвращают значения, для указания пустого списка аргументов функции, как базовый тип для указателей и в операции приведения типов [5].

При вычислении выражений некоторые операции требуют, чтобы операнды имели соответствующий тип, в противном же случае на этапе компиляции выдается сообщение об ошибке. Например, операция взятия остатка от деления (`%`) требует целочисленных операндов. Поэтому в языке С++ есть возможность приведения значений одного типа к другому.

Преобразование типов – это приведение значения переменной одного типа в значение другого типа.

Выделяют *явное* и *неявное* приведения типов. При явном приведении указывается тип переменной, к которому необходимо преобразовать исходную переменную. При неявном приведении преобразование происходит автоматически, по правилам, заложенным в языке программирования С++.

Формат операции явного преобразования типов:

```
имя_типа (операнд)
```

Например, `int(x), float(2/5), long(x+y/0.5).`

Пример.

```
//Взятие цифры разряда сотых в дробном числе
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain( int argc, _TCHAR* argv[]){
 float s,t;
 long  int a,b;
 printf("Введите вещественное число\n");
 scanf("%f", &s);
 t=s*100;
 a=int(t);
 //переменная t приводится к типу  int в переменную a
 b=a%10;
 printf("\nЦифра разряда сотых числа %f равна %d.", s, b);
 system("pause");
 return 0;
}
```

Преобразования типов нужно применять с осторожностью, так как данная операция может приводить к потере информации. Например, после приведения длинного типа к более короткому происходит усечение информации из старших битов.

Пример. Заданы моменты начала и конца некоторого промежутка времени в часах, минутах и секундах (в пределах одних суток). Найти продолжительность этого промежутка в тех же единицах.

Исходными данными для этой задачи являются шесть целых величин, задающих моменты начала и конца интервала, результатами – три целых величины (тип `int`).

Обозначим переменные для хранения начала интервала `hour1, min1` и `sec1`, для хранения конца интервала – `hour2, min2` и `sec2`, а результирующие величины – `hour, min` и `sec`.

Для решения этой задачи необходимо преобразовать оба момента времени в секунды, вычесть первый из второго, а затем преобразовать результат обратно в часы, минуты и секунды. Следовательно, потребуется промежуточная переменная `sum_sec`, в которой будет храниться величина интервала в секундах. Она может иметь весьма большие значения, ведь в сутках 86400 секунд, что выходит за пределы типа `int`. Следовательно, для этой переменной выберем длинный целый тип (`long int`, сокращенно `long`).

Для перевода результата из секунд обратно в часы и минуты используется отбрасывание дробной части при делении целого числа на целое.

```
//Временной интервал. Форматированный ввод-вывод данных
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain( int argc, _TCHAR* argv[]){
 int hour1, min1, sec1, hour2, min2, sec2, hour, min, sec;
 long  int sum_sec;
 printf("Введите время начала интервала (час мин сек)\n");
 scanf("%d%d%d", &hour1,&min1,&sec1);
 printf("Введите время окончания интервала (час мин сек)\n");
 scanf("%d%d%d", &hour2,&min2,&sec2);
 sum_sec = (hour2-hour1)*3600+(min2-min1)*60+sec2-sec1;
 hour = sum_sec/3600;
 min = (sum_sec-hour*3600)/60;
 sec = sum_sec-hour*3600-min*60;
 printf("Продолжительность промежутка от %d:%d:%d до
 %d:%d:%d\n",hour1,min1,sec1,hour2,min2,sec2);
 printf(" равна %d:%d:%d\n",hour,min,sec);
 system("pause");
 return 0;
}
```

При выполнении математических операций производится неявное (автоматическое) преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах. Выполнение преобразования зависит от специфики операций и от типа операнда или операндов.

Преобразование целых типов со знаком:

– *Целое со знаком преобразуется к более короткому целому со знаком*, с потерей информации: пропадают все разряды числа, которые находятся выше (или, соответственно – ниже) границы, определяющей максимальный размер переменной.

– *Целое со знаком преобразуется к более длинному целому со знаком*. Путем размножения знака. То есть все добавленные биты двоичного числа будут заняты тем же числом, которое находилось в знаковом бите: если число было положительным, то это будет, соответственно 0, если отрицательным, то 1.

– *Целое со знаком к целому без знака*. Первым шагом целое со знаком преобразуется к целому со знаком, соответствующему целевому типу, если этот тип данных крупнее. У получившегося значения бит знака не отбрасывается, а рассматривается равноправным по отношению к остальным битам, то есть теперь все биты образуют числовое значение.

– *Преобразование целого со знаком к плавающему типу* происходит без потери информации, за исключением случая преобразования типа `long`

`int` или `unsigned long int` к типу `float`, когда точность часто может быть потеряна.

Преобразование целых типов без знака:

– *Целое без знака преобразуется к более короткому целому без знака или со знаком* путем усечения.

– *Целое без знака преобразуется к более длинному целому без знака или со знаком* путем добавления нулей слева.

– *Целое без знака преобразуется к целому со знаком того же размера*. Если взять для примера, `unsigned short` и `short` – числа в диапазоне от 32768 до 65535 превратятся в отрицательные.

– *Целое без знака преобразуется к плавающему типу*. Сначала оно преобразуется к значению типа `signed long`, которое затем преобразуется в плавающий тип.

Преобразования плавающих типов:

– *Величины типа `float` преобразуются к типу `double`* без изменения значения.

– *Величины `double` преобразуются к `float`* с некоторой потерей точности, то есть, количества знаков после запятой. Если значение слишком велико для `float`, то происходит потеря значимости, о чем сообщается во время выполнения.

При преобразовании величины с плавающей точкой к целым типам она сначала преобразуется к типу `long` (дробная часть плавающей величины при этом отбрасывается), а затем величина типа `long` преобразуется к требуемому целому типу. Если значение слишком велико для `long`, то результат преобразования не определен. Обычно это означает, что на усмотрение компилятора может получиться любой «мусор». В реальной практике с такими преобразованиями обычно сталкиваться не приходится [9].

3.3 Переменные и константы

Данные представляются в программе в виде констант и переменных.

Переменная:

– абстракция, представляемая в программе идентификатором и связанной с ним ссылкой на начальный байт некоторой области памяти, в которой хранится значение переменной и по которому в программе осуществляется доступ к нему (в результате трансляции имя переменной превращается в адрес некоторого участка памяти и во время выполнения программы оно ссылается на этот участок памяти);

– в каждый конкретный момент времени может иметь только одно значение (состояние), которое может изменяться в процессе работы программы;

– характеризуется набором некоторых характеристик, называемых

атрибутом, и состоянием (значением).

Переменная в ее первичном архитектурном понимании – это область памяти, в которой содержатся данные определенного типа. Имя переменной напрямую ассоциируется с ее адресом, содержимое памяти – со значением переменной.

Строка в программе, где содержится описание переменной, по которой транслятор создает ее внутреннее (компьютерное) представление, называется определением переменной.

Например, определения переменных:

```
char symbol;
unsigned short counter;
```

обеспечивают выделение, соответственно, одного и двух байтов памяти:

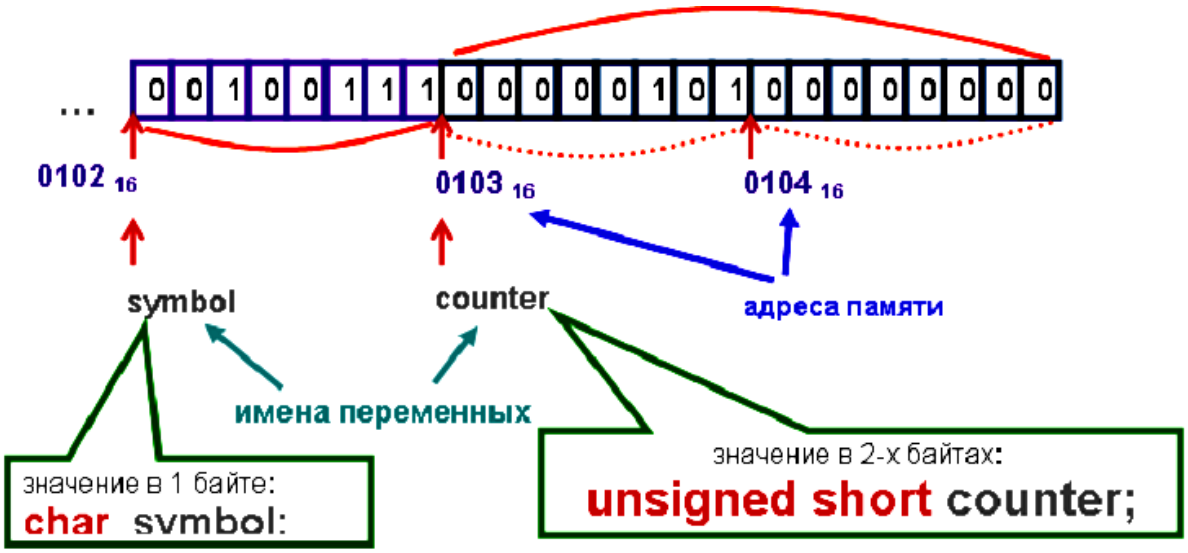


Рисунок 13 – Схема размещения в памяти переменных

Имеется дуализм (двойственность) в использовании имени переменной. В одних случаях имя обозначает хранимое значение, а в других – память (адрес, ссылку, место размещения), его хранящую. Наиболее яркий пример – присваивание. Выражение `a=b` в правой части подразумевает значение переменной `b`, а в левой части – ссылку на `a`.

Объект в программе характеризуют его атрибуты:

– имя (идентификатор);
– адрес;
– значение в конкретный момент времени;
– тип;
– класс памяти (определяет местонахождение и время существования памяти, сопоставленной с именем объекта):

1) место размещения (регистр, стек, сегмент данных, динамическая память и т.п.);

2) время жизни (постоянное, в течение выполнения программы или временное, в течение выполнения блока);

3) область видимости объекта – часть программы, из которой допустим обычный доступ к области памяти, связанной с идентификатором переменной;

4) область действия идентификатора - часть программы, в которой идентификатор объекта можно использовать для доступа к связанной с ним области памяти.

Определение и объявление объекта. Перед тем как использовать объект, его нужно описать (определить или объявить):

– Определения связывают объекты программы с их идентификаторами, резервируют для объектов память и устанавливают их атрибуты.

– Объявления делают идентификаторы объектов известными компилятору.

– В программе определение объекта всегда одно, объявлений может быть несколько.

Описание является определением, если:

– содержит инициализатор;

– полностью описывает функцию (включает тело функции);

– описывает объединение или структуру (включая компоненты);

– описывает класс (включая его компоненты);

Описание является объявлением, если:

– описывает прототип функции;

– содержит спецификатор `extern`;

– описывает статический компонент класса;

– описывает имя класса;

– описывает имя типа, вводимого пользователем (`typedef`).

Примеры описаний программных объектов

```
int a;  //определение целочисленной переменной
char c;  //определение символьной переменной
float x, y;  //определение вещественных переменных
struct complex { float re, im; };
//определение структурного типа (структуры)
struct complex1 {  int re, im; };
complex1 var;
//определение структурного типа (структуры) и переменной этого типа
extern complex funt(complex);
//объявление функции, которая принимает
//в качестве аргумента и возвращает структуру
typedef complex point;
//объявление типа  point  как синонима типа  complex
extern int error_number; //объявление переменной типа  int
int extern c;  //объявление переменной типа  int
```

```
static int a;
```
//определение статической переменной типа `int`
```
float real(complex* p) { return p->re; };
```
//определение функции
```
struct user;
```
//**объявление** структурного типа `user`

Описание может не только ассоциировать тип с именем. Большинство описаний являются также определениями, то есть они определяют для имени сущность, к которой оно относится.

Переменная может быть инициализирована выражением любой сложности, включая вызовы функций.

Определение и инициализация переменных

```
int k, ix, iy=1, iz;
int i = 0xabcd, j=04567, k (1);
//0xabcd – значение в 16-ой с/с; 04567 – в 8-ичной с/с
unsigned long ulk (12345);
float x = 4.7; //или float x (4.7);
double y = 3.141592, z = 0.6E-02;
double salary = 9999.99, wage = salary + 0.01;
bool b( true);
bool b(1);
bool b1 = false;
bool b2 = bool (-25); //b2 получит значение true
char ch1 (':'), symbol ='d';
char c1=58;//числовой код символа «:»
/* однако инициализируя так переменную, мы уменьшаем или даже вообще ликвидируем такое свойство программы, как «переносимость»*/
char first ='\n';//инициализация ESC-последовательностью \n
char ch2 = '\''; // инициализация символом «'»
char ch3 = '\x3f', c3('?');
//ESC-последовательность \x3f задает в 16 с/с код символа «?»
// \x3f = 3f16 = 3*16+f = 3*16+15 = 6310
```

Константы – это данные, которые устанавливаются равными определенным значениям еще до выполнения программы и сохраняют их на протяжении выполнения всей программы. В C++ используются константы следующих видов:

– литералы,

– именованные константы,

– препроцессорные константы,

– константы перечислений.

Литерал – неименованная константа – синтаксически правильная конструкция, представляющая фиксированное значение определенного типа, записываемое непосредственно в тексте программы. Тип константы определяется по записи ее литерала.

В качестве литералов в С++ могут использоваться:

– целые константы:

1) десятичные `33 44 -52 8 -461 0 2010`
2) восьмеричные `0737 0152 -0121`
3) шестнадцатеричные `0X2F56 0x15c -0x2a13b`

– вещественные константы `0.25 0.25f -56.12e-12`

– логические константы `true false`

– символьные (литерные) `'K' 'z' '2'`

– строковые константы `"the first number"`

– неопределенный указатель `NULL`

Именованные константы позволяют сделать запись более наглядной. Идентификаторы именованных констант рекомендуется записывать прописными буквами.

В С++ используются типизированные именованные константы:

– для задания параметров, управляющих:

1) размером структур данных
   (например, массивов: `const int N=10; int mas[N];`),
2) числом итераций в циклах
   `(const int K=10; for ( int i=1; i<K; i++))`

– для задания других значений, изменение которых может потребоваться при отладке или модернизации программы;

– для обозначения часто встречающихся в программе постоянных величин;

– при использовании констант, имеющих общеупотребительные обозначения.

Препроцессорные константы – также возможный способ определения констант в С++. Для этого используется директива препроцессора `#define`:

```
#define IDENTIFICATOR ZAMENA
```

Эта директива вызывает замену в тексте программы каждого появления имени `IDENTIFICATOR` на текст `ZAMENA`.

`#define MAX_LEN 100 //`в программе имя `MAX_LEN` заменится на `100`

С помощью директивы `#define` можно выполнять «настройку» программы. Например, если в программе предполагается работать с массивом, то его размер можно явно определить на этапе препроцессорной обработки заданием именованного литерала:

исходный текст:

```
#define K 40
int main ()
{ int M[K] [K];
float A[K], B[K] [K];…
```

результат препроцессорной обработки:

```
int main ()
{ int M[40] [40];
float A[40], B[40] [40];…
```

При таком описании легко изменять предельные размеры сразу всех

массивов, изменив только одно значение в команде `#define`. В то же время сегодня такой способ определения именованных констант в C++ используется редко [10].

Константы перечислимого типа вводятся с помощью ключевого слова `enum`. Это обычные целочисленные константы, которым приписаны уникальные и удобные для использования обозначения (идентификаторы, отражающие назначение, смысл константы). Они могут быть использованы в любых выражениях, в которых допустим целочисленный тип данных. Используемые идентификаторы должны быть уникальными в пределах контекста объявления `enum`:

```
enum {one=1, two=2, three=3};
```

// `one, two, three` – условные имена, введенные для обозначения констант

```
enum color {red, yellow, green};
```

//значения приписываются по умолчанию: `red=0, color` – название перечислимого типа

```
enum {red=10, yellow=3, green=4};
```

//приписываются указанные значения: `red=10` и т.д.

```
enum {red=10, yellow=3, green=10};
enum {red=10, yellow=3, green=red*10};
enum {red=-2, yellow, green};
```

3.4 Операции и выражения

Выражения относятся к базовым элементам языка и представляют самый нижний уровень конструкций языка программирования.

Выражением называется любая допустимая, записанная по определенным правилам и задающая определенное вычисление, последовательность операций, операндов и разделителей. Выражение есть правило для получения значения. Как частный случай, выражение может содержать только константу или единственную переменную, или только обращение к функции. Один и тот же знак операции может употребляться в различных выражениях и по-разному интерпретироваться в зависимости от контекста. Для изображения операций часто используется несколько символов.

Операции – это специальные комбинации символов, специфицирующие действия по преобразованию различных величин.

Примеры выражений:

```
(a+0.12)/6
 x && y || !z
 (t*sin(x) - 2./5)/(1+exp(cos(x)))
 a=b=c
```

Вычисление выражений выполняется по определенным правилам

преобразования, группировки, ассоциативности и приоритета, которые зависят от типов данных операндов и используемых операций, наличия круглых скобок.

Выражения могут давать в результате: именующее (lvalue) выражение, указывающее на объект; значения переменных (rvalue-выражение); либо не давать никаких значений (например, вызов функции, которая не возвращает значение).

Слово «lvalue» первоначально было придумано для значения «нечто, что может стоять в левой части присваивания». Объект есть область памяти; lvalue есть выражение, ссылающееся на объект. Однако не всякий адрес можно использовать в левой части присваивания; бывают адреса, ссылающиеся на константу. В левой части оператора присваивания допустимы только модифицируемые lvalue – это объекты, удовлетворяющие следующим условиям:

– можно получить адрес объекта;
– можно получить значение объекта;
– это значение легко модифицировать (в объявлении объекта нет спецификатора const).

lvalue выражением может быть:

– идентификатор переменной следующих типов:
  1) целочисленный;
  2) с плавающей точкой;
  3) указатель;
  4) структура (`struct`);
  5) объединение (`union`);
– выражение с индексом (`[ ]`), не являющееся само массивом;
– обращение к члену структуры, объединения, класса (`->` или `.`);
– выражение взятия значения (*) не являющееся массивом;
– любое lvalue в скобках;
– любое lvalue со спецификатором `const` (немодифицируемое lvalue).

Примеры lvalue-выражений:

`a=1; b=a+b;` если `a` и `b` – идентификаторы переменных целого типа (`int a, b;`);

`*P`, где `P` – выражение, дающее непустой указатель.

Напротив, rvalue – это выражение, значение которого вычисляется, или выражение, обозначающее временный объект, для которого нельзя получить адрес и значение которого нельзя модифицировать.

Примеры rvalue-выражений: `a+b`, поэтому запись типа `a+b = a` недопустима.

Результат вычисления выражения характеризуется значением и типом. Операции выполняются в соответствии с их приоритетами.

В выражение могут входить операнды различных типов (по умолчанию, для сохранения значимости и точности, перед вычислениями

выполняются преобразования типов из более коротких в более длинные).

Операции в С++ классифицируют по числу участвующих в них операндов или по типу действия, которое они выполняют.

По числу операндов, участвующих в операции, различают следующие типы:

– унарные (имеющие один операнд);
– бинарные (имеющие два операнда);
– тернарные (имеющие три операнда).

По типам выполняемых действий операции подразделяют на:

– арифметические `+ - * / % -` (изменение знака);
– операции инкремента и декремента `++ --;`
– присваивания `= *= /= %= += -= <<= >>= &= |= ^= ;`
– отношения `< <= > >= == !=;`
– логические `&&(и) ||(или) !(не);`
– поразрядные логические и сдвига `& | ^ ~ << >>;`
– адресные `& *;`
– дополнительные `?: , () [] -> . ->* .* sizeof(тип) ::`

Таблица 2- Операции языка С++

Приоритет	Символ операции
1	`( ) [ ] -> :: .`
2	`! (не) + - (унарные) ++ -- &(адрес) *(указатель) sizeof new delete`
3	`.* ->* Выбор члена косвенный`
4	`* / %`
5	`+ - (бинарные)`
6	`<< >>`
7	`< <= > >=`
8	`== !=`
9	`&(поразрядное и)`
10	`^(исключающее или)`
11	`| (поразрядное или)`
12	`&&`
13	`||`
14	`?:`
15	`= *= /= %= += -= &= ^= |= <<= >>=`
16	`,`

Все операции в С++ разбиты на группы, имеющие определенный приоритет и ассоциативность. Чем выше приоритет группы операций, тем выше она расположена в таблице. В рамках группы приоритеты одинаковы. Ассоциативность определяет порядок выполнения операций, если отсутствуют скобки и операции имеют один приоритет: часть операций выполняется справа налево, часть (унарные операции, условная операция и операции присваивания) – слева направо. Для изменения порядка выполнения операций используются круглые скобки. В таблице 2 приведен перечень всех операций языка, упорядоченных в порядке убывания приоритета [6].

Сложные выражения представляют собой последовательность простых, записанных через запятую «,»:

<Выражение1>,<Выражение2>,...<Выражение n>

По таблице приоритетов операций запятая имеет низший ранг, поэтому простые выражения, разделенные запятой, выполняются последовательно слева направо, а в качестве результата выражения берется тип и значение самого правого выражения.

Например:

```
int m=5,z;
z=(m=m*5,m*3);// результат: m=25, z=75
int d,k;
k=(d=4,d*8);// результат: d=4, k=32 .
c=(a=5, b=a*a);// эквивалентно a=5; b=a*a; c=b;
```

В выражениях можно применять стандартные математические функции из библиотеки `math`. При их использовании необходимо подключить файл прототипов функций `math.h:#include <math.h>`

Наиболее часто используют следующие функции:

– `fabs`(<Вещественное выражение>) // абсолютное значение вещественного числа;

– `abs`(<Целое выражение>) // абсолютное значение целого числа;

– `sqrt`(<Вещественное выражение>) // $\sqrt{x}$;

– `exp`(<Вещественное выражение>) // e^x;

– `log`(<Вещественное выражение>) // ln x;

– `log10`(< Вещественное выражение >) // $\log_{10} x$;

– `sin`(<Вещественное выражение>) // sin x;

– `cos`(<Вещественное выражение>) // cos x;

– `atan`(<Вещественное выражение>) // arctg x;

– `tan`(< Вещественное выражение >) // tg x;

– `acos`(< Вещественное выражение >) // арккосинус;

– `asin`(< Вещественное выражение >) // арксинус;

– `sinh`(<Вещественное выражение>) // гиперболический синус;

– `cosh`(<Вещественное выражение>) // гиперболический косинус.

Кроме этих функций еще достаточно часто используют функции, позволяющие получать последовательности случайных чисел. Их прототипы находятся в файле `conio.h.`

– `rand()` – генерация случайного числа (0-32767);

– `srand`(<Целое число>) – инициализация генератора случайных чисел.

Например:

```
srand((unsigned)time(NULL));
```

//инициализация датчика текущим временем

```
num=rand()/1000.0;
```
// генерация вещественного случайного числа

Для такой инициализации к проекту должен быть подключен файл `time.h: #include <time.h>` [7].

3.5 Простейший ввод-вывод

Ввод-вывод в языке C++ осуществляется потоками байтов [4]. Поток – это просто последовательность байтов. В операциях ввода байты пересылаются от устройства ввода (например, клавиатуры, дисковода или соединений сети) в оперативную память. При выводе байты пересылаются из оперативной памяти на устройства (например, экран дисплея, принтер или дисковод). Язык C++ предоставляет возможности для ввода-вывода как на низком, так и на высоком уровнях. Ввод-вывод на низком уровне обычно сводится к тому, что некоторое число байтов данных следует переслать от устройства в память или из памяти в устройство. При такой пересылке каждый байт является самостоятельным элементом данных. Передача на низком уровне позволяет осуществлять пересылку больших по объему потоков ввода-вывода с высокой скоростью, но такая передача обычно оказывается неудобной для программиста и пользователя. Операции ввода-вывода на высоком уровне осуществляются путем преобразования байтов в такие значащие элементы данных, как целые числа, числа с плавающей запятой, символы, строки и т.д. Стандартные библиотеки C++ имеют расширенный набор средств ввода-вывода, при этом большая часть программ включает заголовочный файл `<iostream.h>,` который содержит основные сведения, необходимые для всех операций с потоками ввода-вывода. Так, например, он включает объекты `cin, cout, cerr, clog,` которые соответствуют стандартным потокам ввода-вывода и стандартным потокам вывода сообщений об ошибках.

Объект стандартного потока ввода `cin` связан со стандартным устройством ввода, обычно с клавиатурой. Операция взять из потока (`cin` – the standard input stream – стандартный поток ввода), показанная в приведенном ниже операторе, означает, что величина переменной `x`

должна быть введена из объекта cin в память cin >> x;.

Объект стандартного потока вывода cout связан со стандартным устройством вывода, обычно с экраном дисплея. Операция поместить в поток (cout - standard output stream – стандартный поток вывода), показанная в приведенном ниже операторе, означает, что величина переменной x должна быть выведена из памяти на стандартное устройство вывода cout << x;.

Объекты cerr и clog связаны со стандартным устройством вывода сообщений об ошибках. Их различие состоит в том, что при использовании cerr сообщение об ошибках выводится мгновенно, тогда, как в случае применения объекта clog сообщения об ошибках помещаются в буфер, где они хранятся до тех пор, пока буфер полностью не заполнится или пока содержимое буфера не будет выведено принудительно.

Примеры ввода-вывода значений.

```
// Вывод строки
# include <iostream.h>
int main( )
{
 cout << “Добро пожаловать в мир C++ ! \n”;
 return 0;
}
// Вывод выражений
# include <iostream.h>
int main( )
{
 cout << “47 плюс 53 равняется”;
 cout << ( 47 + 53 );  //выражение
 cout << endl;
 return 0;
}
// Вычисления по формуле
#include <iostream>
#include <iomanip>
#include <conio.h>
#define _USE_MATH_DEFINES
#include <cmath>
int main()
{
 double x, y, r1, r2, r3, r4;
 cout << "enter a real number x:\n";
  //приглашение для ввода переменной x
 cin >> x; // ввод значения  1.79 с клавиатуры
```

```
 r1=cos(x);
 r2 = pow((exp(r1) + x*x +sin(x)), .25);
 r3 = (sin(M_PI*x*x) + log(x*x));
 r4 = pow(r3, r1);
 y=r2*r4;
cout << "\nfor x=";
cout.width(7); //ширина поля вывода для x
cout <<setiosflags(ios::fixed)<<setprecision(6)<< x;
//форма вывода и точность
cout << setw(8); // ширина поля вывода для " y="
cout <<  " y=" ;
cout << setprecision(8);
//точность - количество знаков после запятой для y
cout.width(15); //ширина поля вывода для y
cout << y<< endl;
_getch();
return 0;
}
```

Для ввода/вывода данных скалярных типов и строк можно использовать стандартные функции ввода/вывода, описанные в библиотеке `stdio`. Для применения этих функций необходимо, чтобы программе был доступен файл `stdio.h`, содержащий прототипы функций из этой библиотеки. В стандарте C++ библиотека фигурирует под именем `<cstdio>`. Для этого необходимо подключить этот файл с помощью директивы препроцессора `include: #include <stdio.h>`

В библиотеке существуют три вида функций, организующих элементарный ввод с клавиатуры и вывод на экран:

– форматный ввод/вывод – для выполнения операций ввода/вывода над скалярными значениями, символами и строками;

– ввод/вывод строк;

– ввод/вывод символов.

Форматный ввод чисел, символов и строк с клавиатуры:

`int scanf(`<Форматная строка>, <Список адресов переменных>);

Форматный вывод чисел, символов и строк на экран:

`int pr intf`(<Форматная строка>, <Список выражений>);

Форматная строка – это строка, которая помимо символов содержит управляющие спецификации вида:

`%[-] [<Целое 1>] [.<Целое 2>] <Формат>`

«-» – выравнивание по левой границе,

<Целое 1> – ширина поля вывода;

<Целое 2> – количество цифр дробной части числа;

<Формат > – формат для ввода/вывода значения.

Основные форматы для ввода и вывода:

d – целое десятичное число;

u – целое десятичное число без знака;

o – целое число в восьмеричной системе счисления;

x – целое число в шестнадцатеричной системе счисления (% 4x – без гашения незначащих нулей);

f – вещественное число;

e – вещественное число в экспоненциальной форме;

c – символ;

p – ближний указатель (адрес);

s – символьная строка, вводит строку до первого пробела.

Кроме этого, форматная строка может содержать Esc-последовательности:

\n – переход на следующую строку;

\t – переход на следующую позицию табуляции;

\r – перевод каретки;

\f – перевод страницы;

\n hhh – вставка символа с кодом ANSI hhh (код задается в шестнадцатеричной системе счисления);

%% – печать знака % [10].

Пример

```
#include <stdio.h>
#include <conio.h>
int main()
{
int i,result;
float fp;
char c,s[81];
result = scanf_s("%d %f %c %s", &i, &fp, &c, 1, s, 80 );
printf("The number of fields input is %d\n", result );
printf("The contents are: %d %f %c %s \n", i, fp, c, s);
_getch();
}
```

Результат работы программы (полужирно выделены вводимые значения):

```
45 67.8
fghjk
The number of fields input is 4
The contents are: 45 67.800003 f ghjk
```

Ввод строк с клавиатуры:

`char* gets`(<Строковая переменная>);

// возвращает копию строки или `NULL`

Вывод строк на экран с переходом на следующую строку:

```
int puts(<Строковая константа или переменная>);
```

Ввод символов с клавиатуры:

```
int getchar();// возвращает символ или EOF
```

Вывод символов на экран:

```
int putchar(<Символьная переменная или константа>);
```

Пример. Программа определения корней квадратного уравнения $Ax^2+Bx+C=0$ при условии, что дискриминант неотрицателен.

```
#include <locale>
#include <stdio.h>
#include <conio.h>
#include <math.h>
// основная программа
int main( int argc, char* argv[])
{
setlocale(0,"russian");
float A,B,C,E,D,X1,X2;
puts("Введите A,B,C:");
scanf_s("%f %f %f",&A,&B,&C);
printf("A=%5.2f B=%5.2f C=%5.2f\n",A,B,C);
E=2*A;
D=sqrt(B*B-4*A*C);
X1=(-B+D)/E;
X2=(-B-D)/E;
printf("X1= %7.3f X2=%7.3f\n",X1,X2);
puts("Нажмите любую клавишу для завершения...");
_getch();
return 0;
}
```

В результате получим следующее (вводимая информация выделена полужирно):

```
Введите A,B,C:
1 -5 6
A= 1.00 B=-5.00 C= 6.00
X1= 3.000 X2= 2.000
Нажмите любую клавишу для завершения...
```

Вопросы для самопроверки

1. Что такое тип данных и каково его назначение?
2. Дайте определение литерала. Приведите примеры литералов?
3. Что такое переменная и как она определяется в C++?

4. Какие операции над данными определены в С++?

5. Какие логические поразрядные операции реализованы в С++ и особенности их выполнения?

6. Какие операции присваивания Вы знаете? В чем особенности их выполнения?

7. Что такое выражение? Дайте определение простого выражения.

8. Что такое приоритет и ассоциативность операций в С++?

9. Что такое форматная строка и как она используется при вводе и выводе данных?

4 Реализация базовых алгоритмических структур

4.1 Линейные программы. Разветвления

Действия, которые должна выполнять программа при реализации алгоритма, задают операторы языка – единые неделимые предложения, задающие законченное описание некоторого действия. Операторы программы могут быть простыми или составными. Простой оператор – это оператор, не содержащий другие операторы. Разделителем простых операторов служит точка с запятой. Специальным случаем простого оператора является **пустой оператор**, состоящий из единственного символа **';'**. **Составной оператор**, или блок, – это любая совокупность простых операторов, заключенная в фигурные скобки **{}**. Составной оператор идентичен простому оператору и может находиться в любом месте программы, где синтаксис языка допускает наличие оператора.

Программную реализацию линейной алгоритмической структуры (структуры «следование») называют линейной программой. Основой при написании линейных программ служат операторы вычисления выражения, присваивания и вызова функций. Кроме этого, могут использоваться невыполняемые операторы (например, описания переменных), операторы организации обработки данных (например, операторы ввода-вывода, вызова функций), составные операторы.

Оператор присваивания предназначен для изменения значений переменных, в том числе и для реализации вычислений «по формуле». Общий вид оператора присваивания:

```
<имя переменной> = <выражение>;
```

Выполнение оператора присваивания приводит к вычислению значения, определяемого выражением (выражение состоит из констант, переменных, обращений к функциям и знаков операций), и присваиванию этого значения переменной, идентифицируемой именем, стоящим слева от символа присваивания.

Присваивание можно рассматривать как операцию с двумя

операндами, причем ее исполнение зависит от типов операндов. Когда употребляется высказывание «x присвоить значение 7.8», то имеется в виду, что x – это ссылка на место в памяти, куда будет помещено значение 7.8. Для того чтобы определить ссылку, компилятор должен определить начальный адрес этого места и длину области памяти (она, естественно, соответствует длине типа данного).

Оператор присваивания неэквивалентен математическому знаку равенства. Запись `b=b+2` является корректным оператором C++ и означает: к значению переменной `b` добавить `2` и результат записать в переменную `b`. Фактически речь идет об увеличении значения `b` на 2.

Для корректного выполнения операции присваивания необходимо выполнение условия совместимости типов операндов. Тип переменной в левой части оператора присваивания должен быть не младше типа выражения в правой части. В противном случае требуется явное преобразование (приведение) типа правой части оператора к типу переменной в левой части.

Пример. Выполнить вычисление по формуле: $z = |x^3 + y^3|$

```
# include<stdio.h>
# include<math.h>
int main()
{
double x,y,z;
printf("Введите x и y");
scanf("%lf%lf",&x,&y);
z=fabs(x*x*x+y*y*y);
printf("z равно %lf",z);
return 0;
}
```

Для организации ветвления в программе в зависимости от значения некоторого логического выражения (условия развилки) предназначаются операторы выбора. В C++ включены два оператора выбора – условный оператор `if` и оператор-переключатель `switch`, которые могут записываться в полной и неполной формах. Оператор `switch` обычно называют оператором множественного выбора, так как он чаще всего используется при выборе среди большого числа вариантов. Если же выбор делается из двух – трех вариантов, то используется оператор `if`.

Условный оператор `if` имеет структуру:

– в полной форме:

```
if  (выражение) Оператор1;
        else    Оператор2;
```

– в неполной форме:

```
if  (выражение) Оператор1;
```

Семантика выполнения условного оператора такова: сначала

вычисляется выражение (его необходимо записывать в скобках). Заметим, что выражение может иметь булевский тип, в нем могут быть только арифметические операции или оно может представлять собой просто переменную (так как любое ненулевое значение соответствует логическому значению `true`, а нулевое – `false`). В зависимости от значения результата (`true, false`), выполняется Оператор1 (если результат равен `true`) или Оператор2 (если результат равен `false`).

В случае неполной формы оператора `if, Оператор1` выполняется только при значении результата `true`. При значении результата `false`, управление передается оператору, следующему непосредственно за оператором `if,` а `Оператор1` пропускается.

Обратите внимание на то, что перед ключевым словом `else` ставится символ «;». Появление (даже случайное) точки с запятой после зарезервированного слова `else` приведет к появлению в программе не синтаксической, а логической ошибки:

```
if (выражение) Оператор1 else ; Оператор2;
```

Например:

```
//внутри условия возможна запись оператора присваивания
int x=5, y=2;
if (x==y) cout << ”yes”;     //сравнение
     else cout << ”no”;      //выведет ”no”
if (x=y) cout << ”yes”;      //присваивание
     else cout << ”no”;      //выведет ”yes”
```

Последнее выражение равносильно следующей последовательности операторов:

```
x=y;
if (x) cout << ”yes”;
   else cout << ”no”;
```

В соответствии с требованиями синтаксиса языка, если в ветви оператора `if` требуется выполнить несколько операторов, то следует воспользоваться составным оператором. При использовании вложенных циклов может возникнуть неоднозначность в понимании того, к какой из вложенных конструкций `if` относится элемент `else.` И здесь следует помнить правило: ключевое слово `else` связывается с ближайшим стоящим перед ним ключевым словом `if.`

Чтобы `else` соответствовал внешнему оператору ветвления, необходимо добавить фигурные скобки:

```
if (a>0)
       {
         if (b>0) cout << ”Yes”;
        }
else   cout << ”No”;
```

При написании программы надо учитывать, что в операторе условия можно определить переменную и присвоить ей значение выражения, например: `if (int k=f(x)) a=k*k; else a=k-1;` область видимости переменной `k` ограничивается условным оператором.

Условная операция позволяет организовать выбор одного из двух выражений в зависимости от результата заданного логического выражения, т.е. реализовать простейший вариант ветвления. Это единственная операция в C++, которая выполняется над тремя операндами, т.е. является тернарной. Операция имеет следующий формат записи:

```
<Выражение 1>?<Выражение 2>:<Выражение 3>
```

При выполнении первым вычисляется значение выражения 1. Если оно истинно, т.е. результат не равен 0, то вычисляется выражение 2, которое и становится результатом. Если при вычислении выражения 1 получается 0, то вычисляется выражение 3, которое становится результатом.

Например:

```
x<0?-x:x;
printf(”%3d%c%”,a,i==n?’ ’:’\n’);
max = (d<=b) ? b : d;
printf(”%c”, (’A’ <=c &&  c <=’Z’) ? (’a’ + c – ’A’): c );
```

Пример. Поиск максимального из двух чисел.

```
#include <iostream>
#include <conio.h>
using namespace std;
int main () {
     double X,Y, Max;
     cout << "Input X <> Y\n";
     if(cin >> X >> Y)
     {
      if (X >= Y) Max = X;
      else   Max = Y;
      cout << "Max =" <<  Max << endl;
     }
      _getch();
     return 0;
}
```

Пример. Поиск максимального из трех чисел (вариант 1).

```
#include <iostream>
#include <conio.h>
using namespace std;
int main () {
      int X, Y, Z, Max;
```

```
      cout << "Input X <> Y <> Z\n";
      if (cin >> X >> Y >> Z)
      {
       if (X > Y)   Max = X;
            else    Max = Y;
       if (Z > Max)  Max = Z;
       cout << "Max =" <<  Max << endl;
      }
     _getch();
     return 0;
}
```

Пример. Поиск максимального из трех чисел (вариант 2).

```
#include <iostream>
#include <conio.h>
using namespace std;
int main () {
      int X, Y, Z, Max;
      cout << "Input X <> Y <> Z\n";
      if (cin >> X >> Y >> Z)
      {
       Max = X>Y ? X : Y;
       Max = Z > Max ? Z : Max;
       cout << "Max =" <<  Max << endl;
      }
     _getch();
     return 0 ;
}
```

Иногда алгоритм задачи содержит ряд альтернативных решений, причем некоторую переменную надо проверять отдельно для каждого значения, которое она может принимать. В зависимости от результатов этой проверки должны выполняться различные действия. Для принятия подобных решений в С++ имеется структура множественного выбора – **оператор `switch`.** Оператор `switch` производит сопоставление значения с множеством констант [11].

Оператор `switch` позволяет выполнить одно из нескольких действий в зависимости от значения выражения-переключателя, которое может иметь целый, символьный, перечисляемый тип.

Структура `switch`, состоящая из ряда меток `case` и необязательной метки `default` (умолчание), имеет следующий вид:

```
switch (выражение выбора)
{
  case значение 1:
```

```
            последовательность операторов 1;
           break;
  case значение 2:
            последовательность операторов 1;
           break;
……….
  case значение N:
            последовательность операторов N;
           break;
default: последовательность операторов;
}
```

При выполнении оператора `switch`, вычисляется выражение, записанное после `switch`. Полученное значение последовательно сравнивается с константами, которые записаны следом за `case`, константы в вариантах `case` должны быть различными. При первом же совпадении выполняются операторы, помеченные данной меткой. Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель.

Если значение выражения, записанного после `switch`, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой `default`. Метка `default` может отсутствовать. При отсутствии ветви `default`, если значение выражения-переключателя не совпадает ни с одним из значений константных выражений, управление передается оператору, стоящему непосредственно после фигурной скобки, замыкающей оператор `switch`.

В операторе `switch` разрешается для нескольких констант указывать одну и ту же последовательность операторов. Допускается вложенность операторов `switch`, когда в любой последовательности операторов используется другой оператор `switch`. Константы или значения константных выражений внутреннего и внешнего операторов `switch` могут совпадать. Они должны быть различными только на одном уровне вложенности. Например:

```
switch (n_day)
{
     case 1:
     case 2:
     case 3:
     case 4:
     case 5: puts("Go work!"); break;
     case 6: printf("%s","");
     case 7: puts("relax!");
```

```
}
char flag1, flag2; …..
switch (flag1)
{
     case ’A’:   switch(flag2)
          {
          case ’A’: cout << ”AA”; break;
          case ’B’: cout << ”BB”; break;
          }
     case ’X’: cout << ”X”; break;
}
```

Пример. Разработать программу, вычисляющую значения нескольких функций. Функция выбирается пользователем из предлагаемого списка по соответствующему коду, вводимому в ответ на запрос пользователем:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int kod,key; float x,y;
     puts("Введите x:"); scanf_s("%f",&x);
     printf("x=%6.3f\n",x);
     puts("Введите код:");
     puts("1 - y=sin(x)");
     puts("2 - y=cos(x)");
     puts("3 - y=exp(x)");
     scanf_s("%d",&kod);
     key=1;
     switch(kod)
     {
          case 1: y=sin(x); break;
          case 2: y=cos(x); break;
          case 3: y=exp(x); break;
          default: key=0;
     }
     if (key) printf("x= %5.2f y=%8.6f\n",x,y);
          else puts("Ошибка!");
     puts("Нажмите любую клавишу для завершения...");
     _getch();
```

```
    return 0;
}
```

4.2 Реализация в программе циклов

Операторы цикла задают многократное выполнение операторов тела цикла в зависимости от некоторого условия. Операторы цикла относятся к управляющим конструкциям всякого языка программирования.

В C++ определены три разных оператора цикла:

– цикл с предусловием `while`:

```
while (выражение-условие)
          тело_цикла;
```

– цикл с постусловием `do-while`:

```
do
          тело_цикла;
while (выражение-условие)
```

– параметрический цикл `for`:

```
for (Инициализация ; Выражение-условие ; Модификация)
тело цикла;
```

Оператор цикла с предусловием `while` – является самой универсальной управляющей конструкцией. С его помощью можно записать любое циклическое действие.

Выражение `while (выражение-условие)` называется заголовком цикла. При входе в цикл вычисляется условие. Если его значение, приведенное к логическому типу, равно `false`, то тело цикла не выполняется, т.е. происходит переход к выполнению следующего за циклом оператора программы. Если же значение условия, приведенное к логическому типу, равно `true`, то происходит вход в цикл и однократное выполнение операторов его тела. Как только достигнут конец тела цикла, управление снова передается на его заголовок, т.е. снова вычисляется значение выражения условие и выполняется его анализ.

Если значение условия все еще равно `true` (что зависит от изменения значений переменных цикла во время предыдущей итерации), то тело цикла выполняется еще один раз, и так далее. Как только очередное вычисление условия цикла дает значение `false`, работа цикла завершается.

Очевидно, что неверное задание выражения (отсутствие модификации переменных, влияющих на значение условия) может привести к бесконечному циклу (зацикливанию) [8].

Пример. Вычисление с помощью оператора `while` суммы чисел натурального ряда от 1 до 10:

```
#include <iostream>//вариант_1
```

```
#include <conio.h>
using namespace std;
const int N=10;
int main ()
{  int  i=1, summa=0;
      //определение и инициализация переменных
      // i является одновременно переменной цикла
    while (i<=N)
    { summa += i; //вычисление суммы
      i++;        //получение нового числа
    }
    cout << summa<< endl;
    _getch();
    return 0;
}

#include <iostream>//вариант_2
#include <conio.h>
using namespace std;
const int N=10;
int main ()
{  int  i=1, summa=0;
      //определение и инициализация переменных
      // i является одновременно переменной цикла
while (i<=N)
   summa += i++;
      //вычисление суммы и получение нового значения i
     cout << summa<< endl;
    _getch();
    return 0;
}
```

Чтобы получить верный результат, необходимо позаботиться об инициализации переменной, в которой накапливается результат: в примере это инициализация переменной `summa (summa=0);` при нахождении произведения натуральных чисел переменная результата инициализировалась бы единицей.

Пример. Вычислить сумму ряда при $x > 1$ с заданной точностью ε:

$S = 1 - 1/x + 1/x^2 - 1/x^3 + \ldots$

Рекуррентная формула определения очередного n+1-го члена ряда:

$$R_{n+1} = - R_n / x .$$

На каждом шаге итерации сумма ряда определяется по формуле:

$$S = S + R_n .$$

Условие выхода из цикла: $|R_n| < \varepsilon$.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main(int argc, char* argv[])
{
  setlocale(0,"russian");
  float s, r,x,eps;
  puts("Введите x, eps:"); scanf_s("%f  %f", &x, &eps);
  s=0; r=1; s+=r;
  while (fabs(r)>eps)
     {
     r=-r/x;  s+=r;
     }
  printf("Результат= %10.7f  r=%10.8f\n", s,r);
  puts("Нажмите любую клавишу для завершения...");
  _getch();
  return 0;
}
```

При выполнении программа выдает следующий результат:

```
Введите x, eps:
3 0.0001
Результат=  0.7499873  r=-0.00005081
Нажмите любую клавишу для завершения...
```

Оператор цикла с постусловием `do-while` – является не столь универсальным, как оператор `while`, поскольку его условие располагается в конце цикла, и операторы его тела будут, в отличие от цикла `while`, всегда выполняться как минимум один раз. Это означает, что оператор не годится для записи цикла, тело которого при определенном условии вообще не должно выполняться.

Оператор `do-while` требует использования составного оператора, когда в его теле необходимо записать более одного оператора.

Общий принцип работы цикла `do-while` такой же, как у цикла с предусловием, но в отличие от `while`, условие проверяется не перед выполнением тела цикла, а после него. Тело цикла `do-while` выполняется до тех пор, пока значение условия равно `true`.

Неверное задание выражения (отсутствие модификации переменных, влияющих на значение условия) может привести к бесконечному циклу (зацикливанию) [11].

Пример. Расчет суммы нечетных чисел от 0 до 10.

```
#include <iostream.h>
```

```
#include <conio.h>
void main()
{
  clrscr();
  int n=1,sum=0;
  do
   {
     sum +=  n;  //  sum = sum +n;
     cout<<"Сейчас n = " << n <<"\t sum = " <<sum <<"\n";
     n += 2;
   }
  while (n <= 10);
  cout << " \nОкончательный результат: \n";
  cout << " n =  " << n << "\t sum = " << sum << "\n";
  cout<<"\n\n";
  cout<<"\nНажмите любую клавишу ...";
getch();
}
```

Результаты работы программы:

```
Сейчас n = 1    sum = 1
Сейчас n = 3    sum = 4
Сейчас n = 5    sum = 9
Сейчас n = 7    sum = 16
Сейчас n = 9    sum = 25
Окончательный результат:
Сейчас n = 11  sum = 25
```

Оператор параметрического цикла `for`. Оператор реализует цикл, для которого известен диапазон и шаг параметра цикла:

`for (Инициализация; Выражение-условие; Модификации) тело цикла;`

`Инициализация` используется для определения и инициализации переменной цикла и выполняется один раз в начале выполнения цикла.

`Выражение-условие` определяет условие выполнения цикла: если его результат, приведенный к типу `bool`, равен `true`, то цикл выполняется.

`Модификации` выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую.

Если `тело цикла` содержит более одного оператора, последние должны заключаться в фигурные скобки.

Любая из частей оператора цикла `for` может быть опущена (но точки с запятой надо оставлять на своих местах). Параметрический цикл `for` реализован как цикл с предусловием, поэтому любой цикл `while`

может быть приведен к эквивалентному ему циклу `for`, и наоборот.

Способы изменения управляющей переменной в структуре оператора `for` покажем на следующих примерах:

– Изменение управляющей переменной от 1 до 100 с шагом 1
```
for (int i = 1; i<=100; i++).
```
– Изменение управляющей переменной от 100 до 1 с шагом –1
```
for (int i = 100; i >=1; i--).
```
– Изменение управляющей переменной от 7 до 77 с шагом 7
```
for (int i = 7; i <= 77; i += 7).
```
– Изменение управляющей переменной от 20 до 2 с шагом –2
```
for (int i = 20; i >= 2; i -= 2).
```
– Изменение управляющей переменной от 7 до 77 с шагом 7
```
for (int i = 7; i <= 77; i += 7).
```

Способы записи структуры оператора `for` покажем на следующих примерах:

– Все части цикла присутствуют
```
for (int i=0,float s=0;i<n;i++)s+=i;
```
– Отсутствует тело цикла
```
for(int i=0,float s=0;i<n;i++,s+=i);
```
– Отсутствует инициализирующее выражение и тело цикла
```
int i=0;float s=0;
for(;i<n;s+=i++);
```
– Отсутствуют инициализирующее и модифицирующие выражения
```
for(;i<n;)s+=i++;
```
– Бесконечный цикл, который ничего не делает
```
for(;;);
```

Пример. Найти сумму N натуральных чисел.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>int main(int argc, char* argv[])
{
  setlocale(0,"russian");
  int i,n,s;
  puts("Введите количество членов последовательности:");
  scanf_s("%d",&n);
  for (i=1,s=0;i<=n;i++) s+=i;
  printf("Сумма=%5d  при n=%3d\n",s,n);
  puts("Нажмите любую клавишу для завершения...");
  _getch();
  return 0;
}
```

Операторы `break и continue` изменяют поток управления. Когда

оператор `break` выполняется в структурах `while, for, do/while` или `switch`, происходит немедленный выход из структуры. Программа продолжает выполнение с первого оператора после структуры. Обычное назначение оператора `break` – досрочно прерывать цикл или пропустить оставшуюся часть структуры `switch`, например:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main()
{
  setlocale(0,"russian");
  int s=0,i,k;
  puts("Введите последовательность до 10 чисел.");
  for(i=0;i<10;i++)
  {
    scanf_s("%d",&k);
    if (k<0) break;
    // по отрицательному числу - выход из цикла
    s+=k;
  }
  printf("Сумма чисел = %d\n",s);
  puts("Нажмите любую клавишу для завершения...");
  _getch();
  return 0;
}
```

Оператор `continue` в структурах `while, for` или `do/while` вызывает пропуск оставшейся части тела структуры и начинается выполнение следующей итерации цикла, например:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main()
{
  setlocale(0,"russian");
  int s=0,i=0,k;
  puts("Введите последовательность до 10 чисел.");
  while(i<10)
  {
    scanf_s("%d",&k);
    if (k<0)  // при вводе отрицательного числа
    {
      puts("Будьте внимательны.");
```

```
        continue; // значение счетчика i и сумма не
меняются
      }
      i++; s+=k;
    }
    printf("Сумма чисел = %d\n",s);
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Программа суммирует 10 вводимых целых положительных чисел. При вводе отрицательного числа выводится предупреждающее сообщение, значение счетчика и сумма не меняются.

4.3 Вложенные циклы

Цикл, организованный в теле другого цикла, называется вложенным. В этом случае внутренний цикл полностью выполняется на каждой итерации внешнего цикла.

Пример программы, иллюстрирующей работу вложенного цикла: даны действительные числа $a_1,\ldots, a_{10}$. Вычислить $a_1 + a_2^2 + \ldots + a_{10}^{10}$.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <time.h>
int main()
{
  setlocale(0,"russian");
  int Sum, p;
  const int n = 10;
  int a[n];
  srand((unsigned)time(NULL));
  for (int i=0; i<n; i++) {
    a[i] = rand()%90+10;
    cout << a[i] << " ";
  }
  Sum = a[0];
  for (int i=1; i<n; i++)
    { p = a[i];
      for (int k=0; k<i; k++)
        p = p*a[i];
```

```
        Sum = Sum + p;
    }
    cout << “Sum= “ << Sum << endl;
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

При программировании вложенных циклов следует иметь в виду, что затраты процессорного времени на выполнение таких конструкций могут зависеть от порядка следования вложенных циклов. По возможности следует делать цикл с наибольшим числом повторений внутренним, а цикл с наименьшим числом повторений – внешним. Это происходит из-за того, что инициализация цикла, т.е. обработка процессором его заголовка с целью определения начального и конечного значения счетчика, а также шага приращения счетчика, требует времени, поэтому необходимо уменьшать количество таких операций. Цикл слева выполняется дольше, чем справа:

```
for (j=1; j<=100000; j++)      for (j=1; j<=1000; j++)
   for (k=1; k<=1000; k++)        for (k=1; k<=100000; k++)
                    a=1;                                a=1;
```

Особенно часто вложенные циклы используются в матричных задачах. Обработка матриц включает следующие типы алгоритмов:

– построение матриц;
– построчная обработка;
– обработка матрицы по столбцам;
– обработка всей матрицы;
– обработка части матрицы [7].

Построение матриц. Элементы матрицы можно задать по некоторому специальному правилу в зависимости от ее индексов. Например: получить целочисленную квадратную матрицу a порядка n, для которой a_{ij}= i+2j:

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    A[i][j] =i+2*j;
```

Матрицу можно построить, используя один или несколько одномерных массивов, либо на основе одной или нескольких определенных к этому времени матриц. Например, даны действительные числа a_1,..., a_{10}, b_1,..., b_{20}. Получить действительную матрицу с размера 20 × 10 для которой $c_{ij}=a_j/(1+|b_i|)$.

```
for (int i=0; i<20; i++)
  for (int j=0; j<10; j++)
    c[i][j]=a[j]/(1 + fabs(b[i]));
```

Например, дана действительная квадратная матрица a порядка n.

Получить две квадратные матрицы b и c, для которых

$$b_{ij}=\begin{cases} a_{ij} & при\ \ j \ge i \\ a_{ji} & при\ \ j < i \end{cases} \quad \text{и} \quad c_{ij}=\begin{cases} a_{ij} & при\ \ j < i \\ -a_{ij} & при\ \ j \ge i \end{cases}$$

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
     if(j<i)
       {b[i][j] = a[j][i];
        c[i][j] = a[i][j];}
     else
       {b[i][j] = a[i][j];
        c[i][j] = -a[i][j];}
```

В таких задачах необходимо установить, от чего и как зависят индексы элементов матрицы и, возможно, значения ее элементов.

Построчная обработка предполагает, что для каждой строки матрицы требуется найти некоторый параметр (сумму, количество элементов строки с некоторым условием, наибольший (наименьший) элементы, определенный элемент (например, 0)) и т.д.

В этом случае не обязательно надо анализировать все элементы строки. Внешний цикл строится по номеру строки, а в одном или нескольких внутренних циклах обрабатывается строка как одномерный массив. При этом полученные характеристики строк можно запоминать в одномерном массиве размерности n или выводить сразу по мере получения.

Например, дана целочисленная квадратная матрица порядка m. Подсчитать число отрицательных элементов каждой строки матрицы.

```
for (int i=0; i<m; i++) //цикл по строкам i
  {for (int k=0, j=0; j<n; j++) //цикл по столбцам j
    if (A[i][j] < 0) k++;//число отрицательных элементов k
   cout << i << "  " << k << endl;
}
```

Обнуления счетчика `k=0;` должно происходить каждый раз при входе во внутренний цикл `for`, так как для каждой строки количество отрицательных элементов необходимо начинать считать с нуля. Оператор вывода значения `k` должен располагаться внутри внешнего, но вне внутреннего цикла.

Обработка по столбцам. Аналогичные вычисления можно выполнять не для каждой строки, а для столбцов. Внешний цикл строят по номеру столбца. Во внутреннем цикле, изменяя первый «левый» индекс, обрабатывают столбец как одномерный массив. Например: дана действительная матрица размером n × m . Найти среднее арифметическое

каждого из столбцов.

```
for (int j=0; j<m; j++) //цикл по столбцам j
  { Sum =0;
     for (int i=0; i<n; i++) //цикл по строкам i
      Sum += A[i][j]; //сумма элементов  j-го столбца
     Sum = Sum/n;  // среднее арифметическое
cout << j << "  " << Sum << endl;
}
```

Обратите внимание, что как и в предыдущем случае, важно место оператора обнуления значения суммы `Sum = 0;` и оператора вывода результата.

Обработка всей матрицы. К такому типу относятся задачи, в которых выполняется анализ всей матрицы в целом.

В таких алгоритмах можно обрабатывать ее как по строкам, так и по столбцам. Но более правильно организовать внешний цикл по номеру строки, а внутренний – по номеру столбца (т.к. элементы матрицы располагаются по строкам). Пример. Все элементы с наибольшим значением в данной целочисленной квадратной матрице порядка 10 заменить нулями.

```
const int n=10;
int A[n][n];
int Max;
//…….
Max=A[0][0];
for (int i=0; i< n; i++)
        for(int  j=0; j< m; j++)
            if (A[i][j] > Max)
                        Max=A[i][j];
 for (int i=0; i< n; i++)
        for(int  j=0; j< m; j++)
            if (A[i][j] = Max)
                        A[i][j]=0;
```

Выполнение следующих заданий не требует привлечения вложенных циклов при работе с матрицами. Подобные не слишком частые ситуации возникают, как правило, тогда, когда обрабатываются или исследуются элементы, образующие «одномерную» часть матрицы: строку, столбец, диагональ и т.д. Например, дана действительная квадратная матрица порядка n, найти:

– сумму элементов главной и побочной диагоналей;

```
const int n=10;
double  Sum_main=0, Sum_secondary =0, A[n][n];
for (int i=0; i<n; i++)
    {Sum_main +=A[i][i];
```

```
        Sum_secondary +=A[i][n-1-i];
    }
```

– найти сумму элементов первого столбца;

```
const int n=10;
double  Sum = 0, A[n][n];
for (int i=0; i<n; i++)
     Sum +=A[i][1];
```

– для данного натурального m ($m \leq 2n$) найти сумму тех элементов матрицы, сумма индексов которых равна m.

```
const int n=10;
int m;
double  Sum = 0, A[n][n];
for (int i=0; i<=n, m-i>=0; i++)
     Sum +=A[i][m-i];
```

Вопросы для самопроверки

1. Какие виды условных операторов вы знаете?
2. В каких случаях в программе используется полный условный оператор? Как он оформляется? Как он работает (что происходит при его выполнении)?
3. В каких случаях в программе используется неполный условный оператор? Как он работает (что происходит при его выполнении)?
4. В каких случаях используются операторы цикла с условием?
5. Может ли тело оператора цикла с условием не выполниться ни разу?
6. В каких случаях используется оператор цикла с параметром? Как он оформляется? Как он работает (что происходит при его выполнении)?
7. Может ли тело оператора цикла с параметром не выполниться ни разу?
8. Чему равно количество повторений тела оператора цикла с параметром, если параметр цикла принимает:
   a. все целые значения от 1 до 10?
   b. все целые значения от a до b?
   c. все нечетные значения от 1 до 20?
   d. все значения от 10 до 100 с шагом 7?
9. Можно ли в теле цикла с параметром не использовать величину – параметр цикла?

5 Структурные типы данных

5.1 Массивы: одномерные массивы

В программировании часто возникают задачи, связанные с обработкой больших объемов данных. Для того, чтобы весь этот объем

данных хранить внутри программы, применяют массивы – простейшую разновидность структурированных типов данных.

Массив – именованная последовательность областей памяти, хранящих однотипные элементы (рис. 14). Каждая такая область памяти называется *элементом массива*. Массивы обладают размерностью (большей или равной единице), которой задается число элементов, содержащихся в них, а также измерением, что предполагает возможность описания в программе одно- и многомерных массивов. Количество элементов в массиве называется его *размером*.

Простейшим аналогом двумерного массива может быть таблица, а трехмерного – несколько таблиц одинакового размера. Математические объекты типа вектор и матрица – примеры аналогов (соответственно одно- и двумерных) массивов.

Тип элемента массива может быть одним из базовых (скалярных), типом другого массива, типом указателя, типом структуры или объединения.

Элементы массива в C++ нумеруется, начиная с нуля. У одномерных массивов после его имени указывается один индекс (порядковый номер), заключенный в прямоугольные скобки [ ], а у многомерных – несколько, каждый из которых заключается в [ ]. Последнее означает, что многомерный массив создается путем определения массива из элементов типа массив [4].

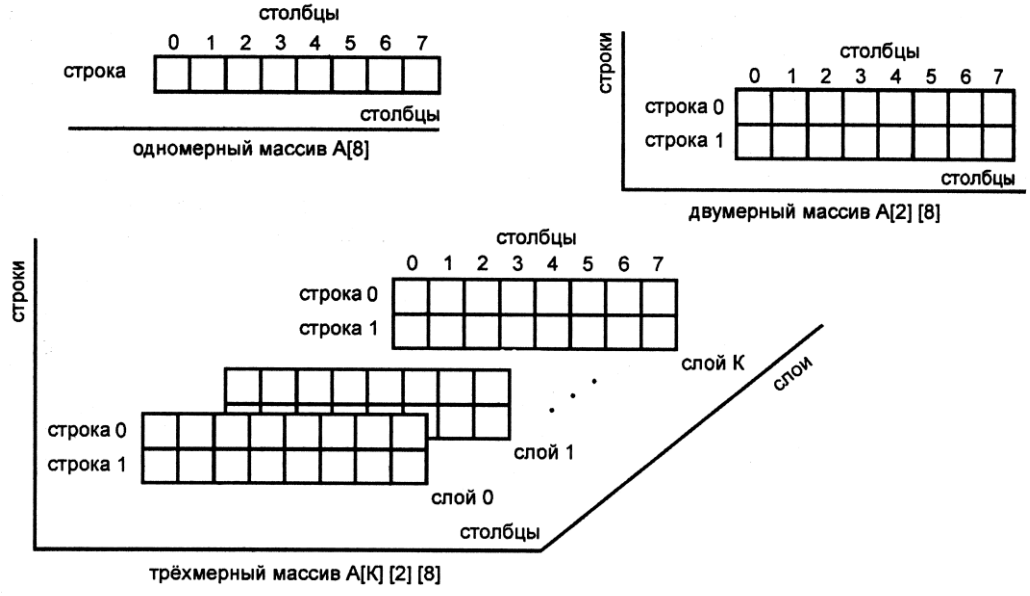


Рисунок 14 - Схематическое представление массивов

Все элементы массива имеют одно имя – имя массива и отличаются индексами – порядковыми номерами в массиве. В определении массива можно задать его размерность по каждому измерению. Допустимо явное

задание массива либо с помощью указателя (объекта, хранящего адрес начала области набора значений). Резервирование памяти для массива выполняется на этапе компиляции программы.

При объявлении массива компилятор выделяет для него последовательность ячеек памяти, для обращения к которым в программе применяется одно и то же имя. В то же время массив позволяет получить прямой доступ к своим отдельным элементам.

Синтаксис определения массива без дополнительных спецификаторов и модификаторов имеет два формата:

```
Тип ИмяМассива[ВыражениеТипаКонстанты];
```

или

```
Тип ИмяМассива[];
```

`ИмяМассива` – идентификатор массива.

`Тип` – тип элементов объявляемого массива. Элементами массива не могут быть функции, файлы и элементы типа `void`.

`ВыражениеТипаКонстанты` – задает количество элементов (размерность) массива. Выражение константного типа вычисляется на этапе компиляции. Данное константное выражение может быть опущено в случаях если:

– при объявлении массив инициализируется;

– массив объявлен как формальный параметр функции;

– массив объявлен как ссылка на массив, явно определенный в другом файле. Например:

```
int a[100]; //массив из 100 элементов целого типа
double d[14]; // массив из 14 элементов типа double
char s[]="Программирование"; //символьный массив
int t=5, k=8;
float wer[2*t+k];
 //массив из 2*t+k элементов вещественного типа
int sample[853];
/*массив из элементов sample[0], sample[1],
  sample[2],...,sample[852] типа int*/
```

равносильно объявлению

```
 const int N_max=853;
 int sample[N_max];
```

В языке C++ не производится проверки границ массивов: таким образом, исполнение кода не остановится при выходе за границы массива. Если переполнение массива происходит во время выполнения оператора присваивания, то лишние значения могут присвоиться другим переменным или включиться в текст программы. С другой стороны, можно объявить массив размером N и указать индекс элемента, выходящий за пределы N, что не приведет к появлению сообщений об ошибке, как на шаге

компиляции, так и на шаге выполнения, даже если это послужит причиной аварийного завершения программы.

В C++ одновременно с объявлением массива можно задать начальные значения всех элементов массива или только нескольких первых его компонент.

Например:

```
float t[5]={1.0, 4.3, 8.1, 3.0, 6.74};
char b[7]={'П','р','и','в','е','т'};
/*в данных примерах длину массива компилятор вычисляет по количеству начальных значений, перечисленных в фигурных скобках*/
int d[10]={1, 2, 3};
char a[10]="Привет";
/*в данных примерах определяется значение только заданных переменных d[0],d[1],d[2] и a[0],a[1],...,d[9], остальные элементы не инициализируются*/
```

Если в определении массива явно указан его размер, то количество начальных значений не может быть больше количества элементов в массиве.

Однако можно заставить C++ автоматически определить размеры этих массивов с помощью инициализации безразмерных массивов. Для этого в операторе инициализации не надо указывать размер массива, и C++ автоматически создаст массив, который сможет содержать присутствующий инициализатор. Компилятор C++ сам сформирует нужное значение по количеству инициализирующих данных:

```
extern long double c[]={7.89L,6.98L,0.5L,56.8L};
// 4 элемента
```

Метод инициализации безразмерных массивов не только менее трудоемок, но и позволяет заменить любое сообщение, без перерасчета размера соответствующего массива.

Адресация элементов массива осуществляется с помощью индексированного имени. Синтаксис обращения к элементу массива:

```
ИмяМассива[ВыражениеТипаКонстанты];
```

или

```
ИмяМассива[ЗначениеИндекса];
```

Обращаться к элементам массива можно также посредством механизма указателей.

Таким образом, чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс).

Например:

`a[0]` – индекс задается как константа,

`d[55]` – индекс задается как константа,

`s[i]` – индекс задается как переменная,

w[4*p] – индекс задается как выражение.

Следует помнить, что компилятор в процессе генерации кода задает начальный адрес массива, который в дальнейшем не может быть переопределен. Начальный адрес массива – это адрес первого элемента массива. Вообще в программе начальным адресом массива считается ИмяМассива либо &ИмяМассива[0]. Имя массива считается константой-указателем, ссылающимся на адрес начала массива.

С элементами объявленного массива можно выполнять все действия, допустимые для обычных переменных этого типа. Например, возможны операторы присваивания:

```
hours[4] = 34;
hours[5] = hours[4]/2;
```

или логические выражения с участием элементов массива:

```
if (number < 4 && hours[number] >= 40) { ... }
```

Присвоить значения набору элементов массива часто бывает удобно с помощью циклов for или while. Для массивов не допустима операция прямого присваивания.

Перед использованием значений элементов массива их необходимо сформировать. Задать значения элементам массива можно на этапе объявления, т.е. выполнить *инициализацию*, а также непосредственным выполнением присваивания.

В большинстве задач с использованием массивов целесообразно выполнять *генерацию* массивов – автоматическое формирование значений элементов.

Для использования функции генерации случайных чисел необходимо подключить библиотеку <time.h>.

Для написания кода генерации массива случайными целыми числами используется:

1. Функция srand(). Синтаксис:

```
void srand(unsigned seed);
```

– функция устанавливает свой аргумент как основу (seed) для новой последовательности псевдослучайных целых чисел, возвращаемых функцией rand(). Сформированную последовательность можно воспроизвести. Для этого необходимо вызвать srand() с соответствующей величиной seed.

Для использования данной функции необходимо подключить библиотечный файл <stdlib.h>.

2. Функция rand(). Синтаксис:

```
int rand(void);
```

– функция возвращает псевдослучайное число в диапазоне от нуля до RAND_MAX. Для использования данной функции необходимо подключить библиотечный файл <stdlib.h>.

3. Константа RAND_MAX определяет максимальное значение случайного числа, которое может быть возвращено функцией rand(). Значение RAND_MAX – это максимальное положительное целое число.

4. Часто в задачах требуется выполнить генерацию массива на произвольном промежутке [a,b). Для этого используются следующие выражения:

```
//генерация случайных целых чисел на [a,b)
   x[i]=rand()%(b-a)+a;
//генерация случайных вещественных чисел на [a,b)
   y[i]= rand()*1.0/(RAND_MAX)*(b-a)+a;
```

Пример .

```
/*Описание функции генерации массива случайными вещественными числами на[a,b)*/
void gen(int k,int a, int b, float x[max]){
  int i;
  srand(time(NULL)*1000);
  //устанавливает начальную точку генерации случайных чисел
  for (i=0;i<k;i++){
   x[i]=(rand()*1.0/(RAND_MAX)*(b-a)+a);
   //функция генерации случайных чисел на [a,b)
   }
}
```

5.2 Массивы: двумерные массивы

Двумерные массивы, являющиеся упорядоченными однотипными объектами, можно отождествлять с прямоугольной матрицей. Двумерные массивы состоят из строк и столбцов.

Синтаксис определения массива без дополнительных спецификаторов и модификаторов имеет два формата:

```
Тип ИмяМассива[ВыражениеТипаКонстанты][ВыражениеТипаКонстанты];
```

или

```
Тип ИмяМассива[][];
```

ИмяМассива – идентификатор массива.

Тип – тип элементов объявляемого массива. Элементами массива не могут быть функции, файлы и элементы типа void.

ВыражениеТипаКонстанты – задает количество элементов (размерность) массива. Выражение константного типа вычисляется на этапе компиляции. Данное константное выражение может быть опущено в случаях, если:

– при объявлении массив инициализируется;

– массив объявлен как формальный параметр функции;

– массив объявлен как ссылка на массив, явно определенный в другом файле. Например:

```
int a[100][50];//массив из 100×50 элементов целого типа
double d[4][10];// массив из 4×10 элементов типа double
int t=5, k=8;
float wer[2*t+k][2*t+k];
//массив из (2*t+k)×(2*t+k) элементов вещественного типа
int sample[853][157];
//массив из 853 строк  и 157 столбцов, элементы типа int
```

равносильно объявлению

```
const int N_max=853,
      int M_max=157;
int sample[N_max][M_max];
```

равносильно объявлению

```
#define N_max 853
#define M_max 157
...
int sample[N_max][M_max];
```

Двумерные массивы располагаются в памяти в порядке быстрого изменения последнего индекса. Двумерные массивы инициализируются так же, как и одномерные. Например:

```
int w[3][3]={
//инициализируется целочисленный массив размерностью 3×3
  {2, 3, 4} //1-я строка
  {3, 4, 8} //2-я строка
  {1, 0, 9} //3-я строка
 };
float w[2][3]={
//инициализируется вещественный массив размерностью 2×3
  {2.1, 3.4, 4.5} //1-я строка
  {5.0, 6.4, 3.9} //2-я строка
 };
```

равносильно инициализации

```
float w[][3]={
  {2.1, 3.4, 4.5} //1-я строка
  {5.0, 6.4, 3.9} //2-я строка
 };
```

Последовательности, выделенные в фигурные скобки, соответствуют строкам массива.

Обращение к элементам двумерного массива осуществляется так же,

как и к элементам одномерного.

```
ИмяМассива[ВыражениеТипаКонстанты][ВыражениеТипаКонстанты];
```

или

```
ИмяМассива[ЗначениеИндекса][ЗначениеИндекса];
```

Например:

`a[0][0]` – индекс задается как константа,

`d[55][5]` – индекс задается как константа,

`s[i][j]` – индекс задается как переменная,

`w[4*p][3+t]` – индекс задается как выражение.

Принцип генерации двумерных массивов такой же, как и одномерных.

```
//Описание функции генерации массива
void gen(int str,int slb, int a, int b,
                                    int m[max_x][max_y]){
  int i,j;
  srand(time(NULL)*1000);
//устанавливает начальную точку генерации случайных чисел
  for (i=0;i<str;i++)
    for (j=0;j<slb;j++)
      m[i][j]=rand()%(b-a)+a);
      //функция генерации случайных чисел на [a,b)
}
```

Двумерные массивы выводятся на экран так же, как и одномерные. Для наглядности вывода целесообразно разделять элементы массива на строки и столбцы.

```
//Описание функции вывода массива
void out (int str,int slb, int m[max_x][max_y]){
  int i,j;
  for (i=0;i<str;i++)  {
    for (j=0;j<slb;j++)
      printf("%4d",m[i][j]);
      printf("\n");
    }
}
```

В программировании двумерные массивы называют также *матрицами*. В задачах на обработку двумерных массивов следует определить способ просмотра массива (по строкам, по столбцам, вдоль диагоналей и т.д.). При этом, как правило, используют кратные циклы, в которых один изменяющийся параметр соответствует пробегу по индексам строк, другой – колонок. При выборе пути обхода матрицы следует учитывать, что параметр внешнего цикла меняется медленнее, чем параметры вложенных в него циклов [10].

Пример. Найти максимальный элемент главной диагонали двумерного целочисленного массива размерностью *n*×*n*, заданного случайными числами на промежутке [-100; 100).

```
/*Описание функции поиска максимального элемента
главной диагонали*/
int maxi(int str,int slb, int m[max_x][max_y]){
  int i,j,e_max=m[0][0];
  for(i=0;i<str;i++)
    for(j=0;j<slb;j++)
      if((i==j)&&(m[i][j]>e_max))
        e_max=m[i][j];
  return e_max;
}
```

Пример. Найти сумму элементов столбца двумерного массива, номер которого задается с клавиатуры.

```
/*Описание функции суммирования элементов заданного
номера столбца матрицы*/
int summa(int str, int slb, int nom,int
m[max_x][max_y]){
  int i,j,sum=0;
  for(i=0;i<str;i++)
    for(j=0;j<slb;j++)
      if(j==nom-1) sum+=m[i][j];
  return sum;
}
```

Пример. Дан двумерный вещественный массив размерностью *n*×*n*, заданный случайными числами на промежутке [-100; 100). Замените все элементы выше главной диагонали на 1.1 и ниже ее на 0.0.

```
//Описание функции замены
void zamena (int str,int slb, double m[max_x][max_y]) {
  int i,j;
  for (i=0;i<str;i++)
    for (j=0;j<slb;j++) {
      if (i>j) m[i][j]=0.0;
      if (i<j) m[i][j]=1.1;
    }
}
```

5.3 Строки

Для представления текстовой информации в языке C++ используются символы (константы), символьные переменные и строки

(строковые константы), для которых в языке С++ не введено отдельного типа в отличие от некоторых других языков программирования [7].

Строка – это последовательность символов, заключенная в двойные кавычки (" ").

Размещая строку в памяти, транслятор автоматически добавляет в ее конце символ '\0' (нулевой символ или нулевой байт, который является признаком конца строки). В записи строки может быть и один символ: ”A” (заключен в двойные кавычки), однако, в отличие от символьной константы 'A' (используются апострофы), длина строки ”A” равна 2 байтам.

В языке С++ *строка* – это пронумерованная последовательность символов (массив символов), она всегда имеет тип char[]. Все символы строки нумеруются, начиная с нуля. Символ конца строки также нумеруется – ему соответствует наибольший из номеров. Таким образом, строка считывается значением типа «массив символов». Количество элементов в таком массиве на 1 больше, чем изображение соответствующей строки, так как в конец строки добавлен нулевой символ '\0'.

Символьная строка в программном коде может располагаться на нескольких строках. Для переноса используется символ '\' с последующим нажатием клавиши ввод. Символ '\' игнорируется компилятором, и следующая строка считается продолжением предыдущей.

Присвоить значение строке с помощью оператора присваивания нельзя, так как для массивов не определена операция прямого присваивания. Поместить строку в символьный массив можно либо при вводе, либо с помощью инициализации:

```
char s1[] = ”ABCDEF”; //инициализация строки
char s2[]={'A','B','C','D','E','F','\0'};
//инициализация строки
```

Операция вычисления размера (в байтах) sizeof действует для объектов символьного типа и строк. Например:

```
// Определение размера строк
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]) {
  char s1[10]="string1";
  int k=sizeof(s1);
  cout<<s1<<"\t"<<k<<"\n";
  char s2[]="string2";
  k=sizeof(s2);
  cout<<s2<<"\t"<<k<<"\n";
  char s3[]={'s','t','r','i','n','g','3','\0'};
```

```
  /*окончание строки '\0' следует соблюдать, формируя
    в программах строки из отдельных символов*/
  k=sizeof(s3);
  cout<<s3<<"\t"<<k<<"\n";
  char *s4="string4";
  //указатель на строку, ее нельзя изменить
  k=sizeof(s4);
  cout<<s4<<"\t"<<k<<"\n";
  system("pause");
  return 0;
}
```

Результат выполнения программы:

`string1 10` – выделено 10 байтов, в том числе под `'\0'`

`string2 8` – выделено 8 байтов (7 + 1 байт под `'\0'`)

`string3 8` – выделено 8 байтов (7 + 1 байт под `'\0'`)

`string4 4` – размер указателя 4 байта

Ввод-вывод стандартного текстового (символьного) потока:

`gets(s)` – функция, которая считывает строку `s` из стандартного потока до появления символа `'\n'`, сам символ `'\n'` в строку не заносится.

`puts(s)` – функция, которая записывает строку в стандартный поток, добавляя в конец строки символ `'\n'`, в случае удачного завершения возвращает значение больше или равное `0` и отрицательное значение в случае ошибки (`EOF = -1`). Например:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]){
  char s[20];
  gets(s);
  puts(s);
  system("pause");
  return 0;
}
```

Результат выполнения программы: при вводе строки `"123 456 789"` чтение данных осуществляется побайтно до символа `'\n'`, то есть в `s` занесется строка `"123 456 789\0"` (управляющая последовательность `'\0'` на экран не выводится, а является признаком конца строки). При выводе строки функция `puts` возвращает в конце строки дополнительно один символ `'\n'`, следовательно, будет выведена строка `"123 456 789\n"` (управляющая последовательность `'\n'` на экран не выводится, а осуществляет перевод курсора на новую строку).

Пример. Вычислите длину введенной строки.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]){
  char st[100];
  int i=0;
  puts("Введите строку:");
  gets(st);
  while(st[i++]);
  printf("Длина введенной строки = %i\n",i-1);
  system("pause");
  return 0;
}
```

Стандартные потоки ввода-вывода символьных данных и строк

`cin` – оператор, который определяет стандартные потоки ввода данных.

`cout` – оператор, который определяет стандартные потоки вывода данных.

`<<` – операция записи данных в поток;

`>>` – операция чтения данных из потока. Например:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]){
  char s[20];
  cin>>s; //ввод строки из стандартного потока
  cout<<s; //вывод строки в стандартный поток
  system("pause");
  return 0;
}
```

Результат выполнения программы: при вводе строки `"123 456 789"` чтение данных осуществляется побайтно до первого пробела, то есть в `s` занесется только первое слово строки `"123\0"`, следовательно, выведется: `"123"`.

Пример. Введите слово и замените в нем все вхождения заглавной латинской `'A'` на малую латинскую `'a'`. Выведите слово после редактирования.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]){
```

```
  char st[80];
  int i;
  cout << "\nВведите слово: ";
  cin >> st;
  for(i=0;st[i]!='\0';i++)
     if (st[i]=='A') st[i]='a';
  cout << "\nСлово после редактирования: "<< st;
  system("pause");
  return 0;
}
```

Пример. Записать введенную строку символов в обратном порядке.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]){
  char st[80];
  char temp;
  int i,len=0;
  printf("\nВведите строку > ");
  scanf("%s",st);
  while (st[len++]); //вычисление длины строки
  len-=2;
//поправка на символ конца строки и нумерацию с нуля
  for(i=0;i<len;i++,len--){
    temp=st[i]; //обмен символов
    st[i]=st[len];
    st[len]=temp;
  }
  printf("\nПолученная строка  > %s",st);
  system("pause");
  return 0;
}
```

5.4 Структуры

В языке C++ возможно формирование производных (пользовательских) типов данных прежде всего на основе массивов, структур и объединений. Комбинирование этих типов данных позволяет программно моделировать достаточно сложные объекты реальности.

Агрегатным типом данных называется тип, конструируемый из элементов независимых (возможно различных) типов.

Структура – это составной объект, в который входят элементы

любых типов, за исключением функций. В отличие от массива, который является однородным объектом (все элементы относятся к одному типу данных), структура может быть неоднородной. Таким образом, структура – это тип данных, сформированный из объектов однородных либо разнообразных типов данных [11].

Структуру можно представить себе как запись, состоящую из нескольких полей или элементов. Структуры обеспечивают удобный способ организации связанных по смыслу переменных. Структуры являются одновременно агрегатным и производным типом данных.

В некоторых языках программирования, в частности в Pascal, структуры называются записями. Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Традиционный пример структуры – строка платежной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и т.д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент (фамилии, имени и отчества); аналогично адрес и даже зарплата. Другой пример – из области графики: точка на плоскости есть пара вещественных координат, шар в пространстве моделируется четырьмя вещественными числами и т. д.

Использование в программе структуры предполагает вначале определение (объявление) структуры-шаблона (типа структуры) и его структурного объекта (структурной переменной). Возможны их раздельное и совместное определения. Для шаблона структуры (с именем или без него) компилятор не выделяет в памяти компьютера место под структуру, а лишь фиксирует правила, необходимые для формирования структурного объекта (структурной переменной).

В момент определения структурного объекта компилятор выделяет место в памяти, где размещаются все компоненты структуры в соответствии с заданным шаблоном [11].

Ключевое слово **`struct`** сообщает компилятору об объявлении структуры. Допустимы три основные формы объявления структур.

1) С поименованным шаблоном:

```
struct ИмяСтруктурногоТипа {ОпределенияЭлементов};
```

где `ИмяСтруктурногоТипа` – идентификатор типа структуры. Следует обратить внимание на точку с запятой, которой заканчивается определяемая структура;

`ОпределенияЭлементов` – список определений типизированных компонентов (полей), из которых образуется шаблон структуры. Он в общем случае представляется так:

```
Тип1 Компонент11 [,Компонент12,...];
```

```
.....................................
ТипК КомпонентК1 [,КомпонентК2,...];
```

Здесь допустимы компоненты различных типов. Все имена компонентов уникальны. Шаблон имеет видимость. Если он объявлен внутри блока, то это локальный шаблон, видимый только внутри него.

Например:

```
struct STUDENT {
  char name[50]; //Ф.И.О
  int passw; //шифр зачетной книжки
 };
```

При указанной форме объявления структуры структурные объекты определяются ниже в виде отдельной строки:

```
ИмяСтруктурногоТипа СписокСтруктур;
```

где `СписокСтруктур` – список идентификаторов, соответствующих именам структурных объектов.

Структурные объекты можно определить так:

```
STUDENT
  group01, group02, //структурные объекты
  *univers, //указатель на структурный объект
  fam[100]; //массив структурных объектов
```

При определении структурного объекта (структурной переменной) `СтруктурныйОбъект` допустима инициализация его компонентов (полей):

```
struct ИмяСтруктурногоТипа СтруктурныйОбъект =
{Инициализатор1,
 Инициализатор2,
 ..............
 ИнициализаторК
 };
```

Для вышеприведённого примера возможно следующее:

```
struct STUDENT group01 = {
  "Сидоров ",
  6374268
 };
```

2) С совмещением определения структуры и структурных объектов:

```
struct ИмяСтруктурногоТипа {ОпределенияЭлементов}
       СписокСтруктур;
```

Структурный объект при такой схеме определения может быть также инициализирован. Например, структура типа `STUDENT` с элементами ФИО и номером зачётной книжки:

```
struct STUDENT {
   char name[50];
   int passw;
```

```
  } *univers,//указатель на структурный объект
fam[100],//массив структурных объектов
group01 = {"Сидоров А.И.", 26276};
//инициализированный объект
```

При рассматриваемом подходе ИмяСтруктурногоТипа (STUDENT) можно опустить. В пределах программы допустим один непоименованный тип структуры.

3) С использованием оператора typedef:

```
typedef struct [ИмяСтруктурногоТипа]
               {ОпределенияЭлементов}
               ОбозначениеСтруктурногоТипа;
```

где ОбозначениеСтруктурногоТипа – синонимы типа структуры. Это упрощает программу при определении структурных объектов. Кроме того, в упрощённом определении имя типа структуры можно опустить.

Например:

```
typedef struct {
  char name[50];
  int passw;
 } NSTUstudent;
//NSTUstudent - псевдоним типа структуры
```

Определение структурного объекта для данного случая можно организовать так:

```
NSTUstudent *pt, //указатель на структуру
groupStudent; //структурная переменная
```

Здесь также допускается инициализация определяемых структурных объектов.

Можно определить структуру с использованием макроопределения:

```
#define Идентификатор struct ИмяТипа
```

Далее следует

```
Идентификатор {ОпределенияЭлементов};
```

Например:

```
#define COMPLEX struct R7
COMPLEX {
         float real;
         float imag;
        };
```

Инициализация структуры заключается в присваивании начальных значений элементам структуры. Структуры могут быть проинициализированы при их объявлении.

Инициализирующая запись – это заключенный в фигурные скобки список, элементы которого разделяются запятыми и являются константами. Любые неинициализированные элементы внешних или статических структур по умолчанию равны 0. Значения

неинициализированных элементов автоматических структур не определены.

Для инициализации структур значения ее полей перечисляются в фигурных скобках. Например:

Первый способ

```
struct Student {
                char name[20];
                int kurs;
                float rating;
               };
Student s={"Королев",1,3.5};
```

Второй способ

```
struct {
        char name[20];
        char title[30];
        float rate;
       } employee={"Петров", "программист",58000};
```

В соответствии с синтаксисом языка компонентами структур могут быть данные любых типов, за исключением функций и структур того же типа, что и определяемый тип. Элементом структуры может быть структура, тип которой уже определен.

Например:

```
stuct mix {int N; double *d;}
struct hole {
             struct mix exit;
             float b;
            }
```

Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование.

```
Student t=s;
```

Информация, содержащаяся в одной структуре, может быть присвоена другой структуре того же типа с помощью одиночного оператора присваивания, то есть не нужно присваивать значение каждого члена по отдельности. Например:

```
#include <stdio.h>
void main(){
struct {
        int a;
        int b;
       } x, y;
  x.a = 10;
  x.b = 20;
  y = x;
```

```
   printf("Содержимое y: %d %d", y.a, y.b);
 }
```

С помощью операций прямого и косвенного выбора (соответственно символы точка '.' и '->') организуется доступ к элементам структур. Первая операция используется со структурными объектами, а вторая – при наличии указателя на структурный объект.

Операция прямого доступа к элементам структуры ('.') – результатом является значение элемента структуры.

Синтаксис операции прямого доступа к элементам структуры:

```
ИмяСтруктуры.ИмяЭлементаСтруктуры
```

Данная операция используется для доступа к элементу структуры с тем, чтобы присвоить ему значение, напечатать его, использовать его значение в арифметической операции и т.д. Эта операция является первичной и находится в самой верхней строке таблицы приоритетов операций языка C++.

Именем структуры можно пользоваться сразу же после его появления в программе. Например:

`employee.name` – массив типа `char`, содержащий значение «Петров»

`employee.rate` – переменная типа `float` со значением `58000.0`.

Если в программе определены две структуры с одинаковыми типами элементов, то для компилятора типы таких структур различны. В этом случае взаимное присваивание можно выполнять лишь на уровне элементов структур. Если объекты принадлежат к одному структурному типу, то возможно косвенное присваивание структур путём присваивания значений одного структурного объекта другому.

Например:

```
struct A {
          int j;
          char titl[10];
          char x;
          } Aa, Ab = {128,"Мир",'Q'};
```

Здесь допустимы операторы присваивания вида `Aa=Ab;`. В этом случае элементы структурного объекта `Aa` будут иметь значения, совпадающие со значениями соответствующих элементов объекта `Ab`.

Пример.

```
//Инициализация структуры и вывод значений ее элементов
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]) {
  struct goods {
```

```
  char* name;
  long price;
  float percent;
  int vol;
  char date[9];
 };
  struct goods coat=
              {"пиджак
         черный",4000,7.5,220,"12.01.09"};
  printf("\n Товар на складе:");
  printf("\n Наименование: %s.", coat.name);
  printf("\n Оптовая цена: %ld руб.", coat.price);
  printf("\n Наценка: %3.1f %%.", coat.percent);
  printf("\n Цена товара: %ld руб.",
          (long)(coat.price*(1.0+coat.percent/100)));
  printf("\n Объем партии: %d штук.", coat.vol);
  printf("\n Дата поставки: %s.",coat.date);
  system("pause");
  return 0;
}
```

Пример. Сложение комплексных чисел.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
typedef  struct {
  double real;
  double imag;
 } complex;
int _tmain(int argc, _TCHAR* argv[]) {
  complex x,y,z;
  printf("\n Введите два комплексных числа:");
  printf("\n Вещественная часть:");
  scanf("%lf", &x.real);
  printf("\n Мнимая часть:");
  scanf("%lf", &x.imag);
  printf("\n Вещественная часть:");
  scanf("%lf", &y.real);
  printf("\n Мнимая часть:");
  scanf("%lf", &y.imag);
  z.real=x.real+y.real;
  z.imag=x.imag+y.imag;
  printf("\n Результат: z.real=%f z.imag=%f", z.real, z.imag);
```

```
  system("pause");
  return 0;
}
```

Массив структур – это массив, каждый элемент которого является структурой. В памяти элементы массива структур размещаются последовательно. Массивы структур широко используются для структурной организации данных в прикладных программах и системном программном обеспечении. Из элементов структурного типа можно организовать массивы также как из элементов стандартного типа.

Для объявления массива структур следует сначала определить структуру, а затем объявить массив переменных данного типа. Как и массивы переменных, массивы структур индексируются с нуля. Например:

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[]) {
  struct Student {
   char name[30];
   char group[10];
   float rating;
  };
  Student mas[35]; //массив структур
  int i;
  //ввод значений массива
  for(int i=0;i<35;i++){
    cout << "\nВведите имя: ";cin >> mas[i].name;
    cout << "\nВведите группу: ";cin >> mas[i].group;
    cout << "\nВведите рейтинг: ";cin >> mas[i].rating;
  }
  //вывод студентов, у которых рейтинг меньше 3
  cout << "Рейтинг < 3: ";
  for(i=0; i<35; i++)
    if(mas[i].rating<3)
      cout << "\n" << mas[i].name;
  system("pause");
  return 0;
}
```

Пример. Программа определяет и печатает название самой высокой вершины из списка.

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

```
struct peak {
  char name[15];//название вершины
  int height; //высота вершины
 } list[30]; //массив структурного типа peak
int N;
void InputDate();
void PrintMaxPic();
int _tmain(int argc, _TCHAR* argv[]) {
  InputDate();
  printf("\n");
  PrintMaxPic();
  system("pause");
  return 0;
}
void InputDate() {
int i;
  printf("Введите количество вершин: ");
  scanf("%d",&N);
  printf("\n");
  for (i=0; i<N; i++) {
    printf("Название: ");
    scanf("%s",&list[i].name);
    printf("Вершина: ");
    scanf("%d",&list[i].height);
  }
}
void PrintMaxPic(){
int i,max=list[0].height,num=0;
  for (i=1; i<N; i++)
    if (list[i].height>max) {
      max=list[i].height;
      num=i;
    }
  printf("Самая высокая вершина - %s, она равна - %d",
          list[num].name,max);
}
```

5.5 Объединения

Объединение – это частный случай структуры.

Объединение подобно структуре, однако в каждый момент времени может использоваться (или, другими словами, быть ответным) только один

из элементов объединения.

Объединение (смеси) – объект, который в каждый момент времени содержит один из нескольких элементов различных типов.

Объединение является структурой данных, члены которой расположены по одному и тому же адресу. Поэтому размер объединения равен размеру его наибольшего члена. В любой момент времени объединение хранит значение только одного из членов.

Объявление объединения определяет имя переменной объединения и специфицирует множество переменных, называемых *элементами объединения*, которые могут быть различных типов. Переменная с типом объединения запоминает любую отдельную величину, определяемую набором элементов объединения [5].

Все компоненты объявления структур, такие как шаблоны, имена типов, имена элементов и т.д. применимы и при объявлении объединений. Единственное отличие состоит в том, что при объявлении объединения вместо ключевого слова `struct` используется `union`. Синтаксис:

```
union [ИмяОбъединения] {
                          ОпределенияЭлементов;
                         } ОбозначениеОбъединения;
```

где `union` – спецификатор типа;

`ИмяОбъединения` – идентификатор;

`ОпределенияЭлементов` – совокупность описаний объектов, каждый из которых служит прототипом одного из элементов объединений.

Например:

```
union {
       char hh[2];
       int ii;
       } CC;
```

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Как и для структурных типов, с помощью `typedef` можно вводить обозначения объединяющих типов. Синтаксис:

```
typedef union [ИмяОбъединения]
        {
         ОпределенияЭлементов;
        } ОбозначениеОбъединения;
```

Например:

```
typedef union uni {
                   double d;
                   int i[4];
```

```
    char ch[8];
} u_name;
```

На основе такого определения типа можно вводить конкретные объединения двумя способами:

```
union uni a, b;
u_name x, y;
```

Объединение применяется для следующих целей:

- инициализации используемого объекта памяти, если в каждый момент времени только один объект из многих является активным;
- интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Для обращения к элементу объединения используются те же конструкции, что и для обращения к элементу структуры:

```
ИмяОбъединения.ИмяЭлемента
(* УказательНаОбъединение).ИмяЭлемента
УказательНаОбъединение->ИмяЭлемента
```

Например:

```
CC.hh
(*pin).mas
pin->mas
```

Чтобы работать с объединением напрямую, надо использовать оператор «точка» (`.`). Если к переменной объединения обращение происходит с помощью указателя, надо использовать оператор «стрелка» (`->`).

Память, которая соответствует переменной типа объединение, определяется величиной для размещения любого отдельного элемента объединения.

В отличие от структуры, объединение может в любой момент времени содержать только один из своих элементов. Объединение позволяет использовать одну область памяти для хранения различных видов данных в разные моменты времени. Фактически, объединение – это структура, в которой все поля начинаются со смещением 0, таким образом, поля накладываются друг на друга.

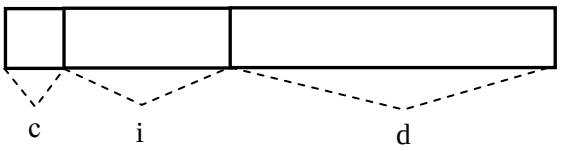
Все поля объединения располагаются по одному и тому же адресу. Размер объединения равен наибольшей из длин его полей. То есть память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память. Объединения применяются для экономии памяти, если известно, что другие поля не потребуются.

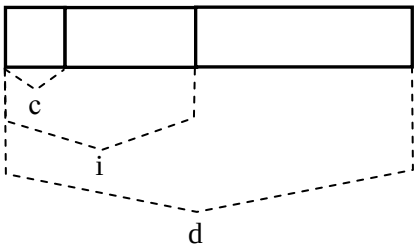
Когда используется наименьший элемент объединения, то переменная типа объединения может содержать неиспользованное пространство. Все элементы объединения запоминаются в одном и том же

пространстве памяти переменной, начиная с одного и того же адреса. Запомненные значения затираются каждый раз, когда присваивается значение очередного элемента объединения [9]. Например:

```
struct s_tag {
              char c;
              int i;
              double d;
             } s_item;
```

```
union u_tag {
              char c;
              int i;
              double d;
             } u_item;
```





Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа. Например:

```
union {
       char fio[30];
       char adres[80];
       int vozrast;
       int telefon;
      } inform;
union {
       int ax;
       char al[2];
      } ua;
```

При использовании объекта `inform` типа `union` можно обрабатывать только тот элемент, который получил значение, т.е. после присвоения значения элементу `inform.fio`, не имеет смысла обращаться к другим элементам. Объединение `ua` позволяет получить отдельный доступ к младшему `ua.al[0]` и к старшему `ua.al[1]` байтам числа `ua.ax`.

Допустимы массивы объединений и указатели на объединения. Объединения могут передаваться функции как параметры и возвращаться функцией.

Операции, применимые к структурам, аналогичны и для объединений, т.е. законны присваивание объединения и копирование его как единого целого, взятие адреса от объединения и доступ к отдельным его элементам.

Объединения часто включаются в структуры, один из элементов которых является ключом, указывающим тип хранимого в памяти

элемента объединения.

Например:

```
//содержит информацию о работающих служащих и пенсионерах
struct mail {
             char id;  // a - active (), r - retired()
             union {
                    struct {
                            char name[30];
                            char dept[10];
                            char location[3];
                           } active;
                    struct {
                            char name[30];
                            char street[20];
                            char city_state[3];
                            char zip[5];
                           } retired;
                  } info;
            } person;
```

Структура типа `struct mail` используется для хранения почтового адреса работающего служащего или пенсионера. При заполнении структуры этого типа информацией в нее заносится порция данных, соответствующая элементам `active` или `retired` объединения `info`. Поле `id` устанавливается равным `'a'` или `'r'` для указания фактически записанного в объединении элемента. При применении объединения используется меньше памяти, чем в случае применения структуры, которая имела бы идентичные поля, но некоторые из них не использовались бы [11].

Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. В таких случаях используются переменные с изменяемой структурой. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь, периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Пример. Информация о геометрических фигурах представляется на основе комбинированного использования структуры и объединения.

```
struct figure {
  double area,perimetr; // общие компоненты
  int type; // признак компонента
  union { // перечисление компонент
          double radius; // окружность
```

```
        double a[2];   // прямоугольник
        double b[3];   // треугольник
      } geom_fig;
} fig1, fig2;
```

В общем случае каждый объект типа `figure` будет состоять из трех компонентов: `area`, `perimetr`, `type`. Компонент `type` называется меткой активного компонента, так как он используется для указания, какой из компонентов объединения `geom_fig` является активным в данный момент. Такая структура называется переменной структурой, потому что ее компоненты меняются в зависимости от значения метки активного компонента (значение `type`). Отметим, что вместо компоненты `type` типа `int`, целесообразно было бы использовать перечисляемый тип. Например, такой

```
enum figure_chess {CIRCLE, BOX, TRIANGLE};
```

Константы `CIRCLE`, `BOX`, `TRIANGLE` получат значения соответственно равные `0, 1, 2`. Переменная `type` может быть объявлена как имеющая перечислимый тип:

```
enum figure_chess type;
```

В этом случае компилятор C++ предупредит программиста о потенциально ошибочных присваиваниях, таких, например, как `figure.type = 40;`

В общем случае переменная структуры будет состоять из трех частей: набор общих компонент, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры имеет следующий вид:

```
struct {
        ОбщиеКомпоненты; МеткаАктивногоКомпонента;
        union {
              ОписаниеКомпоненты1;
              ОписаниеКомпоненты2;
              ..................
              ОписаниеКомпонентыN;
            } ИдентификаторОбъединения;
      } ИдентификаторСтруктуры;
```

Пример. Определения переменной структуры с именем `helth_record`.

```
struct { // общая информация
  char name[25];    // имя
  int age;          // возраст
  char sex;         // пол
  // метка активного компонента(семейное положение)
  enum merital_status ins;
  // переменная часть
  union { // холост
```

```
            // нет компонент
           struct { // состоит в браке
                   char marriage_date[8];
                   char spouse_name[25];
                   int  no_children;
                  } marriage_info;
            // разведен */
           char date_divorced[8];
          } marital_info;
 } health_record;
enum marital_status { SINGLE, // холост
     MARRIGO, // женат
     DIVOREED // разведен
   };
```

Обращаться к компонентам структуры можно при помощи ссылок:

```
  helth_record.name
  helth_record.ins
  helth_record.marriage_info.marriage_date
```

Вопросы для самопроверки

1. Можно ли выполнить прямое присваивание массивов объявленных так: `int x[10], y[10];`?
2. Когда, с какой целью и почему возможно объявление безразмерных массивов?
3. Какие ограничения распространяются на тип массива?
4. Каким образом можно определить объем памяти, выделяемой под массив?
5. Каким образом можно составить выражение для генерации массива случайными целыми числами на заданном промежутке?
6. Когда, с какой целью и почему возможно объявление безразмерных массивов? С одним безразмерным измерением?
7. Приведите возможные обращения к элементу двумерного массива, аналогичные обращению `mas[i][j].`
8. Почему в C++ не выполняется операция прямого присваивания значения строке?
9. Почему символ и строка, состоящая из одного символа, занимают разный объем памяти?
10. Какая из функций, `gets` или `puts`, заносит в поток управляющий символ `'\n'` и с какой целью?
11. В чем принципиальное отличие типов массив и структура?
12. Как располагаются в памяти элементы структуры?
13. Для моделирования каких данных целесообразно использовать структуры?

14. Какими способами можно обратиться к данным структуры?

15. В чем принципиальное отличие размещения в памяти элементов структуры и объединения?

16. Каким образом определяется размер объединения?

17. Какова цель использования объединений в программировании?

18. Какие существуют способы обращения к элементам объединения?

19. С какой целью объединения включаются в структуру в качестве ее полей?

20. Как и с какой целью объявляются переменные с изменяемой структурой?

Библиографический список

1. Незнанов А.А. Программирование и алгоритмизация : учебник для студ. учреждений высш. проф. образования. – М. : Академия, 2010. – 304 с.
2. Сундукова Т.О., Ваныкина Г.В. Структуры и алгоритмы компьютерной обработки данных : учеб. пособие. – Томск : Изд-во ТГПУ им. Л.Н. Толстого, 2011. - 239 с.
3. Единая Система Программной Документации (ЕСПД) – М. : ИПК Издательство стандартов, 2001. – 163 с.
4. Александров Э. Э. Программирование на языке С в Microsoft Visual Studio 2010 : учеб. пособие / Э. Э. Александров, В. В. Афонин. – Саранск : Изд-во Мордов. ун-та, 2010. – 428 с.
5. Стрикелева Л. В. Программирование : учеб. пособие. – Минск : Изд-во БГУ, 2012. – 345 с.
6. Керниган Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – М. : Вильямс, 2007. – 304 с.
7. Подбельский В.В. Практикум по программированию на языке Си: учеб. пособие / В.В. Подбельский. – М. : Финансы и статистика, 2004. – 576 с.
8. Подбельский В.В. Программирование на языке Си: учеб. пособие / В.В. Подбельский, С.С. Фомин. – М. : Финансы и статистика, 2004. – 600 с.
9. Подбельский В.В. Язык Си++ : учеб. пособие / В.В. Подбельский. – М. : Финансы и статистика, 2005. – 560 с.
10. Романов Е.Л. Практикум по программированию на языке С++ : учеб. пособие / Е.Л. Романов. – СПб. : БХВ-Петербург, 2004. – 432 с.
11. С/С++. Структурное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Питер, 2004. – 239 с.

Учебное издание

Солодовникова
Ирина Валентиновна

АЛГОРИТМЫ, СТРУКТУРЫ ДАННЫХ И ПРОГРАММИРОВАНИЕ

Редактор Асылбекова С.

Подписано в печать ___ _____ 2014 г.
Формат 60x90/$_{16}$ Объем 5,8 печ.л. Тираж _____ экз. Заказ___
Издательство КарГТУ. 100027. г. Караганда, Б.Мира, 56.