Общие сведения об алгоритмах

1. Основные определения и понятия
2. Средства изображения алгоритмов
3. Базовые структуры алгоритмов
4. Классификация языков программирования и их краткая характеристика

Предметом курса являются методы и средства составления алгоритмов и программ в целях решения задач на ЭВМ.

Понятие алгоритма является одним из основных понятий современной математики. Существует несколько версий происхождения данного термина. Наиболее распространено мнение, что слово *алгоритм* происходит от имени средневекового узбекского математика Аль-Хорезми, который еще в IX в. (825 г.) определил правила выполнения четырех арифметических действий в десятичной системе счисления. Процесс выполнения арифметических действий был назван *алгоризмом.* Позднее встречались другие варианты использования данного термина: *алгорисмус*, *алгорифм* и, наконец, алгоритм.

Изменялось не только название, но и расширялось значение термина. К началу двадцатого века под алгоритмом понимали любой арифметический или алгебраический процесс, выполняемый по строго определённым правилам. В настоящее время определение алгоритма зависит от области знаний, где оно используется.

Первоначально для записи алгоритмов пользовались средствами обычного языка (словесное представление алгоритмов).

1. Основные определения и понятия

Алгоритмизация – это процесс построения алгоритма решения задачи, результатом которого является выделение этапов процесса обработки данных, формальная запись содержания этих этапов и определение порядка их выполнения.

Алгоритм – это точное предписание, определяющее вычислительный процесс, ведущий от варьируемых начальных данных к искомому результату. (А. Марков)

***Свойства алгоритма*:**

1) детерминированность – точность указаний, исключающая их произвольное толкование;

2) дискретность – возможность разделения вычислительного процесса на отдельные элементарные операции (выполняемые исполнителем по определенным командам);

3) результативность – прекращение процесса через определенное число шагов с выдачей искомых результатов или сообщения о невозможности продолжения вычислительного процесса;

4) массовость – пригодность алгоритма для решения всех задач заданного класса.

Выделяют численные и логические алгоритмы.

Численные алгоритмы – алгоритмы, в соответствии с которыми решение поставленных задач сводится к арифметическим действиям.

Логические алгоритмы – алгоритмы, в соответствии с которыми решение поставленных задач сводится к логическим действиям. Примерами логических алгоритмов могут служить алгоритмы поиска минимального числа, поиска пути на графе, поиска пути в лабиринте и др.

Алгоритмический язык – формализованный язык, предназначенный для однозначной записи алгоритма.

Язык программирования – система слов и правил, используемая для создания компьютерных программ. Большинство компьютеров работает на основе двоичных языков (использующих два знака, 0 и 1), которые называют машинными кодами.

Писать программы на машинном языке очень сложно и утомительно. Программист должен знать числовые коды всех машинных команд, сам распределять память под команды программы и данные. Для повышения производительности труда программистов применяются искусственные языки программирования. При этом требуется перевод программы, написанной на таком языке, на машинный язык. Этот перевод выполняет транслятор.

Согласно ГОСТу 19781-90 «Термины и определения», ***программа*** – это данные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма. Программа содержит наряду с описанием данных команды, в какой последовательности, над какими данными и какие операции должна выполнять машина, а также в какой форме следует получить результат. Это обеспечивают различные операторы. Таким образом, можно сказать, что программа – последовательность машинных команд, предназначенная для достижения конкретного результата.

Для разработки программ используются системы программирования.

Система программирования включает язык программирования и интегрированную среду разработки. Интегрированная среда разработки представляет собой систему программных средств, используемых для разработки программного обеспечения: транслятор (компилятор или интерпретатор), редактор текстов программ, библиотеки стандартных программ и функций, отладчик и др.

2. Средства изображения алгоритмов

Основными способами описания алгоритмов являются:

- словесный;
- формульно-словесный;
- блок-схемный;
- псевдокод;
- структурные диаграммы;

– языки программирования.

Форма записи, состав и количество операций алгоритма зависят от того, кто будет исполнителем этого алгоритма.

Словесный способ содержит описание этапов вычислений на естественном языке в произвольной форме с требуемой детализацией.

Пример.

Пусть задан массив чисел. Требуется проверить, все ли числа принадлежат заданному интервалу. Интервал задается границами *A* и *B*.

Решение.

1. Берем первое число. На **п. 2**.

2. Сравниваем: выбранное число принадлежит интервалу? Если да, то на **п. 3**, если нет – на **п. 6**.

3. Все элементы массива просмотрены? Если да, то на **п. 5**, если нет – то на **п. 4**.

4. Выбираем следующий элемент. На **п. 2**.

5. Печать сообщения: все элементы принадлежат интервалу. На **п. 7**.

6. Печать сообщения: не все элементы принадлежат интервалу. На **п. 7**.

7. Конец.

При этом способе отсутствует наглядность вычислительного процесса, так как нет достаточной формализации.

Формульно-словесный способ – задание инструкций с использованием математических символов и выражений в сочетании со словесными пояснениями.

Пример.

Написать алгоритм вычисления площади треугольника по трем сторонам.

1. Вычислить полупериметр треугольника $p = (a + b + c)/2$. К **п. 2**.

2. Вычислить $S = \sqrt{p(p-a)(p-b)(p-c)}$. К **п. 3.**

3. Вывести S, как искомый результат и прекратить вычисления.

При использовании этого способа может быть достигнута любая степень детализации, более наглядно, но не строго формально.

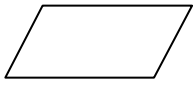
Блок-схемный способ описания алгоритма – это графическое изображение его логической структуры. При этом каждый этап процесса обработки данных представляется в виде геометрических фигур (блоков) и дополняется текстовой надписью.

Правила выполнения схем алгоритмов регламентирует ГОСТ 19.701-90 (единая система программной документации).

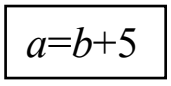
Блоки на схемах соединяются линиями потоков информации.

Основные элементы блок-схем:

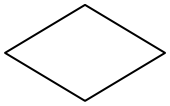
– данные – ввод/вывод данных вне зависимости от физического носителя:



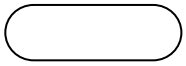
– процесс – выполнение определенной операции или группы операций, приводящее к изменению информации:



– решение – логический блок (выбор направления выполнения алгоритма в зависимости от некоторого условия):



– начало или конец алгоритма, вход в программу или выход из нее:

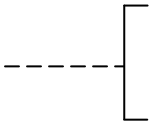


– предопределенный процесс – шаги программы, определенные в функции, подпрограмме:

– соединитель – выход в часть схемы и вход из другой части этой схемы; используется для обрыва линии и продолжения ее в другом месте; соответствующие символы-соединители должны содержать одно и то же уникальное обозначение, например, одну и ту же цифру:

– граница цикла – состоит из двух частей, отображает начало и конец цикла; обе части символа имеют один и тот же идентификатор; условия для инициализации, приращения, завершения и т. д. помещаются внутри символа в начале или в конце в зависимости от расположения операции, проверяющей условие:

– комментарий – используют для добавления пояснительных записей и в случае, если объем текста превышает размеры символа, внутри которого он должен помещаться:



Линии в схемах должны подходить к символу слева или сверху, а исходить справа или снизу и быть направлены к центру символа.

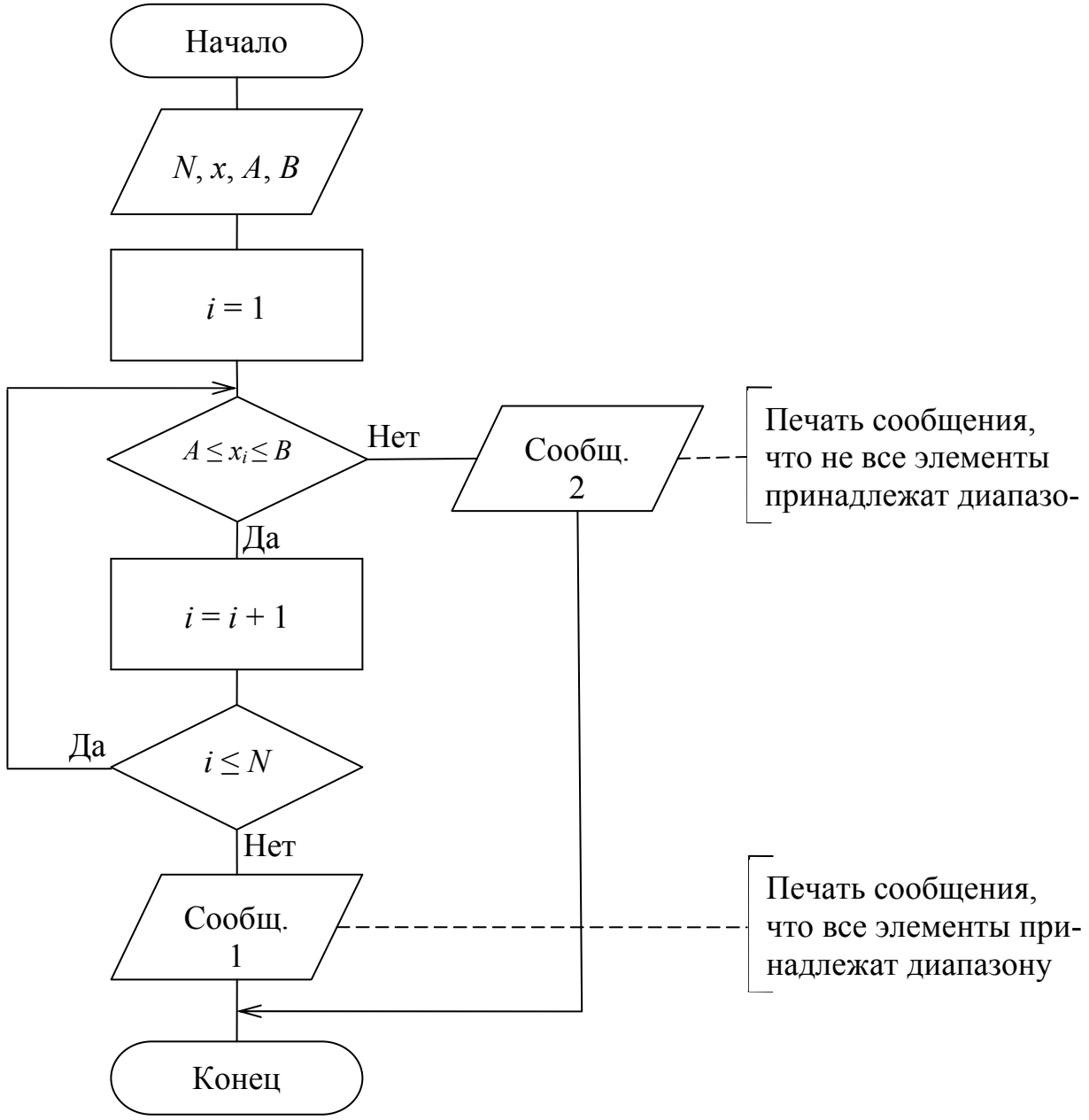


Рисунок 1. Пример блок-схемы задачи, для которой приведен словесный способ описания алгоритма

Псевдокод обычно представляет собой смесь операторов языка программирования и естественного языка. Позволяет формально изображать логику программы, не заботясь при этом о синтаксических особенностях конкретного языка программирования. Является средством представления логики программы, которое можно применять вместо блок-схемы.

Запись алгоритма в виде псевдокода:

Выбираем первый элемент ($i = 1$)

IF $A > x_i$ или $x_i > B$

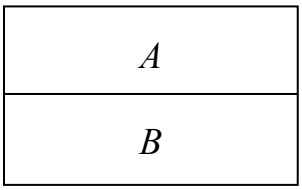
Печать сообщения**,** что не все элементы принадлежат диапазону и переход на конец

ELSE
Переход к следующему элементу ($i = i + 1$)
IF массив не кончился $(i \le n)$
Переход на проверку интервала
ELSE
Печать сообщения, что все элементы принадлежат диапазону
Конец

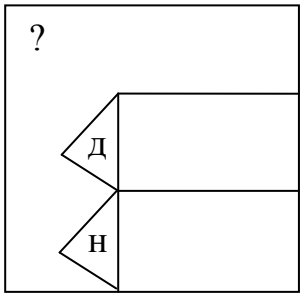
Структурные диаграммы – могут использоваться в качестве структурных блок-схем, для показа межмодульных связей, для отображения структур данных, программ и систем обработки данных. Существуют различные структурные диаграммы: диаграммы Насси – Шнейдермана, диаграммы Варнье, Джексона, МЭСИД и др.

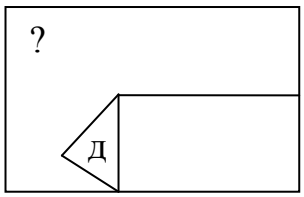
Основные элементы диаграммы МЭСИД:

– следование:

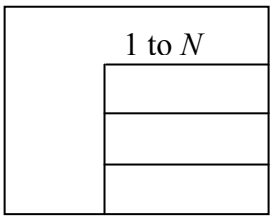


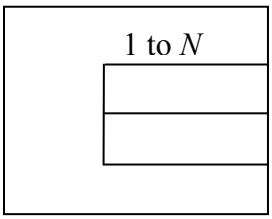
– разветвление:

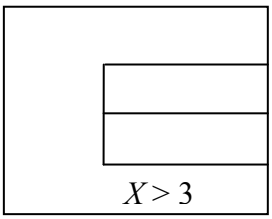




– повторение:







На рис. 2 приведен пример использования диаграммы МЭСИД для алгоритма определения суммы всех положительных элементов одномерного массива.

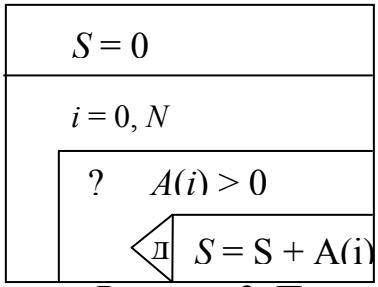
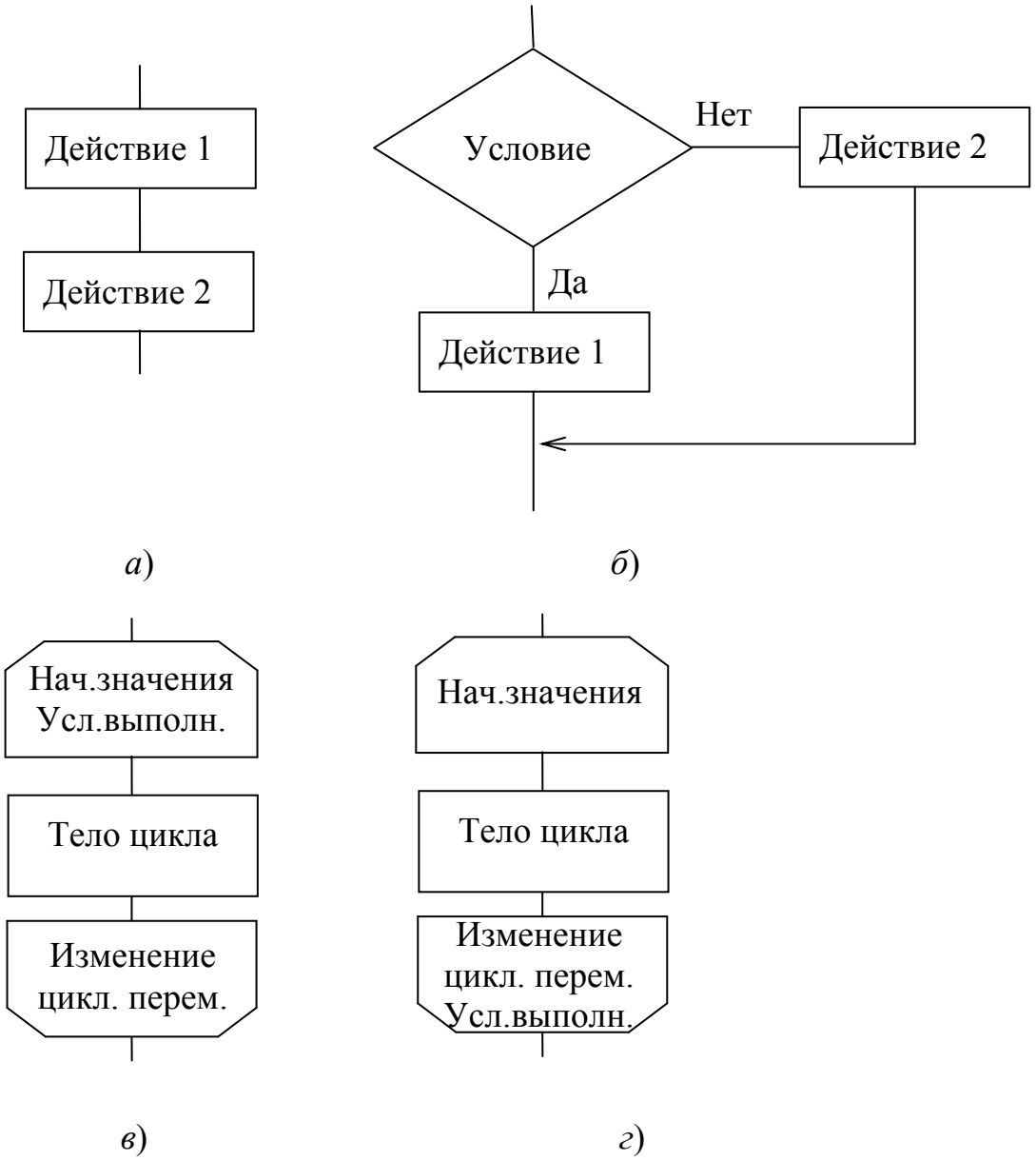


Рисунок 2. Пример использования диаграммы МЭСИД

3. Базовые структуры алгоритмов

Базовые структуры алгоритмов – это определенный набор блоков и стандартных способов их соединения для выполнения типичных последовательностей действий. К основным структурам относятся следующие: линейные (рис. 3, *а*), разветвляющиеся (рис. 3, *б*), циклические (рис. 3, *в*, *г*).



а) *б*)

в) *г*)

Рисунок 3. Базовые структуры алгоритмов

Линейным называется алгоритм, в котором действия осуществляются последовательно друг за другом. Стандартная блок-схема линейного алгоритма приводится на рис. 4 (вычисление произведения двух чисел – *A* и *B*).

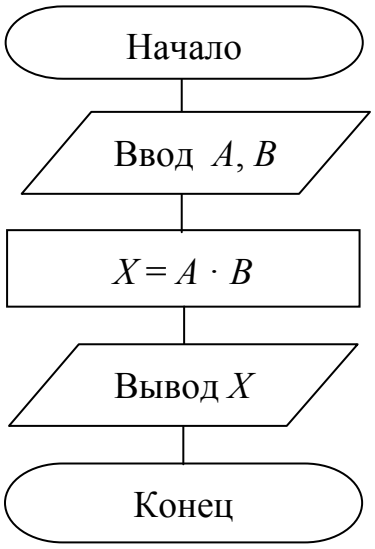


Рисунок 4. Пример линейного алгоритма

Разветвляющимся называется алгоритм, в котором действие выполняется по одной из возможных ветвей решения задачи в зависимости от условий. В отличие от линейных алгоритмов, в которых команды реализуются последовательно, в разветвляющихся алгоритмах имеется условие, в зависимости от истинности которого выполняется та или иная последовательность команд (действий).

В качестве условия в разветвляющемся алгоритме может быть использовано любое понятное исполнителю утверждение, которое может соблюдаться (быть истинно) или не соблюдаться (быть ложно). Такое утверждение может быть выражено как словами, так и формулой. Таким образом, алгоритм ветвления состоит из условия и двух последовательностей команд

Примером может являться разветвляющийся алгоритм, изображенный на рис. 5. Аргументами этого алгоритма являются числа A и B, а результатом – число X. Если условие $A \geq B$ истинно, то выполняется операция умножения чисел ($X = A \cdot B$), в противном случае выполняется операция сложения ($X = A + B$). В результате печатается то значение X, которое получается после выполнения одного из действий.

Циклическим называется алгоритм, в котором определенная последовательность команд (тело цикла) выполняется многократно. Однако слово «многократно» не значит «до бесконечности». Организация циклов, никогда не приводящая к остановке в выполнении алгоритма, является нарушением требования его результативности – получения результата за конечное число шагов.

Перед операцией цикла осуществляются операции присвоения начальных значений тем объектам, которые используются в теле цикла. В цикл входят в качестве базовых следующие структуры: блок проверки условия выполнения цикла и блок, называемый *телом цикла*. Если тело цикла расположено после проверки условий (см. рис. 3, *в* – цикл с предусловием), то может

случиться, что при определенных условиях тело цикла не выполнится ни разу.

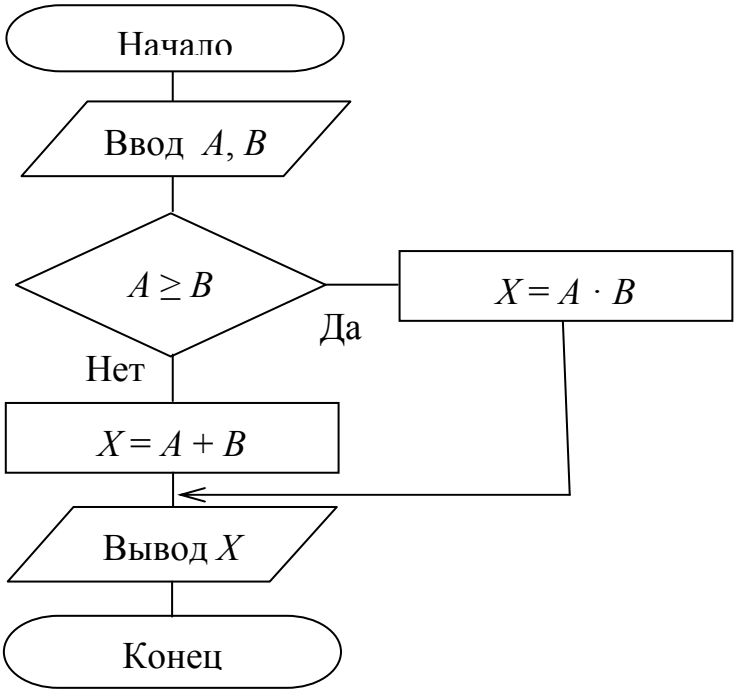


Рисунок 5. Пример разветвляющегося алгоритма

Возможен другой случай, когда тело цикла выполняется по крайней мере один раз и будет повторяться до тех пор, пока истинно условие продолжения цикла. Цикл, в котором его тело расположено перед проверкой условия, называется циклом с постусловием (см. рис. 3, *г*).

Рассмотрим циклический алгоритм *с предусловием* (рис. 6) на примере алгоритма вычисления факториала. N – число, факториал которого вычисляется. Начальное значение $N!$ (в алгоритме $N!$ обозначается как F) принимается равным 1. K будет меняться от 1 до N и вначале также равно 1. Цикл будет выполняться, пока справедливо условие $K \le N$. Тело цикла состоит из двух операций: $F = F \cdot K$ и $K = K + 1$.

Циклические алгоритмы, в которых тело цикла выполняется заданное число раз, реализуются с помощью цикла со счетчиком. Цикл со счетчиком реализуется с помощью рекурсивного увеличения значения счетчика в теле цикла ($K = K + 1$ в алгоритме вычисления факториала).

Решение сложной задачи довольно часто сводится к решению нескольких более простых подзадач. Соответственно, сложный алгоритм может разбиваться на отдельные алгоритмы, которые называются вспомогательными. Каждый вспомогательный алгоритм описывает решение какой-либо подзадачи.

Процесс построения алгоритма методом последовательной детализации состоит в следующем. Сначала алгоритм формулируется в «крупных» блоках (командах), которые могут быть непонятны исполнителю (не входят в его систему команд) и записываются как вызовы вспомогательных алгоритмов. Затем происходит детализация, и все вспомогательные алгоритмы подробно расписываются с использованием команд, понятных исполнителю.

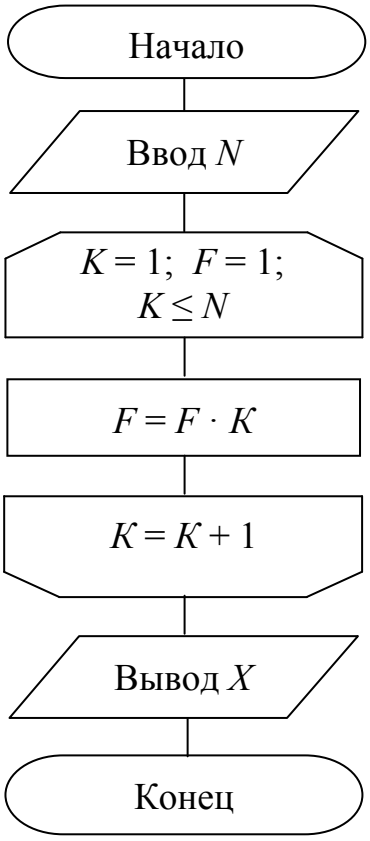


Рисунок 6. Пример циклического алгоритма

1. Классификация языков программирования и их краткая характеристика

Существуют разные варианты классификации языков программирования.

С точки зрения зависимости от конструкции ЭВМ языки программирования делятся на две группы:

1. Машинно-зависимые языки, которые можно применять на одной ЭВМ или на ограниченном подмножестве машин с одинаковой архитектурой.

2. Машинно-независимые языки – их можно использовать на любой ЭВМ. Языки этой группы называют универсальными.

Машинно-зависимые языки (низкоуровневые языки программирования) в зависимости от их близости к машинным языкам делятся на три группы:

– *машинные языки*, которые не применяются в настоящее время, так как громоздки, требуют больших затрат времени на написание, трудно воспринимаемы, используют двоичные коды команд программы. Однако имеют наибольшее быстродействие и требуют минимальных затрат памяти;

– *ассемблерные языки* (языки типа 1 : 1, то есть одна ассемблерная команда после трансляции порождает ровно одну машинную команду) – вместо двоичных, восьмеричных, шестнадцатеричных и десятичных записей кодов операций и адресов используется их наглядная символическая запись, для перевода в двоичные коды используется компилятор, данные и команды размещаются в ОЗУ автоматически (причем без «дырок»), проигрыш по быстродействию 5–10 %;

– *макроассемблеры* (языки типа 1:*n*) – обладают всеми возможностями ассемблера и дополнительно мощным аппаратом макровызовов и макроопределений; основной язык системных программистов; можно разработать свой набор макроопределений; сохраняется высокая эффективность программ.

Машинно-независимые языки включают следующие группы языков.

Высокоуровневый язык программирования – язык программирования, разработанный для быстроты и удобства использования программистом. Основная черта высокоуровневых языков – это абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) очень длинны и сложны для понимания.

Так, высокоуровневые языки стремятся не только облегчить решение сложных программных задач, но и упростить портирование программного обеспечения. Использование разнообразных трансляторов и интерпретаторов обеспечивает связь программ, написанных при помощи языков высокого уровня, с различными операционными системами и оборудованием, в то время как их исходный код остается, в идеале, неизменным.

Такого рода оторванность высокоуровневых языков от аппаратной реализации компьютера помимо множества плюсов имеет и минусы. В частности, она не позволяет создавать простые и точные инструкции к используемому оборудованию. Программы, написанные на языках высокого уровня, проще для понимания программистом, но менее эффективны, чем их аналоги, создаваемые при помощи низкоуровневых языков. Одним из следствий этого стало добавление поддержки того или иного языка низкого уровня (язык ассемблера) в ряд современных профессиональных высокоуровневых языков программирования.

Высокоуровневыми языками программирования являются C, C++, Visual Basic, Java, Python, PHP, Ruby, Perl, Pascal, Delphi и т.д. Подобные языки могут работать с комплексными структурами данных. В большинстве из них интегрирована поддержка строковых типов, объектов, операций файлового ввода-вывода и т. п.

Сверхвысокоуровневый язык программирования (язык программирования сверхвысокого уровня, VHLL – very high-level programming language) – язык программирования с очень высоким уровнем абстракции. В отличие от языков программирования высокого уровня, где описывается принцип «как нужно сде-

лать», в сверхвысокоуровневых языках программирования описывается лишь принцип «что нужно сделать». Термин впервые появился в середине 1990-х годов для идентификации группы языков, используемых для быстрого прототипирования, написания одноразовых скриптов и подобных задач.

Языки сверхвысокого уровня обычно используются для специфических приложений и задач. В связи с этим они могут использовать синтаксис, который никогда не используется в других языках программирования (например, непосредственно синтаксис английского языка). Примером VHLL, распознающего синтаксис английского языка, может служить язык компилятора текстовых квестов Inform версии 7. Также к таким языками часто относят и ICON, который имеет много общего с языками логического программирования.

Новой тенденцией стала разработка языков программирования еще более высокого уровня (*ультра-высокоуровневых*). Такого рода языки характеризуются наличием дополнительных структур и объектов, ориентированных на прикладное использование. Прикладные объекты, в свою очередь, требуют минимальной настройки в виде параметров и моментально готовы к использованию. В качестве примера можно привести созданную в Австралии самообучающуюся нейронную сеть LISA со встроенным языком описания фактов, сущностей и взаимосвязей. Использование ультра-высокоуровневых языков программирования снижает временные затраты на разработку программного обеспечения и повышает качество конечного продукта за счет уменьшения объема исходных кодов.

Базовые элементы и операции языка С++

1. Элементы языка С++
2. Типы данных языка С++
3. Структура программы на языке С++
4. Операции языка С++

В лекции рассматриваются основные элементы и базовые типы данных языка С++, представлена структура программы на языке С++, рассмотрена классификация и правила выполнения основных операций языка С++.

1. Элементы языка С++

Программа на языке С++ формируется с помощью набора лексем – единиц текста программы, имеющих смысл для компилятора и не разбиваемых в дальнейшем. Границы лексем определяются пробелами, знаками пунктуации и операций.

При написании программ на языке С++ используются *латинские прописные* и *строчные буквы*, *арабские цифры*, *специальные символы*:
+ − * / < > = | & ! ~ ' # % ^ ? _ : ; , . ( ) [ ] { } " .

Предложения (*операторы*) языка обычно заканчиваются точкой с запятой.

Исключение:

− директивы препроцессора, начинающиеся с символа #;

− составные операторы и блоки определения функций, которые заключены в фигурные скобки { }.

В языке С++ имеются также управляющие символы, комментарии, ключевые слова, идентификаторы.

Управляющие символы непосредственно на экране не отображаются. Например, ' \t' − горизонтальная табуляция; ' \n' − новая строка.

Комментарий − любые символы, не воспринимаемые компилятором и используемые для пояснения отдельных частей или всей программы:

// однострочный комментарий

/* многострочный комментарий*/

Использование кириллических символов в некоторых случаях возможно и целесообразно (в комментариях, символьных строках, названиях файлов, если это допускает среда выполнения). Но пока единого стандарта на кодировку кириллических символов нет. Поэтому могут быть сложности при переносе таких программ с одного компьютера на другой, при переходе из одной среды выполнения в другую.

Имена (*идентификаторы*) констант и переменных вводятся, чтобы отличать (идентифицировать) различные элементы и производить действия с ними.

Строчные и заглавные буквы − разные символы. Идентификатор *A* отличается от идентификатора *a*.

Рекомендации по ***именованию:***

– использовать имена из постановки задачи;

– давать короткие осмысленные имена, отражающие назначение переменной, функции, объекта или типа;

– не начинать с символа подчеркивания, поскольку такой прием широко используется в библиотеках системы программирования;

– следовать единой системе именования; здесь существуют различные варианты, например:

− начинать с прописной буквы, если требуется подчеркнуть уникальность идентификатора;

− использовать символ подчеркивания или прописные буквы внутри идентификатора для построения хорошо читаемых сложных идентификаторов.

Ключевые слова − зарезервированные в служебных целях идентификаторы, имеющие специальное значение для компилятора: типы данных, операторы. Нельзя использовать как идентификаторы.

Рекомендации по ***комментированию:***

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

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

– не вставлять комментарии в середину строки программы;

– не писать очевидных комментариев;

– начинать комментарий с той же позиции в строке, что и комментируемый текст.

2. Типы данных языка С++

Данные – это факты и идеи, представленные в формализованном виде, позволяющем передавать или обрабатывать их с помощью некоторого процесса (алгоритма).

Данные, известные перед выполнением алгоритма, являются начальными, *исходными данными.* Результат решения задачи – это конечные, *выходные данные.* В задачах нахождения максимума из последовательности чисел и их произведения исходными данными являются числа, а результатами (выходными данными) – соответственно значение максимума и значение произведения.

Данные делятся на переменные и константы.

Переменные – это именованная область памяти, в которой могут храниться различные значения. При этом значение переменной во время выполнения алгоритма можно изменить.

Например, для построения алгоритма вычисления площади круга необходимо объявить две переменные: переменную R, в которую будет заноситься значение радиуса окружности, и переменную S для вычисления площади круга по формуле $S = \pi R^2$.

Свойства переменной:

– переменная называется неопределенной до тех пор, пока она не получит значение:

вводом извне;

занесением константы;

занесением значения другой, ранее определенной переменной;

– в каждый момент времени переменная может либо иметь определенное значение, либо быть неопределенной;

– последующее значение уничтожает (стирает) предыдущее. Выбор (чтение) переменной и ее использование не изменяют значение переменной.

Константы – это данные, значения которых не меняются в процессе выполнения алгоритма. В примере, описанном выше, константой является число π.

Различают следующие виды констант:

1) числовые;

2) символьные;

3) строковые.

Числовые константы разделяют на вещественные и целые:

– *вещественные*:

3.14 эквивалентна 314e–2, также эквивалентна 0.314E1;

–2.5 эквивалентна –0.25e1;

– *целые*:

десятичные: 23, 567, 0, –24 – используются десятичные цифры;

восьмеричные: 00, 01, 07777 – используются восьмеричные цифры, запись числа начинается с нуля;

шестнадцатеричные: 0x0000, 0x0001,…,0x7FFF, …, 0xFFFF, …

Символьные константы – это символ, заключенный в апострофы. Значением символьной константы является числовой код символа.

Примеры символьных констант:

" " – пробел;

"б" – буква «б»;

"\n" – символ новой строки.

Строковые константы – последовательности символов, ограниченные двойными кавычками: "Это строковая константа", "ERROR:".

Для хранения строковой константы требуется $n + 1$ байт, где n – число символов между ограничителями строковой константы. Дополнительный байт последний и требуется для хранения кода 000, который записывается автоматически. Поскольку машинное представление строки имеет последний нулевой байт, то говорят, что строки С++ заканчиваются нулем.

Каждая переменная и константа должны иметь свое уникальное имя. Имена переменных и констант задаются идентификаторами. Идентификатор (по определению) представляет собой последовательность букв и цифр, начинающуюся с буквы или подчеркивания.

Идентификаторы – это имена переменных, типов данных, функций и меток, используемых в программе. Идентификаторы задаются программистом и должны удовлетворять следующим требованиям:

– состоять из 1–32 латинских букв, цифр или знака подчеркивания (_);
– первый символ – буква или подчеркивание;
– недопустимы пробелы в имени и перенос имени со строки на строку;
– не должны совпадать с ключевым словом.

Идентификатор создается объявлением соответствующей ему переменной, типа данных или функции.

Типы данных. С данными тесно связано понятие типа данных. Любой константе, переменной, выражению (с точки зрения обработки на ЭВМ) всегда сопоставляется некоторый тип. Тип данных характеризует множество допустимых значений данных и множество разрешенных операций над ними. Например, если переменная *i* в некотором алгоритме может принимать только значения из множества целых чисел, то ей ставится в соответствие целый тип данных. Также тип данных определяет размер памяти, который занимают переменные и константы данного типа.

Тип объекта указывается в определении объекта с помощью служебного слова (слов) – спецификации типа.

В языке С++ выделяют следующие ***категории типов:***

– базовые типы данных;
– производные (определяемые) типы.

Базовые типы имеют имена, которые являются ключевыми словами языка.

К *базовым* типам относятся: скалярные типы и пустой тип – void.

Скалярные типы делятся на целочисленные и вещественные.

Логический тип, символьные и целые типы данных относятся к целочисленному типу, для которого определены все операции с целыми числами.

Производные типы определяются (образуются) на основе базовых типов. Производные типы делятся на скалярные и структурные (агрегатные).

К скалярным производным типам относятся:

– *указатели* **(***имя_ типа* ***)**;
– *перечисления* **(*enum*** – enumeration) – множество поименованных целых значений;
– *ссылки* **(***имя_типа* **&)**.

Структурными типами являются:

– *массивы (тип_элемента имя_массива[число_элементов]);*
– *структуры (struct);*
– *объединения (union);*
– *классы (class).*

Дерево классификации типов языка С++ приведено на рис. 1.

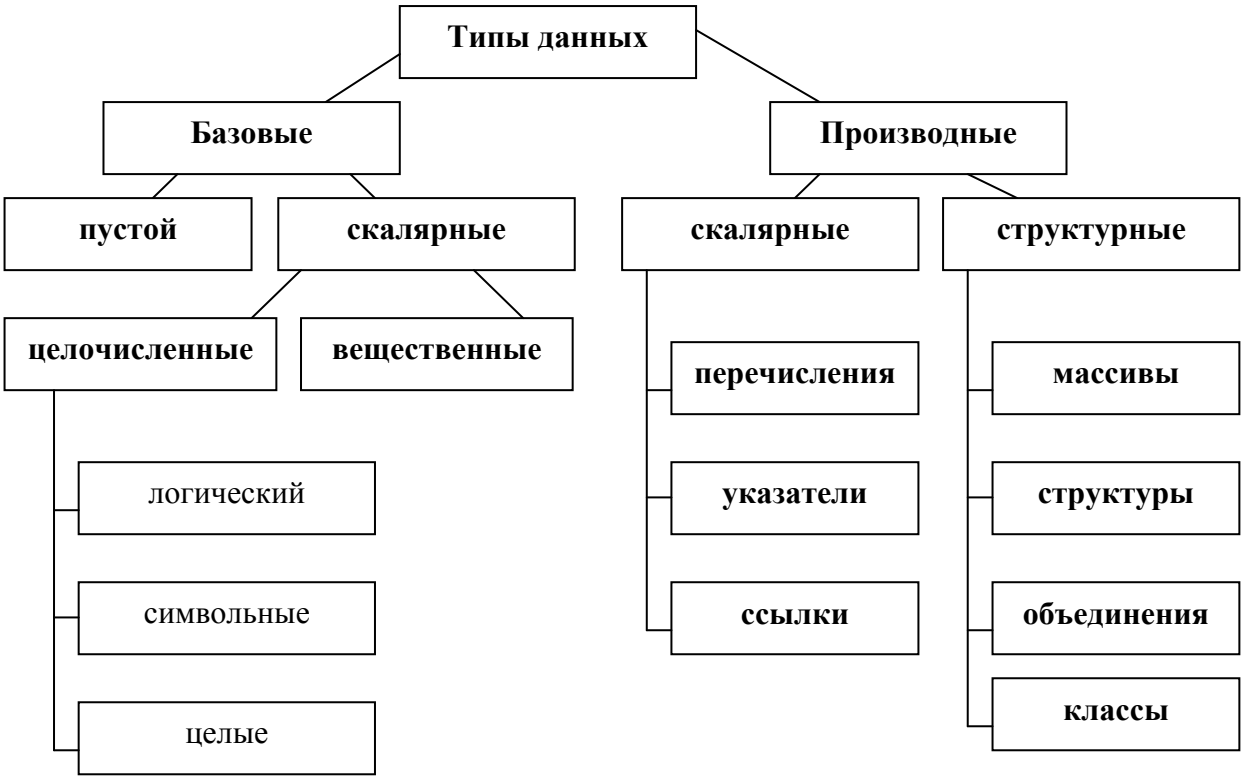


Рисунок 1. Классификация типов языка C++

Таблица 1

Характеристики базовых типов данных языка C++

Type Name	Bytes	Other Names	Range of Values
int	4	signed	–2 147 483 648 to 2 147 483 647
unsigned int	4	unsigned	0 to 4 294 967 295
__int8	1	char	–128 to 127
unsigned __int8	1	unsigned char	0 to 255
__int16	2	short, short int, signed short int	–32,768 to 32,767
unsigned	2	unsigned	0 to 65,535

__int16		short, unsigned short int	
__int32	4	signed, signed int, int	–2,147,483,648 to 2,147,483,647
unsigned __int32	4	unsigned, unsigned int	0 to 4,294,967,295
__int64	8	long long, signed long long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned __int64	8	unsigned long long	0 to 18,446,744,073,709,551,615
bool	**1**	none	**false or true**
char	**1**	none	**–128 to 127** by default
signed char	1	none	–128 to 127
unsigned char	1	none	0 to 255
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

		__int64)	
unsigned long long	8	none (but equivalent to unsigned __int64)	0 to 18,446,744,073,709,551,615
enum	varies	none	See Remarks.
float	**4**	none	**3.4E +10^{-38} (7 digits)**
double	**8**	none	**1.7E +10^{-308} (15 digits)**
long double	same as double	none	same as double
wchar_t	**2**	__wchar_t	**0 to 65,535**

Тип ***void*** не имеет значения и введен для описания функций, не возвращающих значений, или для переменных, тип которых определяют позже.

Тип ***bool*** введен в стандарте C++, раньше он определялся на основе целых типов. *Логические переменные* типа *bool* могут принимать одно из двух значений: *false* (ложь) или *true* (истина). По определению, значение *false* равно 0, а *true* – 1. Логические переменные широко используются в операциях сравнения, логических операциях и логических выражениях. Размер переменной зависит от реализации, но обычно составляет 1 байт. Примеры объявлений:

```
bool A = false, B = true, C = a > b;
```

В *символьной* переменной типа ***char*** может храниться один из ASCII-символов. Например:

```
char m ='f';
```

Одиночные кавычки (апострофы) используются для задания символьного значения переменной. Запись вида ***'f'*** называют *символьной константой* или *символьным литералом.*

Строка символов объявляется следующим образом:

```
char h[20];
char d[]="akademia";
```

В C++ поддерживается два типа строк. Первый (и наиболее популярный) предполагает, что *строка* определяется как символьный массив, который завершается нулевым символом ('\0'). Таким образом, строка с завершающим нулем состоит из символов и конечного нуль-символа. Такие строки широко используются в программировании, поскольку они позволяют программисту выполнять самые разные операции над строками и тем самым обеспечивают

высокий уровень эффективности программирования. Поэтому, используя термин *строка,* C++-программист обычно имеет в виду именно строку с завершающим нулем. Однако существует и другой тип представления строк в C++. Он заключается в применении объектов класса string, который является частью библиотеки классов C++. Таким образом, класс string − не встроенный тип. Он подразумевает объектно-ориентированный подход к обработке строк, но используется не так широко, как строки с завершающим нулем. В этом разделе мы рассматриваем строки с завершающим нулем.

Основы представления строк

Объявляя символьный массив, предназначенный для хранения строки с завершающим нулем, необходимо учитывать признак ее завершения и задавать длину массива на единицу больше длины самой большой строки из тех, которые предполагается хранить в этом массиве. Например, при объявлении массива str, в который предполагается поместить 10-символьную строку, следует использовать следующую инструкцию.

```
char str[11] ;
```

Заданный здесь размер (11) позволяет зарезервировать место для нулевого символа в конце строки.

Для явного задания диапазона можно использовать *модификаторы*:

`signed` (*со знаком*);

`unsigned` (*беззнаковые*);

`short` (*короткое*);

`long` (*длинное*).

В объявлениях символьных типов обычно модификаторы используются, когда значение символа задается с помощью числового значения или требуется «обратить внимание» на зависимость от реализации:

```
unsigned char scode = 87;    // эквивалентно 'W'
```

Для работы с символами из расширенных наборов, таких как UNICODE, применяется тип ***wchar_t***.

Переменные и константы *целых типов* ***int*** также могут объявляться с помощью модификаторов ***signed*** и ***unsigned***. При указании модификаторов ***short*** и ***long*** разрешается опускать имя ***int,*** которое подразумевается по умолчанию. Модификатор ***signed*** также подразумевается по умолчанию.

Размеры целых типов зависят от реализации, но для всех версий языка C++ принято, что выполняется следующая система нестрогих неравенств:

```
sizeof(short) <= sizeof(int) <= sizeof(long).
```

Вещественные типы или типы с плавающей запятой (точкой) представлены тремя размерами, характеризующими точность представления вещественных чисел: ***float*** − одинарной точности; ***double*** − двойной точности; ***long double*** − расширенной точности.

В файлах *limits.h, values.h* и *float.h* определены константы, характеризующие диапазоны значений базовых типов среды выполнения. Используя их, можно создавать программы, пригодные для перенесения в другую среду вы-

полнения. Другой путь построения мобильных (переносимых в другую среду) программ – использование унарной операции *sizeof (имя_типа),* результат выполнения которой равен размеру памяти, необходимой для переменных данного типа. Константы числовых диапазонов среды выполнения можно использовать для создания более корректных вычислительных программ.

Объявление переменных. Все переменные должны быть объявлены до их использования. Общая форма объявления имеет следующий вид:

```
тип имя_переменной;
```

Например, чтобы объявить переменную ***x*** типа ***float***, целочисленную переменную ***y***, символьную ***ch***, логическую ***k*** необходимо записать следующее:

```
float x;
int y;
char ch;
bool k;
```

Используя форму списка, можно в одной записи объявить сразу несколько переменных, разделив их запятыми. Например, следующий оператор объявляет три целые переменные:

```
int a, b, c;
```

Используя спецификатор типа ***typedef,*** можно в своей программе вводить удобные имена для сложных описаний типов. Например:

```
typedef unsigned char cod;
cod p;
```

Здесь определена переменная ***p*** типа ***cod***, который описан спецификатором ***typedef***. Целью такого объявления обычно является введение короткого синонима, определяющего назначение типа данных в программе.

Если в отдельном выражении нужно изменить тип переменной, то можно использовать операцию преобразования типа. Для этого тип данных, который необходим в конкретном случае для переменной, заключается в скобки и записывается в выражении перед именем переменной:

```
float x;
int y=5,z=3;
x=(float)y/z;
```

Инициализация переменных. Переменную можно инициализировать (присвоить ей начальное значение), записав после ее имени знак равенства и начальное значение. Например, следующее объявление позволяет присвоить переменной ***count*** начальное значение ***100*** при объявлении переменной:

```
int count = 100;
```

Инициализатором может быть любое выражение, которое действительно при объявлении переменной. Оно может включать другие переменные и вызовы функций.

Константы базовых типов объявляются так же, как и переменные соответствующих типов, но с указанием модификатора ***const***. Например:

```
const int x=5, y=10;
const double z=1E-5;
const float PI=3.141593;
```

```
const char sf='t';
const char sv[]="Tekst";
```

Константы можно задавать также средствами препроцессора

```
#define HI "Hello"
#define PI 3.14
```

Далее в тексте программы можно использовать только имя **PI**; препроцессор перед компиляцией будет везде его заменять десятичной числовой константой **3.14**. Именно так определены математические и другие числовые константы. Рекомендуется имя константы задавать с помощью заглавных букв, указывая тем самым программисту на константу.

Объекты типа ***const*** не могут быть изменены программой в процессе ее выполнения. Кроме того, объект, адресуемый с помощью указателя, который определен как ***const***, также не может быть модифицирован. Компилятор волен поместить переменные этого типа в память, предназначенную только для чтения (*read-only memory − ROM*). Переменная, определенная как ***const***, получит свое значение либо из явной инициализации, либо посредством выполнения аппаратно-зависимых методов.

Например, в результате выполнения строки

```
const int a =10;
```

будет создана целая переменная с именем ***a*** и со значением ***10***, которое не может быть модифицировано программой. Однако эту переменную можно использовать в выражениях других типов.

3. Структура программы на языке C++

Рассмотрим простейшую программу на языке C++, которая вычисляет произведение двух целых чисел. Полученный результат выводится на экран монитора.

Простейшая программа для среды программирования Builder C++

```
//программа вычисления произведения чисел
//подключается заголовочный файл для
//поддержки системы ввода/вывода C++
#include <iostream.h>
//подключается заголовочный файл библиотеки для работы
//с консольным вводом и выводом – функция  _getch();
#include <conio.h>
//заголовок головной функции программы
int _tmain(int argc, _TCHAR* argv[])
{   // начало тела функции
int a,b;          //объявление переменных a и b
cout<<"a=";       //вывод текста
cin>>a;         //ввод переменной a
cout<<"b=";
cin>>b;
```

```
int y=a*b;
cout<<"y="<<y;    //вывод текста и значения y
getch();          //ожидание нажатия любой клавиши
}   // конец тела функции
```

Простейшая программа для среды программирования Visual Studio C++

```
#include <iostream>
 #include <cmath>
 #include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
int a,b;
cout<<"a=";
cin>>a;
cout<<"b=";
cin>>b;
int y=a*b;
cout<<"y="<<y;
_getch();
return 0;
}
```

Первая строка представляет собой директиву ***препроцессора***, начинается со специального символа # и указывает компилятору, что нужно включить (по-английски ***include*** – включить) информацию, содержащуюся в указанном файле.

Общая структура простой программы на языке C++, содержащей единственную функцию `int _tmain`, представляется в следующем виде:

```
<директивы препроцессора>
int _tmain
{  // начало тела функции
}  // конец тела функции
```

Правила оформления текста программы, направленные на облегчение понимания смысла и повышение наглядности, таковы:

- разделять логические части программы пустыми строками;
- разделять операнды и операции пробелами;
- для каждой фигурной скобки отводить отдельную строку;
- в каждой строке должно быть, как правило, не более одного оператора;
- ограничивать длину строки 60–70 символами;

– отступами слева отражать вложенность операторов и блоков;
– длинные операторы располагать в нескольких строках;
– проводить алгоритмизацию так, чтобы определение одной функции занимало не более одного экрана текста;
– стремиться использовать типовые заготовки фрагментов программ.

4. Операции языка С++

Операции и ***выражения*** задают определенную последовательность действий, но не являются законченными предложениями языка. *Простые выражения* содержат знак операции и операнды. Пример простого выражения с *бинарной* операцией вычитания:

```
3.4 – x.
```

Операции могут проводиться с одним, двумя и тремя *операндами.* Соответственно, различают ***унарные, бинарные*** и ***тернарные*** операции.

Операнд представляет собой элемент-участник операции. Операндами могут быть *константы, переменные, вызовы функций* и *выражения.*

Выражение – это последовательность знаков операций, операндов и круглых скобок, которая определяет вычислительный процесс получения результата определенного типа. *Простейшим выражением* является *константа, переменная* или *вызов функции.* Можно считать, что при выполнении программы результат вычисления выражения заменяет само выражение.

Вызов функции представляет собой указание имени вызываемой функции, за которым в круглых скобках указывается список аргументов (возможно, пустой). Примеры выражений с вызовом функций:

```
a*sin(x+k)+b*cos(x-k)
pow(2,n+1)
```

Можно считать, что во время выполнения программы результат, возвращаемый вызванной функцией, заменяет вызов функции.

Рассмотрим основные операции языка С++.

Арифметические операции

`+,-,*,/` – знаки *бинарных* операций сложения, вычитания, умножения и деления соответственно. Если в операции деления оба операнда целые числа, то результат операции тоже целое число. Следовательно, имеется две разновидности операции деления: *целочисленное деление* (деление с остатком) и *деление без остатка* (результат вещественное число). Например, 5/2 = 2, а 5.0/2 = 2.5.

Для целых чисел определена операция % – нахождение остатка от деления. Например, 5%2 = 1.

Примером *унарной* арифметической операции является операция *изменения знака числа,* обозначаемая символом «минус», который стоит перед одним операндом:

- x + a - 35.2

а также операция определения адреса (&)

Поразрядные операции

Поразрядные (побитовые) операции — это операции над отдельными битами данных и могут применяться только к данным, имеющим тип char или тип int. В поразрядных операциях не могут участвовать данные других типов.

Побитовые логические операции

—& (побитовое И),

—| (побитовое

ИЛИ),

—^ (побитовое исключающее ИЛИ).

~ (двоичное дополнение) позволяют выполнить установку значений битов. Операции производятся по всем известным таблицам истинности.

Пример. Проверка значения бита.

```
char bb=0x64; // = 100 в дес.с.с. и 1100100 - в дв.с.с.
if (bb & 4) cout<<"\nТретий бит равен 1";
else cout<<"\nТретий бит равен 0";
if (bb & 8) cout<<"\nЧетвертый бит равен 1";
else cout<<"\nЧетвертый бит равен 0";
if (bb & 32) cout<<"\nШестой бит равен 1";
else cout<<"\nШестой бит равен 0";
```

Побитовые операции сдвига >> и<< сдвигают все биты переменной, соответственно, вправо или влево. Общая форма операторов сдвига:

переменная >> количество_разрядов

переменная << количество_разрядов

При сдвиге битов в один конец числа, другой конец заполняется нулями.

Операции сдвига можно использовать для быстрого умножения (и деления) на степени числа 2.

Пример. Умножение и деление на степени числа 2.

```
int a,b;
a=3; b=2048;
int n=a<<5; // n= a*32
int m=b>>5; // m=b/32
int x= (a<<3) + (a<<2); // x=a*12 = a*8 + a*4
```

Замечание. В последнем операторе примера показано, как применить операции сдвига при умножении на числа, отличные от степеней 2, в данном случае – умножение на 12. Скобки в операторе x= (a<<3) + (a<<2) необходимы, – это связано с тем, что сложение имеет более высокий приоритет, чем операции сдвига.

Операции присваивания

В результате операции присваивания переменная получает новое значение.

Общая форма оператора присваивания:

идентификатор = выражение;

Оператор присваивания может присутствовать в любом выражении языка, в результате выполнения которого значение правого операнда присваивается левому операнду. Пример выражения с операцией присваивания:

```
v=0.5*a*t*t+7
```

Результатом выполнения операции присваивания является присвоенное значение.

Поэтому возможно следующее выражение *множественного присваивания*

```
z=y=x=t=0
```

В результате вычисления данного выражения переменные *z, y, x, t* примут значение 0*,* если все они одного *типа.*

Пример. Оператор присваивания в арифметическом выражении.

```
int a, b, c;
c=(a=2)+(b=3);
```

Несколько операций присваивания могут быть объединены, например,

```
int i, j;
i = j = 0; //присваивание 0 обеим переменным
i = (j = 3)*2 +7;
```

Такое присваивание называют ***множественным***.

При множественном присваивании вычисления производятся справа налево:

```
i = j= k = m=1;
```

Сначала переменная m получает значение 1, затем k получает значение m, j – значение результата этого присваивания, и в завершении – i получает значение j.

В языках C и C++ используются также ***составные операции присваивания***.

Общий синтаксис составного оператора присваивания:

a op= b;

где op= является одним из операторов: +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=.

Запись a op= b эквивалентна записи a = a op b.

Операции инкремента и декремента

Операции инкремента или, по-русски, увеличения (++) и декремента, т.е. уменьшения (--) дают возможность компактной записи для изменения значения переменной на единицу.

Выражение n++ является *постфиксной* формой оператора инкремента. Значение переменной n увеличивается *после того*, как ее текущее значение употреблено в арифметическом выражении. Аналогично, выражение m-- является *постфиксной* формой оператора декремента.

Существует и *префиксная* форма этих операторов: ++n, --m. При использовании такой формы текущее значение переменной сначала увеличивается или уменьшается и только потом используется в арифметическом выражении.

Пример.

```
int n, m,k,j;
n=m=2;
k=++n * 2; // k=6
j=2 * m++; // j=4
```

Пример. Возможная неоднозначность совместного использования операторов в префиксной и постфиксной формах.

```
#include <iostream>
using namespace std;
int main()
{ int i=2, j=2;
int k=(i++) * (i++) * i; //k=8
int m=(++j) * (j++) * j; //m=?
cout<<"k= "<<k<<" i= "<<i<<" m= "<<m<<" j= "<<j;
return 0;
}
```

Стандартная рекомендация – вместо эквивалентных (по значению) операторов присваивания использовать операторы инкремента и декремента. Это связано с тем, что компиляторы создают для них более эффективный код.

Преобразование типов в операции присваивания

Если в операции присутствуют переменные разных типов, компилятор производит, если это возможно, преобразование типов. Значение выражения в правой части оператора присваивания преобразуется к типу переменной в левой части.

Пример. Потеря информации при преобразовании.

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
int i, j;
double x=0.5;
i=x;
j=1/2+1/2;
return 0;
}
```

Операция "запятая"

Одно выражение может состоять из набора выражений, разделенных запятыми; например,

```
d=(b=a+1),c=a+2;
a=1,2,3;
```

такие выражения вычисляются слева направо. Результатом всего выражения будет результат самого правого выражения списка. Эту операцию, еѐ ещѐ называют ***операцией последовательного вычисления***, используют в тех случаях, когда нужно вычислить несколько выражений, а по правилам синтаксиса допускается только одно выражение, как, например, в операторе for.

Пример. Несколько выражений в выражении инициализации оператора цикла for.

for (**x = 4,u = a[n],k=n-1**;k>=0;k--) u = u*x + a[k];

в данном случае, группа выражений x = 4,u = a[n],k=n-1 рассматривается как одно и имеет значение k=n-1.

Операция запятая обладает самым низким приоритетом, это означает, что оператор,

```
y=1,2;
```

воспринимается как

```
(y=1),2;
```

Пример. Операция запятая – расстановка скобок влияет на результат.

```
using namespace std;
int main()
{
int n,m;
n=m=1,2;
cout<<"\n n="<<n<<" m="<<m<<'\n';
n=(m=1,2);
cout<<"\n n="<<n<<" m="<<m<<'\n';
n=m=(1,2);
cout<<"\n n="<<n<<" m="<<m<<'\n';
return 0;
}
```

Пример. Варианты использования операции **—**запятая

```
double x,y;
x=1.2;
y=1,2;
int a,b,c,d;
a=1,2,3;
d=(b=a+1),c=a+2;
```

Преобразование типов

В арифметических операциях все операнды предварительно приводятся к одному типу. Чтобы избежать потери точности, используется принцип перехода от операнда меньшего типа к большему, например, если встречаются переменные, принадлежащие к типам int, float и double, то производится преобразование к типу double. Этот процесс преобразования типов называется *продвижением типов*(*type promotion*)

(подробнее, см., напр., Шилдт Г. *Полный справочник по C* и *Программирование на C и C++*. –http://cpp.com.ru/).

В C++ имеется операция *явного приведения типа*, т.е. можно указать, к какому типу необходимо преобразовать значение выражения. Общая форма оператора явного приведения типа:

(Тип) выражение

Пример. Явное приведение типа в арифметическом выражении.

```
double x;
int m;
cin>>m;
x= (double) m/2 + (double)7/3;
```

В языке C++ используется также приведение типа в форме вызова функции

Тип(выражение)

Пример.

```
double x;
int m;
cin>>m;
x= double(m/2) + double(7)/3;
```

Операции сравнения и логические операции

<, >, ==, >=, <=, != являются знаками *операций отношения* (меньше, больше, равно, больше или равно, меньше или равно, не равно), используемыми в *логических выражениях;*

&&, ||, ! – знаки *логических операций* И, ИЛИ и НЕ;

Условная тернарная операция

? и : – знаки, используемые в *условной тернарной операции.* Например, результатом выражения `y=(a>0)?a:0` будет значение y = *a,* если *a* > 0 и нулевое значение y = 0 в противном случае.

*, & – знаки *адресных операций;*

, – знак специальной операции, задающей последовательность выражений («*операция запятая*»). Например, выражение `i=1, j=4, k=7;` состоит из трех простых выражений.

[*i*] – квадратные скобки, используются при работе с массивами для задания размеров массива и доступа к элементу массива;

( ) – скобки, используемые для изменения очередности выполнения операций в выражениях и при указании аргументов функций;

sizeof (имя_типа) или *sizeof выражение* – операция вычисления размера типа или значения в байтах. Например, *sizeof a* – размер памяти в байтах для хранения переменной *a*.

Приоритеты операций

Приоритеты операций задают последовательность вычислений в сложном выражении.

В C++ умножение и деление имеют более высокий приоритет, чем сложение, поэтому они будут вычислены раньше. Их собственные приоритеты равны, поэтому умножение и деление будут вычисляться слева направо. Самый низкий приоритет у операции —запятая.

Далее приведен список операций (с рядом из них предстоит еще познакомиться) в порядке убывания приоритета:

( ) (скобки функций), [ ] (скобки для индексов массивов),->, .(операции доступа к элементам структур),

!, ~,++,--,+(унарный плюс),-(унарный минус), (type),*(операция указателя—разыменование),

& (операция адреса), sizeof ,

*,/,%,+,-

<<, >>,

<,<=,>, — >=,

= =,!=,

&,^,|,&&,||,?:,

=,+=,-=, —*,=,/=,%=, , .

Порядок вычисления выражения определяется расположением знаков операций, круглых скобок и приоритетами выполнения операций. Например, результат вычисления выражения (11 + 25/5 + 7) не равен результату вычисления выражения (11 +25)/(5+7).

Выражения с наивысшим приоритетом вычисляются первыми. Приоритеты операций приведены в табл. 1. Операции, находящиеся в начале таблицы, имеют более высокий приоритет.

Если в выражении содержится несколько операций одного приоритета на одном и том же уровне, то их обработка производится в соответствии с порядком выполнения. Например:

```
y=x/2+a*5%15;        // y=(x/2)+((a*5)%15)
```

Как видим, смысл символа операции зависит от контекста, т. е. тех условий, в которых используется операция.

Компилятор по *имени операции* и *типам операндов* определяет возможность ее выполнения и необходимые для этого машинные действия и свои собственные действия. После выполнения ***операции*** получается результат определенного *типа,* подставляемый в текст программы вместо знака операции и операндов, участвующих в данной операции. Например, можно считать, что в результате выполнения операции умножения ***(t · 5.2)*** заменится ***15.6****,* если ***t = 3***.

Таблица 1

Приоритеты и порядок выполнения операций

Приоритет	Знаки операций	Названия операций	Порядок выполнения
1	: :	расширение области видимости	слева
2	. –> [ ] ( ) + + – – *typedef*	выбор элемента по имени выбор элемента по указателю выбор элемента по индексу вызов функции или конструирование значения постфиксный инкремент постфиксный декремент идентификация типа	слева
3	*sizeof* + + – – ~ ! + – & * *new* *delete* (имя_типа)	размер операнда в байтах префиксный инкремент префиксный декремент инверсия (поразрядное НЕ) логическое НЕ унарный плюс унарный минус адрес разыменование (разадресация) выделение памяти или создание освобождение памяти или уничтожение преобразование типа	справа
4	.* –>*	выбор элемента по имени через указатель выбор элемент по указателю через указатель	слева
5	* / %	умножение деление остаток от деления целых	слева

		(деление по модулю)	

Окончание табл.1

Приоритет	Знаки операций	Названия операций	Порядок выполнения
6	+ –	сложение вычитание	слева
7	<< >>	сдвиг влево сдвиг вправо	слева
8	< > <= >=	меньше больше меньше или равно больше или равно	слева
9	= = ! =	равно не равно	слева
10	&	поразрядное И	слева
11	^	поразрядное исключающее ИЛИ	слева
12	|	поразрядное ИЛИ	слева
13	&&	логическое И	слева
14	||	логическое ИЛИ	слева
15	?:	условная	справа
16	= *=, /=, %=, +=,–=, <=, >>=, &=,^=, |=	простое присваивание составные присваивания	справа
17	*throw*	генерация исключения	справа
18	,	последовательность выражений	слева

Операторы так же, как операции и выражения, задают определенную последовательность действий компилятора или вычислительной машины, но, в отличие от выражений, являются законченными предложениями языка. Они могут содержать несколько выражений, разделенных запятой.

Обычно операторы заканчиваются *точкой с запятой. Пустой оператор* содержит только точку с запятой. Простые операторы можно объединить в *составной оператор* или *блок* с помощью фигурных скобок.

Операторы имеют названия, отражающие их назначение или осуществляемые действия. Так, оператор, в котором используется операция присваивания, обычно называют ***оператором присваивания.*** Например, `y=(a*x+b)*x+c;`

Примером *оператора-выражения* является `count--;`

При его выполнении значение переменной *count* уменьшится на единицу.

Операторы, в которых объявляются константы, переменные, заголовки функций и типы данных, называются *операторами объявления.* Имеются операторы с другими названиями.

Операторы, задающие действия вычислительной машины, могут быть помечены идентификатором, который называется *меткой* и заканчивается двоеточием. Метки используются для перехода к ним с помощью оператора ***go-to*** *(безусловного перехода)*.

УСЛОВНЫЕ ОПЕРАТОРЫ

1. Оператор условия if
2. Тернарная операция
3. Оператор выбора switch
4. Логические основы алгоритмизации

В лекции рассматриваются различные способы реализации разветвляющегося вычислительного процесса. Для программирования разветвлений в языке С++ предназначены условный оператор (if... else), тернарная операция (?:), оператор выбора (switch).

1. Оператор условия if

Разветвляющийся вычислительный процесс в простейшем случае описывается в языке С++ с помощью оператора, имеющего следующий формат:

if(выражение) оператор1;
[else оператор2;]

Здесь оператор может быть как простым, так и составным.

Условный оператор ***if*** используется для организации перехода после принятия решения. Он позволяет управлять ходом вычислительного процесса в зависимости от значения (истинно или ложно) выражения-условия. При этом ***else***-часть может быть (рис. 1, *а*), а может и отсутствовать (рис. 1, *б*).

Выполнение оператора ***if*** начинается с вычисления выражения, которое может представлять собой комбинацию операций отношения (**<,>,<=,>=,==,!=**) и логических выражений (***И*** – **&&**, ***ИЛИ*** – **||** , ***НЕ*** – **!**).

Например:

(A>B&& B<C)

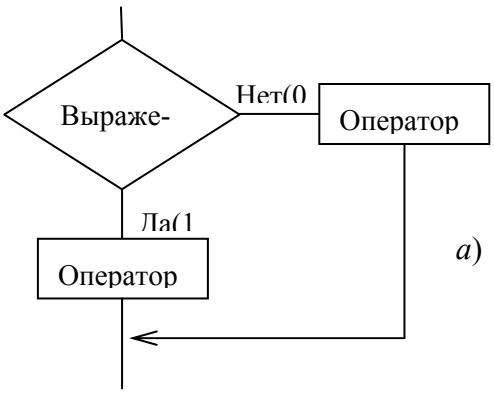
Далее выполнение идет по следующей схеме (рис. 1):

– если выражение истинно (т.е. отлично от нуля), то выполняется ***оператор1***;

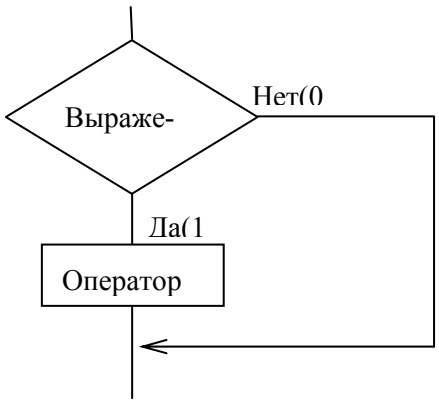
– если выражение ложно (т.е. его значение равно 0), то выполняется ***оператор2*** (рис. 1, *а*);

– если выражение ложно и отсутствует ключевое слово ***else*** (рис. 1, *б*), то выполняется следующий за ***if*** оператор программы.

После выполнения оператора ***if*** управление передается на следующий оператор программы.



а)



б)

Рис. 1. Блок-схема выполнения оператора ***if***

Пример 1. Составить программу на языке С++ для расчета значения переменной *y*. Значение переменных *a* и *b* целого типа ввести с клавиатуры.

$$y = \begin{cases} a + b, \text{ если } a \le b \\ a - b, \text{ если } a > b \end{cases}$$

Решение:

```
#include<conio.h>
#include<iostream.h>
int  _tmain()
{
 int a, b, y;
cout<<"Введите значение a и b: ";
cin>>a>>b;
 if (a<=b) y=a+b;
 else y=a-b;
cout<<"\ny="<<y;
 getch();
}
```

Пример 2. Составить программу расчета частного двух вещественных чисел; в случае деления на ноль вывести соответствующее сообщение. Исходные данные ввести с клавиатуры.

Решение:

```
int _tmain()
{
float a,b,d;
cout<<"Введите два числа:";
cin>>a>>b;
 if(b= =0)
cout<<"Отношение не определено \n";
 else
{
d=a/b;
cout<<"Отношение =\n"<< d;
}
getch();
}
```

В приведенном примере в предложении ***if*** используется один оператор, а в предложении ***else*** – составной оператор из двух операторов.

Допускается использование вложенных операторов ***if***. Оператор ***if*** может быть вложен во фразу ***if*** или во фразу ***else*** другого оператора ***if***.

Чтобы сделать программы более понятными, рекомендуется группировать операторы и фразы во вложенных операторах ***if***, используя фигурные скобки. Если же фигурные скобки опущены, компилятор связывает каждое ключевое слово ***else*** с наиболее близким ***if***, для которого нет ***else***. Например:

```
. . .
int a=2,b=7,c=3;
if (a>b)
{
     if(b<c)
       c=b;
}
else
        c=a;
cout<<"c="<<c;
. . .
```

Результат выполнения программы: ***c=2***.

Если в этой программе опустить фигурные скобки в операторе ***if***, то программа будет иметь следующий вид:

```
. . .
int a=2,b=7,c=3;
if(a>b)
   if(b<c)
      c=b;
else
       c=a;
cout<<"c="<<c;
. . .
```

В этом случае ключевое слово ***else*** относится ко второму оператору ***if***.

Результат выполнения программы: ***c=3***.

Отметим, что здесь в ***if*** используется полное выражение. Если оно не содержит знаков операций отношения или тождества, то выполняется проверка на равенство выражения нулю, и условие можно записывать в сокращенном виде. Так, запись ***if(p)*** эквивалентна ***if(p!=0).*** Например:

```
#include<conio.h>
#include<iostream.h>
int _tmain()
{
float a;
a=0.6;
if(a)
cout<<"истина a="<<a <<"\n";
else cout<<"ложь "<<a;
a=0;
if(a)
cout<<"ложь a="<<a;
else cout<<"истина a="<<a;
getch();
}
```

Часто встречается конструкция вида:

if (выражение)
оператор;
else if (выражение)
оператор;
else if (выражение)
оператор;
else if (выражение)
оператор;
else оператор;

Приведенная последовательность инструкций ***if*** – самый общий способ описания многоступенчатого принятия решения. Выражения вычисляются по порядку; как только встречается ***выражение*** со значением «истина», выполняется соответствующий ему ***оператор***; На этом последовательность проверок завершается. Здесь под словом ***оператор*** имеется в виду либо один оператор, либо группа операторов в фигурных скобках.

Последняя ***else***-часть срабатывает, если все предыдущие условия не выполняются. Иногда в последней части не требуется производить никаких действий, в этом случае фрагмент

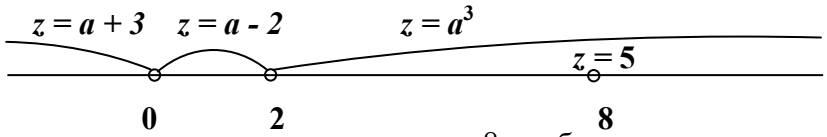
else оператор;

можно опустить или использовать для фиксации ошибочной (невозможной) ситуации.

Пример 3. Составить программу на языке С++ для расчета значения переменной z. Значение переменной a вещественного типа ввести с клавиатуры.

$$z = \begin{cases} 5, \text{ если } a = 8, \\ a + 3, \text{ если } a \le 0, \\ a - 2, \text{ если } 0 < a < 2, \\ a^3, \text{ если } a \ge 2. \end{cases}$$

Для решения задач такого типа удобно представить возможные варианты на координатной оси.



В данном случае вариант для значения $a = 8$ необходимо рассматривать прежде, чем для интервала $a \ge 2$.

Решение:

```
#include<conio.h>
// подключается заголовочный файл библиотеки
//математических функций
#include<math.h>
#include<iostream.h>
int _tmain()
{
float a, z;
cout<<" a=";
cin>>a;
 if (a= =8) z=5;
 else if (a<=0) z=a+3;
```

```
 else if (a>0&&a<2) z=a-2;
 else  z=pow(a,3);   //a в степени 3
cout<<" z="<<z;
getch();
}
```

Инструкцию ***if*** целесообразно использовать, когда рассматриваются интервалы возможных значений переменных. В случае если вариант определяется конкретным значением целого или символьного типа, подобный подход нерационален.

Пример 4. Составить программу для определения значения переменной *x* путем действия над переменными *y* и *z* в соответствии с введенным с клавиатуры знаком операции: −, + или *.

Решение:

```
. . .
char sign;
int x,y,z;
cin>>sign>>y>>z;
if(sign= ='-')
   x=y-z;
else if (sign= ='+')
   x=y+z;
else if (sign= ='*')
   x=y*z;
else cout<<"неверный знак операции\n";
cout<<x;
```

Приведенный фрагмент программы подтверждает, что такие конструкции получаются длинными и не всегда понятными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора ***switch***.

2. Тернарная операция

В языке С++ имеется короткий способ записи оператора *if… else.* Для этого используют тернарную операцию (операцию условия). Она имеет следующую форму записи:

(условное выражение) ? выражение1 : выражение2

Если условное выражение истинно, то выполняется *выражение 1*, если ложно – *выражение 2.*

Пример 5. Составить программу на языке С++ для расчета значения переменной *y*.

$$y = \begin{cases} a + b, \text{ если } a \le b, \\ a - b, \text{ если } a > b. \end{cases}$$

Решение:

```
int _tmain()
{
int a,b;
cout<<"a=";
cin>>a;
cout<<"b=";
cin>>b;
```

```
int y=a<=b?a+b:a-b;
cout<<"\ny="<<y;
getch();
}
```

Тернарную операцию удобно использовать в случаях выбора значения из двух возможных. Применение этой операции не является обязательным, так как тех же результатов можно достичь при помощи оператора if... else. Однако получаемые при использовании операции условия выражения более компактны и их применение приводит к получению более компактного машинного кода.

Пример 3а. Составить программу на языке С++ для расчета значения переменной *z* с использованием тернарной операции.

$$z=\begin{cases}5, \textit{если } a=8\\ a+3, \textit{ если } a\le 0\\ a-2, \textit{если } 0<a<2\\ a^3, \textit{если } a\ge 2\end{cases}$$

Решение:

```
. . .
float a, z;
cout<<" a=";
cin>>a;
bool A=a==8, B=a<=0, C=a>0&&a<2;
z=A?5:(B?a+3:(C?a-2:pow(a,3)));
cout<<" z="<<z;
. . .
```

3. Оператор выбора switch

Оператор ***switch*** вызывает передачу управления к одному из нескольких операторов, в зависимости от значения выражения.

Формат:

switch(выражение)
{
[объявление]
. . .
[case константа 1: оператор 1; [break;]]
. . .
[case константа k: оператор k; [break;]]
[default: оператор;]
}

Выражение, следующее за ключевым словом ***switch***, может быть любым, допустимым в языке С++, в котором проводятся обычные арифметические преобразования. Результат выражения должен быть целочисленным.

Значение этого выражения является критерием для выбора из нескольких вариантов. Тело оператора ***switch*** состоит из нескольких операторов, помеченных ключевым словом ***case*** с последующей константой (целая или символьная константа).

Все константы в операторе ***switch*** должны быть уникальны (не могут иметь одинаковое значение). Кроме операторов, помеченных ключевым словом ***case***, может быть, но не обязательно, один фрагмент, помеченный ключевым словом ***default***.

Оператор 1 ... Оператор k могут быть пустыми либо содержать один или более операторов. Причем их не требуется заключать в фигурные скобки.

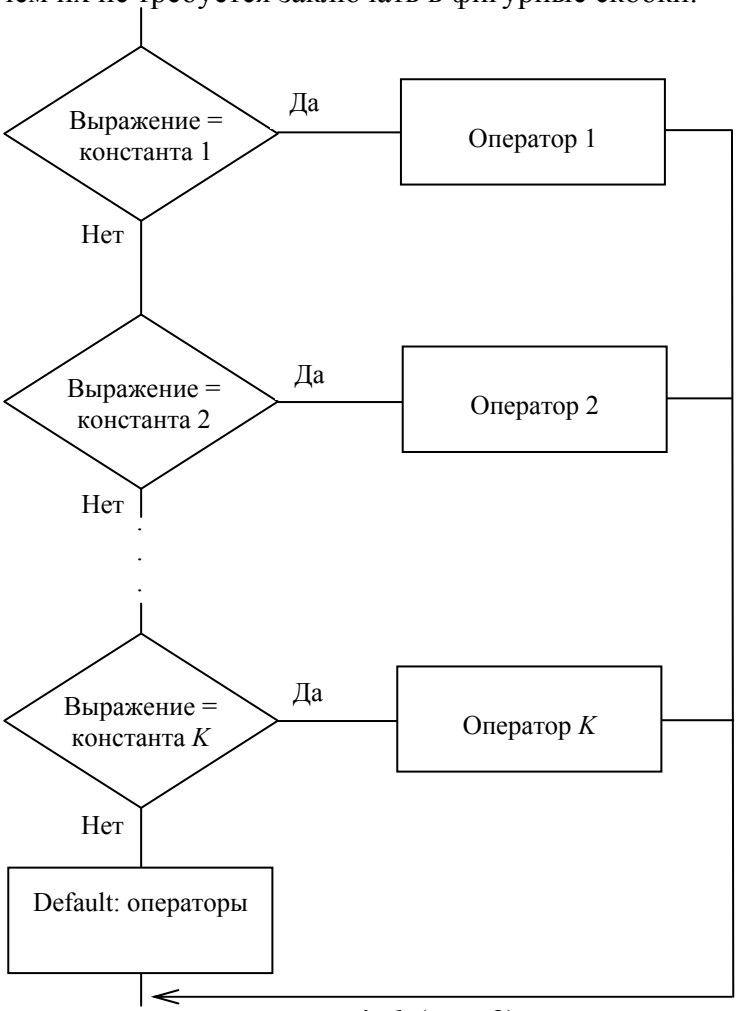


Схема выполнения оператора ***switch*** (рис. 2):

– вычисляется выражение в круглых скобках;

– вычисленное значение последовательно сравнивается с константными выражениями, следующими за ***case***;

– если сравнение не найдено и есть вариант ***default***, то управление передается на оператор, помеченный ключевым словом ***default***;

– если одно из константных выражений оказывается равным значению выражения в круглых скобках, то управление передается на оператор, следующий за совпадающим вариантом;

– если сравнение не найдено и варианта ***default*** нет, то управление передается на следующий за ***switch*** оператор.

Вариант ***default*** может быть не последним в теле оператора ***switch***. Ключевые слова ***case*** и ***default*** определяют метки, после которых в обязательном порядке ставится двоеточие.

Метки ***case*** и ***default*** в теле инструкции ***switch*** существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора ***switch***, и не изменяют поток управления. Все операторы между начальным оператором и концом тела оператора ***switch*** выполняются вне зависимости от меток, если только какой-то из операторов не передаст управление из тела оператора ***switch.*** Таким образом, программист должен сам позаботиться о выходе из ***case***, если это необходимо.

Наиболее распространенным средством немедленного выхода из оператора ***switch*** является оператор переходов ***break***.

Пример 6. Составить программу на языке С++ для расчета значения переменной *y*, где

$$y=\begin{cases} x+2, \text{если } x=1, \\ x+5, \text{если } x=2, \\ 1, \text{если } x=7, \\ 0, \text{в остальных случаях.} \end{cases}$$

Решение:

```
int _tmain()
{
int x;
int y;
  cin>>x;
switch(x)
{
case 1: y=x+2; break;
case 2: y=x+5; break;
case 7: y=1; break;
default:y=0; break;
}
  cout<<"y="<<y;
getch();
}
```

при $x = 1$ выполняется $y = x + 2$;
при $x = 2$ – $y = x + 5$;
при $x = 7$ – $y = 1$;
в остальных случаях – $y = 0$;

Пример 6а. Составить программу на языке С++ для расчета значения переменной *y*, где

$$y=\begin{cases} x+2, \textit{ если } x=1 \textit{ или } x=8 \\ x+5, \textit{ если } x=2 \textit{ или } x=3 \textit{ или } x=5 \\ 1, \textit{ если } x=7 \\ 0, \textit{ в остальных случаях} \end{cases}$$

Решение:

```
int _tmain()
{
int x, y;
  cin>>x;
switch(x)
{ case 1:
  case 8: y=x+2; break;
  case 2:
  case 3:
```

```
  case 5: y=x+5; break;
  case 7: y =1; break;
  default: y=0; break;
}
cout<<"y="<<y;
getch();
}
```

4. Логические основы алгоритмизации

Важной составляющей алгоритмов являются *логические условия*. Вычисление значений логических условий происходит в соответствии с аксиомами *алгебры логики*.

Начало исследований в области формальной логики связывают с работами Аристотеля в IV в. до н. э. Однако математические подходы к этим вопросам впервые были указаны Джорджем Булем, который положил в основу математической логики алгебру логики (в честь ее создателя алгебру логики называют *булевой алгеброй,* а логические значения – *булевыми*). Алгебра логики используется при построении основных узлов ЭВМ – шифратора, дешифратора, сумматора.

Алгебра логики оперирует с высказываниями. Под высказыванием понимают повествовательное предложение, относительно которого имеет смысл говорить, истинно оно или ложно. Например, выражение «Расстояние от Москвы до Киева больше, чем от Москвы до Тулы» истинно, а выражение «4 < 3» – ложно.

Высказывания принято обозначать большими буквами латинского алфавита: *A*, *B*, *C*... *Z*. Если высказывание *C* истинно, то пишут *C* = 1 (*C* = *t*, true), а если оно ложно, то *C* = 0 (*C* = *f*, false).

В алгебре высказываний над высказываниями можно производить определенные логические операции, в результате которых получаются новые высказывания. Истинность полученных высказываний зависит от истинности исходных высказываний и использованных для их преобразования логических операций.

Для образования новых высказываний наиболее часто используются логические операции, выражаемые словами **НЕ**, **И**, **ИЛИ**.

Конъюнкция (логическое умножение). Соединение двух (или нескольких) высказываний в одно с помощью союза **И** называется операцией, логического умножения, или конъюнкцией. Эту операцию принято обозначать знаками «$\wedge$, &» или знаком умножения. Сложное высказывание *A* & *B* истинно только в том случае, когда истинны оба входящих в него высказывания. Истинность такого высказывания задается следующей таблицей:

A	*B*	*A* & *B*
false	false	false
false	true	false
true	false	false
true	true	true

Дизъюнкция (логическое сложение). Объединение двух (или нескольких) высказываний с помощью союза **ИЛИ** называется операцией логического сложения, или дизъюнкцией. Эту операцию обозначают знаками «$\vee$, || » или знаком сложения. Сложное высказывание *A* || *B* истинно, если истинно хотя бы одно из входящих в него высказываний. Таблица истинности для логической суммы высказываний имеет вид

A	*B*	*A* || *B*
false	false	false
false	true	true

true	false	true
true	true	true

Инверсия (логическое отрицание). Присоединение частицы **НЕ (NOT)** к данному высказыванию называется операцией отрицания (инверсии). Она обозначается $\neg A$ и читается *не A* . Если высказывание *A* истинно, то *B* ложно, и наоборот. Таблица истинности в этом случае имеет вид

A	$\neg A$
false	true
true	false

Высказывания, образованные с помощью логических операций, называются сложными. Истинность сложных высказываний можно установить, используя таблицы истинности. Например, истинность сложного высказывания $\neg A$ & $\neg B$ определяется следующей таблицей:

A	B	$\neg A$	$\neg B$	$\neg A$ & $\neg B$
false	false	true	true	true
false	true	true	false	false
true	false	false	true	false
true	true	false	false	false

Высказывания, у которых таблицы истинности совпадают, называются равносильными. Для обозначения равносильных высказываний используют знак «=» ($A = B$).

В алгебре высказываний можно проводить тождественные преобразования, заменяя высказывания равносильными.

Исходя из определений дизъюнкции, конъюнкции и отрицания, устанавливаются свойства этих операций и взаимные распределительные свойства.

Коммутативность (перестановочность):

$A \wedge B = B \wedge A$;

$A \vee B = B \vee A$.

Законы де Моргана

$\neg(A \wedge B) = \neg A \wedge \neg B$ (условно его можно назвать первым);

$\neg(A \vee B) = \neg A \vee \neg B$ (второй) – описывают результаты отрицания переменных, связанных операциями И, ИЛИ.

Пример. Требуется сформулировать логическое выражение для отбора информации о курсантах, у которых в дате рождения стоит декабрь 1992 г. или январь 1993 г. Построим сначала простые высказывания, а затем объединим их в логическое выражение:

A – «Месяц рождения декабрь»;

C – «Месяц рождения январь»;

B – «Год рождения 1992»;

D – «Год рождения 1993»;

Логическое выражение на базе простых высказываний имеет следующий вид:

((A & B) || (C & D))

ОПЕРАТОРЫ ПЕРЕХОДОВ И ЦИКЛОВ

1. Операторы переходов
2. Операторы циклов

В лекции рассматриваются различные способы реализации циклического вычислительного процесса, а также возможности использования операторов перехода. Для программирования циклов в языке С++ предназначены операторы: ***for***, ***while***, ***do while***.

1. Операторы переходов

Операторы переходов осуществляют безусловную передачу управления.

Оператор break. Оператор ***break*** встречается в теле оператора ***switch*** и в операторах цикла. Он завершает работу самой внутренней циклической или ***switch***-конструкции, содержащей ***break***, после чего управление переходит к следующему оператору. Подробно порядок использования оператора ***break*** в циклах будет рассмотрен ниже.

Оператор return. Оператор ***return*** завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию. Управление передается в вызывающую функцию в точку, непосредственно следующую за вызовом.

Формат оператора ***return***:

return [выражение];

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое функцией значение не определено. Выражение может быть заключено в круглые скобки.

Если оператор ***return*** отсутствует в вызываемой функции, то управление автоматически передается в вызывающую функцию после выполнения последнего оператора функции. Возвращаемое функцией значение в этом случае не определено. Если функция не возвращает значения, ее следует объявить с типом ***void***.

Таким образом, оператор ***return*** используется в двух случаях:

– если надо немедленно выйти из функции;

– если функции должна возвращать значение.

Пусть имеется пользовательская функция с именем ***imin***, которая позволяет произвести выбор между двумя альтернативами (и результирующими значениями), основанный на некотором условии. Обычно это реализуется условным оператором ***if***. Например, так:

```
int   imin(int a, int b)
{
if(a<b) return(a);
else return (b);
}
```

Функция ***imin*** возвращает меньшее из двух чисел. Оператор ***return*** используется для возврата значения.

Этот же пример можно записать, используя тернарную операцию:

```
int imin(int a, int b)
{
return((a<b)?a:b);
}
```

Более того, можно даже записать ***imin*** как строку директивы препроцессора:

```
#define imin(a,b) ((a<b)?a:b)
```

Рассмотрим другой пример:

```
void print(char a)
{
if(x==0)
 {
```

```
    cout<<"Это плохой аргумент \n";
    return;
   }
    cout<<"Введен аргумент"<<a;
}
```

В этом случае оператор ***return*** используется для выхода из функции в случае, если аргумент равен нулю.

Оператор goto. Хотя для написания структурированных программ не рекомендуется использовать операторы перехода, в языке С++ имеется оператор перехода **goto**.

Формат оператора goto:

goto имя;

. . .

имя: оператор

Оператор ***goto*** передает управление на оператор с меткой ***имя***. Помеченный оператор должен находиться в той же функции, что и оператор ***goto***, и метка должна быть уникальна.

Если помеченный оператор встречается в любом другом контексте, он выполняется без учета метки.

Метка - это идентификатор. Каждая метка должна отличаться от других меток той же функции.

Любой оператор в составном операторе может иметь метку. Используя оператор ***goto***, можно передать управление внутрь составного оператора. Однако надо быть осторожным, передавая управление внутрь составного оператора, так как составной оператор может содержать объявление переменных с инициализацией. Так как обычно объявления располагаются в составном операторе до выполнимых операторов, то при передаче управления внутрь составного оператора на выполнимый оператор будет обойдена инициализация. Результат в этом случае непредсказуем.

Существует лишь один случай, когда программисты профессионалы допускают использование goto, — это выход из вложенного набора циклов при обнаружении ошибок *(break* дает возможность выхода лишь из одного цикла).

Оператор continue. Оператор ***continue*** работает подобно оператору ***break***, но в отличие от него прерывает выполнение тела цикла и передает управление на следующую итерацию.

Формат оператора continue:

continue;

Этот оператор подробно будет рассмотрен ниже при описании циклических вычислительных процессов.

Оператор ***continue*** прерывает самый внутренний из объемлющих его циклов.

2. Операторы цикла

Циклическим называется вычислительный процесс, содержащий участок с многократным исполнением.

Повторяющийся многократно участок программы называется ***циклом***.

Цикл выполняется до тех пор, пока выполняется некоторое ***условие*** – условие продолжения цикла. Если выполняется всегда, то цикл бесконечный (зацикливание программы). В зависимости от условия осуществляется выход из цикла либо его повторное выполнение обычно с новыми значениями исходных данных. Цикл может повторяться заранее известное количество раз (с известным количеством повторений) и неизвестное количество раз. Условие выхода из цикла может проверяться либо до, либо после самого цикла.

Для программирования циклических вычислительных процессов можно использовать рассмотренные ранее операторы ***goto*** и ***if***. Это не является рациональным способом решения и обычно не рекомендуется, но позволяет наглядно продемонстрировать организацию цикла.

Пример 1. Составить программу для расчета таблицы умножения числа 239 на все числа в интервале от 5 до 100, используя операторы ***goto*** и ***if***.

Решение:

```
#include <conio.h>
#include <iostream.h>
int  _tmain()
{
       int i,k=239;

       i=5;
  lm:      cout<<k<<"*"<<i<<"="<<k*i<<"\n";
       i++;
       if (i<=50) goto lm;

       getch();
}
```

Однако в языке С++ имеются специальные операторы цикла ***while***, ***for*** и ***do-while***.

2.1. Оператор while

Оператор цикла ***while*** является наиболее общим и может использоваться вместо других типов циклических конструкций. В принципе можно сказать, что по-настоящему для программирования необходим только оператор ***while***, а другие типы циклических конструкций служат лишь для удобства написания программ.

Оператор ***while*** имеет следующий формат:

инициализация переменной цикла;
while (условие продолжения цикла)
{
оператор;
изменение значения переменной цикла;
}

где ***оператор*** – тело цикла (представляет собой оператор или составной оператор).

Таким образом, пока истинно условие продолжения цикла, выполняется оператор и изменяется значение переменной цикла. Для описания условие продолжения цикла, используются те же операции отношения и логические выражения, что и для оператора условия ***if***. Выражение обязательно должно стоять в круглых скобках.

Схема выполнения оператора цикла ***while***:

1) устанавливается начальное значение переменной цикла;

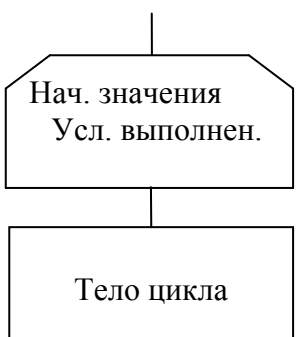
2) вычисляется выражение–условие продолжения цикла;

3) если выражение «ложно» (равно нулю), то тело цикла не выполняется, а управление передается на следующий за ***while*** оператор;

4) если выражение «истинно» (не нуль), то тело цикла выполняется и изменяется значение переменной цикла;

5) процесс повторяется с шага 2.

Блок-схема выполнения оператора цикла ***while***:



Из блок-схемы видно, что тело цикла не выполнится ни разу, если ***выражение*** – «ложь».

Пример 2. Составить программу на языке С++ для расчета значения переменной *y*:

$$y=\sum_{i=1}^{10}\left[ax^2+m\right], \text{ где } x=\begin{cases}1+m^i, \text{если } i \text{ четное},\\ i+m, \text{если } i \text{ нечетное}.\end{cases}$$

Решить задачу, если *i* изменяется от 1 до 10 с шагом 1 при *a* = 1, *m* = 2.5. Исходные данные ввести явной инициализацией.

Решение:

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
int _tmain()
{
double y=0, x, m=2.5;
int i, a=1;
     i=1;
     while (i<=10)
{
   if (i%2= =0) x=1+pow(m,i);
   else x=i+m;
   y=y+a*x*x+m;
   i=i+1;
}
     cout<<"y="<<y;
     getch();
}
```

Оператор ***while*** удобно использовать в ситуациях, если тело цикла не всегда надо выполнять. Он заменяет необходимость отдельной проверки перед циклом.

2.2. Оператор for

Оператор цикла ***for*** является одним из основных операторов цикла, которые имеются во всех универсальных языках программирования, включая С++. Однако версия цикла ***for***, используемая в С++, обладает большей мощностью и гибкостью.

Основная идея, заложенная в его функционирование, заключается в том, что операторы, находящиеся внутри цикла, выполняются фиксированное число раз, в то время как переменная цикла (известная еще как индексная переменная) пробегает определенный ряд значений.

Приведем основной формат оператора цикла ***for***:

for(выражение 1; выражение 2; выражение 3)

оператор;

Так же, как и в цикле ***while***, ***оператор*** в теле цикла ***for*** обычно является либо оператором, либо составным оператором.

Заметим, что параметры оператора цикла ***for***, заключенные в круглые скобки, должны разделяться точкой с запятой (***;***), которая делит, в свою очередь, пространство внутри скобок на три сектора. Каждый параметр, занимающий определенную позицию (позиционный параметр), означает следующее:

выражение 1 устанавливает начальное значение переменной цикла (выполняет ее инициализацию);

выражение 2 задает условие продолжения цикла;

выражение 3 изменяет значение переменной цикла;

оператор – оператор или составной оператор, которые выполняются неоднократно (тело цикла).

Выполнение оператора цикла ***for*** можно представить в виде блок-схемы:

Отсюда видно, что переменная цикла инициализируется перед циклом и изменяется каждый раз после выполнения цикла.

Схема выполнения инструкции ***for***:

1) вычисляется ***выражение 1***, используемое для установки начального значения переменных;

2) вычисляется ***выражение 2***;

3) если значение выражения 2 не равно нулю (истина), выполняется ***оператор***;

4) вычисляется ***выражение 3***;

5) вновь вычисляется ***выражение 2***;

6) как только ***выражение 2*** становится равным нулю, управление передается на оператор, следующий за телом цикла ***for***.

Существенно то, что проверка ***выражения 2*** (условия цикла) всегда выполняется в начале цикла. Это значит, что цикл может ни разу не выполниться, если ***выражение 2*** сразу будет ложным.

Пример 3. Составить программу для расчета значения функции *z*:

$z = x^a + e^{-x}$,

в точках x = 0,5; 0,7; 0,9; …; 1,5. Значение переменной a = **5** ввести с клавиатуры.

Решение:

```
#include <conio.h>
#include <iostream.h>
#include <math.h>
int   _tmain()
{
float a,x,z;
    cout<<"vvod a\n";
    cin>>a;
for(x=0.5;x<1.7;x=x+0.2)
{
z=pow(x,a)+ exp(-x);
    cout<<"x="<<x<<"\t z="<<z<<"\n";
}
   getch();
}
```

Основной вариант оператора цикла ***for*** эквивалентен следующей конструкции, выраженной с помощью оператора цикла ***while***:

выражение 1;
while(выражение 2)
{
оператор;
выражение 3;
}

Допускаются некоторые варианты оператора цикла ***for***, которые повышают его гибкость в определенных ситуациях.

В операторе цикла for можно опускать одно, несколько или даже все выражения, однако о необходимости наличия точек с запятой нужно помнить всегда. Если опустить ***выражение 2***, то это будет равносильно тому, что значение ***выражения 2*** всегда будет иметь значение 1 (истина) и цикл никогда не завершится (бесконечный цикл). Пример простейшего бесконечного цикла:

for(;;) оператор;

Для выхода из такого цикла обычно используется оператор ***break***:

```
int _tmain()
{
char n;
for(;;)
{
cout<<"Введите еще раз или символ f\n";
cin>>n;
if (n= ='f') break;
}
getch();
}
```

Другим вариантом использования оператора цикла ***for*** является использование операции ***запятая***, которая позволяет вводить несколько переменных в тело цикла.

Пример 4. Составить программу для расчета значения функции $y = \sum b + c$, если b изменяется от 5 до 10 с шагом 1, а c – от 2 до 17 с шагом 3.

Решение:

```
int _tmain()
{
int b,c,y=0;
```

```
   for(b=5,c=2;b<=10;b=b+1,c=c+3)
{
      y=b+c;
      cout<<"\n b="<<b<<"\t c="<<c<<"\t y="<<y;
}
getch();
}
```

Здесь ***выражение 1*** и ***выражение 3*** состоят из двух выражений, инициализирующих и модифицирующих переменные ***b*** и ***c***.

2.3. Оператор do while

Формат оператора цикла ***do while***:

инициализация переменной цикла;
do
{
оператор;
изменение значения переменной цикла;
}
while (условие продолжения цикла);

Тело цикла выполняется до тех пор, пока выражение в скобках не примет ***ложное*** значение.

Блок-схема выполнения оператора цикла ***do while***:

Порядок выполнения цикла ***do while***:

1) выполняется ***оператор***, изменяется значение переменной цикла;

2) вычисляется ***выражение***. Если выражение не равно нулю (истина), то выполнение продолжается с шага 1. Если выражение равно нулю (ложно), то выполнение передается следующему оператору программы.

Основное отличие цикла ***while*** от цикла ***do while*** состоит в том, что операторы внутри ***do while*** всегда выполняются хотя бы один раз (так как проверка условия выполнения цикла осуществляется после выполнения последовательности операторов, составляющих тело цикла).

Пример 5. Составить программу для расчета значений функции *y*: $y = \prod_{i=3}^{\infty} \frac{e^i}{i}$, где *i* изменяется от 3 с шагом 2 . Решить задачу, учитывая лишь те элементы произведения, для которых выполняется условие: $e^i / i \le 10\,000$.

Решение:

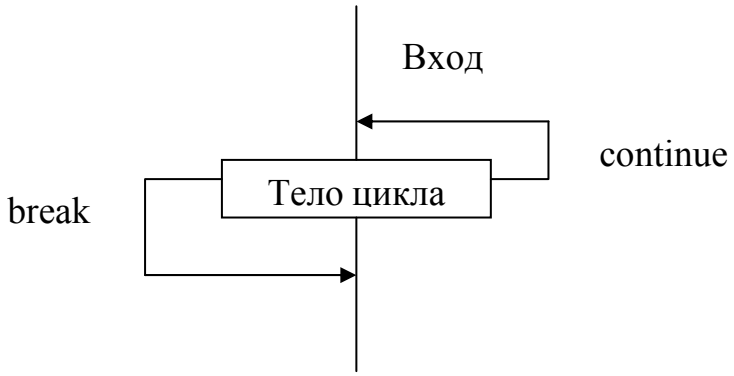
```
int _tmain()
{
double y=1.0,i;
i=3.0;
  do
{
y=y*(exp(i)/i);
i+=2;
}
     while((exp(i)/i)<=10000);
cout<<"\t"<<y;
getch();
}
```

Чтобы прервать цикл до того, как условие станет ложным, можно использовать инструкцию ***break***.

В теле всех рассмотренных операторов цикла могут быть использованы операторы переходов:

– ***break*** (немедленный выход из цикла);

– ***continue*** (прекращение очередной и начало следующей итерации):



Оператор ***break*** является наиболее важным из этих трех операторов и уже встречался при рассмотрении оператора выбора *switch.* Оператор *break* может использоваться в циклах всех трех типов. Выполнение оператора *break* приводит к выходу из цикла, в котором он содержится, и переходу к следующему за циклом оператору. Если

оператор *break* находится внутри вложенных циклов, то его действие распространяется только на тот цикл, непосредственно в котором он находится.

Пример 1. Использование оператора *break.*

Требуется определить задуманное число с 10 попыток.

```
int i=1,rez;
while    ( i++<=10 )
{
cin>>rez;
if   ( rez==15 ) break;
cout<<"\n Popytka neud.\n"; }
```

```
if   ( i!=12 )   cout<<"\nVy ugadali!";
```

В этом примере при угадывании числа происходит прекращение выполнения цикла с помощью оператора *break.*

Оператор ***continue*** может использоваться только среди операторов тела цикла. Этот оператор вызывает пропуск оставшейся части итерации внутри цикла и переход к следующей итерации.

Пример 2. Использование оператора *continue.*

Вводятся числа месяца для обработки. Необходимо осуществить проверку правильности ввода. Число 31 обозначает конец обработки.

```
int den;
while    ( den!=31)
{cin>>den;
if   (den<1||den>31)   continue; } ...  // Обработка числа den
```

В данном примере неправильный ввод значения приводит к пропуску части итерации, предназначенной для обработки этого значения.

Заметим, что такое изменение условия *if* `(den<1||den>31)` на обратное *if(den>0&&den<32)* позволяет исключить использование *continue.* С другой стороны, с помощью *continue* иногда можно сократить некоторые программы, особенно если они включают в себя вложенные операторы *if... else.*

При программировании циклических вычислительных процессов следует учитывать следующее:

1) выражение выхода из цикла должно быть составлено таким образом, чтобы оно когда-нибудь выполнялось, иначе будет зацикливание;

2) в теле цикла могут быть другие циклы – вложенные.

Пример 6. Составить программу на языке С++ для расчета значений функции y: $y = b^2 a^3$, если b изменяется от –1 до 11 с шагом 1, переменная c от 5 до 15 с шагом 2.

Решение:

```
int _tmain()
{
int y,a,b;
    for(b=-1;b<=11;b++)
    for(a=5;a<=15;a=a+2)
{
y=b*b+a*a*a;
cout<<"\n b="<<b<<"\t a="<<a<<"\t y="<<y;
}
getch();
}
```

Пример 7. Необходимо вывести на экран заполненный символами * прямоугольный треугольник, высота которого равна N.

Решение:

```
int _tmain()
{
int i,j,N;
cout<<"Введите N\n";
cin>>N;
i=1;
while(i<=N)
{
 j=1;
 while (j<=i)
```

```
    {
      cout<<'*';
     j=j+1;
     }
 i=i+1;
 cout<<"\n";
}
getch();
}
```

Во внешнем цикле устанавливается очередная строка вывода (параметр ***i***), а во внутреннем (параметр ***j***) в очередную строку выводится ровно ***i*** символов ***.

Результат выполнения программы:

Введите N
5

```
    *
    **
    ***
    ****
   *****
```

Исследование возможностей использования оператора for.

В языке С++ оператор цикла *for* является более гибким средством, чем аналогичные операторы циклов в других языках программирования, например, Паскале. Эта гибкость является следствием использования трех выражений в скобках после *for.* Кроме описанных выше, существует еще много других возможностей применения этого типа циклов. Рассмотрим некоторые из них.

1. Можно применять операцию уменьшения для счета в порядке убывания.

Пример 1. Счет в порядке убывания.

Требуется вычислить $r = y^5$. Возможное решение имеет вид:

```
{int i,y,r;
for  (i=5,y=2,r=1;  i>=1;  i-- )
r=r*y;
     cout<<"r="<<r<<"\n";}
```

При желании можно организовать счет двойками, тройками, десятками и т. д.

Пример 2. Приращение при счете, отличное от 1.

```
for  ( int n=5;  n<61;  n+=15)   cout<<n<<"\n";
```

Можно в качестве счетчика использовать не только цифры, но и символы.

Пример 3. Использование символов в качестве счетчика.

Требуется напечатать алфавит. Возможное решение имеет вид:

```
for  ( char chr='A';  chr<='Z';  chr++)  cout<<chr<<"\t";
```

4. Можно проверять любое другое условие, отличное от числа итераций.

5. Можно задать возрастание значений счетчика не в арифметической, а в геометрической прогрессии.

1. В качестве третьего выражения можно использовать любое правильно составленное выражение. Оно будет вычисляться в конце каждой итерации.

Пример 5. Использование в качестве счетчика выражения.

```
int z=0;
for  ( int k=1;   z<=196;  z+=5*k+23 )   cout<<z<<"\n";
```

7. Можно пропускать одно или несколько выражений. (При этом нельзя пропускать символы «точка с запятой».)

Пример 6. Неполный список выражений в заголовке тела цикла.

```
float n=2,k=5,p;
for(p=2; p<=202;)  p=p+n/k; cout<<p;
```

8. Первое выражение не обязательно должно инициализировать переменные, оно может быть любого типа.

Пример 7. Произвольное первое выражение в заголовке цикла.

```
int p=1;  for( cout<<"vvod thisel:"; p<=30;p++)  cin>>p;
```

9. Переменные, входящие в выражения спецификации цикла, можно изменять в теле цикла.

Пример 8. Изменение управляющих переменных в теле цикла.

```
. . . .
delta=0.1;
for  (k=l;  k<500;  k+=delta)        if  (a>b)     delta=0.5;
. . . .
```

10. Использование операции «запятая» в спецификации цикла позволяет включать несколько инициализирующих и корректирующих выражений.

Пример 9. Использование операции «запятая» в спецификации цикла.

```
int y=2;
for  (int i=1,   r=1;  i<=10;  i++,   r*=y )
cout<<y<<" ^ "<<i<<" = "<<r<<"\n";
```

Большая свобода выбора вида выражений, управляющих работой цикла *for,* позволяет с помощью этого оператора делать гораздо больше дополнительных действий, чем выполнять просто фиксированное число итераций.