Глава 4
АЛГОРИТМИЗАЦИЯ И ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

В процессе изучения данной главы рекомендуется установить следующее программное обеспечение:

для операционной системы Windows:

- систему объектно-ориентированного программирования Microsoft Visual Studio;

для операционных систем Windows и Linux:

- систему объектно-ориентированного программирования Lazarus.

4.1. Алгоритм и кодирование основных алгоритмических структур

4.1.1. Алгоритм и его свойства

Алгоритмы могут описывать процессы преобразования самых разных объектов. Широкое распространение получили вычислительные алгоритмы, которые описывают преобразование числовых данных. Само слово «алгоритм» происходит от *algorithmi* — латинской формы написания имени выдающегося математика IX века аль-Хорезми, который сформулировал правила выполнения арифметических операций.

Алгоритм обладает следующими свойствами.

Результативность. Выполнение алгоритма должно завершиться за конечное число шагов.

Дискретность. Алгоритм должен обеспечивать преобразование объекта из начального состояния в конечное состояние за определённое число дискретных шагов.

Массовость. Один и тот же алгоритм может применяться к большому количеству однотипных объектов.

Детерминированность (определённость). Исполнитель должен выполнять команды алгоритма в строго определённой последовательности. Каждое действие алгоритма должно быть строго определено, не допускать разночтений. Алгоритм должен выдавать один и тот же результат для разных исходных данных.

Понятность. Алгоритм должен содержать только команды, входящие в систему команд исполнителя и записанные на понятном для исполнителя языке.

Алгоритм — это строго детерминированная последовательность действий, описывающая процесс преобразования объекта из начального состояния в конечное (исходных данных в результат), записанная с помощью понятных исполнителю команд.

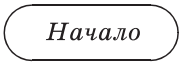
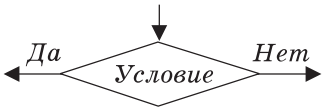
Существуют разные способы записи алгоритмов: словесный (на естественных языках), графический (блок-схемы), описание на языках программирования.

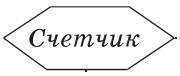
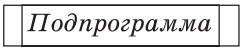
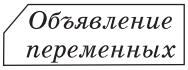
Блок-схемы алгоритмов. Блок-схема представляет собой графическую форму записи алгоритмов. Она позволяет сделать алгоритм более наглядным и выделить в нём основные алгоритмические структуры (линейная, ветвление и цикл). Если исполнителем алгоритма является человек, то он может по блок-схеме легко проследить выполнение алгоритма, так как элементы блок-схем соединены стрелками, указывающими шаги выполнения алгоритма.

Элементы алгоритма изображаются на блок-схеме с помощью различных геометрических фигур, внутри которых записывается программный код (табл. 4.1).

Таблица 4.1

Элементы блок-схем

Элемент блок-схемы	Назначение элемента блок-схемы
	Прямоугольник с закруглёнными углами применяется для обозначения начала или конца алгоритма
	Параллелограмм предназначен для описания ввода или вывода данных, имеет один вход вверху и один выход внизу
	Прямоугольник применяется для описания линейной последовательности команд, имеет один вход вверху и один выход внизу
	Ромб служит для обозначения условий в алгоритмических структурах «ветвление» и «выбор», имеет один вход верху и два выхода (налево, если условие истинно, и направо, если условие ложно). Применяется также для составления алгоритмической структуры «цикл»

Элемент блок-схемы	Назначение элемента блок-схемы
	Шестиугольник используется в алгоритмической конструкции «цикл со счётчиком»
	Прямоугольник в прямоугольнике применяется для вызова отдельно описанного алгоритма (подпрограммы)
	Прямоугольник со срезанным углом применяется для объявления переменных или ввода комментариев

Вопросы и задания

Какие из нижеперечисленных правил являются алгоритмами:
- орфографические правила;
- правила выполнения арифметических операций;
- правила техники безопасности;
- правила перевода чисел из одной системы счисления в другую?

Ответ обоснуйте.

4.1.2. Алгоритмические структуры «ветвление» и «цикл»

Алгоритмическая структура ветвление. В отличие от линейных алгоритмов, в которых команды выполняются последовательно одна за другой, в алгоритмическую структуру ветвление входит условие: в случае истинности этого условия реализуется одна последовательность команд, а в случае ложности — другая.

В **алгоритмической структуре ветвление** одна или другая серия команд выполняется в зависимости от истинности **условия.**

Алгоритмическая структура ветвление может быть зафиксирована графически с помощью блок-схемы (рис. 4.1). В блок-схеме на рис. 4.1 альтернативные последовательности команд обозначены словами *Серия 1* и *Серия 2*.

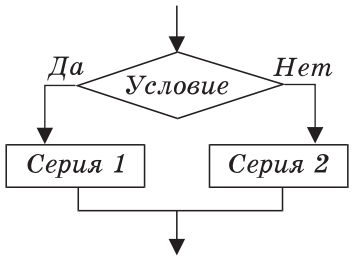


Рис. 4.1

На языках объектно-ориентированного программирования, которые мы будем рассматривать в этой главе, алгоритмическая структура ветвление

кодируется с использованием оператора **If**. После первого ключевого слова **If** должно быть записано условие. После ключевого слова **Then** (в языке Visual C# оно отсутствует) идёт последовательность команд (*Серия 1*), которая должна выполняться, если условие принимает значение «истина». После ключевого слова **Else** размещается последовательность команд (*Серия 2*), которая должна выполняться, если условие принимает значение «ложь».

В сокращённой форме оператора ключевое слово **Else** отсутствует. Тогда, если условие ложно, выполнение оператора условного перехода заканчивается и выполняется следующая строка программы.

Для реализации ветвления со многими вариантами серий команд используется алгоритмическая конструкция выбор. Конструкция «выбор» может быть зафиксирована графически с помощью блок-схемы (рис. 4.2).

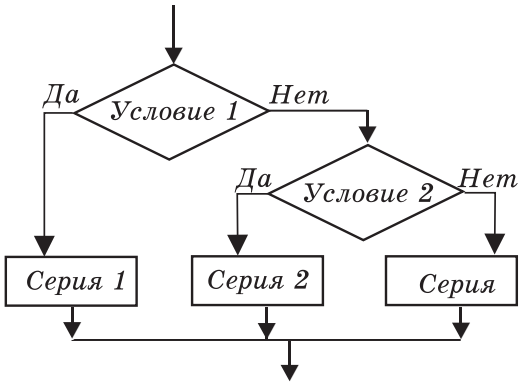


Рис. 4.2

В структуру выбора входят несколько **условий**, проверка которых осуществляется по порядку их записи в структуре выбора. При истинности одного из условий (*Условие 1*, *Условие 2* и т. д.) выполняется соответствующая последовательность команд (*Серия 1*, *Серия 2* и т. д.). Если ни одно из условий не является истиным, то будет выполнена последовательность команд *Серия*.

В алгоритмической конструкции **выбор** выполняется одна из нескольких последовательностей команд при истинности соответствующего условия.

На языках объектно-ориентированного программирования конструкция выбор кодируется с использованием оператора

выбора. На языке программирования Visual Basic .NET оператор выбора начинается с ключевых слов **`Select Case`**, на языке Visual C# — с ключевого слова **`switch`**, а на языке Lazarus — с ключевого слова **`Case`**.

После ключевого слова записывается выражение (переменная или арифметическое выражение). Заданное выражение сравнивается с определёнными значениями. При истинности одного из условий начинает выполняться соответствующая серия команд. Если ни одно из условий не истинно, то выполняется серия команд после ключевого слова **`Else`** (в языках Visual Basic .NET и Lazarus) или ключевого слова **`default`** (в языке Visual C#).

В сокращённой форме оператора ключевое слово **`Else`** (**`default`**) отсутствует. Тогда, если все условия ложны, выполнение оператора выбора заканчивается и выполняется следующая строка программы.

Алгоритмическая структура цикл. В алгоритмическую структуру цикл входит серия команд, выполняемая многократно. Такая последовательность команд называется **телом цикла**.

В **алгоритмической структуре цикл** серия команд **(тело цикла)** выполняется многократно.

Циклические алгоритмические структуры бывают двух типов:

- циклы со счётчиком, в которых тело цикла выполняется определённое количество раз;
- циклы по условию, в которых тело цикла выполняется, пока истинно (или ложно) заданное условие.

Алгоритмическая структура цикл может быть описана графически с помощью блок-схемы (рис. 4.3).

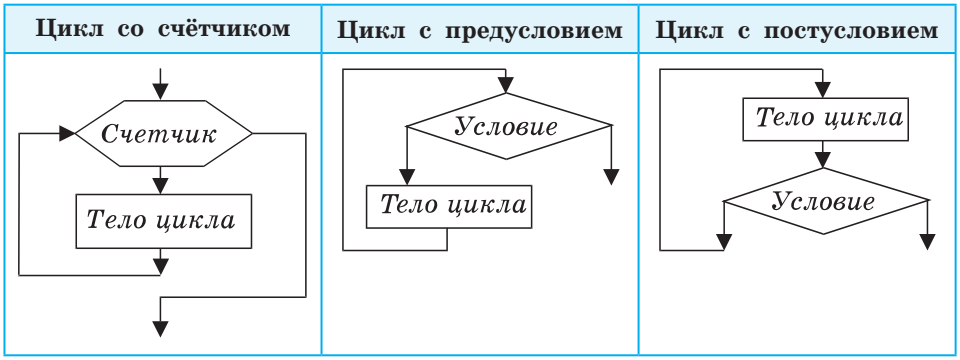


Рис. 4.3

Цикл со счётчиком используется, когда заранее известно, какое количество повторений тела цикла необходимо выполнить. Количество повторений задаётся с помощью счётчика.

Цикл со счётчиком реализуется при помощи оператора **For**. В заголовке цикла устанавливается начальное значение переменной **Счётчик**, определяется величина её конечного значения и величина изменения значения за один шаг. Затем располагаются многократно выполняемые операторы тела цикла.

Цикл с условием используется, когда заранее неизвестно, какое количество раз должно повториться тело цикла. В таких случаях количество повторений зависит от некоторого условия.

Цикл называется **циклом с предусловием**, если условие выхода из цикла стоит в начале, перед телом цикла. В случае ложности условия цикл с предусловием не выполнится ни разу.

В языках объектно-ориентированного программирования цикл с предусловием реализуется с помощью оператора **While** (в языке Visual Basic .NET — **Do While**). Проверка условия выхода из цикла проводится до начала цикла с помощью ключевого слова **While**, которое обеспечивает выполнение цикла, пока условие истинно. Как только условие примет значение «ложь», выполнение цикла закончится.

Цикл называется **циклом с постусловием**, если условие выхода из цикла стоит в конце, после тела цикла. Цикл с постусловием выполняется обязательно, как минимум один раз, независимо от того, истинно условие или нет.

Цикл с постусловием реализуется с помощью оператора **Do** (в языке Lazarus — **Repeat**). Проверка условия выхода из цикла производится после цикла с помощью ключевого слова **Until**. Как только условие примет значение «истина», выполнение цикла закончится. В языке Visual Basic .NET используется также ключевое слово **While**. Оно обеспечивает выполнение цикла до тех пор, пока условие не станет ложным, т. е. пока условие имеет значение «истина». Как только условие примет значение «ложь», выполнение цикла закончится.

Пример

Данный пример демонстрирует использование алгоритмических структур.

Рассмотрим *алгоритм перевода целых десятичных чисел в двоичную систему счисления на естественном языке*:

1) Ввести десятичное целое число.
2) В цикле с предусловием, пока исходное целое десятичное число или целое частное больше 0, выполнить вычисления:
   2.1) Вычислить остаток от деления исходного целого десятичного числа или целого частного на основание новой системы (на 2).
   2.2) Выполнить целочисленное деление целого десятичного числа или целого частного на основание новой системы (на 2).
   2.3) Записать полученный остаток от деления слева от двоичного числа (остатки, записанные в обратном порядке, образуют двоичное число).
3) Вывести двоичное целое число.

На рисунке 4.4 изображена блок-схема этого алгоритма. Команды в блоках записаны на языке Visual Basic .NET.

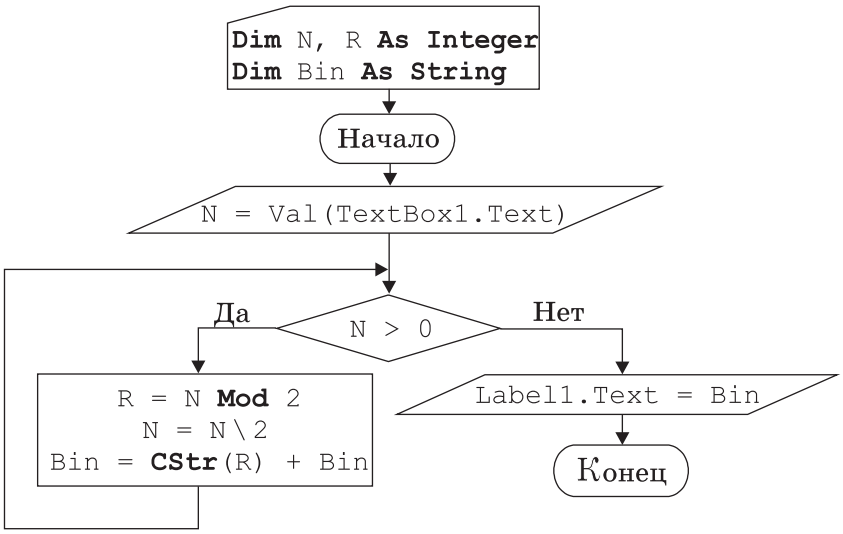


Рис. 4.4

Вопросы и задания

1. Какие типы алгоритмических конструкций использованы в приведённом в параграфе алгоритме перевода десятичных чисел в двоичное представление?

2. Опишите алгоритм перевода чисел из двоичной системы счисления в десятичную. Оформите ответ в форме блок-схемы для числа 1011.
3. Как работает автомат для покупки газет и журналов? Объясните его работу с использованием алгоритмической конструкции проверки условия при выборе издания и выдаче сдачи автоматом. Оформите ответ с помощью блок-схемы.
4. Приведите примеры использования алгоритмической конструкции проверки условия в различных приборах. Какой алгоритм управляет датчиком включения и отключения света в подъезде дома («умный свет») или автоматическими дверями в магазине? Оформите свой пример с помощью блок-схемы.
5. В различных социальных сетях и журналах часто используются опросы или тесты. Опишите с помощью блок-схемы алгоритм автоматического тестирования при условии, что в тесте предлагается пять вопросов и на каждый вопрос — три возможных ответа. Тестируемый может выбрать только один из предложенных вариантов ответа. Каждый ответ имеет свой балл, который учитывается в суммарном балле тестируемого. По итогам прохождения теста набранный балл сравнивается со шкалой результатов, и для тестируемого на экран выводится сообщение в соответствии с тем диапазоном баллов, в который попал суммарный результат.

   Используйте описание алгоритма на естественном языке, предложенное ниже, для построения блок-схемы «Тестирование».

   i — счётчик цикла обработки пяти вопросов (i меняется от 1 до 5).
   n — номер ответа (n меняется от 1 до 3).
   1) **Начало цикла:** Для i от 1 до 5:
      1.1) Вывести на экран вопрос i и три возможных ответа, перенумерованных как 1, 2, 3 и баллы для ответов ($B1$, $B2$, $B3$).
      1.2) Ввести с клавиатуры номер ответа n.
      1.3) В ячейку суммы S добавить балл, соответствующий выбранному ответу ($S = S + Bn$).
   2) **Конец цикла по i**
   3) Если $S < X1$, то вывести сообщение *1*, если $S >= X1$ *AND* $S < X2$, то вывести сообщение *2*, если $S >= X2$, то вывести сообщение *3*.

4.1.3. Подпрограммы. Рекурсивные алгоритмы

Подпрограммы. Во многих алгоритмах те или иные действия могут повторяться для различных исходных данных. Например, пусть нужно составить алгоритм для вычисления площади круговой пластины с круглыми отверстиями (рис. 4.5). Очевидно, для этого нужно вычислить площадь круга, соответствующего внешнему контуру пластины, и вычесть из нее сумму площадей круговых отверстий.

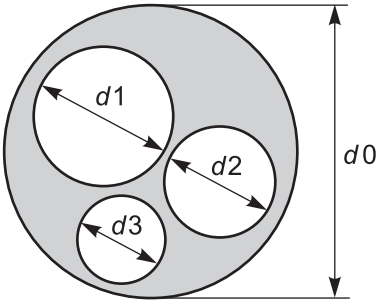


Рис. 4.5

В алгоритме для вычисления площади пластины потребуется записать четыре одинаковых вычислительных блока, в которых рассчитывается площадь круга диаметром, соответственно, $d0$, $d1$, $d2$, $d3$ (рис. 4.6).

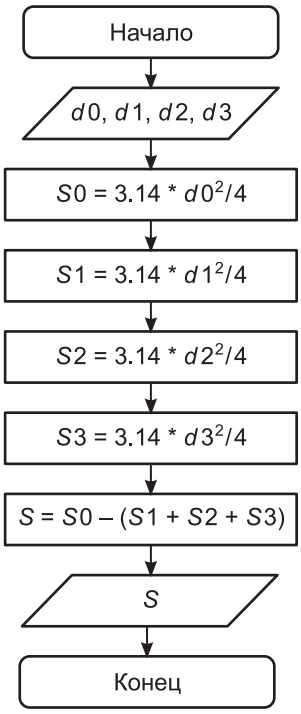


Рис. 4.6

Существует возможность упростить такой алгоритм, вынеся из него однотипные действия в отдельный, подчинённый алгоритм и обращаясь к подчинённому алгоритму из основного (главного) алгоритма с требуемыми исходными данными (рис. 4.7). Такой подчинённый алгоритм реализуется в виде **подпрограммы**. Когда подпрограмма завершает работу, происходит *возврат* из неё в основную программу, которой передаются результаты работы подпрограммы.

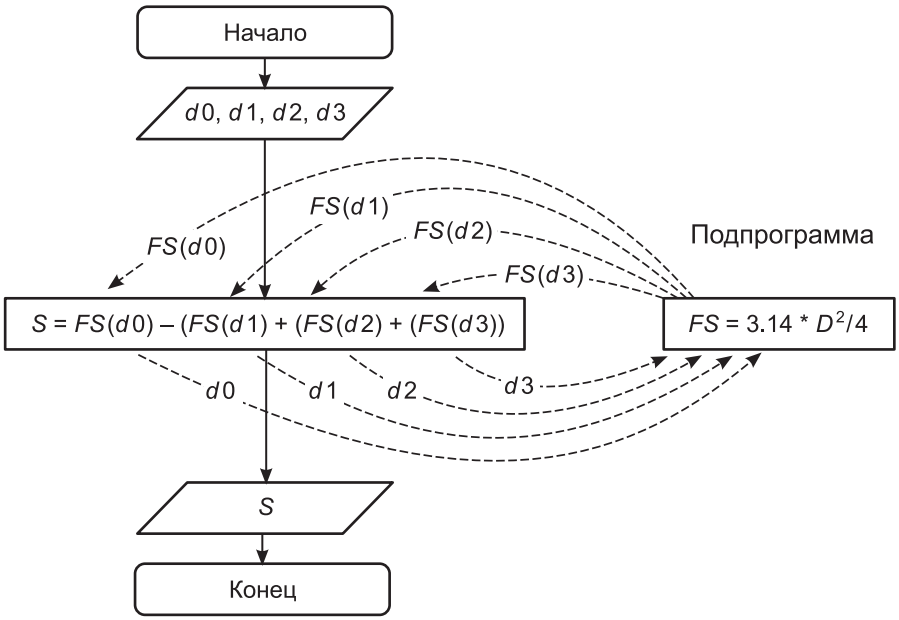


Рис. 4.7

Процедуры и функции. Возможны два вида подпрограмм, различающиеся принципами их вызова и количеством возвращаемых ими значений.

Функция — это подпрограмма, которая, принимая одно или несколько исходных данных, возвращает единственное значение, являющееся результатом работы этой функции. Такая подпрограмма аналогична математическим функциям (например, тригонометрическим, которые по заданному значению угла вычисляют его синус, косинус, тангенс и т. д.). Обращение к функции (вызов функции) обычно записывается так же, как это делается в математике, например:

`X = SIN(A)` — вызов функции, вычисляющей значение синуса угла, величина которого задана в переменной *A*; результат, возвращенный этой функцией, записывается в переменную *X*;

`Y = SQRT(1 - COS(A)*COS(A))` — реализация вычисления значения синуса угла по формуле $\sqrt{1 - \cos^2(A)}$: здесь результаты, возвращаемые функциями вычисления косинуса угла A, сразу же используются в арифметическом выражении, значение которого, в свою очередь, затем используется в качестве исходных данных для функции вычисления квадратного корня (SQRT).

Обычно при написании алгоритма подпрограммы-функции после выполнения всех вычислений полученные результаты нужно записать в специальную переменную. Именно её значение возвращается в основной алгоритм в качестве результата работы функции.

В современных языках программирования предусматривается большое количество различных готовых функций (**стандартных**, или **библиотечных**), которые достаточно вызывать из основного алгоритма. Однако при необходимости можно создавать свои собственные (**пользовательские**) функции, реализующие любой алгоритм.

Процедура — подпрограмма, которая может принимать любое количество исходных данных и возвращать любое количество значений в качестве результатов своей работы. Обращение к процедуре в основном алгоритме записывается отдельной командой: и исходные данные, и переменные, в которые должны быть записаны результаты работы процедуры, записываются в скобках списком через запятую (сначала перечисляются исходные данные, а затем — переменные-результаты). Вызовы процедур не могут использоваться в арифметических выражениях (в отличие от функций).

Формальные и фактические параметры. Для удобства и большей универсальности при написании подпрограмм переменные, используемые в алгоритме подпрограммы, обычно полностью независимы от переменных, используемых в основном алгоритме (даже если совпадают имена этих переменных). Принято считать, что подпрограмма использует отдельное *адресное пространство*. Соответственно, переменные, используемые внутри подпрограммы, называют **локальными**.

Чтобы обеспечить передачу данных из основного алгоритма в подпрограмму и возврат из подпрограммы результатов её работы, используется механизм формальных и фактических параметров.

- При написании подпрограммы в её заголовке записывается «шаблон» вызова этой подпрограммы. Он включает имя подпрограммы и записанный в скобках список локальных переменных, используемых внутри подпрограммы. Эти переменные представляют собой передаваемые в подпрограмму *исходные* (*входные*) данные, а для процедуры — также список локальных переменных, в которые в процедуре записываются резуль-

таты вычислений. Все эти переменные, записанные в заголовке подпрограммы, называют **формальными параметрами.**

- При вызове подпрограммы из основного алгоритма в строке обращения к подпрограмме записывается её имя, а в скобках — список передаваемых ей реальных исходных значений и (для процедуры) список переменных, в которые после выполнения процедуры будут записаны результаты её работы. В качестве исходных данных могут быть записаны константы, переменные, арифметические выражения или подпрограммы-функции. Все эти исходные данные и переменные для записи результатов называют **фактическими параметрами.**

Для правильной работы подпрограммы необходимо соблюдать правило: *количество, порядок записи и типы формальных и фактических параметров должны совпадать*[1)].

Например:

	Имя подпрограммы	1-е исходное данное (целое число)	2-е исходное данное (вещественное число)	3-е исходное данное (текстовый символ)	1-й результат (число)	2-й результат (текст)
Заголовок подпрограммы:	PODPROG	(A,	X,	T,	Y,	S)
Правильный вызов подпрограммы:	PODPROG	(5,	2.3,	"@",	Z,	TXT)
Неправильный вызов подпрограммы:	PODPROG	("$",	3,	2.5,	"&",	7)
Суть ошибки:		Формальный параметр — целое число, а фактически задан символ	Формальный параметр — вещественное число, а фактически задано целое число	Формальный параметр — текстовый символ, а фактически задано вещественное число	В качестве фактического параметра должна быть записана переменная	В качестве фактического параметра должна быть записана переменная

1) Кроме особых случаев, когда часть фактических параметров может быть пропущена. Такая возможность реализуется особыми приёмами программирования. Вместо отсутствующих фактических параметров в подпрограмме тогда используются некие заранее оговоренные и занесённые в алгоритм подпрограммы значения «по умолчанию».

Процедуры — обработчики событий. Современные программы для компьютеров обычно пишутся по объектному принципу. Рассматривается некий набор **объектов** — как отображаемых визуально (изображаемые на экране окна, кнопки и другие элементы интерфейса), так и не имеющих визуального представления (например, аудиозапись). Каждый такой объект может иметь свой набор **свойств**, значения которых можно задать при создании объекта, а затем изменять при выполнении программы. Кроме того, рассматривается некоторый набор возможных **событий** — ситуаций, как возникающих при работе самого компьютера и его периферийных устройств (скажем, замятие бумаги в принтере), так и инициированных пользователем (например, в качестве события может выступать щелчок кнопкой мыши или нажатие определённой клавиши на клавиатуре).

Для каждого объекта и каждого возможного события, совершаемого над ним, разрабатывается отдельная процедура — алгоритм, определяющий действия компьютера при возникновении того или иного события. Такая процедура называется **обработчиком события**.

При отсутствии событий компьютер находится в «ждущем» режиме — не выполняет никаких действий, которые можно было бы наблюдать со стороны. Если же для какого-то объекта произойдёт то или иное событие, для которого была предусмотрена процедура-обработчик, то компьютер запустит в работу именно её. Если произойдёт несколько событий для одного и того же или для разных объектов (одновременно или поочерёдно, когда новое событие происходит до завершения обработки предыдущего), то компьютер может запустить несколько соответствующих обработчиков, — если только их выполнение не противоречит друг другу.

Рекурсивные алгоритмы. Возможна ситуация, когда в ходе выполнения подпрограммы её алгоритмом предусматривается вызов самой этой же подпрограммы. Такой приём называют **рекурсией**, а подобные алгоритмы с «самовызовом» называют рекурсивными алгоритмами.

Пример использования рекурсии — вычисление факториала.

Факториалом натурального числа называют значение, равное произведению этого числа на все числа натурального ряда, меньшие данного числа. Факториал обозначается восклицательным знаком, записанным после исходного числа. Например, факториал числа 4 вычисляется так: $4! = 4 \cdot 3 \cdot 2 \cdot 1$.

Создадим для решения этой задачи рекурсивный алгоритм в виде подпрограммы, которой в качестве исходного данного будет передаваться значение 4.

Предположим, что мы уже знаем, чему равен факториал предыдущего натурального числа. Тогда для вычисления 4! достаточно умножить это уже известное значение 3! на число 4.

Однако мы пока не знаем, чему равен 3!. Вычислить его мы можем... вызвав эту же самую подпрограмму, которую мы пишем для вычисления факториала числа 4. Но теперь в эту подпрограмму в качестве исходного данного передается число 3. И вычисление значения 3! производится по тому же принципу: число 3 умножается на 2!.

Поскольку величина 2! нам тоже пока неизвестна, её мы вычислим снова вызовом этой же подпрограммы, которой будет передано уже число 2. И здесь вычисление производится тоже умножением числа 2 на факториал предыдущего числа 1 (и тоже вызовом нашей же подпрограммы с передачей ей исходного данного — числа 1).

Но вот чему равен факториал единицы, мы знаем точно: он равен единице. И этот факт можно отразить в нашей подпрограмме при помощи оператора ветвления типа: «ЕСЛИ переданное число равно 1, то результат равен 1».

Запишем теперь (на псевдокоде; это условный язык — смесь языка программирования и естественного (русского, английского) языка) соответствующую подпрограмму:

```
FAKTOR(N) // N - формальный параметр, соответствующий
          // заданному числу
    ЕСЛИ N = 1 ТО РЕЗУЛЬТАТ = 1
    ИНАЧЕ РЕЗУЛЬТАТ = N * FAKTOR(N-1)
                                  // рекурсивный вызов
КОНЕЦ ПОДПРОГРАММЫ
```

А теперь посмотрим, как работает этот алгоритм, например, для числа 4 (из основного алгоритма производится вызов подпрограммы в виде `F = FAKTOR(4)`). Для этого будем отслеживать изменение значения переменной N (рис. 4.8).

Как видим, в процессе работы подпрограммы сначала формируется цепочка её *вложенных вызовов* — до тех пор, пока не сработает **условие завершения рекурсии** (в данном случае $N = 1$). После этого в каждом из выполненных вложенных вызовов — начиная с самого последнего и в обратном порядке — производится вычисление результата и возврат в предыдущий вложенный вызов подпрограммы. Такая обратная цепочка рекурсивных возвратов выполняется, пока не произойдёт возврат в самый первый вызов подпрограммы. А когда в ней будет вычислен окончательный результат, происходит возврат в основной алгоритм.

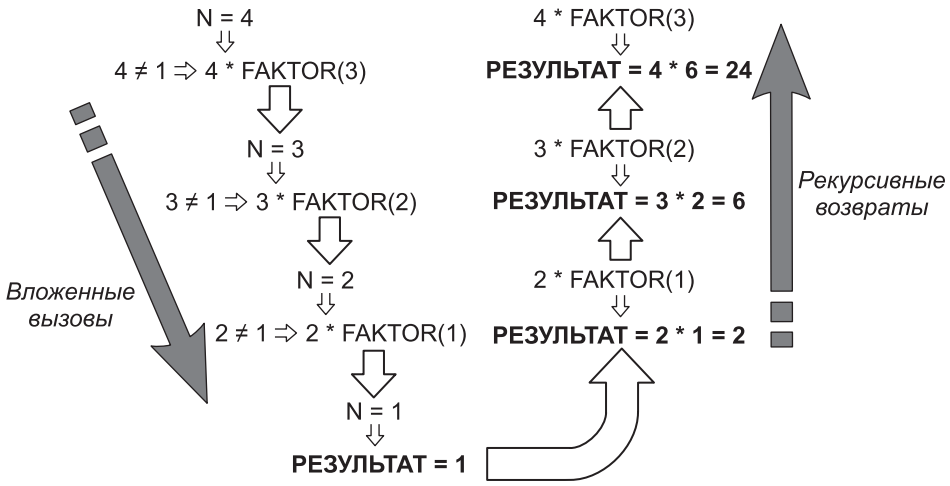


Рис. 4.8

Важно помнить, что использовать рекурсию нужно оптимально. Некоторые задачи более просто и изящно решаются при помощи рекурсии. Но во многих случаях гораздо быстрее и проще разработать обычный, нерекурсивный алгоритм. Например, для вычисления факториала можно использовать обычный цикл (ниже приведен фрагмент алгоритма):

```
F = 1
ПОКА N > 1 выполнять:
  F = F * N
  N = N - 1
КОНЕЦ ПОКА
```

Вопросы и задания

1. Что такое подпрограмма? Для чего используются подпрограммы?
2. Сформулируйте основные различия между подпрограммой-функцией и подпрограммой-процедурой. В каких случаях предпочтительно использовать тот или иной из этих двух видов подпрограмм? Приведите примеры.
3. Что понимается под стандартными функциями? Как вы считаете, почему их называют стандартными? Найдите в Интернете информацию о том, почему такие функции называют библиотечными.
4. Как работает механизм передачи данных между основным алгоритмом и подпрограммой при помощи формальных и фактических параметров? Какое основное правило при этом должно соблюдаться?

5. Что такое обработчики событий? Чем работа таких процедур отличается от работы обычных подпрограмм-процедур?
6. Что такое рекурсия? Как работают рекурсивные алгоритмы?
*7. Приведите свой пример построения рекурсивного алгоритма. Опишите последовательность вложенных вызовов и рекурсивных возвратов при работе вашего рекурсивного алгоритма.

4.1.4. Приёмы отладки программ. Трассировка программ

Созданная программа не обязательно сразу работает правильно — при её разработке могут быть допущены ошибки либо могут быть не учтены все возможные варианты исходных данных. Поэтому необходима проверка программы, поиск, выявление и исправление допущенных в ней возможных ошибок.

Поиск и исправление ошибок в программе называют её **отладкой**.

Первый шаг отладки — проверка синтаксиса (правильности написания текста программы по созданному алгоритму). Обычно эта проверка выполняется автоматически средой программирования. Обнаруженные возможные ошибки компьютер выделяет визуально (цветом, подчёркиванием и т. д.), чтобы разработчик алгоритма обратил на них своё внимание.

Важно помнить, что одна допущенная ошибка может стать причиной «обнаружения» других предполагаемых ошибок (например, если в начале алгоритма вы забыли объявить тип используемой переменной, то возникнут ошибки при её последующем использовании). Поэтому ошибки надо исправлять с начала алгоритма и каждый раз, исправив очередную ошибку, повторять проверку программы заново: возможно, какие-то из ранее обнаруженных ошибок исчезнут сами собой.

Второй шаг отладки — проверка правильности работы программы на каком-то одном типовом примере. Цель этой проверки — убедиться, что в программе не допущено семантических (смысловых) ошибок.

Такую проверку можно выполнить путём **трассировки программы** — её пробного выполнения вручную для конкретных исходных данных, с контролем получаемых значений переменных на каждом шаге. Эти значения переменных удобно записывать в виде таблицы, которую называют **таблицей трассировки**.

Приведём *пример трассировки программы* вычисления факториала натурального числа:

```
НАЧАЛО
ВВОД N
F = 1
ПОКА N > 1 выполнять:
  F = F * N
  N = N - 1
КОНЕЦ ПОКА
ВЫВОД F
КОНЕЦ
```

Проведём трассировку программы для значения N, равного 6. Моменты изменения значений переменных будем для наглядности выделять серым фоном. Заметим, что в остальных случаях в строках таблицы повторяются те значения переменных, которые были получены ранее и в данный момент сохраняются в них. Если же переменная ещё не имеет никакого значения, в соответствующей ячейке таблицы ставится прочерк.

	N	F
ВВОД N	6	–
F = 1	6	1
ПОКА N > 1	6 > 1 — цикл выполняется	1
F = F * N	6	1 · 6 = 6
N = N - 1	6 – 1 = 5	6
ПОКА N > 1	5 > 1 — цикл выполняется	6
F = F * N	5	6 · 5 = 30
N = N - 1	5 – 1 = 4	30
ПОКА N > 1	4 > 1 — цикл выполняется	30
F = F * N	4	30 · 4 = 120
N = N - 1	4 – 1 = 3	120
ПОКА N > 1	3 > 1 — цикл выполняется	120
F = F * N	3	120 · 3 = 360
N = N - 1	3 – 1 = 2	360
ПОКА N > 1	2 > 1 — цикл выполняется	360
F = F * N	2	360 · 2 = 720
N = N - 1	2 – 1 = 1	720
ПОКА N > 1	1 = 1 — цикл прекращается	720
ВЫВОД F	1	720

Нетрудно видеть, что для команды ветвления или для цикла с условием в таблице трассировки записывается соответствующее условие, и, в зависимости от его истинности, записывается соответствующий вывод, согласно которому далее выполняются те или иные команды алгоритма. Для цикла в таблице трассировки соответствующие команды из тела цикла повторяются столько раз, сколько реально выполняется этот цикл при заданных исходных данных (соответствующие блоки для наглядности разделены двойной линией границы).

В последней строке таблицы трассировки, соответствующей команде вывода результата, для соответствующей переменной (F) мы видим полученное значение (720).

Проверяем его правильность, зная математическую формулу расчета факториала:

$$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720.$$

Совпадение результата, полученного при трассировке программы, со значением результата, вычисленным по исходной математической формуле, показывает, что алгоритм для выбранного исходного значения работает правильно. Однако это не гарантирует его правильную работу при всех возможных исходных данных. Поэтому необходим **третий шаг отладки программы — её тестирование**.

Для выполнения тестирования алгоритма необходимо тщательно продумать и подготовить **набор тестов** — такой набор возможных исходных данных, который исчерпывает все возможные ситуации.

Так, для программы, связанной с математическими расчётами, необходимо предусмотреть следующие варианты исходных данных:

- типичные для решаемой задачи значения исходных данных (например, при решении задачи на движение автомобилей это будут значения скоростей порядка десятков километров в час, а для задачи на движение космических кораблей — значения скоростей в несколько километров в секунду);
- граничные значения данных (если заранее известны диапазоны их изменения);
- значения данных вне границ допустимого диапазона (например, при работе с натуральными числами — нулевые или отрицательные значения) — это необходимо для того, чтобы выявить, как алгоритм реагирует на заведомые ошибки[1)].

[1)] Как правило, в профессионально разработанных программах должен быть предусмотрен контроль вводимых значений исходных данных, обнаружение их несоответствия корректным диапазонам и либо исправление этих данных (если ситуация однозначна), либо выдача пользователю соответствующего сообщения об ошибке и запрос на повторный ввод данных.

Если программа предполагает ввод нескольких исходных данных, то необходимо предусмотреть в наборе тестов *все* возможные их комбинации.

Разработка корректного и полного набора тестов для каждой отлаживаемой программы представляет собой отдельную, достаточно сложную задачу. Однако сделать это необходимо. Иначе варианты исходных данных, приводящие к сбою в работе программы, могут выявиться уже при работе с ней её пользователей.

При отладке программ важно помнить следующее правило: *чем на более раннем этапе разработки будет обнаружена ошибка, тем легче и дешевле её исправить*. Так, обнаружение и исправление ошибки при разработке программы потребуют лишь повторной её отладки. Но если ошибка обнаружится уже при коммерческой эксплуатации созданной программы, то это приведёт к значительным потерям — как моральным (страдает репутация разработчика как создателя высококачественных программ), так и финансовым (необходимость информирования пользователей программы об ошибке, выпуска и распространения по всем пользователям исправленной версии программы и т. д.),

Вопросы и задания

1. Что такое отладка программы? Для чего она нужна?
2. Перечислите основные этапы отладки программы. Опишите назначение каждого этапа и характер обнаруживаемых с его помощью недочётов программы.
3. Что такое таблица трассировки? Как она составляется? Приведите свой пример программы и составленной для неё таблицы трассировки.
4. Как формируется набор тестов для тестирования программы? Какие варианты исходных данных рекомендуется включать в этот набор тестов?

*5. Приходилось ли вам при работе на компьютере сталкиваться со сбоями в работе программ? Как вы считаете — вызвано ли это недостаточно тщательным тестированием этих программ при их разработке? Предположите, какие меры разработчику этих программ потребовалось (или потребуется) предпринять для исправления этих ошибок и оцените примерные затраты на это исправление.

4.1.5. Типовые алгоритмы

Некоторые алгоритмы используются настолько часто, что считаются типовыми. Рекомендуется знать принципы построения этих алгоритмов, их принципы работы и уметь опознавать эти алгоритмы (уметь определять по виду алгоритма, какую задачу он решает).

Алгоритм нахождения наибольшего из нескольких чисел

Задано два, три, четыре или более исходных чисел. Требуется определить (и, например, вывести на экран) максимальное из них.

Принцип решения задачи

Исходные числа при их вводе записываются в соответствующие переменные (например, *a*, *b*, *c*).

Определить наибольшее из двух чисел позволяет оператор ветвления, условие в котором сводится к сравнению этих двух чисел:

```
ЕСЛИ a>b ТО Max = a ИНАЧЕ Max = b
```

Если требуется найти максимальное из трёх чисел, то проще всего использовать несколько операторов ветвления с соответствующими сложными условиями[1]:

```
ЕСЛИ (a>=b) И (a>=c) ТО Max = a
ЕСЛИ (b>=a) И (b>=c) ТО Max = b
ЕСЛИ (c>=a) И (c>=b) ТО Max = c
```

Возможен и алгоритм, использующий вложенные операторы ветвления (рис. 4.9).

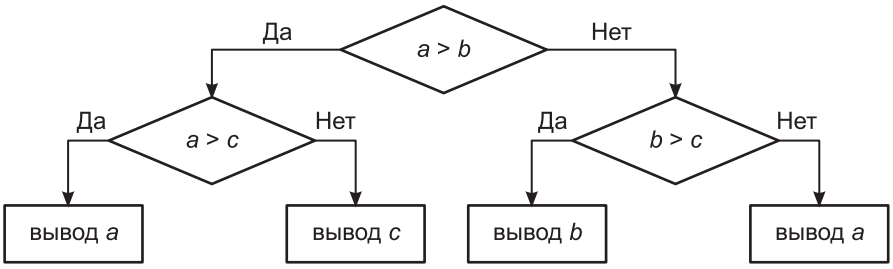


Рис. 4.9

[1] В условии используется сравнение «больше или равно», чтобы обеспечить корректную работу алгоритма в случае равенства чисел (очевидно, для равных чисел в качестве «наибольшего» можно выбрать любое из них).

Если требуется найти максимальное из четырёх и более чисел, то используется прием последовательного сравнения чисел с «предполагаемым максимумом». Сначала мы предполагаем, что максимальным является первое число. Далее каждое число по очереди сравнивается с текущим значением предполагаемого максимума: если это число больше, чем предполагаемый максимум, то в качестве предполагаемого максимума берётся уже это число. В результате после проверки всех чисел в переменной, в которой хранится предполагаемый максимум, окажется наибольшее число.

```
Max = a
ЕСЛИ b>=Max ТО Max = b
ЕСЛИ c>=Max ТО Max = c
ЕСЛИ d>=Max ТО Max = d
...
```

Выполним трассировку алгоритма для исходных чисел 2, 3, 1, 5:

	Условие	*a*	*b*	*c*	*d*	*Max*
`Max = a`		2	3	1	5	2
ЕСЛИ `b >= Max` **ТО** `Max = b`	да	2	3	1	5	3
ЕСЛИ `c >= Max` **ТО** `Max = c`	нет	2	3	1	5	3
ЕСЛИ `d >= Max` **ТО** `Max = d`	да	2	3	1	5	5

Алгоритм поиска наименьшего из двух, трёх, четырёх и более чисел реализуется аналогично, но в условиях операторов ветвления используется сравнение «меньше» (либо «меньше или равно»).

Подобный алгоритм обладает *масштабируемостью*: его легко дополнять для обработки практически любого количества чисел. Однако для поиска максимума или минимума из множества числовых значений удобнее использовать массивы и циклы.

Анализ записи числа в позиционной системе счисления (разбор числа на цифры)

Задано десятичное число. Требуется выделять и обрабатывать его отдельные цифры (от младших разрядов к старшим).

Принцип решения задачи

Для разбора числа на цифры используется пара операций, как правило, имеющихся в любом языке программирования:

- поиск остатка от деления (операция MOD);
- целочисленное деление (операция DIV).

Очевидно, что операция вычисления остатка от деления исходного числа на 10 приводит к выделению последней цифры этого числа. Например: `12345 MOD 10 = 5`.

Операция целочисленного деления исходного числа на 10 приводит к «отсечению» последней цифры этого числа. Например: `12345 DIV 10 = 1234`.

Соответственно, для разбора исходного числа *X* на отдельные цифры можно использовать цикл с предусловием:

```
ПОКА X > 0 ВЫПОЛНЯТЬ
C = X MOD 10  // в переменную C выделяется
              // очередная цифра
  // обработка выделенной цифры
  X = X DIV 10  // от числа отсекается
                // обработанная цифра
КОНЕЦ ЦИКЛА
```

Если использовать операции MOD и DIV для деления не на 10, а на другое число, то алгоритм будет выполнять разбор исходного числа на цифры соответствующей системы счисления. Например, использование операций MOD 2 и DIV 2 позволяет выделять цифры записи числа в двоичной системе счисления.

Поиск НОД заданных натуральных чисел

Заданы два натуральных числа. Требуется вычислить их наибольший общий делитель (НОД).

Принцип решения задачи

Операция поиска **наибольшего общего делителя (НОД)** двух заданных натуральных чисел обычно выполняется по **алгоритму Евклида**, который был впервые описан греческим математиком Евклидом около 300 лет до нашей эры. Идея алгоритма Евклида очень проста: из большего числа вычитается меньшее до тех пор, пока числа не станут равны. Полученное значение — это и есть НОД двух исходных чисел (рис. 4.10).

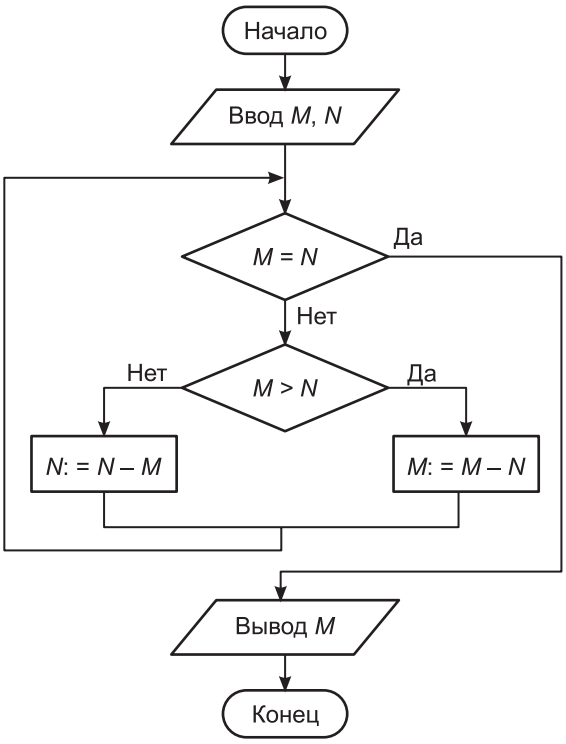


Рис. 4.10

Возможен также **модифицированный алгоритм Евклида**, работающий быстрее традиционного. Смысл модификации в том, что вместо последовательности нескольких одинаковых вычитаний меньшего числа из большего можно использовать операцию вычисления остатка от деления большего числа на меньшее. Например, для пары чисел 65 и 18:

Традиционный алгоритм Евклида	Модифицированный алгоритм Евклида
65 – 18 = 47	65 MOD 18 = 11
47 – 18 = 29	
29 – 18 = 11	

Для реализации модифицированного алгоритма Евклида используются две вспомогательные переменные. Кроме того, перед началом работы алгоритма необходимо обеспечить, чтобы первое из чисел обязательно было больше второго.

```
ЕСЛИ M > N ТО A = M, B = N ИНАЧЕ A = N, B = M
NOD = A
// теперь гарантировано, что A >B
ПОКА NOD <> 0 ВЫПОЛНЯТЬ
NOD = A MOD B
A = B, B = NOD
КОНЕЦ ЦИКЛА
ВЫВЕСТИ A
```

Выполним трассировку алгоритма для пары чисел 156 и 15:

Команда	NOD <> 0	A	B	NOD
ЕСЛИ M > N ТО A = M, B = N ИНАЧЕ A = N, B = M	–	156	15	–
NOD = A	–	156	15	156
ПОКА NOD <> 0 ВЫПОЛНЯТЬ	Да	156	15	156
NOD = A MOD B		156	15	6
A = B, B = NOD		15	6	6
ПОКА NOD <> 0 ВЫПОЛНЯТЬ	Да	15	6	6
NOD = A MOD B		15	6	3
A = B, B = NOD		6	3	3
ПОКА NOD <> 0 ВЫПОЛНЯТЬ	Да	6	3	3
NOD = A MOD B		6	3	0
A = B, B = NOD		3	0	0
ПОКА NOD <> 0 ВЫПОЛНЯТЬ	Нет	3	0	0
ВЫВЕСТИ A		3	0	0

Вы можете самостоятельно оценить выгодность модифицированного алгоритма Евклида, заменив операции A MOD B на соответствующее количество операций вычитания числа *B* из числа *A* и определив их количество.

Проверка числа на простоту

Задано натуральное число. Требуется определить, является ли это число простым.

Простое число — это натуральное число, которое делится нацело только на единицу или на само себя.

Принцип решения задачи

Обычно такая задача решается **методом перебора делителей** — берётся исходное число *N*, и производятся попытки разделить его нацело на все натуральные числа от 2 до $N - 1$. При этом заранее предусматривается отдельная *переменная-флаг*:

- изначально ей присваивается значение, которое будет обозначать, что деление нацело невозможно (например, значение 0);
- в цикле имеется оператор ветвления, условие которого проверяет, делится ли исходное число нацело на очередной проверяемый делитель; если да, то в переменную-флаг записывается другое значение (например, 1).

После завершения работы цикла отдельный оператор ветвления проверяет значение переменной-флага: если в ней осталось значение 0, то можно сделать вывод, что исходное число простое (ни один проверяемый делитель не подошёл), а если её значение изменилось на 1, это означает, что исходное число не простое.

Ниже приведен фрагмент алгоритма определения простоты числа *N*. Для корректного решения задачи нужно также добавить перед этим фрагментом операторы ветвления, контролирующие возможную ошибкуввода отрицательного или нулевого числа, а также числа, равного 1 (которое, как известно из математики, хотя и удовлетворяет условию «делится только на 1 и на само себя», но простым числом не является).

```
FLAG = 0 // исходное значение переменной-флага
ЦИКЛ I ОТ 2 ДО N - 1 С ШАГОМ 1
ЕСЛИ N MOD I = 0 ТО FLAG = 1
  // если остаток от деления отсутствует,
  // то число N делится нацело, и оно не простое
КОНЕЦ ЦИКЛА
ЕСЛИ FLAG = 1 ТО ВЫВЕСТИ "Число составное"
              ИНАЧЕ ВЫВЕСТИ "Число простое"
```

Можно сделать этот алгоритм более оптимальным. Вспомним, что составное число можно представить как произведение меньшего сомножителя на больший, например:

$$12 = \underbrace{2 \cdot 6 = 3 \cdot 4} = \underbrace{4 \cdot 3 = 6 \cdot 2}.$$

Обратите внимание на группы умножений, выделенные скобками: они «зеркально симметричны». А значит если деление исходного числа нацело обнаружено для какого-то из меньших сомножителей (в первой группе умножений они записаны первыми), то проверять соответствующие им бóльшие сомножители уже не требуется. «Граница» между этими «симметричными» группами умножений примерно соответствует квадратному корню из исходного числа, поэтому в описанном выше алгоритме достаточно выполнять цикл не до $N - 1$, а до ближайшего натурального значения, большего $\sqrt{N}$. При этом используются две стандартные подпрограммы-функции, которые есть в любом языке программирования:

`SQRT()` — вычисление квадратного корня от значения, заданного в скобках;
`INT()` — определение целой части вещественного значения, заданного в скобках (в отличие от математической операции округления, эта функция просто отбрасывает дробную часть заданного числа, поэтому для округления в бóльшую сторону нужно к полученному целому значению дополнительно прибавить единицу).

```
FLAG = 0 // исходное значение переменной-флага
ЦИКЛ ДЛЯ I ОТ 2 ДО INT(SQRT(N))+1 С ШАГОМ 1
ЕСЛИ N MOD I = 0 ТО FLAG = 1
  // если остаток от деления отсутствует,
  // то число Nделится нацело, и оно не простое
КОНЕЦ ЦИКЛА
ЕСЛИ FLAG = 1 ТО ВЫВЕСТИ "Число составное"
              ИНАЧЕ ВЫВЕСТИ "Число простое"
```

Вопросы и задания

1. Опишите основную идею алгоритмов поиска наибольшего/наименьшего из нескольких заданных чисел. Объясните принцип работы алгоритма поиска максимального/минимального значения путём последовательного сравнения чисел с предполагаемым максимумом/минимумом.

2. Объясните принцип работы алгоритма разбора десятичной записи числа на отдельные цифры.

*3. Напишите на основе наразбора записи числа на отдельные цифры свой алгоритм перевода десятичного числа в его запись в двоичной системе счисления.

4. Объясните принцип работы модифицированного алгоритма Евклида. На примере определения НОД двух конкретных выбранных вами чисел продемонстрируйте и оцените выгодность модифицированного алгоритма Евклида по сравнению с традиционным (считайте для простоты, что каждая команда алгоритма выполняется в течение 0,1 секунды и сравните соответствующие затраты времени).

5. Как выполняется проверка заданного числа на простоту? Объясните суть идеи оптимизированного алгоритма проверки числа на простоту.

4.2. История развития языков программирования

Выполнение алгоритма может быть автоматически реализовано техническими устройствами, среди которых особое место занимает компьютер. При этом говорят, что компьютер исполняет программу (последовательность команд), реализующую алгоритм.

Алгоритм, записанный на «понятном» компьютеру языке программирования, называется **программой**.

Машинный язык. На заре компьютерной эры, в 40–50-е годы XX века, программы писались на машинном языке и представляли собой очень длинные последовательности нулей и единиц. Составление и отладка таких программ были чрезвычайно трудоёмким делом. Программы на машинных языках были машинно зависимыми, т. е. для каждой ЭВМ необходимо было создавать свою собственную программу, так как в ней в явной форме учитывались аппаратные ресурсы ЭВМ.

Ассемблер. В начале 1950-х годов были созданы языки программирования ассемблеры. Вместо одних только нулей и единиц программисты теперь могли пользоваться операторами (`MOV`, `ADD`, `SUB` и т. д.), которые были похожи на слова английского языка. Для преобразования текста программы на ассемблере в «понятный» компьютеру машинный код использовался компилятор, который загружался в оперативную память ЭВМ.

Программы на ассемблере были также машинно зависимыми, т. е. ассемблеры для различных процессоров существенно различались между собой.

Первые языки программирования высокого уровня. С середины 1950-х годов начали создаваться первые языки программирования высокого уровня. Эти языки были машинно независимыми, так как использовали универсальную компьютерную логику и не были привязаны к типу ЭВМ. Однако для каждого языка и каждого типа ЭВМ требовалась разработка собственных компиляторов, которые загружались в оперативную память.

Языки программирования высокого уровня создавались и использовались для решения разных задач:

- язык FORTRAN (*FORmula TRANslator* — транслятор формул) был предназначен для научных и технических расчётов;
- язык COBOL (*Common Business-Oriented Language* — стандартный язык для делового применения) в основном предназначался для коммерческих приложений, обрабатывавших большие объёмы нечисловых данных;
- язык BASIC (*Beginner's All-Purpose Symbolic Instruction Code* — универсальный язык символьных инструкций для начинающих) отличается простотой изучения и был ориентирован на начинающих программистов.

Алгоритмические языки программирования. С начала 1980-х годов начали создаваться алгоритмические языки программирования, которые позволили программистам перейти к структурному программированию. Отличительной чертой этих языков было использование операторов ветвления, выбора и цикла и отказ от хаотического использования оператора `goto`. Наибольшее влияние на переход к структурному алгоритмическому программированию оказали:

- язык Pascal (назван его создателем Никлаусом Виртом в честь Блеза Паскаля) — алгоритмический язык, который позволяет легко кодировать основные алгоритмические структуры;
- язык C (произносится «Си»), позволяющий создавать быстро и эффективно выполняющийся программный код.

Языки объектно-ориентированного программирования. С 1990-х годов начали создаваться объектно-ориентированные языки программирования. В основу этих языков были положены программные объекты, которые объединяли данные и методы их обработки. Необходимо подчеркнуть, что в языках объектно-ориентированного программирования сохранялся алгоритмический

стиль программирования. С течением времени для этих языков были разработаны интегрированные среды разработки, позволяющие визуально конструировать графический интерфейс приложений:

- язык C++ является прямым потомком алгоритмического языка C;
- язык Object Pascal был разработан компанией Borland на основе алгоритмического языка Pascal. После создания интегрированной среды разработки система программирования получила название Delphi, а кроссплатформенная свободно распространяемая версия — Lazarus;
- язык Visual Basic был создан корпорацией Microsoft на основе языка QBasic для разработки приложений с графическим интерфейсом в среде операционной системы Windows.

Языки программирования для компьютерных сетей. В 1990-е годы в связи с бурным развитием Интернета были созданы языки, обеспечивающие межплатформенную совместимость:

- язык Java, полноценный объектно-ориентированный язык, был разработан фирмой Sun Microsystems для создания сетевого программного обеспечения;
- язык JavaScript, язык сценариев веб-страниц, разработан компанией Netscape.

На подключённых к Интернету компьютерах с различными операционными системами (Windows, Linux, macOS и др.) могли выполняться одни и те же программы. Исходная программа на таких языках компилируется в промежуточный код, который исполняется на компьютере встроенной в браузер виртуальной машиной.

Языки программирования на платформе .NET. В настоящее время используют интегрированную систему программирования Visual Studio на платформе .NET Framework, разработанную корпорацией Microsoft. Эта платформа предоставляет возможность создавать приложения в различных системах объектно-ориентированного программирования, в которых для создания программного кода используются объектно-ориентированные языки программирования, в том числе:

- на языке Visual Basic .NET, созданном на основе языка Visual Basic и сохраняющем простоту своих предшественников;
- на языке Visual C# (читается «С-шарп»), созданном на основе языков C++ и Java.

Вопросы и задания

1. В чём состоит преимущество языков высокого уровня перед машинным языком и ассемблерами?
2. Каковы характерные особенности алгоритмических языков программирования? Объектно-ориентированных языков программирования?
3. Какими причинами было обусловлено появление языков Java и JavaScript?
4. На основе каких языков возникли языки программирования на платформе .NET?

4.3. Введение в объектно-ориентированное программирование

4.3.1. Объекты: свойства и методы

Объекты (`Objects`). Основной единицей в объектно-ориентированном программировании является **программный объект**, который объединяет в себе как описывающие его **свойства**, так и действия объекта (процедуры) — **методы**. Если говорить образно, то объекты — это «существительные», свойства объекта — это «прилагательные», а методы объекта — это «глаголы».

Программные объекты обладают **свойствами**, имеют **методы** и для них можно описать реакцию на **события**.

Классы объектов являются «шаблонами», определяющими наборы свойств, методов и событий, по которым создаются объекты. Основными классами объектов являются объекты, реализующие графический интерфейс проектов.

Объект, созданный по «шаблону» класса объектов, является **экземпляром класса** и **наследует** весь набор свойств, методов и событий данного класса. Каждый экземпляр класса имеет уникальное для данного класса имя. Различные экземпляры класса обладают одинаковым набором свойств, однако значения этих свойств у них могут различаться.

Свойства объектов (`Properties`). Каждый объект обладает определённым набором свойств. Существует несколько основных ситуаций, в которых можно менять свойства объектов. Во время разработки проекта `[design]` можно установить первоначальные значения свойств объекта.

В режиме выполнения проекта **[run]** можно устанавливать или менять значения свойств объекта в ходе исполнения программного кода. Для присваивания свойству объекта нового значения необходимо указать в левой части строки программного кода имя объекта, а затем — название свойства. В правой части строки необходимо записать конкретное значение свойства. Например, программный код вывода в поле с именем `Label1` текста в различных языках программирования будет выглядеть следующим образом.

Язык Visual Basic .NET:[1)]
```
Label1.Text = "Текст"
```

Язык Visual C#:
```
label1.Text = "Текст";
```

Язык Lazarus:
```
Label1.Caption := 'Текст';
```

В программном коде для доступа к свойствам и методам используется **точечная нотация (dot-запись)**, при которой имена объектов, свойств и методов отделяются друг от друга знаком точки «.».

Методы объектов (Methods). Чтобы объект выполнил какую-либо операцию, необходимо применить метод, которым он обладает. Многие методы имеют аргументы, которые позволяют задать параметры выполняемых действий. Обратиться к методу объекта можно тоже с использованием точечной нотации, причём аргументы метода заключаются в скобки. Например, для добавления элемента в список в языках объектно-ориентированного программирования используется метод **Add()**.

Язык Visual Basic .NET:
```
ListBox1.Items.Add("Элемент списка")
```

Язык Visual C#:
```
listBox1.Items.Add("Элемент списка");
```

Язык Lazarus:
```
ListBox1.Items.Add('Элемент списка');
```

1) Набранные тексты этой и следующих программ размещены в электронном приложении к главе 4.

Вопросы и задания

1. Чем различаются понятия «класс объектов» и «экземпляр класса»?
2. В экземпляре класса можно изменить набор свойств? Набор методов? Значения свойств?

4.3.2. События

События (`Events`). Событие представляет собой изменение некоторого состояния, распознаваемое объектом. Для реакции на это изменение могут быть описаны те или иные методы — обработчики, обрабатывающие события в программном коде. Событие может создаваться пользователем (например, щелчок мышью или нажатие клавиши) или быть результатом воздействия других программных объектов. Каждый элемент управления может реагировать на различные события, однако есть события (например, `Click` — щелчок мышью), на которые реагирует большинство типов элементов управления.

Обработчик события. Для каждого события можно запрограммировать обработчик события (*событийную процедуру*). Если пользователь производит какое-либо воздействие на элемент графического интерфейса (например, щелчок мышью), в качестве обработчика выполняется некоторая последовательность действий в форме процедуры.

Каждый обработчик события представляет собой процедуру, которая реализует определённый алгоритм. Создание программного кода обработчика события производится с использованием алгоритмических структур различных типов (линейная, ветвление или цикл).

Обработчик события представляет собой процедуру, которая начинает выполняться после реализации определённого события.

Имя обработчика события (событийной процедуры) включает в себя имя объекта и имя события. После имени событийной процедуры в скобках указываются параметры, которые позволяют правильно обработать событие. В событийной процедуре на языках .NET существуют два параметра, а в языке Lazarus — один параметр.

Далее на трёх языках программирования показан пустой обработчик события `Click` элемента управления «кнопка» — `Button1`.

Язык Visual Basic .NET:

```
Private Sub Button1_Click(sender As Object,
e As EventArgs) Handles Button1.Click
End Sub
```

Язык Visual C#:

```
private void button1_Click(object sender,
EventArgs e)
{
}
```

Язык Lazarus:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

Первый параметр, `sender`, предоставляет ссылку на объект, который вызывает событие. Например, при щелчке мышью по кнопке наступает событие **Click** данной кнопки и её адрес передаётся обработчику события и сохраняется в аргументе `sender`.

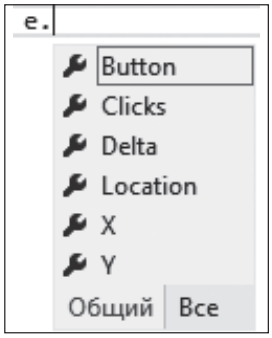


Рис. 4.11

В языках программирования на платформе .NET используется и второй параметр `e`, который передаёт данные, характерные для обрабатываемого события. Этот параметр обычно имеет тип **EventArgs**, однако существуют события, которые требуют особого типа данных. Например, если обрабатываются события мыши, то используется параметр `MouseEventArgs`. С помощью этого параметра можно получить сведения о координатах мыши, какая была нажата кнопка и сколько было сделано щелчков. Для вывода всех свойств аргумента `e` (рис. 4.11) достаточно ввести в процедуре обработчика события: `e.`

Вопросы и задания

В чём состоит различие между обработчиками событий в языках программирования на платформе .NET и в языке Lazarus?

4.3.3. Проекты и приложения

Проект (Project). С одной стороны, система объектно-ориентированного визуального программирования является системой про-

граммирования, так как позволяет кодировать алгоритмы на данном языке. С другой стороны, система объектно-ориентированного визуального программирования является средой проектирования, так как позволяет осуществлять визуальное конструирование графического интерфейса.

Результатом процессов программирования и проектирования является проект, который объединяет в себе программный код и графический интерфейс.

В объектно-ориентированном программировании проект может включать несколько **форм**, причём каждой форме, с помощью которой реализуется графический интерфейс проекта, соответствует свой программный модуль формы. Подробно о графическом интерфейсе проекта рассказано в параграфе 4.7.

Кроме того, в состав проекта могут входить отдельные самостоятельные программные модули.

Проект включает в себя **программные модули форм** и самостоятельные **программные модули** в виде отдельных файлов. Проект может быть запущен на выполнение только из системы объектно-ориентированного программирования.

Решения (`Solution`). В системах объектно-ориентированного программирования Visual Basic .NET и Visual C# проекты объединяются в решения, а в системе Lazarus — в группы. Решение (группа) включает один или несколько проектов, которые в упорядоченном виде в системах Visual Basic .NET и Visual C# отображаются в *Обозревателе решений*, а системе Lazarus — в окне *Обозреватель кода*. Решение (группа) создаётся автоматически при создании нового проекта, а при необходимости к решению можно добавлять новые проекты. Решения (группы) позволяют работать с несколькими проектами в пределах одного экземпляра системы объектно-ориентированного программирования.

Интерпретаторы и компиляторы. Чтобы процессор мог выполнить программу, эта программа и данные, с которыми она работает, должны быть загружены в оперативную память.

Итак, мы создали программу на языке программирования (некоторый текст) и загрузили её в оперативную память. Теперь мы хотим, чтобы процессор её выполнил, однако процессор «понимает» команды на машинном языке, а наша программа написана на языке программирования. Как быть?

Необходимо, чтобы в оперативной памяти находилась программа-переводчик **(транслятор)**, автоматически переводящая

нашу программу с языка программирования на машинный язык. Компьютер может выполнять программы, написанные только на том языке программирования, транслятор которого размещён в оперативной памяти компьютера.

Трансляторы языков программирования бывают двух типов: интерпретаторы и компиляторы.

Интерпретатор — это программа, которая обеспечивает последовательный «перевод» инструкций программы на машинный язык с одновременным их выполнением. Поэтому при каждом запуске программы на выполнение эта процедура повторяется. Достоинством интерпретаторов является удобство отладки программы (поиска в ней ошибок), так как возможно пошаговое её исполнение, а недостатком — сравнительно малая скорость выполнения.

Компилятор действует иначе. Он переводит весь текст программы на машинный язык и сохраняет его в исполняемом файле (обычно с расширением exe). Затем этот уже готовый к исполнению файл, записанный на машинном языке, можно запускать на исполнение многократно. Достоинством компиляторов является большая скорость выполнения программы, а недостатком — трудоёмкость отладки, так как невозможно пошаговое выполнение программы.

Системы объектно-ориентированного программирования позволяют программисту контролировать в интегрированной среде выполнение программ с помощью **отладчика**. Это даёт возможность отлаживать программу пошагово.

Итак, как мы отмечали ранее, сохранённый проект может выполняться только в самой системе программирования. Чтобы преобразовать проект в *приложение*, которое может выполняться непосредственно в среде операционной системы, необходимо выполнить компиляцию проекта, в процессе которой приложение сохраняется в исполняемом файле (с расширением exe).

Приложение интегрирует программный код и графический интерфейс в одном исполняемом файле, который может запускаться непосредственно в операционной системе.

Этапы разработки проектов. Создание проектов и приложений в системах объектно-ориентированного программирования можно условно разделить на несколько этапов.

1. *Создание графического интерфейса проекта.* На форму помещаются элементы управления, которые должны обеспечить взаимодействие проекта с пользователем.

2. *Установка значений свойств объектов графического интерфейса.* В режиме конструирования задаются значения свойств формы и элементов управления, помещённых ранее на форму.
3. *Создание и редактирование программного кода.* Создаются заготовки обработчиков событий (двойной щелчок мышью по элементу управления вызывает заготовку обработчика события, которое для данного элемента управления используется наиболее часто). Затем в редакторе программного кода производится ввод и редактирование программного кода обработчиков событий.
4. *Сохранение проекта.* Так как проекты включают в себя несколько файлов, рекомендуется для каждого проекта создать отдельную папку на диске. Сохранение проекта производится с помощью пунктов меню *Файл*.
5. *Компиляция проекта в приложение.* Создаётся приложение — исполняемый файл (exe).

Вопросы и задания

1. В чём состоит различие между интерпретаторами и компиляторами?
2. В чём состоит различие между проектом и приложением?

4.4. Система объектно-ориентированного программирования Microsoft Visual Studio

4.4.1. Интегрированная среда разработки языков Visual Basic .NET и Visual C#

Состав Visual Studio. Microsoft Visual Studio — это инструмент разработки программ, позволяющий писать программы на нескольких языках программирования .NET. В состав Visual Studio[1)] входят следующие языки программирования (рис. 4.12):

- Visual Basic .NET;
- Visual C# (произносится «Си-шарп»);
- Visual C++ (произносится «С плюс плюс»).

1) Здесь и далее в зависимости от используемых версий Visual Studio и Lazarus вид диалоговых окон может несколько различаться, но смысл имеющихся на них надписей и диалоговых элементов (кнопок, меню и пр.) остаётся одним и тем же.

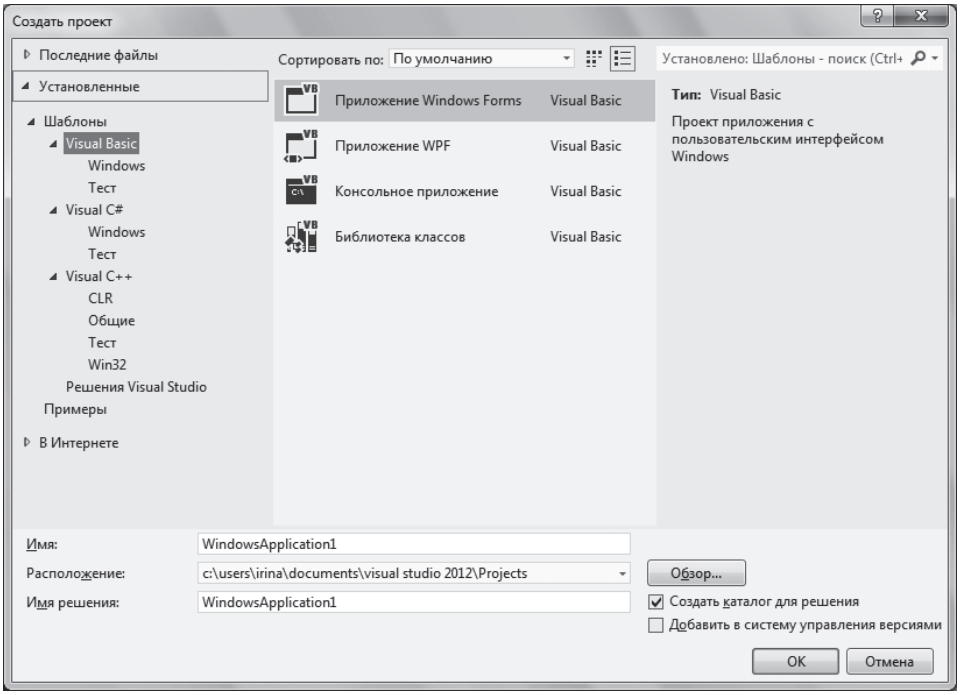


Рис. 4.12

Интегрированная среда разработки. Visual Studio является системой визуального объектно-ориентированного программирования на платформе .NET Framework. Система программирования Visual Studio предоставляет пользователю для работы со всеми языками .NET удобный, причём одинаковый, графический интерфейс **IDE** (*Integrated Development Environment* — интегрированная среда разработки) — см. на примере Visual Basic (рис. 4.13).

Визуальное конструирование графического интерфейса проекта выполняется в **конструкторе форм** (*Windows Forms Designer*), который по умолчанию размещается на вкладке *Form1.vb [Конструктор]*. Конструктор форм располагается в центре окна интегрированной среды разработки и содержит **форму** (по умолчанию **`Form1`**), являющуюся основой графического интерфейса проекта. На **форму** можно поместить различные **элементы управления**: кнопки (`Button`), текстовые поля (`TextBox`), надписи (`Label`) и т. д.

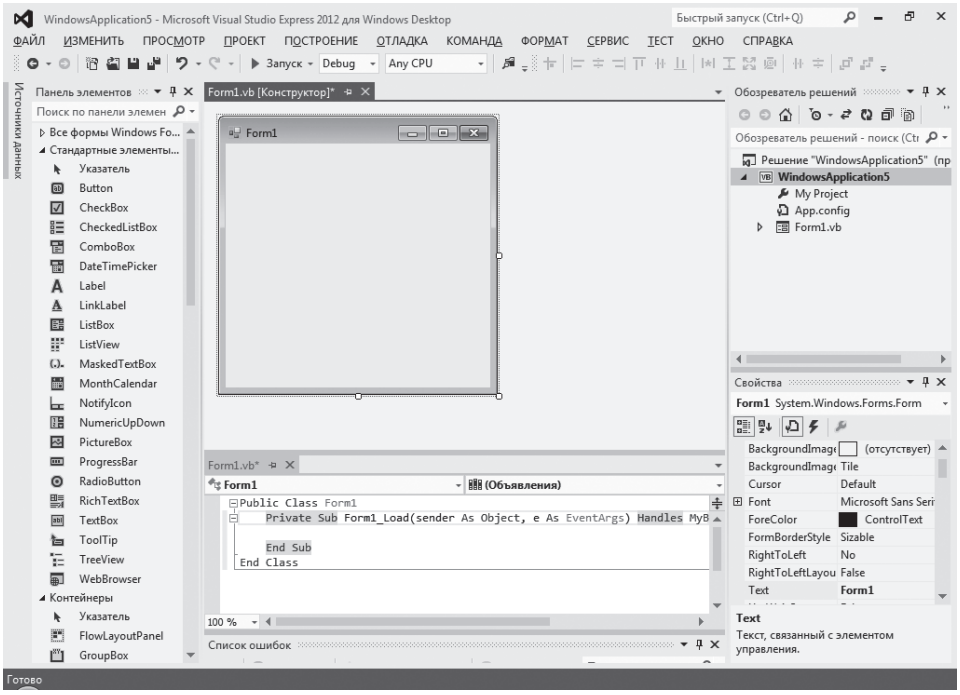


Рис. 4.13

Пиктограммы элементов управления располагаются на панели *Панель элементов*, которая размещается в левой части окна интегрированной среды разработки.

С формой связан программный код проекта, для ввода и редактирования которого служит **редактор программного кода** (по умолчанию размещается на вкладке *Form1.vb*). Редактор программного кода поддерживает язык проекта, т. е. предлагает подсказки при вводе программы, выделяет синтаксические ошибки, структурирует программный код отступами и т. д.

Для перехода в окно программного кода можно просто щёлкнуть по ярлыку вкладки *Form1.vb* или ввести команду [*Просмотр—Код*]. Для обратного перехода в конструктор форм можно щёлкнуть по ярлыку вкладки *Form1.vb* [*Конструктор*] или ввести команду [*Просмотр—Конструктор*].

Первые части названий окна конструктора форм и окна редактора программного кода совпадают не случайно, так как в них создаётся и редактируется один и тот же файл `Form1.vb`. Этот файл и другие файлы проекта можно увидеть в окне *Обозреватель решений*, которое размещается в правом верхнем углу интегрированной среды разработки.

Справа внизу располагается окно *Свойства*. Оно содержит список свойств, относящихся к выбранному объекту (форме или элементу управления на форме). В левом столбце находятся названия свойств, а в правом — их значения. Установленные по умолчанию значения могут быть изменены.

Настройка интегрированной среды разработки. Рекомендуется провести настройку интегрированной среды разработки. Для этого необходимо ввести команду [*Сервис—Параметры...*] и в появившемся диалоговом окне *Параметры* установить нужные параметры для элементов интерфейса: *Конструктор Windows Form*, *Отладка* и др.

В процессе создания графического интерфейса проекта важно упорядоченно разместить на форме элементы управления. Для этого необходимо в левой части диалогового окна *Параметры* выбрать пункт *Конструктор Windows Form*, а в правой части установить шаг сетки на форме и её видимость (рис. 4.14).

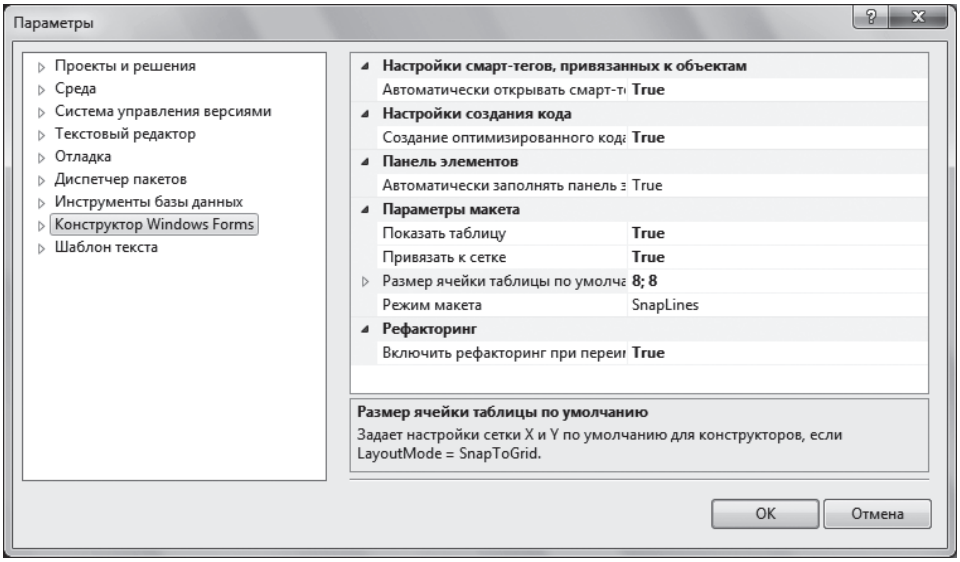


Рис. 4.14

В процессе создания, редактирования и отладки программного кода проекта целесообразно пронумеровать строки программы. Для этого необходимо в левой части диалогового окна *Параметры* выбрать пункт [*Текстовый редактор—Все языки*], а в правой части установить флажок *Номера строк* (рис. 4.15).

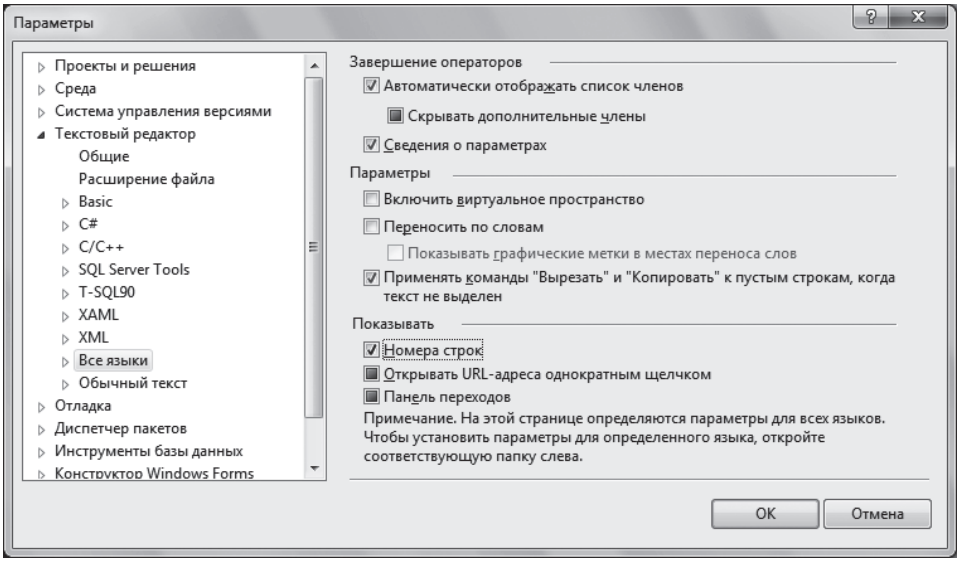


Рис. 4.15

4.5. Система объектно-ориентированного программирования Lazarus

Система объектно-ориентированного программирования Lazarus позволяет визуально создавать графический интерфейс к проектам на языке программирования Object Pascal.

Окно системы программирования Lazarus. Система программирования Lazarus предоставляет пользователю удобный графический интерфейс в процессе разработки проекта. После запуска Lazarus появится окно системы программирования, которое включает в себя (рис. 4.16):

- строку заголовка *Lazarus*;
- строку **главного меню** под строкой заголовка;
- кнопки с пиктограммами наиболее часто используемых команд под строкой главного меню.

Окно *Конструктор форм*. В центре располагается окно *Конструктор форм*. Для создания нового проекта необходимо ввести команду [*Проект—Создать проект*]. Окно представляет собой **форму** (в данном случае *Form1*), на которой происходит визуальное конструирование графического интерфейса разрабатываемого проекта. Размеры формы можно менять, перетаскивая мышью правую или нижнюю границу формы.

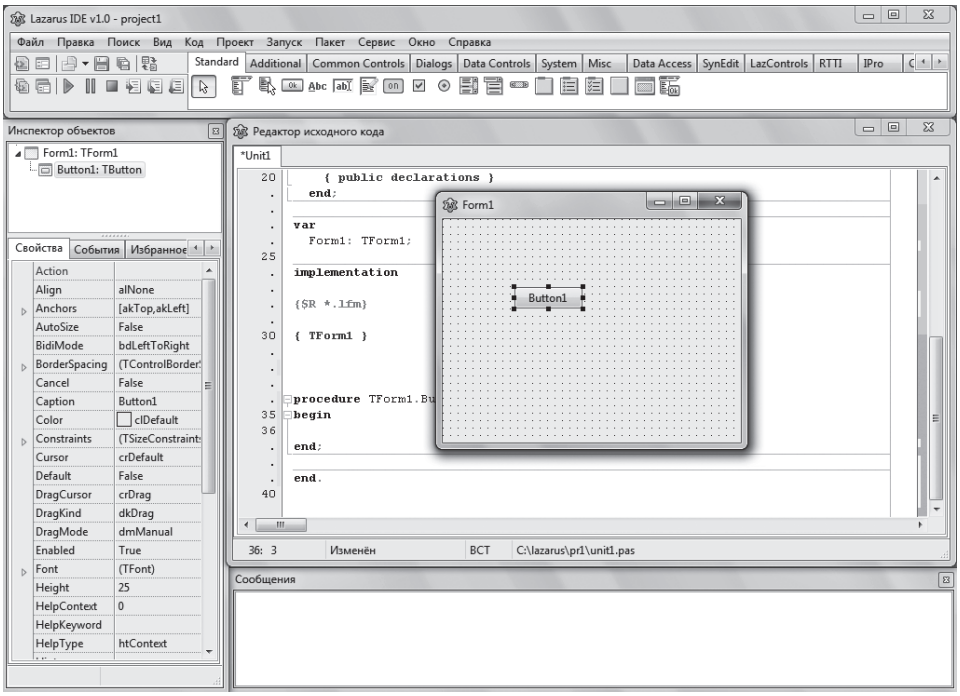


Рис. 4.16

Окно *Конструктор форм* вызывается командой [*Вид—Переключить форму/модуль*].

Первоначально форма пуста, в дальнейшем в процессе создания графического интерфейса проекта на ней размещаются **элементы управления.**

Окно *Редактор исходного кода*. С формой связан программный модуль, содержащий программные коды процедур. Для ввода и редактирования текста программы служит окно *Редактор исходного кода* (в данном случае *Unit1.pas*), которое вызывается командой [*Вид—Редактор исходного кода*].

Вкладки элементов управления. Эти вкладки содержат пиктограммы элементов управления. Стандартный набор элементов управления размещается на вкладке *Standard* и включает 18 классов объектов — это командная кнопка `TButton`, текстовое поле `TEdit`, надпись `TLabel` и т. д.

На вкладках *Additional*, *Common Controls* и других располагаются дополнительные элементы управления: графическое поле `TImage`, список изображений `TImageList` и др.

Выбрав щелчком мышью нужный элемент, мы можем поместить его на форму проектируемого приложения. Процесс размещения на форме элементов управления аналогичен рисованию графических примитивов с использованием графического редактора.

Фактически мы размещаем на форме экземпляры определённых классов объектов. Например, выбрав класс объектов `TButton`, мы можем разместить на форме неограниченное количество экземпляров этого класса, т. е. командных кнопок `Button1`, `Button2`, `Button3` и т. д.

Окно *Инспектор объектов*. Слева располагается окно *Инспектор объектов*. В верхней части окна размещается *дерево объектов*, отображающее перечень объектов графического интерфейса проекта (размещаемые на форме элементы управления).

На вкладке *Свойства* содержится список свойств, а на вкладке *События* — перечень событий, относящихся к выбранному объекту (форме или элементу управления на форме). На рисунке 4.10 выбран объект `Button1` из класса `TButton`.

Вкладка *Свойства* разделена на две колонки. В левой колонке находятся имена свойств, а в правой — их значения. Установленные по умолчанию значения могут быть изменены. Свойством объекта является количественная или качественная характеристика этого объекта (размеры, цвет, шрифт и др.). Для некоторых свойств предусмотрена возможность выбора из раскрывающегося списка значений, например, из списка можно выбрать значение цвета фона элемента управления (свойство `Color`).

Вкладка *События* также разделена на две колонки. В левой колонке находятся имена событий, а в правой можно ввести их значения.

Окно *Инспектор объектов* вызывается командой [*Вид*—*Инспектор объектов*].

Практическая работа 4.1

Создание проекта «Консольное приложение»

Задание. В системе объектно-ориентированного программирования Visual Studio создать консольное приложение, т. е. приложение, не имеющее графического интерфейса. Приложение должно выводить название языка программирования в окне командной строки.

Электронное приложение к главе 4.

Создание проекта «Консольное приложение» на языках программирования Visual Basic .NET и Visual C#

1. Запустить систему Microsoft Visual Studio.
2. Создать новый проект командой [*Файл—Создать проект...*]. В появившемся диалоговом окне *Создать проект* выбрать язык программирования Visual Basic .NET или Visual C# и тип приложения *Консольное приложение*. В окне *Код* появится заготовка программы.

Создание программного кода на языке Visual Basic .NET

3. Ввести программный код пользователя между служебными словами:

```
Sub Main()
  ...
End Sub
```

Для вывода текста в окно программной строки использовать оператор `Console.WriteLine()`
Чтобы консольное приложение не закрылось и можно было увидеть результаты работы программы, в конце программы разместить оператор `Console.Read()`.

```
Sub Main()
  Console.WriteLine ("Язык программирования Visual
                      Basic")
  Console.Read()
End Sub
```

Теперь можно запустить консольное приложение.

4. Ввести команду [*Отладка—Начать отладку*]. В окне командной строки будет выведено (рис. 4.17):

```
Язык программирования Visual Basic
```

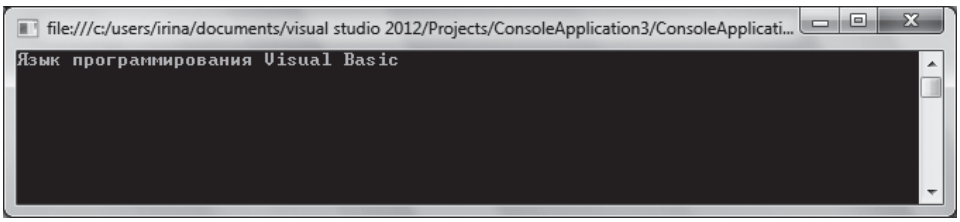


Рис. 4.17

Создание программного кода на языке Visual C#

3. На языке Visual C# программный код пишется между фигурными скобками после служебных слов.

```
static void Main(string[] args)
{
  …
}
```

Ввести программный код:

```
static void Main(string[] args)
{
  Console.WriteLine("Язык программирования C#");
  Console.Read();
}
```

Теперь можно запустить консольное приложение.

4. Ввести команду [*Отладка—Начать отладку*].
В окне командной строки будет выведено (рис. 4.18):

```
Язык  программирования  C#
```

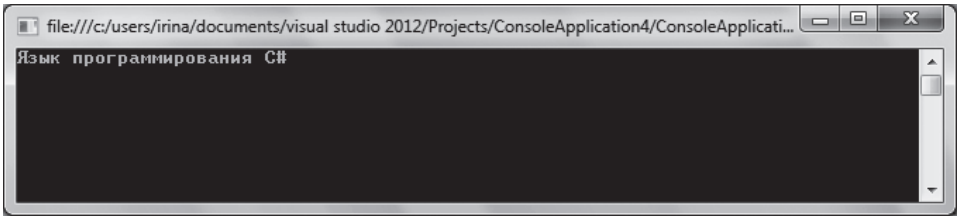


Рис. 4.18

Создание проекта «Консольное приложение» на языке программирования Lazarus

1. Запустить систему программирования Lazarus.
2. Создать новый проект командой [*Проект—Создать проект…*].
В появившемся диалоговом окне *Создать новый проект* выбрать тип приложения *Консольное приложение* (рис. 4.19).

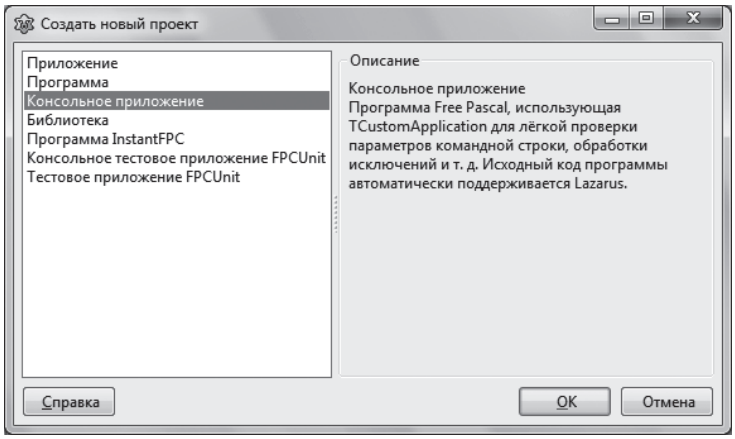


Рис. 4.19

3. В окне *Редактор исходного кода* появится заготовка программы.
   Ввести программный код пользователя между служебными словами **begin** и **end**.

Для вывода текста в окно программной строки использовать оператор `writeln()`.

```
begin
  writeln('Language Lazarus');
  readln;
end.
```

4. Сохранить программный код консольного приложения командой [*Файл—Сохранить как...*].

Теперь можно запустить консольное приложение.

5. Ввести команду [*Запуск—Запустить*].
   В окне командной строки будет выведено (рис. 4.20): `Language Lazarus`

Рис. 4.20

4.6. Переменные в языках объектно-ориентированного программирования

Переменная в программировании означает область оперативной памяти компьютера, в которую может быть занесено и храниться некоторое значение.

Тип переменной. Тип переменной определяется типом данных, которые могут быть значениями переменной. Различные типы переменных требуют для своего хранения в оперативной памяти компьютера различное количество ячеек (байтов) и могут принимать различные диапазоны значений.

В объектно-ориентированных языках программирования переменные могут быть следующих типов:

- **Byte**, **Short** (**SmallInt** в Lazarus), **Integer** (**Int** в Visual C#) и **Long** (**LongInt** в Lazarus) — значениями являются целые числа;
- **Single** (**float** в Visual C#, **real** в Lazarus), **Double** и **Decimal** (в Visual Basic .NET и Visual C#) — значениями являются вещественные числа;
- **Char** и **String** — значениями являются символы и последовательности символов;
- **Boolean** (**bool** в Visual C#) — переменные принимают логические значения **True** (истина) или **False** (ложь).

Имя переменной. Имя переменной обозначает адрес ячейки пямяти, где хранится значение переменной.

Имя каждой переменной (идентификатор) уникально и не может меняться в процессе выполнения программы. Имя переменной:

- должно начинаться с буквенного символа или с подчёркивания «_»;
- может содержать только буквенные символы, десятичные цифры и подчёркивания;
- должно содержать, по крайней мере, один буквенный или цифровой символ, если оно начинается с подчёркивания.

Рекомендуется для большей понятности текстов программ для программиста в имена переменных включать приставку, которая обозначает тип переменной. Тогда имена целочисленных переменных будут начинаться с приставок **byt**, **int** и **lng**, содержащих вещественные числа — **sng** и **dbl**, строковых — **chr** и **str**, логических — **bln**.

Объявление переменной. Важно, чтобы исполнитель программы (компьютер) «понимал», переменные какого типа используют-

ся в программе. При объявлении переменной используется ключевое слово и указывается её тип.

Область действия переменной. Область действия переменной, т. е. область в программе, где она доступна для использования, может быть локальной или глобальной.

Локальная переменная доступна только внутри процедуры или программного модуля, и к ней невозможно обращение из другой процедуры или модуля. Если переменная определена внутри процедуры, то она может быть вызвана только в этой процедуре, если она определена внутри программного модуля, то она может быть вызвана только в этом модуле.

К **глобальным переменным** может быть произведено обращение из всех программных модулей проекта.

Присваивание переменным значений. Переменная может получить или изменить значение с помощью **оператора присваивания**. В языках объектно-ориентированного программирования Visual Basic .NET и Visual C# в качестве оператора присваивания используется знак «=», а в языке Lazarus — знак «:=».

При выполнении оператора присваивания переменная, имя которой указано слева от знака равенства, получает значение, равное значению правой части. Это может быть, как в алгоритмических языках, выражение (арифметическое, строковое или логическое), а также значение свойства объекта или результат выполнения метода.

Вопросы и задания

1. В чём состоит разница между типом, именем и значением переменной?
2. Что происходит в оперативной памяти компьютера в процессе присваивания значения переменной?
3. Для чего нужно объявлять переменные в программе?

4.7. Графический интерфейс

Система объектно-ориентированного программирования позволяет визуализировать процесс создания графического интерфейса разрабатываемого проекта. Графический интерфейс необходим для реализации интерактивного диалога пользователя с исполняемым проектом.

Форма. Основой для создания графического интерфейса разрабатываемого проекта является **форма**, представляющая собой окно, в котором размещаются элементы управления. Необходимо отметить, что графический интерфейс проекта может включать в себя несколько форм.

Форма — это объект, представляющий собой окно на экране, в котором размещаются элементы управления.

Элементы управления. Визуальное конструирование графического интерфейса проекта состоит в том, что на форму с помощью мыши помещаются и «рисуются» те или иные **элементы управления.**

Перечислим классы элементов управления и их назначение в графическом интерфейсе проекта:

- текстовые поля `TextBox` (`Edit` в языке Lazarus), надписи `Label` и списки `ListBox` и `ComboBox` для ввода и вывода данных;
- графические поля `PictureBox` (`Image` в языке Lazarus) для вывода графики;
- командные кнопки `Button`, переключатели `RadioButton` и флажки `CheckBox` для организации интерактивного диалога пользователя с проектом;
- главное меню `MainMenu` для создания меню формы;
- панель инструментов `ToolBar` для создания панели инструментов формы;
- коллекция изображений `ImageList` для хранения изображений;
- диалоги `ColorDialog` и `FontDialog` для выбора цвета и шрифта;
- диалоги `OpenFileDialog` и `SaveFileDialog` для выбора файла при открытии и сохранении.

На форму может быть помещено несколько экземпляров одного класса элементов управления. Например, несколько кнопок, каждая из которых обладает индивидуальными значениями свойств (надпись, размеры и др.).

Элементы управления — это объекты, являющиеся элементами графического интерфейса проекта и реагирующие на события, производимые пользователем или другими программными объектами.

Форма и каждый класс элементов управления обладают определённым набором свойств, методов и событий. Однако есть

свойства, методы и события, которыми обладают формы и большинство элементов управления. С помощью свойств `Width` и `Height` устанавливается размер формы или элемента управления; с помощью метода `Show()` можно показать форму или элемент управления; на событие `Click` они реагируют.

Системы объектно-ориентированного программирования Visual Basic .NET и Visual C# имеют одинаковый набор элементов управления, который практически совпадает с набором элементов управления языка Lazarus (табл. 4.2). Это позволяет конструировать практически одинаковые графические интерфейсы проектов во всех вышеперечисленных системах объектно-ориентированного программирования.

Таблица 4.2

Некоторые элементы управления в языках Visual Basic .NET, Visual C# и Lazarus

Элемент управления	Visual Basic .NET	Visual C#	Lazarus
Текстовое поле	`TextBox1`	`textBox1`	`Edit1`
Надпись	`Label1`	`label1`	`Label1`
Список	`ListBox1`	`listBox1`	`ListBox1`
Поле со списком	`ComboBox1`	`comboBox1`	`ComboBox1`
Счётчик	`NumericUpDown1`	`numericUpDown1`	`SpinEdit1`
Ползунок	`TrackBar1`	`trackBar1`	`TrackBar1`
Переключатель	`RadioButton1`	`radioButton1`	`RadioButton1`

Автоматическая генерация кода элементов графического интерфейса. При размещении на форме элементов управления в системах объектно-ориентированного программирования производится автоматическая генерация программного кода. Устанавливаются значения свойств формы и элементов управления, которые определяют местоположение элемента графического интерфейса, его размеры, цвет, параметры шрифта и др.

В языках программирования Visual Basic .NET и Visual C# этот программный код сохраняется в файле программного кода

формы в разделе *Код, автоматически созданный конструктором форм Windows*. В языке Lazarus этот программный код сохраняется в отдельном файле project1.lpr, связанном с формой.

Вопросы и задания

1. Какие элементы управления целесообразно использовать при конструировании графического интерфейса проекта, если необходимо: вводить и выводить данные? Организовать диалог с пользователем?
2. Что происходит в системе программирования после помещения на форму элемента управления?

Практическая работа 4.2

Создание проекта «Переменные»

Задание. Создать в системе объектно-ориентированного программирования Visual Studio или Lazarus проект, который позволит продемонстрировать использование различных типов переменных, арифметических, строковых и логических выражений и операции присваивания. Каждый тип переменных разместить на отдельной вкладке.

Электронное приложение к главе 4.

Создание графического интерфейса проекта на языках Visual Basic .NET, Visual C# и Lazarus

Для создания вкладок разместить на форме элемент управления `TabControl1` (`PageControl1` в Lazarus).

Создание вкладок на языках Visual Basic .NET, и Visual C#

1. Выделить элемент управления `TabControl1` и в окне *Свойства* у свойства *TabPages* активизировать значение (*Коллекция*).
2. В появившемся окне *Редактор коллекции TabPage* (рис. 4.21) создать три вкладки на панели инструментов, нажав три раза кнопку *Добавить*.
   В списке *Свойства TabPage:* в свойстве *Text* ввести названия вкладок: *Числовые*, *Строковые*, *Логические*.

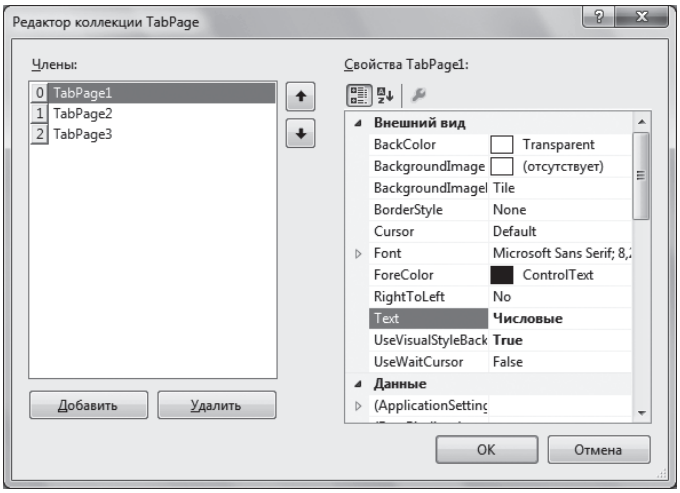


Рис. 4.21

Создание вкладок на языке Lazarus

1. Выделить элемент управления **PageControl1**, щёлкнуть правой кнопкой мыши и в контекстном меню выбрать пункт *Добавить страницу*. Выполнить эти действия три раза.
2. В окне *Инспектор объектов* в свойстве *Caption* ввести названия вкладок: *Числовые*, *Строковые*, *Логические*.

Настройка графического интерфейса проекта на языках Visual Basic .NET, Visual C# и Lazarus

3. На вкладке *TabPage1* (*TabSheet1* в Lazarus) разместить:
   - для ввода чисел — два текстовых поля `TextBox1` (`Edit1` в Lazarus) и `TextBox2` (`Edit2` в Lazarus);
   - для вывода чисел — надписи `Label1`, `Label2` и `Label3`;
   - для создания обработчика события — кнопку `Button1`;
   - для вывода поясняющих текстов — пять надписей.
4. На вкладке *TabPage2* (*TabSheet2* в Lazarus) разместить:
   - для ввода символа и строки — два текстовых поля `TextBox3` (`Edit3` в Lazarus) и `TextBox4` (`Edit4` в Lazarus);
   - для вывода строки — надпись `Label9`;
   - для создания обработчика события — кнопку `Button2`;
   - для вывода поясняющих текстов — три надписи.
5. На вкладке *TabPage3* (*TabSheet3* в Lazarus) разместить:
   - для вывода логического значения — надпись `Label13`;

- для создания обработчика события — кнопку Button3;
- для вывода поясняющих текстов — пять надписей.

На языке программирования создать обработчик события, реализующий вывод значений переменных. Ниже для каждой вкладки приведён код на одном из языков программирования.

Создание программного кода на языке Visual Basic .NET

```
Dim bytA, bytB As Byte, intC As Integer,
    sngD As Single, dblE As Double
Private Sub Button1_Click(sender As Object, e As
            EventArgs) Handles Button1.Click
    bytA = Val(TextBox1.Text)
    bytB = CByte(TextBox2.Text)
    intC = bytA / bytB
    sngD = bytA / bytB
    dblE = bytA / bytB
    Label1.Text = intC
    Label2.Text = sngD
    Label3.Text = dblE
  End Sub
```

Создание программного кода на языке программирования Visual C#

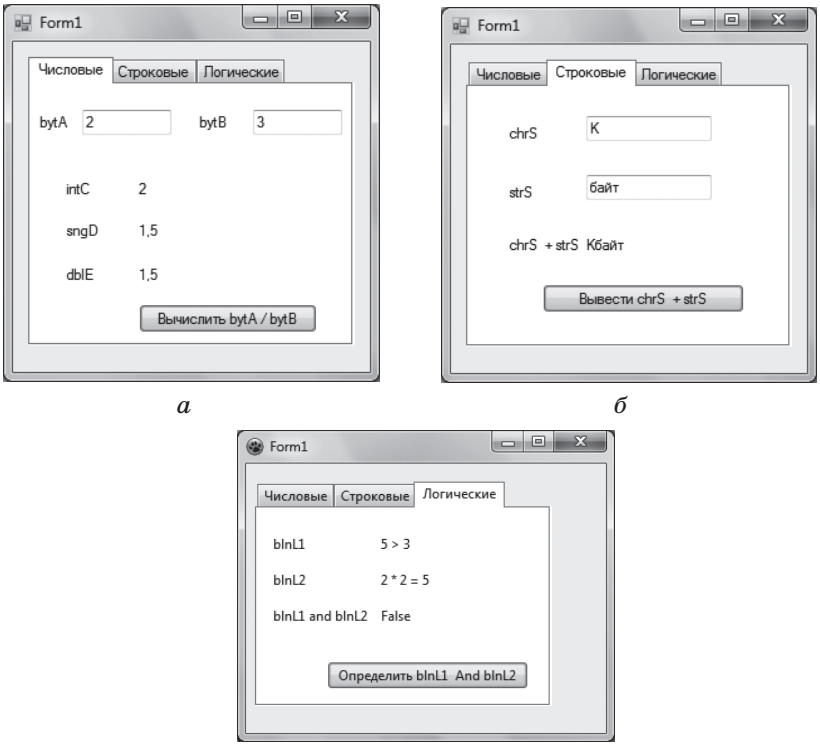
```
char chrS;
String strS;
private void button2_Click(object sender,
                           EventArgs e)
{chrS = Convert.ToChar(textBox3.Text);
  strS = textBox4.Text;
  label9.Text = chrS + strS;
}
```

Создание программного кода на языке Lazarus

```
var
blnL1: boolean;
blnL2: boolean;
blnL3: boolean;
procedure TForm1.Button3Click(Sender: TObject);
begin
  blnL1 := 5 > 3;
  blnL2 := 2 * 2 = 5;
  blnL3 := blnL1 and blnL2;
  Label13.Caption := BoolToStr(blnL3,True);
end;
```

Запуск проекта на языках Visual Basic .NET, Visual C# и Lazarus

1. Ввести команду [*Отладка—Начать*] (на языке Lazarus [*Запуск—Запустить*]) или щёлкнуть по кнопке ▶ на панели инструментов окна интегрированной среды разработки.
2. На вкладке *Числовые* в текстовые поля ввести числа и щёлкнуть по кнопке *Вычислить bytA/bytB*. На надписи будут выведены значения числовых переменных различных типов (рис. 4.22, *а*).
   На вкладке *Строковые* в текстовые поля ввести символ и строку и щёлкнуть по кнопке *Вывести chrS + strS*. На надпись будет выведена строка (рис. 4.22, *б*).
   На вкладке *Логические* щёлкнуть по кнопке *Определить lblL1 And lblL2*. На надпись будет выведено значение логического выражения (рис. 4.22, *в*).



а *б* *в*

Рис. 4.22

Практическая работа 4.3

Создание проекта «Отметка»

Задание. Создать в системе объектно-ориентированного программирования Visual Studio или Lazarus проект, который в зависимости от количества ошибок, введённых с использованием различных элементов управления (списка, поля со списком, текстового поля, счётчика, ползунка и переключателей), выводит на надпись отметку.

Электронное приложение к главе 4.

Создание графического интерфейса проекта на языках Visual Basic .NET, Visual C# и Lazarus

1. Разместить на форме (рис. 4.23):
   - для ввода количества ошибок — список **ListBox1**, поле со списком **ComboBox1**, текстовое поле **TextBox1** (**Edit1** в Lazarus), счётчик **NumericUpDown1** (**SpinEdit1** в Lazarus), ползунок **TrackBar1** и переключатели **RadioButton1**, **RadioButton2**, **RadioButton3**, **RadioButton4**, **RadioButton5**;
   - для вывода отметки — надпись **Label1**;
   - для вывода поясняющих текстов — надписи **Label2** и **Label3**.
2. Сделать графический интерфейс более привлекательным. Для этого присвоить форме и элементам управления новые значения свойств с помощью диалогового окна *Свойства* (*Инспектор объектов* в Lazarus).

Для каждого элемента управления существует одно событие (событие по умолчанию), чаще всего возникающее для данного элемента управления (табл. 4.3). После двойного щелчка по элементу управления в редакторе программного кода открывается заготовка процедуры обработки такого события.

3. Создать программный код.

На языке программирования Visual Basic .NET объявим переменную и создадим обработчик события, реализующий вывод отметки на надпись в зависимости от количества ошибок, выбранных в списке. Создадим заготовку обработчика события щелчком по списку и введём программный код. Для преобразования количества ошибок, выбранного как элемент списка строкового типа, в целое число используем функцию **CByte()**. Для реализации выбора отметки используем оператор **Select Case**.

Таблица 4.3

События по умолчанию некоторых элементов управления в языках Visual Basic .NET, Visual C# и Lazarus

Элемент управления	Visual Basic .NET	Visual C#	Lazarus
Текстовое поле	TextChanged	TextChanged	Change
Надпись	Click	Click	Click
Список	SelectedIndexChanged	SelectedIndexChanged	Click
Поле со списком	SelectedIndexChanged	SelectedIndexChanged	Change
Счётчик	ValueChanged	ValueChanged	Click
Ползунок	Scroll	Scroll	Change
Переключатель	CheckedChanged	CheckedChanged	Click

Создание программного кода на языке программирования Visual Basic .NET

```
Dim N As Byte
Private Sub ListBox1_SelectedIndexChanged
            (sender As Object, e As EventArgs)
Handles ListBox1.SelectedIndexChanged
N = CByte(ListBox1.Text)
Select Case N
  Case 0
  Label1.Text = "Отлично"
  Case 1
  Label1.Text = "Хорошо"
  Case 2
  Label1.Text = "Удовлетворительно"
  Case 3
  Label1.Text = "Плохо"
  Case Else
  Label1.Text = "Очень плохо"
End Select
End Sub
```

На языке программирования Lazarus объявим переменную и создадим обработчик события, реализующий вывод отметки на надпись в зависимости от количества ошибок, введённых в текстовое поле. Для преобразования введённого в текстовое поле количества ошибок строкового типа в целое число используем функцию **StrToInt()**. Для реализации выбора отметки используем оператор **Case**.

Создание программного кода на языке программирования Lazarus

```
var
N: byte;
procedure TForm1.Edit1Change(Sender: TObject);
begin
N := StrToInt(Edit1.Text);
Case N Of
  0: Label1.Caption := 'Отлично';
  1: Label1.Caption := 'Хорошо';
  2: Label1.Caption := 'Удовлетворительно';
  3: Label1.Caption := 'Плохо';
  Else Label1.Caption := 'Очень плохо';
end;
end;
```

На языке программирования Visual C# объявим переменную и создадим обработчик события, реализующий вывод отметки на надпись в зависимости от количества ошибок, введённых с помощью ползунка. Так как свойство ползунка **Value** имеет тип **int**, объявим переменную такого же типа. Для реализации выбора отметки используем оператор **switch**.

Создание программного кода на языке программирования Visual C#

```
int n;
private void trackBar1_Scroll(object sender,
                              EventArgs e)
{n = trackBar1.Value;
  switch (n)
  {case 0:
  label1.Text = "Отлично";
  break;
  case 1:
  label1.Text = "Хорошо";
  break;
  case 2:
  label1.Text = "Удовлетворительно";
  break;
  case 3:
  label1.Text = "Плохо";
  break;
  default:
  label1.Text = "Очень плохо";
  break;
  }
}
```

Запуск проекта на языках Visual Basic .NET, Visual C# и Lazarus

Загрузка проекта в систему объектно-ориентированного программирования производится путём активизации в папке проекта основного файла проекта:

- язык Visual Basic .NET — файл с расширением vbproj;
- язык Visual C# — файл с расширением csproj;
- язык Lazarus — файл с расширением lpr.

Запустим проект, после этого система программирования перейдёт в режим выполнения проекта [*Отладка*], в котором редактирование графического интерфейса или программного кода невозможно.

1. Ввести команду [*Отладка—Начать отладку*] ([*Запуск—Запустить*] на языке Lazarus) или щёлкнуть по кнопке ▶ на панели инструментов окна интегрированной среды разработки.
2. Ввести количество ошибок (например, 2) с использованием элементов управления:
   - списка;
   - поля со списком;
   - текстового поля;
   - счётчика;
   - ползунка;
   - одного из переключателей.

   На надпись будет выведена отметка (в данном случае «Удовлетворительно») (см. рис. 4.23).

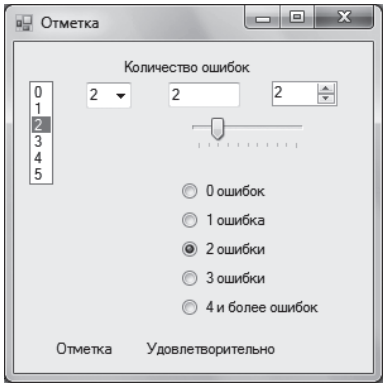


Рис. 4.23

Практическая работа 4.4

Создание проекта «Перевод целых чисел»

Задание. Создать проект, реализующий перевод целых десятичных чисел в двоичную систему счисления.

Электронное приложение к главе 4.

Создание графического интерфейса проекта на языках Visual Basic .NET, Visual C# и Lazarus

1. Разместить на форме (рис. 4.24):
   - текстовое поле TextBox1 (Edit1 в Lazarus) для ввода целого числа в десятичной системе счисления;
   - надпись Label1 для вывода целого двоичного числа;
   - кнопку Button1 для создания обработчика события, реализующего перевод целых чисел в двоичную систему счисления;
   - надписи Label2 и Label3 для вывода поясняющей информации.

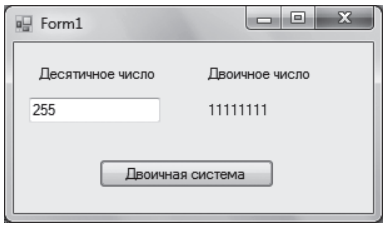


Рис. 4.24

2. Сделать графический интерфейс более привлекательным. Присвоить элементам управления новые значения свойств с помощью диалогового окна *Свойства*.

Создание обработчика события, реализующего перевод целых десятичных чисел в двоичную систему счисления, на языке программирования Visual Basic .NET

В языке программирования Visual Basic .NET комментарий в строке начинается с символа апострофа «'». В языке программирования Visual C# строки комментария начинаются с двух символов косой черты «//»; в языке Lazarus комментарий заключается в фигурные скобки «{}».

1. Программный код с комментариями:

```
Dim N As Integer 'десятичное число
Dim R As Integer 'остаток от деления исходного
                 'целого десятичного числа или
                 'целого частного на основание
                 'новой системы
Dim Bin As String 'двоичное число в строковой форме
Private Sub Button1_Click(sender As Object,
e As EventArgs) Handles Button1.Click
  '1.Ввести десятичное целое число и другие
  'начальные значения
     N = Val(TextBox1.Text)
     Label1.Text = ""
```

```
        Bin = ""
'2.В цикле с предусловием, пока исходное целое
'десятичное число или целое частное больше 0,
'выполнить вычисления:
        Do While N > 0
'2.1.Вычислить остаток от деления исходного целого
'десятичного числа или целого частного на основание
'новой системы (на 2).
          R = N Mod 2
'2.2.Выполнить целочисленное деление целого
'десятичного числа или целого частного на основание
'новой системы (на 2).
          N = N \ 2
'2.3. Записать полученный остаток от деления слева
'от двоичного числа (остатки, записанные в обратном
'порядке, образуют двоичное число).
          Bin = CStr(R) + Bin
        Loop
'3.Вывести двоичное целое число
        Label1.Text = Bin
        End Sub
```

Создание обработчика события, реализующего перевод целых десятичных чисел в двоичную систему счисления, на языке программирования Visual C#

Программный код:

```
private void button1_Click(object sender, EventArgs e)
        {
            int N;
            int R;
            string Oct;
                N = Convert.ToInt32(textBox1.Text);
                label1.Text = "";
                Oct = "";
                while (N > 0)
                    {R = N % 2;
                     N = Convert.ToInt32(N/2);
                     Oct = Convert.ToString(R)+ Oct;
                    }
                label1.Text = Oct;
        }
```

Создание обработчика события, реализующего перевод целых десятичных чисел в двоичную систему счисления, на языке программирования Lazarus

Программный код:

```
procedure TForm1.Button1Click(Sender: TObject);
var N, R: integer;
       Bin: string;
begin
     N := StrToInt(Edit1.Text);
     Label1.Caption := '';
     Bin := '';
     while N > 0 Do
         begin
             R := N mod 2;
             N := N div 2;
             Bin := FloatToStr(R) + Bin;
         end;
     Label1.Caption := Bin;
end;
```

Запуск проекта на языках Visual Basic .NET, Visual C# и Lazarus

Запустить проект на выполнение и ввести в текстовое поле десятичное число (например, 255).

Щёлкнуть по кнопке — на надписи будет выведено двоичное число (см. рис. 4.18).

ЭОР к главе 4 на сайте ФЦИОР (http://fcior.edu.ru)

- Алгоритмы. Ветвление и выбор
- Блок-схемы. Циклический алгоритм
- Идентификаторы, ключевые слова, литералы
- Общий обзор .NET и синтаксиса C#
- Понятие алгоритма, его свойства. Линейный алгоритм
- Теория алгоритмов. Основные понятия