Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PASCAL В СРЕДЕ LAZARUS

Учебное пособие для студентов и преподавателей вузов

Петрозаводск
Издательство ПетрГУ
2013

УДК 681.3.06
ББК 32.973.2-018
П784

Печатается по решению редакционно-издательского совета Петрозаводского государственного университета

Издается в рамках реализации комплекса мероприятий Программы стратегического развития ПетрГУ на 2012-2016 гг.

Рецензенты:
А. Г. Варфоломеев, кандидат физико-математических наук, доцент ПетрГУ;
Ю. В. Маркаданов, кандидат технических наук, доцент ПетрГУ

П784 **Программирование на языке Pascal в среде Lazarus**: учебное пособие для студентов и преподавателей вузов / сост. : В. Б. Ефлов, Ю. В. Никонова; – Петрозаводск : Изд-во ПетрГУ, 2013. – 53 с.

ISBN 978-5-8021-1702-6

В учебном пособии описываются интерфейс системы визуального программирования Lazarus, состав и характеристика элементов проекта приложения, приемы программирования на языке Object Pascal, дается описание синтаксических конструкций языка, операторы, директивы. Приводится методика работы с основными типами данных. Рассматриваются визуальные компоненты, используемые для создания интерфейса приложений; техника работы с текстовой информацией, кнопками и переключателями, а также формами, которые являются центральной частью любого приложения.

Учебное пособие предназначено для студентов и преподавателей, а также для школьников и лиц, самостоятельно изучающих программирование на языке «Паскаль» в среде Lazarus.

УДК 681.3.06
ББК 32.973.2-018

ISBN 978-5-8021-1702-6

СОДЕРЖАНИЕ

1. ЯЗЫК ПРОГРАММИРОВАНИЯ OBJECT PASCAL

1.1. Основные понятия

Паскаль – язык профессионального программирования, который назван в честь французского математика и философа Блеза Паскаля (1623–1662) и разработан в 1968–1971 гг. Никлаусом Виртом. Первоначально был предназначен для обучения, но вскоре стал использоваться для разработки программных средств в профессиональном программировании.

Паскаль популярен по следующим причинам:

1. Прост для обучения.
2. Отражает фундаментальные идеи алгоритмов в легко воспринимаемой форме, что предоставляет программисту средства, помогающие проектировать программы.
3. Позволяет четко реализовать идеи структурного программирования и структурной организации данных.
4. Использует простые и гибкие структуры управления: ветвления, циклы.
5. Надежен для разрабатываемых программ.

Программы на языке Паскаль имеют блочную структуру:

1. Блок типа PROGRAM – имеет имя, состоящее только из латинских букв и цифр. Его присутствие не обязательно, но рекомендуется для быстрого распознавания нужной программы среди других листингов.

2. Программный блок, состоящий в общем случае из 7 разделов:

· раздел описания модулей (uses);

· раздел описания меток (label);

· раздел описания констант (const);

· раздел описания типов данных (type);

· раздел описания переменных (var);

· раздел описания процедур и функций;

· раздел описания операторов.

Заголовок программы начинается со слова *Program* (программа), за которым следует произвольное имя, придуманное программистом:

Program <имя программы>;

Раздел описания переменных начинается со слова *Var* (variables — переменные), за которым идет список имен переменных через запятую. Тип указывается после двоеточия.

В стандарте языка Паскаль существуют два числовых типа величин: вещественный и целый. Слово *integer* обозначает *целый* тип (является идентификатором целого типа). *Вещественный* тип обозначается словом *real*. Например, раздел описания переменных может быть таким:

var a, b: integer; c, d: real;

Идентификаторы переменных составляются из латинских букв и цифр; первым символом обязательно должна быть буква.

Раздел операторов – основная часть программы. Начало и конец раздела операторов программы отмечаются служебными словами *begin* (начало) и *end* (конец). В самом конце программы ставится точка:

begin

< операторы >

end.

1.2. Типы данных

Тип определяет множество значений, которые могут принимать элементы программы, и совокупность операций, допустимых над этими значениями.

Например, значения -34 и 67 относятся к целочисленному типу и их можно умножать, складывать, делить и выполнять с ними другие арифметические операции, а значения abed и sdfhi23 относятся к строковому типу, и их можно сцеплять (складывать), но нельзя делить или вычитать.

Типы данных можно разделить на следующие группы:

· простые;

· структурные;

· указатели;

· процедурные;

· вариантные.

Простые и структурные типы включают в свой состав другие типы, например целочисленные или массивы. Приводимое деление на типы в некоторой мере условно — иногда указатели причисляют к простым типам, а строки, которые относятся к структурным типам, выделяют в отдельный тип.

Важное значение имеет понятие совместимости типов, которое означает, что типы равны друг другу или один из них может быть автоматически преобразован к другому. Совместимыми, например, являются вещественный и целочисленный тип, так как целое число автоматически преобразовывается в вещественное, но не наоборот.

1.1.1. Простые типы данных

Простые типы не содержат в себе других типов, и данные этих типов могут одновременно содержать одно значение. К ***простым*** относятся следующие типы:

· целочисленные;

· литерные (символьные);

· логические (булевы);

· вещественные.

Все типы, кроме вещественного, являются *порядковыми,* то есть значения каждого из этих типов образуют упорядоченную конечную последовательность. Номера соседних значений в ней отличаются на единицу.

Целочисленные типы включают целые числа. Наиболее часто используется тип *integer*, допускающий значения в диапазоне от -2 147 483 648 до 2 147 483 647.

Для записи целых чисел можно использовать цифры и знаки плюса и минуса, если знак числа отсутствует, то число считается положительным. При этом число может быть представлено как в десятичной, так и в шестнадцатиричной системе счисления. Если число записано в шестнадцатиричной системе, то перед ним ставится знак $ (без пробела), а допустимый диапазон значений — ОТ $00000000 ДО $FFFFFFFF.

Значениями ***литерного типа*** являются элементы из набора литер, то есть отдельные символы. В Object Pascal определен литерный тип *char*, который занимает один байт, а для кодирования символов используется код американского национального института стандартов ANSI (American National Standards Institute).

В Object Pascal к ***логическому*** относится тип *Boolean*. Этот тип представлен двумя возможными значениями: *True* (истина) и *False* (ложь). Для представления логического значения требуется один байт памяти.

Интервальные типы описываются путем задания двух констант, определяющих границы допустимых для данных типов значений. Эти границы и определяют интервал (диапазон) значений. Компилятор для каждой операции с переменной интервального типа, если это возможно, проверяет, находится ли значение переменной внутри установленного для нее интервала, и в случае его выхода за границы выдает сообщение об ошибке. Во время выполнения программы при выходе значения интервального типа за границы интервала сообщение об ошибке не выдается, однако значение переменной будет неверным. Интервал можно задать только для порядкового типа, то есть для любого простого типа, кроме вещественного. Обе константы, определяющие интервал, должны принадлежать одному из простых типов. Значение первой константы должно быть меньше значения второй. Формат описания интервального типа:

Type <Имя типа> = <Константа1> .. <Константа2>;

Вещественные (действительные) типы включают в себя вещественные числа. Наиболее часто используется тип *Real*, обеспечивающий точность 15–16 цифр мантиссы.

Запись вещественных чисел возможна в форме с фиксированной и в форме с плавающей точкой. Вещественные числа с фиксированной точкой записываются по обычным правилам арифметики. Целая часть отделяется от дробной десятичной точкой. Перед числом может указываться знак + или –. Если знак отсутствует, то число считается положительным. Для записи вещественных чисел с плавающей точкой указывается порядок числа со знаком, отделенный от мантиссы знаком E (или e).

1.1.2. Структурные типы данных

Структурные типы имеют в своей основе один или более других типов, в том числе и структурных. К структурным типам относятся:

· строки;

· записи;

· массивы;

· файлы;

· множества;

· классы.

Строки обеспечивает тип *string*, который представляет строку с максимальной длиной около 2 x 1031 символов. Символы строки кодируются в коде ANSI. Так как строки фактически являются массивами символов, то для обращения к отдельному символу строки можно указать название строковой переменной и номер (позицию) этого символа в квадратных скобках, например *strName* [i] .

Массивом называется упорядоченная индексированная совокупность однотипных элементов, имеющих общее имя. Элементами массива могут быть данные различных типов, включая структурированные. Каждый элемент массива однозначно определяется *именем* массива и *индексом* (номером этого элемента в массиве) или индексами, если массив многомерный. Для обращения к отдельному элементу массива указываются имя этого массива и номер (номера) элемента, заключенный в квадратные скобки, например:

arrl[3, 35], arrl[3] [35] ИЛИ arr3[7].

Количество индексных позиций определяет мерность массива (одномерный, двумерный и т. д.), при этом мерность массива не ограничивается. В математике аналогом одномерного массива является вектор, а двумерного массива — матрица. Индексы элементов массива должны принадлежать порядковому типу. Разные индексы одного и того же массива могут иметь различные типы. Наиболее часто типом индекса является целочисленный тип.

Множество представляет собой совокупность элементов, выбранных из предопределенного набора значений. Все элементы множества принадлежат одному порядковому типу, число элементов в множестве не может превышать 256. Формат описания множественного типа:

Set of <Тип элементов>;

Переменная множественного типа может содержать любое количество элементов своего множества — от нуля до максимального. Значения множественного типа заключаются в квадратные скобки. Пустое множество обозначается как [ ].

1.2. Выражения

При выполнении программы осуществляется обработка данных, в ходе которой с помощью выражений вычисляются и используются различные значения. ***Выражение*** представляет собой конструкцию, определяющую состав данных, операции и порядок выполнения операций над данными. Выражение состоит из:

· операндов;

· знаков операций;

· круглых скобок.

В простейшем случае выражение может состоять из одной переменной или константы. Тип значения выражения определяется типом операндов и составом выполняемых операций.

Операнды представляют собой данные, над которыми выполняются действия. В качестве операндов могут использоваться константы (литералы), переменные, элементы массивов и обращения к функциям.

Операции определяют действия, которые выполняются над операндами.

Операции могут быть унарными и бинарными. ***Унарная операция*** относится к одному операнду, и ее знак записывается перед операндом, например –x.

Бинарная операция выражает отношение между двумя операндами, и знак ее записывается между операндами, например X+Y.

Круглые скобки используются для изменения порядка выполнения операций.

В зависимости от типов операций и операндов выражения могут быть: *арифметическими, логическими* и *строковыми.*

1.3. Операторы

Операторы представляют собой законченные предложения языка, которые выполняют некоторые действия над данными. Операторы Pascal разделить на две группы:

· простые;

· структурированные.

Например, к простым операторам относится оператор присваивания, к структурированным — операторы разветвлений и циклов.

Правила записи операторов:

Операторы разделяются точкой с запятой. Точка с запятой является разделителем операторов, и ее отсутствие между операторами является ошибкой.

Наличие между операторами нескольких точек с запятой не является ошибкой, так как они обозначают пустые операторы. Отметим, что лишняя точка с запятой в разделе описаний и объявлений является синтаксической ошибкой.

Точка с запятой может не ставиться после слова *begin* и перед словом *end*, так как они являются операторными скобками, а не операторами.

В условных операторах и операторах выбора точка с запятой не ставится после слова *then* и перед словом *else*. Отметим, что в операторе цикла с параметром наличие точки с запятой сразу после слова *do* синтаксической ошибкой не является, но в этом случае тело цикла будет содержать только пустой оператор.

1.3.1. Простые операторы

Простыми называются операторы, не содержащие в себе других операторов.

К ним относятся:

· оператор присваивания;

· оператор перехода;

· пустой оператор;

· оператор вызова процедуры.

Оператор присваивания является основным оператором языка. Он предписывает вычислить выражение, заданное в его правой части, и присвоить результат переменной, имя которой расположено в левой части оператора. Переменная и выражение должны иметь совместимый тип, например вещественный и целочисленный, но не наоборот. Допустимо присваивание любого типа данных, кроме файлового. Формат оператора присваивания:

<Имя переменной> := <Выражение>;

Вместо имени переменной можно указывать элемент массива или поле записи. Отметим, что знак присваивания := отличается от знака равенства = и имеет другой смысл. Он означает, что сначала вычисляется значение выражения и затем оно присваивается указанной переменной. Поэтому при условии, что x является числовой переменной и имеет определенное значение, будет допустимой и правильной следующая конструкция: x := x + 1;

Пример. Операторы присваивания

```
Var x, y: real;
n: integer;
stroka: string;
n := 17 * n - 1;
stroka := 'Дата ' + DateToStr(Date);
x := -12.3 * sin(pi / 4);
y := 23.789E+3;
```

Оператор перехода предназначен для изменения естественного порядка выполнения операторов программы. Он используется в случаях, когда после выполнения некоторого оператора требуется выполнить не следующий по порядку, а какой-либо другой, помеченный меткой оператор. Метка, стоящая перед оператором, отделяется от него двоеточием. Меткой может быть идентификатор или целое число без знака в диапазоне 0 – 9999, и все метки должны быть предварительно объявлены в разделе объявления меток того блока процедуры, функции или программы, в котором эти метки используются. Формат оператора перехода: *goto* <Метка>;

Пример. Использование оператора перехода.

```
Label ml;
goto ml;
ml: <Оператор>;
```

Передавать управление с помощью оператора перехода можно на операторы, расположенные в тексте программы выше или ниже оператора перехода.

Запрещается передавать управление операторам, находящимся внутри структурированных операторов, а также операторам, находящимся в других блоках (процедурах, функциях).

Пустой оператор представляет собой точку с запятой и может быть расположен в любом месте программы, где допускается наличие оператора. Как и другие операторы, пустой оператор может быть помечен меткой. Пустой оператор не выполняет никаких действий и может быть использован для передачи управления в конец цикла или составного оператора.

Оператор вызова процедуры служит для активизации стандартной или предварительно описанной пользователем процедуры и представляет собой имя этой процедуры со списком передаваемых ей параметров.

1.3.2. Структурированные операторы

Структурированные операторы представляют собой конструкции, построенные по определенным правилам из других операторов. К структурированным операторам относятся:

· составной оператор;

· операторы цикла (повтора);

· условный оператор;

· оператор доступа.

· операторы выбора;

Составной оператор представляет собой группу из произвольного числа любых операторов, отделенных друг от друга точкой с запятой, и ограниченную операторными скобками *begin* и *end*. Формат составного оператора:

begin <Оператор1>; ... ; <ОператорЫ>; end;

Независимо от числа входящих в него операторов, составной оператор воспринимается как единое целое и может располагаться в любом месте программы, где допускается наличие оператора. Наиболее часто составной оператор используется в условных операторах и операторах цикла. Составные операторы могут вкладываться друг в друга.

Пример. Составной оператор.

```
begin
Beep;
Editl.Text := 'Ошибка';
Exit;
end;
```

Условный оператор обеспечивает выполнение или невыполнение некоторых операторов в зависимости от соблюдения определенных условий. Условный оператор в общем случае предназначен для организации разветвления программы на два направления и имеет формат:

if <Условие> then <0ператор1> [else <0ператор2>];

Условие представляет собой выражение логического типа. Оператор работает следующим образом: если условие истинно (имеет значение *True*), то выполняется оператор1, в другом случае выполняется оператор2. Оба оператора могут быть составными. Условный оператор может быть записан в сокращенной форме, когда слово *else* и оператор после него отсутствуют. В этом случае при невыполнении условия не выполняется никакой оператор.

Для организации разветвлений на три направления и более можно использовать несколько условных операторов, вложенных друг в друга. При этом каждое *else* соответствует тому *then*, которое непосредственно ему предшествует. Из-за возможных ошибок следует избегать большой вложенности условных операторов друг в друга.

Пример. Условные операторы

```
if x > 0 then x := x + 1 else x := 0;
if q = 0 then a := 1;
```

Оператор выбора является обобщением условного оператора и позволяет сделать выбор из произвольного числа имеющихся вариантов, то есть организовать разветвления на произвольное число направлений. Этот оператор состоит из выражения, называемого **селектором**, списка вариантов и необязательной ветви *else*, имеющей тот же смысл, что и в условном операторе.

Формат оператора выбора:

```
case <Выражение-селектор> of
<Список1> : <Оператор1>;
<CnncoKN> : <OnepaTopN>
else <Оператор>;
end;
```

Выражение-селектор должно быть порядкового типа. Каждый вариант представляет собой список констант, отделенных двоеточием от относящегося к данному варианту оператора, возможно, составного. Список констант выбора состоит из произвольного количества значений и диапазонов, отделенных друг от друга запятыми. Границы диапазона записываются двумя константами через разделитель "..". Тип констант должен совпадать с типом выражения-селектора.

Оператор выбора выполняется следующим образом:

· вычисляется значение выражения селектора;

· производится последовательный просмотр вариантов на предмет совпадения значения селектора с константами и значениями из диапазонов соответствующего списка;

· если для очередного варианта этот поиск успешный, то выполняется оператор этого варианта. После этого выполнение оператора выбора заканчивается;

· если все проверки оказались безуспешными, то выполняется оператор, стоящий после слова *else* (при его наличии).

Пример. Оператор выбора

```
case DayNumber of
1 .. 5 : strDay := 'Рабочий день';
6, 7 : strDay := 'Выходной день'
else strDay := '';
end;
```

В зависимости от значения целочисленной переменной DayNumber, содержащей номер дня недели, присваивается соответствующее значение строковой переменной *strDay*.

Операторы цикла используются для организации циклов (повторов). **Цикл** представляет собой последовательность операторов, которая может выполняться более одного раза. Группу повторяемых операторов называют телом цикла. Для построения цикла в принципе можно использовать ранее рассмотренные условный оператор и оператор перехода. Однако в большинстве случаев удобно использовать операторы цикла. Всего имеется три вида операторов цикла:

· с параметром;

· с предусловием;

· с постусловием.

Обычно, если количество повторов известно заранее, то применяется оператор цикла с параметром, в противном случае — операторы с пост- или предусловием (чаще используется оператор с предусловием).

Выполнение оператора цикла любого вида может быть прервано с помощью оператора перехода *goto* или предназначенной для этих целей процедуры без параметров *Break*, которая передает управление на следующий за оператором цикла оператор.

С помощью процедуры без параметров *continue* можно задать досрочное завершение очередного повторения тела цикла, что равносильно передаче управления в конец тела цикла.

Операторы циклов могут быть вложенными друг в друга.

Оператор цикла с параметром имеет два следующих формата:

for <Параметр> := <Выражение1> to <Выражение2> do <Оператор>;

и

for <Параметр> := <Выражение1> downto <Выражение2> do <Оператор>;

Параметр цикла представляет собой переменную порядкового типа, которая должна быть определена в том же блоке, где находится оператор цикла, <Выражение1> и <Выражение2> являются соответственно начальным и конечным значениями параметра цикла и должны иметь тип, совместимый с типом параметра цикла.

Оператор цикла обеспечивает выполнение тела цикла, которым является оператор после слова *do*, до полного перебора всех значений параметра цикла от начального до конечного с соответствующим шагом. Шаг параметра всегда равен 1 для первого формата цикла и –1 — для второго формата.

То есть значение параметра последовательно увеличивается (*for … to*) или уменьшается (*for … downto*) на единицу при каждом повторении цикла.

Цикл может не выполниться ни разу, если для цикла *for ... to* значение начального выражения больше конечного, а для цикла *for ... downto*, наоборот, значение начального выражения меньше конечного.

Пример. Циклы с параметром

```
var n, k: integer;
s := 0;
for n := 1 to 10 do s := s + m[n];
for к := 0 to 2 do
for n := 5 to 10 do begin
arrl[к, n] := 0;
arr2[k, n] := 1;
end;
```

В первом цикле выполняется расчет суммы из десяти значений массива Т. Во втором случае два цикла вложены один в другой, и в них пересчитываются значения элементов двумерных массивов arrl и агг2.

Оператор цикла с предусловием целесообразно использовать в случаях, когда число повторений тела цикла заранее неизвестно и тело цикла может не выполняться. Во многом этот оператор аналогичен оператору *repeat...until*, но проверка условия выполняется в начале оператора.

Формат оператора цикла с предусловием:

while <Условие> do <Оператор>;

Оператор тела цикла выполняется до тех пор, пока логическое выражение не примет значение *False*, то есть, в отличие от цикла с постусловием, цикл выполняется при значении логического выражения *True*.

Пример. Оператор цикла с предусловием

Рассмотрим расчет суммы из десяти значений массива т.

```
var x: integer;
sum: real;
т: array[l .. 10] of real;
x : = 1; sum := 0;
while x <= 10 do begin
sum := sum + m[x];
x := x + 1;
end;
```

Если перед первым выполнением цикла условие не выполняется (значение логического выражения равно *False*), то тело цикла не выполняется ни разу, и происходит переход на следующий за оператором цикла оператор.

Оператор доступа служит для удобной и быстрой работы с составными частями объектов, в том числе с полями записей. Напомним, что для обращения к полю записи необходимо указывать имя записи и имя этого поля, разделенные точкой. Аналогичным путем образуется имя составной части какого-либо объекта, например, формы или кнопки. Оператор доступа имеет следующий основной формат:

with <Имя объекта> do <Оператор>

В операторе, расположенном после слова *do*, для обращения к составной части объекта можно не указывать имя этого объекта, которое уже задано после слова *with*.

Пример. Использование оператора доступа

```
// Составные имена пишутся полностью
Forml.Canvas.Pen.Color := clRed;
Forml.Canvas.Pen.Width := 5;
Forml.Canvas.Rectangle(10, 10, 100, 100);
ИЛИ
// Использование оператора доступа
with Forml.Canvas do begin
Pen.Color := clRed;
Pen.Width := 5;
Rectangle(10, 10, 100, 100);
end;
```

В обоих приведенных примерах на форме красным пером толщиной пять пикселов рисуется прямоугольник. Для обращения к свойствам и методу (процедуре) поверхности рисования формы удобно использовать оператор доступа (второй вариант).

1.5. Подпрограммы

Подпрограмма представляет собой группу операторов, логически законченную и специальным образом оформленную. Подпрограмму можно вызывать неограниченное число раз из различных частей программы. Использование подпрограмм позволяет улучшить структурированность программы и сократить ее размер.

По структуре подпрограмма почти полностью аналогична программе и содержит заголовок и блок, однако в блоке подпрограммы отсутствует раздел подключения модулей. Кроме того, заголовок подпрограммы по своему оформлению отличается от заголовка программы.

Работа с подпрограммой имеет два этапа:

· описание подпрограммы;

· вызов подпрограммы.

Любая подпрограмма должна быть предварительно описана, после чего допускается ее вызов. При описании подпрограммы указывается ее имя, список параметров и выполняемые подпрограммой действия. При вызове подпрограммы указываются имя подпрограммы и список аргументов (фактических параметров), передаваемых подпрограмме для работы.

Программист может создавать свои подпрограммы, которые также называют пользовательскими.

Подпрограммы делятся на ***процедуры*** и ***функции***, которые имеют между собой много общего. Основное различие между ними заключается в том, что функция может возвращать под своим именем в качестве результата значение и соответственно может использоваться в качестве операнда выражения.

С подпрограммой взаимодействие осуществляется *по управлению* и *по данным*. Взаимодействие *по управлению* заключается в передаче управления из программы в подпрограмму и организации возврата в программу.

Взаимодействие *по данным* заключается в передаче подпрограмме данных, над которыми она выполняет определенные действия. Этот вид взаимодействия может осуществляться следующими основными способами:

· с использованием файлов;

· с помощью глобальных переменных;

· с помощью параметров.

Наиболее часто используется последний способ. При этом различают ***параметры*** и ***аргументы***.

Параметры (формальные параметры) являются элементами подпрограммы и используются при описании алгоритма, выполняемого подпрограммой.

Существует несколько видов параметров:

· значение;

· константа;

· переменная;

· нетипизированная константа и переменная.

Аргументы (фактические параметры) являются элементами вызывающей программы. Они замещают параметры при вызове подпрограммы. При этом осуществляется проверка на соответствие типов и количества параметров и аргументов. Имена параметров и аргументов могут различаться, однако их количество и порядок следования должны совпадать, а типы параметров и соответствующих им аргументов должны быть совместимыми.

Для прекращения работы подпрограммы можно использовать процедуру *Exit,* которая прерывает выполнение операторов подпрограммы и возвращает управление вызывающей программе.

1.6. Особенности объектно-ориентированного программирования

Язык Object Pascal является объектно – ориентированным расширением языка Pascal и реализует концепцию объектно – ориентированного программирования. Это означает, что создаваемое приложение состоит из объектов, которые взаимодействуют между собой. Каждый объект имеет свои свойства, то есть характеристики (атрибуты) этого объекта, методы, определяющие поведение этого объекта, и события, на которые реагирует объект.

В языке Object Pascal классы являются специальными типами данных и используются для описания объектов. Соответственно объект, имеющий тип какого-либо класса, является экземпляром этого класса или переменной этого типа.

Класс представляет собой особый тип записи, имеющий в своем составе такие элементы (члены), как поля, свойства и методы. ***Поля*** класса аналогичны полям записи и служат для хранения информации об объекте. ***Методами*** называются процедуры и функции, предназначенные для обработки полей. ***Свойства*** занимают промежуточное положение между полями и методами.

С одной стороны, свойства можно использовать как поля, например присваивая им значения с помощью оператора присваивания; с другой стороны, внутри класса доступ к значениям свойств выполняется методами класса.

Пример. Описание класса

```
type
TColorCircle = class(TCircle);
FLeft,
FTop,
FRight,
FBottom: Integer;
Color: TColor;
end;
```

Здесь класс *TColorCircle* создается на основе родительского класса *TCircle*. В отличие от родительского, новый класс дополнительно содержит четыре поля типа *integer* и одно поле типа *TColor*.

Если в качестве родительского используется класс *Tobject*, который является базовым классом для всех классов, то его имя после слова *class* можно не указывать. Тогда первая строка описания будет выглядеть так:

```
type TNewClass = class.
```

Для различных элементов класса можно устанавливать разные права доступа (видимости), для чего в описании класса используются отдельные разделы, обозначенные специальными спецификаторами видимости.

Разделы *private* и *protected* содержат защищенные описания, которые доступны внутри модуля, в котором они находятся. Описания из раздела *protected*, кроме того, доступны для порожденных классов за пределами названного модуля.

Раздел *public* содержит общедоступные описания, которые видимы в любом месте программы, где доступен сам класс.

Раздел *published* содержит опубликованные описания, которые в дополнение к общедоступным описаниям порождают динамическую информацию о типе (Run-Time Type Information, RTTI). По этой информации при выполнении приложения производится проверка на принадлежность элементов объекта тому или иному классу.

Одним из назначений раздела *published* является обеспечение доступа к свойствам объектов при конструировании приложений. В *Инспекторе объектов* видны те свойства, которые являются опубликованными.

Если спецификатор *published* не указан, то он подразумевается по умолчанию, поэтому любые описания, расположенные за строкой с указанием имени класса, считаются опубликованными.

1.6.1. Поля

Поле класса представляет собой данные, содержащиеся в классе. Поле описывается как обычная переменная и может принадлежать к любому типу.

Пример. Описание полей

```
type TNewCiass = class(TObject)
private
FCode: integer;
FSign: char:
FNote: string;
end;
```

Здесь новый класс *TNewCiass* создается на основе базового класса *Tobject* и получает в дополнение три новых поля *Fcode, FSign и FNote*, имеющих, соответственно, целочисленный, символьный и строковый типы. Согласно принятому соглашению, имена полей должны начинаться с префикса *f* (от англ. Field — поле).

При создании новых классов класс-потомок наследует все поля родителя, при этом удаление или переопределение этих полей невозможно.

Напомним, что изменение значений полей обычно выполняется с помощью методов и свойств объекта.

1.6.2. Свойства

Свойства реализуют механизм доступа к полям. Каждому свойству соответствует поле, содержащее значение свойства, и два метода, обеспечивающих доступ к этому полю. Описание свойства начинается со слова *property*, при этом типы свойства и соответствующего поля должны совпадать. Ключевые слова *read* и *write* являются зарезервированными внутри объявления свойства и служат для указания методов класса, с помощью которых выполняется чтение значения поля, связанного со свойством, или запись нового значения в это поле.

1.6.3. Методы

Метод представляет собой подпрограмму (процедуру или функцию), являющуюся элементом класса. Описание метода похоже на описание обычной подпрограммы модуля. Заголовок метода располагается в описании класса, а сам код метода находится в разделе реализации. Имя метода в разделе реализации является составным и включает в себя тип класса. Метод, объявленный в классе, может вызываться различными способами, что зависит от вида этого метода. Вид метода определяется

модификатором, который указывается в описании класса после заголовка метода и отделяется от заголовка точкой с запятой. Приведем некоторые модификаторы:

· *virtual* – виртуальный метод;

· *dynamic* – динамический метод;

· *override* – переопределяемый метод;

· *message* – обработка сообщения.

По умолчанию все методы, объявленные в классе, являются статическими и вызываются как обычные подпрограммы.

Методы, которые предназначены для создания или удаления объектов, называются *конструкторами* и *деструкторами*, соответственно. Описания данных методов отличаются от описания обычных процедур только тем, что в их заголовках стоят ключевые слова *constructor* и *destructor*. В качестве имен конструкторов и деструкторов в базовом классе *Tobject* и многих других классах используются имена *Create* и *Destroy*.

1.6.4. Сообщения и события

В основе многих операционных систем лежит использование механизма *сообщений*, которые «документируют» все производимые действия, например нажатие клавиши, передвижение мыши или тиканье таймера. Приложение получает сообщение в виде записи заданного типа. Система программирования преобразовывает сообщение в свой формат. Для обработки сообщений, посылаемых ядром операционной системы и различными приложениями, используются специальные методы, описываемые с помощью модификатора *message*, после которого указывается идентификатор сообщения. Метод обработки сообщения обязательно должен быть процедурой, имеющей один параметр, который при вызове метода содержит информацию о поступившем сообщении. Имя метода программист выбирает самостоятельно, для компилятора оно не имеет значения.

Обычно в системе программирования не возникает необходимость обработки непосредственных сообщений операционной системы, т. к. в распоряжение программиста предоставляются события, работать с которыми намного удобнее. *Событие* представляет собой свойство процедурного типа, предназначенное для обеспечения реакции на те или иные действия. Присваивание значения этому свойству (событию) означает указание метода, вызываемого при наступлении события. Соответствующие методы называют *обработчиками событий*.

1.6.5. Библиотека визуальных компонентов

Библиотека визуальных компонентов (Visual Component Library, VCL) содержит большое количество классов, предназначенных для быстрой разработки приложений. Библиотека написана на Object Pascal. В VCL содержатся невизуальные и визуальные

компоненты, а также другие классы, начиная с абстрактного класса *TObject*. При этом все компоненты являются классами, но не все классы являются компонентами.

Все классы VCL расположены на определенном уровне иерархии и образуют дерево (иерархию) классов. Знание происхождения объекта оказывает значительную помощь при его изучении, так как потомок наследует все элементы объекта-родителя. Так, если свойство *caption* принадлежит классу *TControi*, то это свойство будет и у его потомков, например у классов TButton и TCheckBox и у компонентов — кнопки Button и независимого переключателя CneckBox соответственно.

2. LAZARUS RAD И РАЗРАБОТКА В СРЕДЕ LAZARUS

2.1. Lazarus

Lazarus — бесплатная и свободная графическая среда разработки программного обеспечения на языке Object Pascal для компилятора **Free Pascal Compiler** (FPC), также свободного программного обеспечения. Оба продукта распространяются под лицензией GPL (*General Public License – универсальная общественная лицензия GNU, Универсальная общедоступная лицензия GNU* или *Открытое лицензионное соглашение GNU*).

Object Pascal (Объектный Паскаль) – язык программирования, разработанный в фирме Apple Computer в 1986 году группой Ларри Теслера, который консультировался с Никлаусом Виртом. Произошел от более ранней объектно – ориентированной версии языка Паскаль.

Lazarus является кросплатформенной средой разработки, т. е. позволяет создавать программы в графическом окружении, практически неизменном, для широкого класса операционных систем: Linux, FreeBSD, Mac OS X, Microsoft Windows, Android. Для создания исполняемой программы достаточно снова собрать ее в соответствующей операционной системе. Является практически единственной простой средой разработки, в которой можно создавать приложения программистам знакомым с Delphi.

Среда разработки базируется на библиотеке визуальных компонентов *Lazarus Component Library* (LCL), которая содержит достаточное число компонент, позволяющих создавать формы при помощи визуального проектирования графического интерфейса пользователя (GUI, англ. *graphical user interface).* В Lazarus также возможно разрабатывать консольные приложения и динамически подгружаемые библиотеки: в Windows dynamic-link library, или DLL; в OS X – библиотеки dylib (dynamic shared library); в Linux – библиотеки .so (shared object library). Динамические библиотеки используются для разработки других программ.

Языки интерфейса: русский, английский и еще 36 языков (апрель 2013 года).

2.2. Установка Lazarus для Linux, Windows

Наиболее прост вариант установки Lazarus в Ubuntu и его довольно распространенных формах, использующих различные рабочие окружения. Рассмотрим установку Lazarus в Xubuntu 12.04.2, где реализована среда окружения XFCE.

XFCE [2] — свободная среда рабочего стола для UNIX-подобных операционных систем, таких как Linux, NetBSD, OpenBSD, FreeBSD, Solaris и т. п. Конфигурация данной среды полностью управляется мышью, конфигурационные файлы скрыты от пользователя.

Приступим к установке, предполагая, что среда разработки еще не установлена,

что можно проверить в меню приложений Xubuntu в пункте разработка. Если там отсутствует приложение Lazarus, то далее в этом параграфе – информация для его установки.

Стартуем менеджер пакетов Synaptic, причем настоятельно рекомендуется, чтобы информация о системе система была обновлена до текущего актуального состояния репозиториев Ubuntu. Для этого в пункте меню Synaptic достаточно выбрать пункт «Обновить».

В строке «Быстрый поиск» набираем имя искомого приложения и, подводя курсор мыши к чекбоксу, рядом с приложением выбираем пункт меню «Отметить для установки». При этом все необходимые зависимости будут выбраны автоматически. Вам потребуется лишь согласиться с последующей их установкой. Далее в меню выбираем пункт «Применить». Через некоторое время приложение будет установлено. Lazarus установлен. В окне терминала можно прочитать информацию о версии, установленной ОС Xubuntu, и некоторых ее параметрах. Убедиться в успешности установки можно, выбрав в меню приложений раздел «Разработка», и запустив появившееся в разделе приложение, на момент написания текста пособия — это Lazarus (0.9.30.2-2). После запуска приложения в интерфейсе XFCE появится многооконное приложение среды разработки. Язык Lazarus основан на библиотеке визуальных компонентов *Lazarus Component Library* (LCL).

Установка Lazarus для Windows платформы практически ничем не отличается от установки для Linux. Предварительная подготовка состоит в получении с сайта http://sourceforge.net/projects/lazarus/files/Lazarus Windows 32 bits/ текущей актуальной версии пакета. После инсталляции и запуска приложения, значок которого по умолчанию будет установлен на рабочий стол, Вы увидите набор окон среды разработки Lazarus. В Linux, также как и в Windows, основными возможностями Lazarus являются: возможность легкого переноса Delphi-программ с графическим интерфейсом в различные ОС (Linux; FreeBSD, Mac OS X, Microsoft Windows); максимальная приближенность редактора форм и инспектора объектов к Delphi; интерфейс отладки; простой переход для Delphi программистов благодаря близости LCL к VCL; интерфейс и редактор, полностью поддерживающие UTF-8; мощный редактор кода, включающий систему подсказок, гипертекстовую навигацию по исходным текстам, автозавершение кода и рефакторинг; форматирование кода по умолчанию средствами встроенного редактора, используя механизмы Jedi Code Format; поддержка двух стилей ассемблер – Intel и AT&T; поддержка множества типов синтаксиса Pascal – Object Pascal, Turbo Pascal, Mac Pascal, Delphi; собственный формат управления пакетами.

2.3. IDE Lazarus

В данном разделе приведено текущее состояние интерфейса Lazarus. Далее все свойства среды разработки, как и примеры, будем рассматривать в окружении Lubuntu или Xubuntu. В качестве менеджеров рабочего стола используется LXDE, XFCE. Рабочий стол LXDE использует оконный менеджер Openbox и нетребователен к системным ресурсам.

Рассмотрим основные компоненты среды разработки, отметив, что элементы среды разработки, достаточно универсальны и встречаются практически во всех подобных средах RAD, например, в продуктах Microsoft, также как и в продуктах для других операционных систем.

Когда Lazarus стартует в первый раз, на рабочем столе появляется набор несвязанных окошек. Первое, расположенное в самом верху рабочего стола, имеет название **Lazarus Editor vXXXXXX – project1** (название зависит от используемой версии и названия открытого проекта). Это главное окно управления проектом и оно содержит *Главное Меню* и *Палитру Компонентов* (Component Palette).

Строкой ниже располагается *Главное Меню* с обычными пунктами *File*, *Edit*, *Search*, *View* и некоторыми специфичными для Lazarus. Ниже слева располагается набор кнопок, предоставляющих быстрый доступ к некоторым функциям главного меню, и справа – *Палитра Компонентов*.

Под окном редактора Lazarus слева располагается окно *Инспектора Объектов*, а справа от него – *Редактор Исходного кода* (Lazarus Source Editor). Могут присутствовать и другое окно меньшего размера, озаглавленное *Form1*, расположенное поверх *Редактора Исходного Кода*. Если его в данный момент не видно, то можно переключиться к нему, нажав клавишу **F12**, которая позволяет переключаться между *Редактором Исходного Кода* и *Окном Формы*. **Окно формы** – это то место, где вы разрабатываете графический интерфейс программы, а в **Редакторе Исходного Кода** отображается разрабатываемый вами Pascal-код приложения.

Когда начинается разработка нового проекта (или впервые запускается Lazarus), по умолчанию создается стандартная форма, которая обычно содержит кнопки **Свернуть**, **Развернуть** и **Закрыть**. Если вы щелкните мышкой в любом месте формы, вы увидите ее свойства в *Инспекторе Объектов* у левого края экрана.

Другие окна, которые могут появиться в процессе работы: *Инспектор Проектов*, содержащий сведения о файлах, включенных в проект, и позволяющий вам добавлять и удалять файлы из проекта; окно **Messages** (**Сообщения**), отображающее сообщения компилятора, ошибки и отчеты по вашему проекту.

2.3.1. Главное меню Lasarus

Главное текстовое меню содержит следующие пункты: **File, Edit, Search, View, Project, Run, Components, Tools, Environment, Windows, Help.** И в локализованном (русифицированном) варианте – **Файл, Правка, Поиск, Вид, Проект, Запуск, Пакет, Сервис, Окружение, Окно, Справка.**

Любой пункт можно выбрать, если навести на него курсор и нажать левую кнопку мыши или использовать горячие клавиши.

Меню *Файл* (File)

- **Создать модуль (New Unit)**: создать новый модуль (Unit, исходный код на Pascal).
- **Создать форму (New Form)**: создать новую форму. То есть два файла: саму форму и соответствующий ей файл с Pascal-кодом.
- **Создать... (New...)**: открывает всплывающее окно, в котором приведены различные типы проектов, которые можно создать.
- **Открыть (Open)**: открывает диалоговое окно, при помощи которого можно найти и открыть существующий файл.
- **Возвратить (Revert)**: отменяет сделанные изменения и возвращает файл в исходное состояние.
- **Сохранить (Save)**: сохранить текущий файл под этим же именем. Если имя еще не дано, то система скажет об этом (подобно пункту **Сохранить как**).
- **Сохранить как (Save As)**: позволяет выбрать папку и имя, под которым сохранить текущий файл.
- **Закрыть (Close)**: закрывает текущий файл с выводом сообщения о том, следует ли сохранить сделанные изменения.
- **Закрыть все файлы редактора (Close all editor files)**: закрывает все файлы, открытые в данный момент в редакторе. Выдается сообщение о сохранении изменений.
- **Очистить каталог (Clean directory)**: выводится диалог со строками задания фильтров, при помощи которых можно очистить текущий каталог.
- **Печать (Print)**: использует системный принтер для печати выбранного файла.
- **Перезапуск (Restart):** повторное включение Lazarus, что полезно, если рабочие файлы безнадежно испорчены.
- **Выход (Quit)**: выход из Lazarus с выводом запроса на сохранение всех открытых файлов.

Меню *Правка* (Edit)

- **Отменить (Undo):** отменяет последнее выполненное действие.
- **Вернуть (Redo):** вновь выполняет ранее отмененное действие.
- **Вырезать (Cut):** удаляет выделенный текст или другой элемент и помещает его в буфер обмена.
- **Копировать (Copy):** создает копию выделенного текста, не затрагивая текст, и помещает в буфер обмена.
- **Вставить (Paste):** помещает содержимое буфера обмена в позицию курсора. Если есть выделенный текст, то он будет заменен содержимым буфера обмена.
- **Увеличить отступ выделенного (Indent selection)**: сдвигает выделенный текст вправо на количество позиций, указанных в настройках **Окружение** => **Настройки редактора** => **Общие** => **Отступ блока** (Environment => Editor options => General => Block indent). Эта опция удобна для форматирования исходного Pascal-кода для создания блочной структуры.
- **Уменьшить отступ выделенного (Unindent selection)**: удаляет один уровень отступа, сдвигая текст влево на количество позиций, указанных в **Отступе блока** (Block indent).
- **Заключить выделенное в (Enclose selection)**: открывает всплывающее меню с набором опций для заключения выделенного текста в программные скобки (begin ... end; try ... except; try ... finally; repeat ... until; { ... } и т. д.).
- **Закомментировать (Comment selection)**: закомментировать выделенный текст, добавив в начало каждой строки //.
- **Раскомментировать (Uncomment selection):** удаляет символы комментария.
- **Верхний регистр выделенного (Uppercase selection)**: преобразует выделенный текст в верхний регистр.
- **Нижний регистр выделенного (Lowercase selection)**: преобразует выделенный текст в нижний регистр.
- **Преобразовать ТАБ в пробелы выделенном (Tabs to spaces in selection)**: преобразует все символы табуляции в выделенном тексте в некоторое количество пробелов, настраиваемое посредством **Окружение** => **Настройки редактора** => **Общие** => **Ширина ТАБа** (Environment => Editor options => General => Tab widths). Вставляется не фиксированное количество пробелов, а только число необходимое для завершения данного табулятора.
- **Разбить строки в выделенном (Break lines in selection)**: если строка выделенного текста длиннее 80 символов или числа указанного в **Окружение** => **Настройки редактора** => **Дисплей** => **Правая граница** (Environment => Editor options =>

Display => Right Margin), то строка текста будет разорвана на границе слова и продолжится со следующей строки.

- **Сортировать выделенное (Sort selection)**: сортирует строки (слова или параграфы) в алфавитном порядке.
- **Выделить (Select)**: выделяет текстовый блок.
- **Вставить из таблицы символов (Insert from character map)**: позволяет вставить специальные символы, такие как символы с акцентами, которые берутся из всплывающей таблицы символов.
- **Вставить текст (Insert text)**: вызывает всплывающее меню, позволяющее вставить шаблоны текста, такие как ключи CVS (Author, Date, Header и т. д.) или заметку о GPL, имя пользователя или текущую дату и время.
- **Завершить код (Complete code)**: завершает код под курсором. Действия зависят от контекста и позволяют сохранить довольно много времени.
- **Выделить процедуру (Extract procedure)**: использует выделенный текст (один или несколько операторов) для построения новой процедуры.

Меню *Поиск* (Search)

- **Найти (Find)**: позволяет ввести строку поиска, а также опции поиска, такие как «чувствительность к регистру», «искать целые слова», «целые выражения», «область и направление поиска».
- **Найти следующее (Find Next), Найти сзади (Find previous)**: искать далее указанную ранее строку в соответствующем направлении.
- **Найти предыдущее (Find previous)**: искать далее указанную ранее строку в соответствующем направлении.
- **Найти в файлах (Find in files)**: искать строку в файлах: всплывающее окно с опциями во всех открытых файлах, во всех файлах проекта или поиск в каталогах; можно задать маску-фильтр для типов файлов.
- **Замена (Replace)**: подобна **Найти.** Появляется диалоговое окно со областью ввода искомой строки и текста для замены, а также опций чувствительности к регистру, направления и т. д.
- **Найти инкрементно (Incremental find)**: поиск строки в то время, пока вы вводите ее.
- **Переход к строке (Goto line)**: перемещает курсор в указанную строку файла.
- **Переход назад (Jump back)**: перемещается по файлу назад, к предыдущей закладке (необходимо использовать **Добавить точку перехода в историю**).
- **Переход вперед (Jump forward)**: переместиться к следующей закладке.
- **Добавить точку перехода в историю (Add jump point to history)**: добавляет

закладку или точку перехода в файл.

- **Перейти к следующей ошибке (Jump to next error)**: перейти к позиции в исходном файле к следующей обнаруженной ошибке.
- **Перейти к предыдущей ошибке (Jump to previous error)**: перейти к позиции в исходном файле к предыдущей обнаруженной ошибке.
- **Установить свободную закладку (Set a free bookmark)**: пометить текущую строку, где находится курсор, к следующей (произвольно) номерованной закладке, и добавить ее в список закладок.
- **Найти другой конец блока кода (Find other end of code block)**: если курсор стоит на **begin**, то осуществляется поиск соответствующего **end** и наоборот.
- **Найти начало блока кода (Find code block start)**: перемещается к **begin** процедуры или функции, в теле которой находится курсор.
- **Найти объявления под курсором (Find Declaration at cursor)**: поиск участка кода, где описан выбранный идентификатор. Оно может быть в этом же файле или в любом другом, открытом в редакторе. Если файл еще не открыт, то он будет открыт.
- **Открыть имя файла под курсором (Open filename at cursor)**: открывает файл, имя которого выделено курсором. Полезно для просмотра Include–файлов или файлов, содержащих другие модули Units, используемые в проекте.
- **Перейти к директиве $I{nclude} (Goto include directive)**: если курсор помещен в файле, который включен Included в другой файл, то происходит перемещение в то место другого файла, из которого вызывается включенный.
- **Найти ссылки на идентификатор (Find Identifier Reference)**: выдает все строки в текущем файле или текущем проекте, или во всех вложенных файлах, в которых упоминается идентификатор.
- **Переименовать идентификатор (Rename Identifier)**: позволяет разработчику переименовать идентификатор.
- **Список процедур (Procedure List)**: производит список всех процедур и функций в текущем файле, с номерами строк, где они определены.

Меню *Вид* (View)

Управляет отображением на экране различных окон и панелей.

- **Инспектор Объектов (Object Inspector)**: окно, отображающее возможности текущей формы. Чуть выше располагается окно, содержащее древовидную структуру текущего проекта. Нижняя, главная панель имеет две вкладки: **Properties (**свойства) и **Events (**События).
- **Редактор исходного кода (Source Editor)**: основное окно для редактирования исходных текстов. Функции и вид *Редактора исходного кода* имеют настройки,

вызываемые из основного меню выбором: **Окружение** => **Параметры** => **Опции Редактора**.

- **Обозреватель кода (Code Explorer)**: это окно обычно расположено справа и отображает в древовидной форме структуру кода в текущем блоке или программе.
- **Модули... (Units...)**: открывается диалоговое окно с перечислением файлов модулей текущего проекта. Флажок **Множественное выделение** позволяет открывать одновременно несколько файлов.
- **Формы... (Forms...)**: открывается диалоговое окно со списком форм в текущем проекте, позволяющее выбрать одну или более форм для отображения.
- **Показать зависимости модулей (View Unit Dependencies)**: открывается диалоговое окно, показывающее древовидную структуру зависимостей текущего открытого файла модуля.
- **Переключатель Форма/Модуль (Toggle form / unit view) F12**: позволяет помещать на верхний уровень отображения либо форму, либо *Редактор Исходного кода*, и дает фокус.
- **Сообщения (Messages)**: окно, отображающее сообщения компилятора, показывающие ход успешной компиляции или перечисление найденных ошибок.
- **Результат поиска (Search Results)**: окно, которое отображает результаты поиска в файлах.
- **Окна отладки (Debug windows)**: открывает меню с несколькими опциями управления и конфигурирования работы отладчика.

Меню *Проект*

- **Создать проект ... (New Project)**: создание нового проекта. Диалоговое окно позволяет выбрать тип создаваемого проекта.
- **Создать проект из файла (New Project from file)**: появляется окно для выбора файла, из которого следует создать проект.
- **Открыть проект ... (Open Project)**: открывается созданный и сохраненный проект. Появляется окно выбора со списком файлов Lazarus Project Information (.lpi).
- **Закрыть проект (Close Project)**: закрывает проект, предлагая пользователь сохранить изменения или отказаться от них.
- **Сохранить проект (Save Project)**: поход на подменю **Файл** => **Сохранить**: сохраняются все файлы текущего проекта; если ранее не сохранялись – будет запрошено имя файла(ов), подобно **Сохранить проект Как**.
- **Сохранить проект как... (Save Project as)**: запрашивается имя файла сохраняемого проекта.
- **Опубликовать проект (Publish Project)**: создается полная копия проекта.

- **Инспектор проекта (Project Inspector)**: открывается диалог древовидного отображения файлов в текущем проекте. Можно добавлять, удалять или открывать выбранные файлы, а также изменять опции проекта.
- **Параметры проекта... (Project Options)**: открывается диалог с вкладками для установки параметров проекта.
- **Добавить файл редактора в проект (Add editor file to Project)**: добавляет редактируемый файл в проект.
- **Убрать из проекта ... (Remove from Project)**: выдает меню файлов, доступных для удаления из проекта.
- **Просмотреть исходный код проекта (View Source)**: независимо от того, какой файл редактируется, выдает основной файл программы (.lpr) или основной файл .pas, если отсутствует файл .lpr.

Меню *Запуск*

- **Собрать (Build)**: запускается сборка (т. е. компиляция) любых файлов проекта, которые были изменены со времени последней сборки.
- **Собрать все (Build all)**: запускается сборка всех файлов проекта, независимо от наличия изменений.
- **Быстрая компиляция (Quik Compile)**: быстрая компиляция с поиском ошибок.
- **Прервать сборку (Abort build)**: останавливает процесс сборки на ходу.
- **Запуск (Run)**: это обычный путь запуска компилятора, и если компиляция прошла успешно, запускается приложение.
- **Приостановить (Pause)**: задерживается выполнение работающей программы. Это позволяет проверить промежуточные результаты. Выполнение программы можно продолжить повторным выбором **Запуск.**
- **Показать точку исполнения (Show execution point).**
- **Шаг со входом (Step into)**: применяется совместно с отладчиком, запуская программу на один шаг вплоть до точки, помеченной в исходном тексте.
- **Шаг в обход (Step over)**: вызывает пошаговое выполнение вплоть до помеченного оператора, пропускает его и продолжает работу с нормальной скоростью.
- **Запуск до курсора (Run to cursor)**: запускает выполнение, пока не встретится оператор, где находится курсор, и произойдет остановка. Выполнение продолжится с нормальной скоростью после выбора **Запуск.**
- **Останов (Stop)**: прекращается выполнение программы. Продолжить выполнение выбором **Запуск (Run)** нельзя; можно запустить сначала.
- **Параметры Запуска (Run Parameters)**: открывается многостраничное окно, позволяющее передать программе опции командной строки и параметры; настроить

отображение выполняемой программы; изменить некоторые переменные системного окружения.

- **Сброс отладчика (Reset debugger)**: отладчик приводится в исходное состояние, так что точки останова, значения переменных и т. д. будут «забыты».
- **Собрать файл (Build file)**: происходит компиляция только файла, открытого в **Редакторе**.
- **Запустить файл (Run file)**: компиляция, сборка и выполнение только открытого файла.
- **Параметры сборки + запуска (Configure Build+Run file)**: открывается многостраничное окно с опциями сборки только данного файла, когда выбрано **Собрать Проект (Build Project)**, позволяющее выбрать рабочий каталог, использовать различные макросы и т.д. Затем файл собирается и выполняется.

Меню *Пакет*

- **Новый пакет (New packege)**: отображается окно выбора пакета для установки с возможностями настройки работы пакета.
- **Открыть файл пакета (Open Package)**: открывается один из файлов выбранного пакета.
- **Открыть загруженный пакет (Open Package File)**: открывается недавно загруженный пакет.
- **Добавить активный модуль в пакет (Add Active Unit to Package)**: файл модуля (текущий в редакторе) помещается в пакет.
- **Диаграмма пакетов (Package Graph)**: отображается graph, показывая взаимосвязи используемых пакетов.
- **Настройка установленных пакетов (Configure custom components)**: если созданы некоторые компоненты, здесь их можно настроить.

Меню *Сервис*

- **Настроить внешние средства**: позволяет добавлять различные внешние средства (обычно макросы) в инструментарий пакета разработчика.
- **Быстрая проверка синтаксиса**: выполняет быструю проверку синтаксиса исходного текста без реальной компиляции.
- **Исправить незаконченный блок**: полезная утилита при работе со сложной блочной структурой, когда в каком-либо блоке пропущен «end».
- **Исправить несоответствие IFDEF/ENDIF**: полезно при работе со сложной или вложенной структурой макро, если есть подозрение, что пропущена директива ENDIF.

- **Создать строку ресурсов**: делает выбранную строку ресурсной, помещая ее в секцию ресурсных строк.
- **Разница Diff**: сравниваются два файла для нахождения различий. Имеются опции для игнорирования пробелов в начале или конце строк, а также разницы в концах строк: CR+LF или LF.
- **Проверить файл LFM в редакторе**: позволяет проверить файл LFM, содержащий настройки текущей формы.
- **Преобразовать модуль Delphi в Lazarus**: помогает перенести приложения Delphi в Lazarus; внося необходимые изменения в исходный файл.
- **Преобразовать файл DFM в LFM**: для переноса из Delphi в Lazarus: преобразует файлы описания формы из Delphi в Lazarus.
- **Собрать Lazarus**: запускается обновление Lazarus с недавно загруженными или скорректированными файлами CVS.
- **Параметры сборки Lazarus**: позволяет определить, какие части Lazarus должны быть пересобраны и как.

Меню *Окружение*

- **Опции *Окружения***: открывается многостраничное окно с вкладками.
- **Файлы**: позволяет указать пути: папки по умолчанию, компилятора, папки исходных текстов и временной папки для компиляции.
- **Рабочий стол**: опции языка, автосохранения, свойств рабочего стола, подсказок палитры компонентов и командных кнопок.
- **Окно**: для настройки размера и поведения различных окон.
- **Редактор Форм**: выбор цветов для редактирования форм.
- **Инспектор Объектов**: выбор цветов и высоты элементов.
- **Резервирование**: настройка резервирования редактируемых файлов.
- **Именование**: настройка расширения имени в именовании файлов Паскаля ('.pp' или '.pas'), сохранять ли файлы с именами в нижнем регистре, выполнять ли автоудаление или автопереименование.
- **Опции *Редактора***: многостраничное окно с вкладками:
    - **Общие** – настройка поведения: авто-отступ, подсветка скобок, редактирование перетаскиванием, прокрутка, подсветка синтаксиса, показ подсказок, размер отступа блока и вкладок, лимит откатов;
    - **Отображение** – опции показа номеров строк, границ, размера и типа шрифта редактора; имеется предварительный просмотр, показывающий расцветку синтаксиса (комментариев, директив, пунктуации, ошибок и точек остановок);

- **Привязки клавиш** – опции выбора схемы Lazarus или Turbo Pascal;
- **Цвет** – позволяет выбор цветовой схемы для различных языков: Object Pascal, C++, Perl, HTML, XML и скриптов оболочки. Имеется панель предварительного просмотра для выбранного языка;
- **Code Tools** – позволяет настроить возможности наподобие завершения идентификаторов, специфические шаблоны для завершения кода.

- **Опции *Отладчика***: многостраничное окно с вкладками:
  - **Общие** – выбор отладчика: none, GNU debugger (gdb) или gdb через SSH, указание пути к отладчикам, и специфические опции для выбранного отладчика;
  - **Протокол событий** – указание, когда очищать журнал при работе, и какие сообщения отображать;
  - **Исключения языка** – выбор исключений для игнорирования;
  - **Исключения ОС** – возможность добавлять некоторые сигналы, применимые в текущей операционной системе (не осуществлено).
- **Опции *Code Tool***: многостраничное окно с вкладками:
  - **Общие** – указание дополнительных путей к исходным кодам, методики перехода;
  - **Создание Кода** – определяется политика вставки созданных элементов программ;
  - **Слова** – определяется, как пишутся ключевые слова языка Паскаль – в верхнем или нижнем регистре, или с первой заглавной буквой;
  - **Разрыв строк** – установка правил разрыва;
  - **Пробел** – установка правил автоматического добавления.
- **Редактор определений *Code Tools***: здесь можно видеть все внутренние определения для грамматического разбора исходников. Видны все определения, модули, исходники, включая пути к папкам с исходниками.
- **Пересмотреть папку исходного кода FPC**: повторный просмотр папки. Lazarus использует исходники fpc для генерации правильной обработки событий и проверки объявлений.

Меню *Окно*

Содержит список открытых файлов и доступных окон вроде **Редактор Исходного Кода**, **Инспектор Объектов** и **Инспектор Проекта**. Щелчком на имени одного из окон оно выводится «наверх» и получает фокус.

Меню *Справка*

Имеется три выбора:

- **Оперативная Справка** – открывается окно браузера с картинкой бегущего гепарда и несколькими связями на веб-сайты Lazarus, FreePascal и WiKi.
- **Параметры справки** – открывается меню с опциями выбора инструмента просмотра и баз данных для чтения информации Справки.
- **О Проекте Lazarus** – отображается многостраничное окно с информацией о установленной версии и участниках проекта Lazarus.

Кнопочная панель

Маленькая панель в левой верхней части основного окна, слева от палитры компонентов, имеет набор кнопок, повторяющих наиболее часто применяемые выборы основного меню: **создать модуль**, **Открыть** (со стрелкой вниз для отображения списка недавно использованных файлов), **Сохранить**, **Сохранить все**, **Создать форму**, **Переключить Форма/Модуль** (т. е. показать либо форму, либо модуль исходного кода), **Показать модули**, **Показать формы**, **Запуск** (т. е. компиляция и выполнение), **Пауза**, **Шаг со входом**, **Шаг в обход** (последние два – функции отладчика).

2.3.2. Палитра компонентов

Это панель инструментов с вкладками, каждая из которых представляет собой набор иконок, составляющих функциональную группу компонентов. Самая левая иконка на каждой вкладке в виде стрелочки называется **Средством выбора.**

Если навести курсор мыши на иконку палитры компонентов без нажатия, появится название данной компоненты. Каждое название начинается с *T*, что означает *Tun*, а точнее *Класс* компоненты. При выборе компоненты для размещения на форме, Class добавится в секцию **type** раздела **interface** модуля (обычно в виде части на TForm1), и **instance** (образец) этого класса добавится в секцию **var** (обычно как переменная Form1). Все **Methods** (методы), разработанные для формы или ее компонент (процедуры или функции), будут помещены в раздел **implementation** модуля.

Вкладки (их имена достаточно понятны и не требуют разъяснений):

1. Standart
2. Additional
3. Common Controls
4. Dialogs
5. Misc

6. Data Controls
7. Data Access
8. System
9. SynEdit

Компоненты палитры *StdCtrls*, *ComCtrls* и *ExtCtrls* содержат определения и описания многих из наиболее часто используемых элементов управления для построения форм и других объектов в приложениях Lazarus. Рассмотрим использование палитры компонентов на примере использования подпалитры *Common*.

Многие из используемых компонент на палитре имеют соответствующий базовый (родительский) класс, например такие, как *TCustomButton*, *TCustomMemo* или *TCustomScrollBar*. Некоторые свойства и методы, имеющие отношение к размещенному на форме элементу определяются, соответственно, более полно в классе *TCustomXXX* и наследуются от базового класса.

Если вы разместили на форме компонент редактора, то не требуется явно добавлять код для его создания. Компонент автоматически создается IDE вместе с формой, и уничтожается, когда форма разрушается.

Существует несколько способов определения и изменения свойств компонента. Если компонент разместить на дизайнере формы и посмотреть на свойства в инспекторе объектов, то можно наблюдать за изменением некоторых свойства при перемещении компонента по форме. Кроме того, с помощью инспектора объектов, можно самостоятельно и независимо выбрать значение, связанное со свойством объекта, и ввести новое значение. Также можно явно изменить свойства объекта, если ввести новое значение свойства в редакторе исходного кода. Новое значение также отобразится в инспекторе объектов.

Таким образом, существует возможность использовать три разных способа определения свойства каждого объекта:

- с помощью перемещения при помощи мыши на форме;
- путем установки значения параметров в инспекторе объектов;
- посредством написания кода в редакторе.

Рассмотрим наиболее распространенные свойства компонентов, определенных в этих объектах. Нестандартные или контрольно-специфические свойства могут быть найдены в справке среды разработки для индивидуального управления ими. Дополнительную помощь можно получить, выбрав свойство или ключевое слово, либо в **Инспекторе Объектов**, либо в **Редакторе Исходного Кода**, и нажав клавишу F1. В этом случае будет осуществлен переход на соответствующую страницу помощи в документации проекта.

Перечислим некоторые наиболее часто используемые свойства:

- **Действие** (*Action*). Основное действие или событие, связанное с объектом. Если выбрано действие *Exit*, то кнопка может вызвать действие *Close*.
- **Выравнивание** (*Align*). Определяет способ, в соответствии с которым объект должен быть выровнен по отношению к родительскому объекту. Возможные значения *alTop* (размещены в верхней части и используют всю доступную ширину), *alBottom, alLeft* (расположены слева и используют всю доступную высоту), *alRight, alNone* (разместить в любом месте на родительском элементе управления) или *alClient* (занимает все свободное пространство рядом и выравнивается по верхней, нижней, левой или правой стороне базового объекта)
- **Якорь** (*Anchor*). Используется для сохранения положения элемента управления на определенном расстоянии от края родительского элемента управления.
- **Автовыбор** (*AutoSelect*). Используется для выделения всего текста, когда объект получает фокус или нажата клавиша *Enter*.
- **BorderSpacing**. Пространство между закрепленным элементом управления и его родителем.
- **Caption**. Текст, который отображается на элементе или вблизи элемента контроля. По умолчанию *Caption* устанавливается такой же, как и *Имя*, а программист изменяет название на осмысленный текст.
- **CharCase**. Указывает, как текст отображается в элементе управления редактирования текста.
- **Constraints**. Устанавливает минимальный и максимальный размеры для элемента управления.
- **Color**. Цвет, который будет использоваться для отрисовки элемента управления или цвета текста, который в нем содержится.
- **Enabled**. Логическое свойство для определения, могут ли элементы быть выбраны, и разрешены ли выполнения действий с ними. Если элемент контроля не включен, он помечен серым цветом на форме.
- **Font**. Шрифт, используемый для написания текста, связанного с элементом управления.
- **Hint**. Короткий кусок информативного всплывающего текста, который появляется, если курсор мыши наведен на элемент управления.
- **Items**. Перечень тех сущностей, которые объект содержит, например группу изображений, серии строк текста, ряд действий в *ActionList*, и т. д.

- **Lines**. Массив строк, содержащих текстовые данные в контрольной группе с более чем одной строкой данных.
- **Name**. Идентификатор, под которым элемент управления определен в программе.
- **PopUpMenu**. Окно, содержащее контекстно - зависимое меню, которое появляется при нажатии правой кнопки мыши на объект, элемент управления.
- **Position** (or *Top, Left*) Определяет, где объект или элемент управления находится в базовых форме или окне.
- **ReadOnly**. Логическое свойство, которое, если имеет значение *True*, означает, что содержимое элемента управления может быть прочитано пользователем или вызывающей программой, но не может быть переписано или изменено.
- **ShowHint**. Позволяет отображать небольшое окно с контекстной справкой или другим описанием, которое будет отображаться при наведении курсора мыши. Размещено над элементом управления.
- **Size** (or Height and Width). Размер элемента управления.
- **Style**. Набор вариантов, доступных для стиля, который зависит от рода элемента управления.
- **TabOrder**. Целое число, определяющее, где в последовательности вкладок на форме этот элемент управления расположен.
- **TabStop**. Логическое свойство, которое при значении *True* размещает элемент в последовательности объектов, до которых пользователь может добраться при последовательном нажатии клавиши табуляции.
- **Text**. Строка текста, которая представляет фактические данные, которые этот объект содержит. В частности, относится к таким типам объектов, как *Text, Memo* и *StringList*.
- **Visible**. Если содержит значение *True*, то объект можно увидеть в форме; если – *False*, то объект скрыт.
- **WordWrap**. Логический флаг, определяющий, включен ли перенос слов.

События (Event Actions)

Многие действия перечислены на вкладке «События» *Инспектора объектов* (Object Inspector). При выборе записи в списке *ComboBox,* появляется выпадающий список, показывающий любые действия, которые уже определены в IDE, и позволяет вам выбрать одно из них, чтобы оно было связано с этим событием. Также возможно выбрать многоточие, и тогда будет осуществлен переход в область редактора исходного кода, где можно начать определять собственные инструкции действий для выбранного события.

Для большинства элементов управления достаточно обеспечить написание кода для *OnClick*; хотя для более сложных объектов может быть необходимо также обеспечить действие *OnEntry* (когда курсор мыши входит в управления и передает ему фокус) и *OnExit (*когда курсор мыши покидает элемент управления) или написать обработчик событий для *OnChange* или *OnScrol*.

Всплывающее меню, которое появляется при нажатии правой кнопкой мыши на объекте в дизайнере форм, содержит в качестве своего первого пункта *Создать событие по умолчанию,* и выбор этой опции, будет иметь тот же эффект, что и выбор с многоточием в инспекторе объектов по умолчанию для данного события. Как правило, на событие *OnClick* средствами IDE будет создано поле со стандартным программным кодом в редакторе, где можно ввести код для обработчика выбранного события.

По отношению к событийной модели общей стратегией в объектно - ориентированном программировании является предоставление *ActionList* с объектом для ввода, удаления или редактирования некоторого количества предварительно определенных действий, из которых наиболее подходящие могут быть выбраны для использования в каждом конкретном случае и для каждого элемента управления.

Перечислим некоторые наиболее часто встречающиеся события:

- **OnChange.** Действия, которые будут приняты, если любое изменение будет обнаружено.
- **OnClick**. Предлагаемое действие, когда нажата кнопка мыши.
- **Click**. Метод для эмуляции в коде действия однократного нажатия на элементе управления.
- **OnDragDrop**. Предлагаемое решение во время события OnDragDrop.
- **OnEditingDone**. Предлагаемое решение, когда пользователь закончил все изменения или модификации объекта.
- **OnEntry**. Действие в результате события, когда курсор мыши входит в область, занимаемую объектом, и переносит фокус на этот объект.
- **OnExit**. Предлагаемое действие, когда мышь перемещается из области объекта с потерей фокуса на объекте.
- **OnKeyPress**. Действие, по событию соответствующее любому нажатию кнопки.
- **OnKeyDown**. Действия, активирующиеся, когда клавиша нажата, а фокус находится в элементе управления.
- **OnKeyUp**. Действия, выполняющиеся, когда клавиша освобождена, в то время как фокус находится в этом элементе управления.
- **OnMouseMove.** Действия, реализующиеся, когда курсор мыши

перемещается над элементом управления, находящемся в фокусе.

- **OnMouseDown**. Действия, выполняющиеся, когда кнопка мыши нажата и в то же время находится в элементе управления, находящемся в фокусе.
- **OnMouseUp**. Действия, выполняющиеся, когда кнопка мыши не нажата, а курсор находится над этим элементом управления. Означает, что кнопка мыши была ранее нажата и была освобождена. Случай, когда курсор входит в область элемента управления, но кнопка мыши еще не была нажата, определен событиями OnEntry или OnMouseEnter.
- **OnResize**. Действие, совершаемое при изменении размеров элемента управления.

Конструкторы и деструкторы

Эти два специальные метода, связанные с каждым элементом управления:

- Конструкторы: такие как, например, создание, выделения памяти и системных ресурсов, которые необходимы объекту. Они также вызывают конструктор любого подобъектов, присутствующего в классе.
- Деструкторы: удаляют объект и освобождают память и другие ресурсы, ранее выделенные объекту. Если вы вызываете метод Destroy для уничтожения объекта, который ранее не был инициализирован, то он сгенерирует ошибку. Всегда используйте метод Free для освобождения объектов, потому, что он проверяет, является ли значение объекта равно нулю перед вызовом метода Destroy.

Файлы Lazarus

Когда выполняется сохранение, то на самом деле сохраняется два файла:
xxx.pas и yyy.lpr.

Файл проекта (lpr) и файл модуля (pas) должны иметь разные имена, потому что Lazarus присваивает имя модулю (в исходном коде) такое, как и имя файла модуля, а программе – по имени файла проекта. Это необходимо сделать, в противном случае компилятор может впоследствии не найти модуль по ссылке на него в файле проекта. Во избежание противоречий, ошибок следует изменить все упоминания Unit1 на xxx.

Ниже приведена краткая справка по каждому файлу:

again.exe: основной исполняемый файл программы. Win32 добавляет расширение «exe». Linux этого не делает. В Linux это файл будет иметь большой размер вследствие того, что включает отладочную информацию. Запустите утилиту «strip» чтобы удалить ее и значительно снизить размер исполняемого файла.

again.lpi: это основной файл проекта Lazarus (Lazarus Project Information); эквивалент основного файла приложения в Delphi с расширением .dpr. Он сохраняется в XML–формате.

again.lpr: исходный код основной программы. Несмотря на специфичное для Lazarus расширение, на самом деле это обычный Pascal-код. Он содержит строку Uses, помогающую компилятору найти все необходимые модули. Отметим, что программа называется не аналогично имени данного файла.

againu.lfm: это файл, в котором Lazarus хранит описание формы. Lazarus использует его для создания файла ресурсов, который включает секцию инициализации модуля againu.pas. Файл Delphi с расширением .dfm может быть преобразован в lfm-формат в IDE Lazarus из главного меню: Tools=>Convert DFM file. Описание каждого объекта начинается со следующих строк:

object xxxx

then there follows a list of properties

(including embedded or nested objects) then an

end

again.lrs: это автоматически генерируемый файл ресурсов. Заметьте, что это не файл ресурсов Windows.

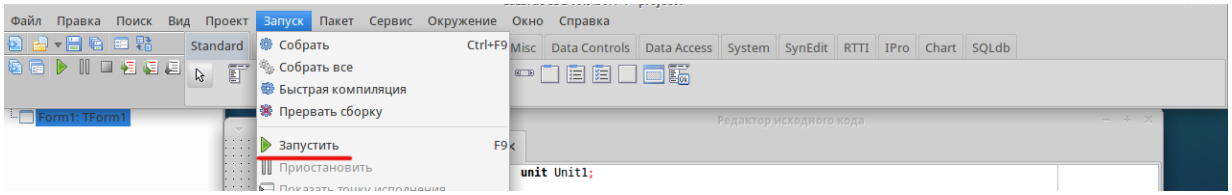
againu.pas: модуль, содержащий код формы. Обычно это единственный файл, который прикладному программисту необходимо редактировать или проверять.

again.ppu: это скомпилированный модуль, который получает связанные в исполняемый файл определения, указанные в разделе Uses.

ppas.bat: это простой скрипт, связывающий программу для создания выполняемого файла. Если компиляция успешна, он удаляется компилятором.

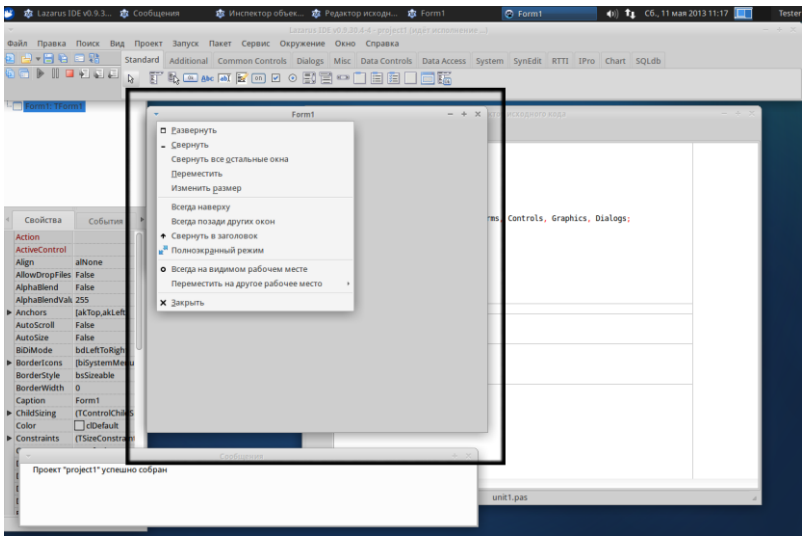
2.4. Разработка приложений в среде Lazarus

Первое приложение на Lazarus нами уже написано! Для того чтобы в этом убедиться, достаточно в пункте меню «Запуск» выбрать подпункт «Запуск» (подчеркнут красным на рисунке):



или просто нажать на клавишу F9.

На следующем рисунке видно окно исполняемой программы — серый квадрат, а также результат успешной сборки проекта с именем по умолчанию project1.



Прежде чем перейти к следующим шагам, закроем окно текущей программы, либо нажав на крестик в верхнем правом углу, либо выбрав в меню пункт *Закрыть*. Когда закрытие происходит первый раз, будет задан вопрос о подтверждении закрытия с правом выбора дальнейших действий.

Перейдем к разработке следующего приложения, которое будет иметь на базовой форме другие объекты управления. На экране присутствуют стандартные окна IDE Lazarus: главное окно вверху, *Инспектор объектов* (Object Inspector) слева, занимающий большую часть экрана *Редактор Кода* Lazarus (Lazarus Source Editor), и готовое к использованию окно Form1 поверх окна *Редактора кода*.

Во-первых, изменим заголовок окна программы. Для этого перейдем к окну *Инспектор объектов => Свойства => Caption* и зададим в правом редактируемом поле название программы (в данном случае формы). Пусть это будет *Первая программа*, после чего нажмем клавишу Enter на клавиатуре. Заголовок программы изменился.

В главном окне сверху, под строкой меню, располагается строка вкладок. Если вкладка *Standard* еще не выбрана, выберите ее, щелкнув по ней левой кнопкой мыши. Затем найдите иконку *Button* (прямоугольник с текстом *Ok* на нем) и щелкните по ней мышкой. Затем щелкните в окне *Form1*, где-нибудь слева от середины. Появится затененный прямоугольник с надписью *Button1*. Вновь щелкните на иконке *Button* на вкладке *Standard* и щелкните на *Form1* где-нибудь справа от центра: появится прямоугольник с надписью *Button2*.

Теперь щелкните на *Button1* чтобы выбрать ее. Инспектор Объектов отобразит свойства объекта *Button1*. Недалеко от верхнего края располагается свойство с именем *Caption*, в котором отображается значение *Button1*. Щелкните в этой строке и измените *Button1* на *Нажми меня*. Если вы нажмете клавишу *Enter* или щелкнете в другой строке, то увидите, что надпись на первой кнопке *Form1* изменилась на *Нажми меня*. Если размера кнопки не хватает для отображения всего текста, тогда в свойствах данной кнопки

необходимо найти *Width* и изменить ее. В демонстрационной программе стандартное значение изменено от 75 до 100, что потребовалось для данного набора шрифтов, установленных в системе в качестве основных, а также их параметров.

Теперь щелкните в *Инспекторе объектов* на вкладке *Events* (События) и вы увидите различные события, на которые может реагировать кнопка. Среди них *OnClick, OnEnter, OnExit* и т. д. Щелкните в строке справа от *OnClick*: появится маленькая кнопка с троеточием (...). Если вы ее нажмете, то автоматически перенесетесь в *Редактор кода*, и курсор окажется в начале участка кода, содержащего следующий фрагмент:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  {здесь наберите:}  Button1.Caption := 'Нажми еще';
  {Редактор уже вставил завершение процедуры end}
end;
```

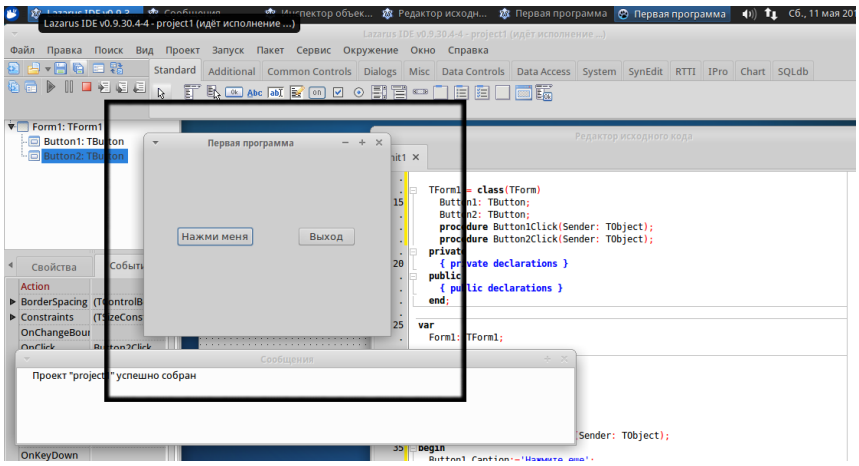
Обратим внимание, что текста в фигурных скобках не будет, это комментарии позволяющие понять действия или логику работы этой процедуры.

Нажмите F12 для переключения от *Редактора кода* к окну формы *Form1*. Теперь отредактируем свойства кнопки *Button2*: щелкните на *Button2* для отображения ее свойств в *Инспекторе объектов*. Измените свойство *Caption* на *Выход* вместо *Button2*. Теперь перейдите на вкладку событий (Events) и щелкните в строке *OnClick*. Щелкните на кнопке с тремя точками и перенеситесь в *Редактор кода*, в тело другой процедуры:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  {здесь наберите:}   Close;
  {Редактор уже вставил завершение процедуры}
end;
```

Теперь нажмите F12 чтобы увидеть форму *Form1* вновь. Теперь вы можете попытаться скомпилировать. Простейшим способом сделать это является выбор в главном меню пункта *Run* а в появившемся подменю пункта *Run*. Вы также можете просто нажать клавишу F9. Сначала произойдет компиляция, а затем (если все в порядке) линковка и запуск вашей программы.

Отобразится несколько текстовых окон, и будут выведены различные сообщения компилятора, а потом вновь появится окно формы *Form1* с заголовком *Первая программа*, но уже без точечной сетки; это и есть главное окно приложения, и оно ожидает нажатия кнопок или любого другого действия:

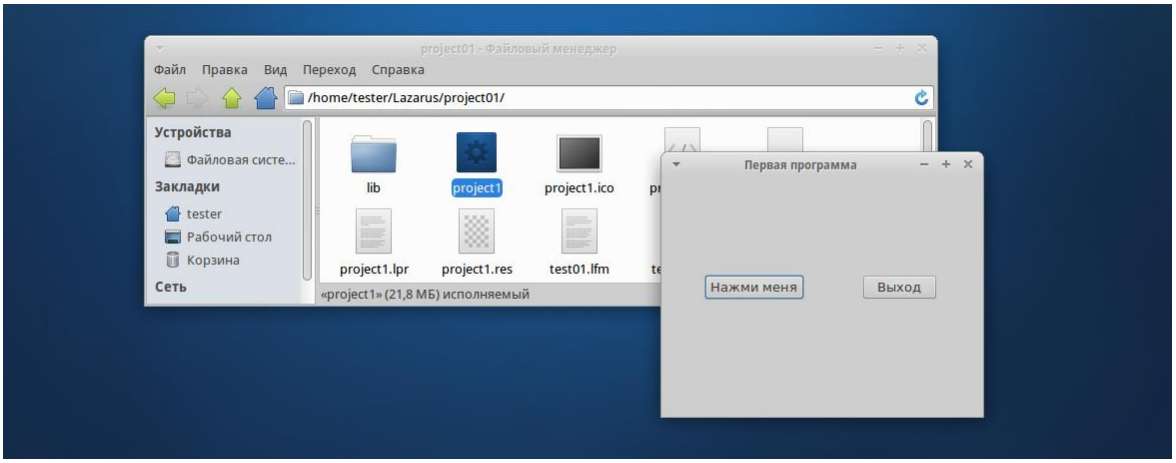


Попробуйте щелкнуть по кнопке *Нажми меня*. Надпись на ней сменится на *Нажми еще*. Если вы нажмете еще раз, то на кнопке так и останется надпись *Нажми еще*.

Теперь щелкните по кнопке с надписью *Выход*. Окно закроется и программа завершится. Вновь появится окно формы *Form1* с точечной сеткой, готовое для дальнейшего редактирования.

Теперь можно сохранить свою работу, выбрав последовательно пункты меню Project => Save Project As => имя_вашего_файла.pas/. Далее в меню *Пуск* выберем пункт *Собрать все* и посмотрим в файловом менеджере на файлы, генерируемые IDE Lazarus в операционных системах Linux.

Набор файлов практически идентичен набору файлов в Windows. Следует обратить внимание на то, что в Linux нет расширения для исполняемого файла, он называется просто *project1*. Можно щелкнуть по нему левой кнопкой мыши и снова увидеть окно программы с заголовком *Первая программа*:



Если закрыть Lazarus, то все созданные нами файлы сохранятся, а программу можно будет стартовать независимо от активности среды разработки.

Еще раз модифицируем первую программу. Вновь стартуем IDE Lazarus и откроем сохраненный проект. Для этого после старта среды разработки выберем пункт меню: *Проект => Открыть недавний проект*.

На форме Form1 щелкните на кнопке *Нажми меня* (Button1), чтобы выбрать ее. В *Инспекторе объектов* перейдите на вкладку событий (Events), щелкните на строке справа от события *OnClick*, щелкните на кнопке с многоточием, чтобы перейти к соответствующему участку кода в *Редакторе исходного кода.*

Изменим код на приведенный ниже:

```
procedure TForm1.Button1Click(Sender: TObject);
{Используем свойство Tag, устанавливая его в положения 0 или 1}
 begin
  if Button1.tag =0 then
  begin
    Button1.caption := 'Нажми еще раз';
    Button1.tag := 1
  end else
  begin
    Button1.caption := 'Нажми меня';
    Button1.tag := 0;
  end
 end;
```

Сохраните проект, перекомпилируйте и запустите. Левая кнопка на форме первой программы теперь циклически меняет свой текст с одного сообщения на другое.

2.5. Простейший калькулятор (преобразование типов)

Снова обратим внимание на *Редактор исходного кода*, который предназначен для создания и редактирования текста программы. Этот текст составляется по специальным правилам и описывает алгоритм работы программы.

Первоначально окно кода содержит минимальный исходный текст, обеспечивающий нормальное функционирование пустой формы в качестве полноценного окна, реализованного IDE в текущей операционной системе. В ходе работы над проектом программист вносит в него необходимые дополнения, чтобы придать программе нужную функциональность.

До этого момента времени приложения создавались автоматически вместе со стартом Lazarus. Теперь же по умолчанию IDE открывает последний существующий проект, который далее не потребуется. Закроем существующий проект и создадим новый. В списке предлагаемых для реализации создаваемых новых проектов выберем

Приложение.

Результатом этого выбора будет обновление всех окон IDE, приведенных к исходному состоянию, позволяющему начать разработку нового проекта. В окне редактора будет отображен некоторый начальный автоматически сгенерированный код:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;
type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.lfm}
end.
```

В дальнейшем мы будем вставлять в окно текст программы между двумя последними строками этого кода, не обращая внимания на код выше, но редактировать его строго нежелательно.

Процесс создания Lazarus-программы разбивается на две фазы: фазу конструирования формы и фазу кодирования. Итак — этапы конструирования:

- Конструирование формы осуществляется с помощью выбора компонентов из палитры и размещения их на форме.
- Программист может перемещать любой размещенный на форме компонент и изменять его размеры с помощью мыши.
- Чтобы придать компоненту нужные свойства, используется страница **Properties** *Инспектора объектов*.
- Чтобы компонент мог откликаться на то или иное событие, программист должен создать обработчик события и указать его имя на странице **Events**

Инспектора объектов.

- Обработчик события оформляется в виде процедуры, имеющей составное имя. Первая часть имени представляет собой имя класса для формы, вторая часть отделяется от первой точкой и может быть произвольной. Если Lasarus автоматически формирует заготовку для обработчика, то вторая часть имени представляет собой объединение имени компонента и имени события без предлога **On**.
- Тело процедуры ограничено словами begin ... end и состоит из отдельных предложений (операторов) языка Object Pascal. В конце каждого предложения ставится точка с запятой.
- Свойства компонента могут изменяться на этапе выполнения программы.

Рассмотрим далее ***основные этапы разработки программы.*** Выражение «написать программу» отражает только один из этапов создания компьютерной программы, когда разработчик программы (программист) действительно пишет команды (инструкции) на бумаге или при помощи текстового редактора.

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

- Спецификация (определение, формулирование требований к программе).
- Разработка алгоритма.
- Кодирование (запись алгоритма на языке программирования).
- Отладка.
- Тестирование.
- Создание справочной системы.
- Создание установочного дистрибутива программы.

Спецификация. Спецификация, определение требований к программе — один из важнейших этапов, на котором подробно описывается исходная информация, формулируются требования к результату, поведение программы в особых случаях (например, при вводе неверных данных), разрабатываются диалоговые окна, обеспечивающие взаимодействие пользователя и программы.

Разработка алгоритма. На этапе разработки алгоритма необходимо определить последовательность действий, которые надо выполнить для получения результата. Если задача может быть решена несколькими способами и, следовательно, возможны различные варианты алгоритма решения, то программист, используя некоторый критерий, например скорость решения алгоритма, выбирает наиболее подходящее решение. Результатом этапа разработки алгоритма является подробное словесное описание алгоритма или его блок-схема.

Кодирование. После того как определены требования к программе и составлен алгоритм решения, алгоритм записывается на выбранном языке программирования. В результате получается исходная программа.

Отладка. Отладка — это процесс поиска и устранения ошибок. Ошибки в программе разделяют на две группы: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки — наиболее легко устраняемые. Алгоритмические ошибки обнаружить труднее. Начальный этап отладки можно считать законченным, если программа правильно работает как минимум на одном-двух наборах входных данных.

Кодирование. После того как определены требования к программе и составлен алгоритм решения, алгоритм записывается на выбранном языке программирования. В результате получается исходная программа.

Отладка. Отладка — это процесс поиска и устранения ошибок. Ошибки в программе разделяют на две группы: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки — наиболее легко устраняемые. Алгоритмические ошибки обнаружить труднее. Этап отладки можно считать законченным, если программа правильно работает на одном-двух наборах входных данных.

Создание дистрибутива программы. Установочный дистрибутив (полноценный установочный CD/DVD) создаются для того, чтобы пользователь мог самостоятельно, без помощи разработчика, установить программу на свой компьютер. Обычно помимо самой программы на установочном диске находятся файлы справочной информации и инструкция по установке программы (Readme-файл).

Следует понимать, что современные программы, в том числе разработанные в Lazarus, в большинстве случаев (за исключением самых простых программ) *не могут быть установлены на компьютер пользователя путем простого копирования,* так как для своей работы требуют специальных библиотек и компонентов, которых может и не быть у конкретного пользователя. Поэтому установку программы на компьютер пользователя должна выполнять специальная программа, которая помещается на установочный диск. Как правило, установочная программа создает отдельную папку для устанавливаемой программы, копирует в нее необходимые файлы и, если надо, выполняет настройку операционной системы путем внесения дополнений и изменений в реестр.

На первом этапе создания программы программист должен определить последовательность действий, которые необходимо выполнить, чтобы решить поставленную задачу, т. е. разработать алгоритм.

Алгоритм — это точное предписание, определяющее процесс перехода от исходных данных к результату. Алгоритм может быть представлен в виде словесного описания или графически — в виде блок-схемы с использованием специальных символов. При программировании в Lazarus алгоритм решения задачи представляет собой совокупность алгоритмов процедур обработки событий.

Так как алгоритм программы, реализующей простейший калькулятор, достаточно тривиален, не будем представлять его в виде блок-схемы. Воспользуемся тремя отдельными полями: два – для ввода данных и одно – для вывода результата. Переименуем форму, дав ей имя «Калькулятор», и разместим на форме три стандартных поля *Edit*, четыре кнопки для основных арифметических действий *Button1–4*, а также кнопку *Button5* для выхода из программы. Размеры и размещение объектов на форме можно выбрать произвольно. В свойствах же формы желательно запретить изменение ее размеров.

Для «наведения порядка» выделим все кнопки, обведя их мышью с нажатой левой кнопкой, и для выделенной области (объектов) вызовем контекстное меню. Нажав в этой области правую кнопку мыши, перейдем к пункту *Выравнивание* этого меню: горизонтально распределим кнопки равномерно, а вертикально — выровняем их по верхнему краю. Также можно выровнять и поля *Edit*, которые мы будем использовать в качестве полей ввода аргументов. Свяжем с кнопкой *Выход* код, который будет завершать программу, повторив действия из первого примера.

Для определения глобальных и текстовых переменных запишем программный код в области между строками **implementation** и **{$R *.lfm}**:

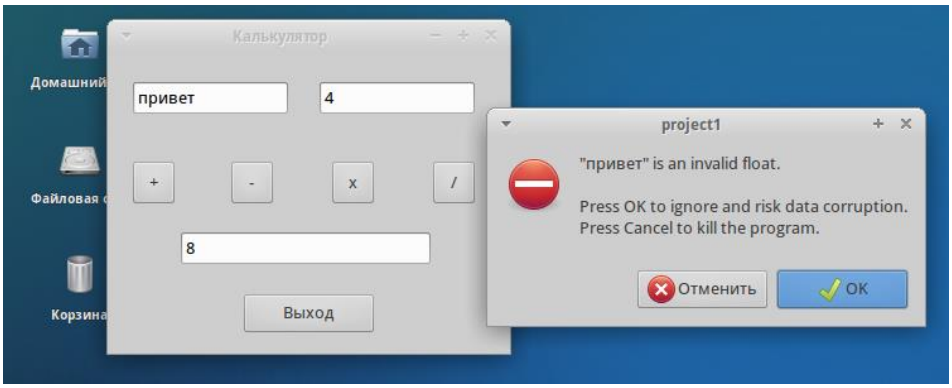
```
implementation
var    arg1, arg2, result: real;
       str1, str2, str_result: string;
{$R *.lfm},
где объявления переменных задаются
var <имя переменной>: <тип переменной элементов>;
```

Итак, выше и в программном коде определены шесть глобальных переменных: три переменных вещественного типа и три строковых переменных.

Дополним программу минимальным кодом для реализации функционирования калькулятора. Для каждой кнопки, связанной символом в заголовке с конкретной операцией, введем для события (либо дважды щелкнув по кнопке, либо выбрав в меню многоточие) ButtonClick, следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  str1:=Form1.Edit1.Text;
  arg1:=StrToFloat(str1);
  str2:=Form1.Edit2.Text;
  arg2:=StrToFloat(str2);
  result:= arg1+arg2;
  str_result:=FloatToStr(result);
  Form1.Edit3.Text:=str_result;
end;
```

После определения действий для всех кнопок, выполним компиляцию. Если ошибок не обнаружено, можно выполнить получившуюся программу. Программа весьма неэффективна, содержит много потенциальных пробелов и служит лишь учебным целям, причем на первом этапе знакомства с *Паскалем* и средой *Lazarus*. Попробуем ввести в первое поле что-то отличное от числа, например слово *Привет* и выполнить любую из операций на калькуляторе. Пользователя проинформируют о том, что введенное значение *Привет* – неверное число с плавающей точкой:



Собственно с этого момента и начинается труд программиста. Теперь необходимо протестировать программу в разных режимах и модифицировать код так, чтобы ошибок во время исполнения не возникало.

СПИСОК ЛИТЕРАТУРЫ

1. Алексеев Е. Р. Free Pascal и Lazarus: учебник по программированию / Е. Р. Алексеев, О. В. Чеснокова, Е. В. Кучер. – М.: ДМК-пресс, 2010. – 438 с.
2. Алексеев Е. Р., Самоучитель по программированию на Free Pascal и Lazarus / Е. Р. Алексеев, О. В. Чеснокова, Е. В. Кучер – Донецк: ДонНТУ УНИТЕХ, 2011. – 503 с.
3. Гофман В. Э. Delphi. Быстрый старт / В. Э. Гофман, А. Д. Хомоненко — СПб.: БХВ-Петербург, 2003. — 288 с.
4. Мансуров К. Т. Основы программирования в среде Lazarus / К. Т. Мансуров – Издательство: Интернет-издание – 2010. – 772 с.
5. Немнюгин С. А. Turbo Pascal: Программирование на языке высокого уровня: учебник для вузов / С. А. Немнюгин – СПб.: Питер, 2007. – 496 с.
6. Павловская Т. А. Программирование на языке высокого уровня / Т. А. Павловская – СПб.: Питер Пресс, 2009. – 432 с.
7. Суркова Е. В. Лабораторный практикум по программированию на языке Pascal: методические указания / Е. В. Суркова – Ульяновск: УлГТУ, 2007. – 59 с.

Учебное издание

Составители:

Ефлов *Владимир Борисович*

Никонова *Юлия Васильевна*

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PASCAL
В СРЕДЕ LAZARUS
Учебное пособие для студентов и преподавателей вузов

Редактор А. В. Ермашова

Компьютерная верстка
и оформление обложки Ю. В. Никоновой

Подписано в печать

Формат 60 х 84 1/8 Бумага офсетная.

Уч.-изд. л. 2,5 Тираж 50 экз Изд. № 184

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Отпечатано в типографии Издательства ПетрГУ
185910, Петрозаводск, пр. Ленина, 33