**Подпрограммы**

Вначале языки программирования были проще, они выполнялись строго сверху-вниз, один оператор за другим. Такие языки еще называли **линейными**. Типичный пример линейных языков - Бейсик. Единственную возможность организовать хоть какую-то логику в таких языках предоставлял оператор безусловного перехода GOTO, который в зависимости от условия, "перепрыгивал" на заранее расставленные метки. В современных языках программирования GOTO тоже остался, наверное, для любителей антиквариата. Но его применение может привести к трудно обнаруживаемым логическим ошибкам времени выполнения (run-time errors). Использование GOTO в современном программировании считается дурным тоном. Мы не будем изучать эту возможность, поскольку для организации логики есть куда более "продвинутые" средства! Одним из таких средств являются **подпрограммы**.

*Подпрограмма - это часть кода, которую можно вызвать из любого места программы неопределенное количество раз.*

Другими словами, подпрограммы подобны строительным кирпичикам, из которых, в конце концов, получается здание - программа. Без подпрограмм можно обойтись, если вы пишете небольшую учебную программу на пару десятков строк кода. А если это серьезное приложение, с парой сотен модулей, в каждом из которых могут быть тысячи строк кода? Как такую программу написать, не разбивая задачу на отдельные части? Подпрограммы помогают улучшить код, структурировать его. Поэтому языки высокого уровня, которые позволяют использовать подпрограммы, называют ещё **процедурно-ориентированными** языками. И наш компилятор FPC тоже относится к таким языкам.

Подпрограммы бывают двух типов: **процедуры** и **функции** - первые просто выполняют свою работу, вторые еще и возвращают результат этой работы.

**Процедуры**

На самом деле, мы уже неоднократно использовали процедуры. Например, когда генерировали событие нажатия на кнопку. Это событие - процедура. Процедура начинается с ключевого слова procedure и имеет следующий синтаксис:

procedure <имя процедуры>(<список параметров>);

const

<объявление констант>;

type

<объявление новых типов>;

var

<объявление переменных>;

<описание вложенных процедур и функций>;

begin

<тело процедуры>;

end;

Большая часть указанного синтаксиса является необязательной - процедура может не иметь параметров, констант, пользовательских типов данных, переменных и т.п. - она может быть очень простой, например:

procedure ErrorMessage;

begin

ShowMessage('Ошибка!' +#13 + 'На ноль делить нельзя!');

end;

Такую процедуру можно вызвать из любого места программы, но процедура обязательно должна быть описана выше - ведь иначе компилятор не будет знать о ней. Есть еще возможность предварительно объявить процедуру, но об этом чуть позже. Итак, если эта процедура описана выше, то мы можем вызвать её, просто указав её имя:

ErrorMessage;

Компилятор перейдет к процедуре и выполнит её код (в данном случае - выведет сообщение об ошибке). После этого компилятор вернется назад, и выполнит следующий за вызовом процедуры оператор.

**Параметры**

Параметры подпрограмм - достаточно важная тема, поговорим об этом подробней. В процедуру (или в функцию) можно передать какие то исходные данные, чтобы процедура их обработала. Такие данные называются **параметрами**, или **формальными параметрами**. Пример - пользователь ввел какое-то число (а на самом деле, строку из цифровых символов), нам нужно удвоить его, а результат сообщить пользователю. Описать подобную процедуру можно следующим образом:

procedure Udvoenie(st: string);

var

r: real;

begin

//полученную строку преобразуем в число:

r:= StrToFloat(st);

//теперь удвоим его:

r:= r \* 2;

//теперь выведем результат в сообщении:

ShowMessage(FloatToStr(r));

end;

Этот пример уже сложнее, правда? На самом деле, всё просто. Давайте разберем, что тут к чему. Итак, строка объявления процедуры:

procedure Udvoenie(st: string);

объявляет процедуру Udvoenie с параметром строкового типа st. Это означает, что теперь мы можем вызвать процедуру, передав ей в качестве параметра какую-то строку. Параметр st условно можно считать внутренней переменной процедуры, в которую компилятор скопирует передаваемую процедуре строку. Этот способ передачи данных в подпрограмму называется **параметром по значению**. Допустим, в дальнейшем мы вызвали процедуру таким образом:

Udvoenie('123.4');

Компилятор сделает вызов процедуры, передав в параметр st указанное значение '123.4'. Или же мы можем вызвать процедуру иначе, передав в неё значение, которое хранится в какой то другой строковой переменной:

myst:= '123.4';

Udvoenie(myst);

Результат будет таким же. Тут важно помнить, что тип передаваемого значения обязательно должен совпадать с типом параметра. Если параметр у нас string, то и передавать ему нужно значение типа string. Компилятор копирует это значение в параметр. Другими словами, если внутри процедуры мы изменим значение параметра st, это никак не отразится на переменной myst, поскольку мы изменим копию данных, а не сами данные.

Пойдем дальше. А дальше мы объявляем вещественную переменную r:

var

r: real;

Здесь она необходима, ведь нам нужно умножить значение параметра на два, поэтому мы вынуждены будем преобразовать строковое представление числа в настоящее число - ведь строку на два не умножишь! Результат поместим в r:

begin

//полученную строку преобразуем в число:

r:= StrToFloat(st);

Служебным словом begin мы начинаем тело процедуры. Стандартной функцией StrToFloat(st) мы преобразуем строковое значение параметра st в число, и присвоим это число переменной r.

Далее всё просто:

//теперь удвоим его:

r:= r \* 2;

//теперь выведем результат в сообщении:

ShowMessage(FloatToStr(r));

end;

Мы удваиваем значение r, результат этого помещаем снова в r, затем стандартной функцией FloatToStr(r) преобразуем полученное число в строку, и выводим эту строку в сообщении ShowMessage(). Вот, собственно, и всё.

Теперь мы сможем вызывать эту процедуру, когда необходимо, и передавать ей различные числа в виде строки. А уж функция сама позаботится обо всех необходимых преобразованиях, об удвоении числа и выводе результатов на экран.

Кстати, сами данные, которые мы передаём в подпрограмму, называются **аргументами** или **фактическими параметрами**. В примере вызова процедуры

myst:= '123.4';

Udvoenie(myst);

переменная myst - аргумент.

В качестве параметров в процедуре можно использовать не одну, а множество переменных. Если они имеют одинаковый тип, то их имена разделяют запятыми, а тип указывается в конце сразу для всех параметров. Например:

procedure MyStrings(st1, st2, st3: string);

Если параметры имеют разные типы, их разделяют точкой с запятой:

procedure MyProc1(st: string; r1:real);

procedure MyProc2(st1, st2, st3:string; r1:real);

**Практическое задание**

Откройте **Lazarus** с новым проектом. Как всегда, назовем главную форму (свойство Name) fMain, сохраним проект в папку **05**, при этом назовем проект, например, **MyPodprog**, а модулю дадим имя **Main**.

В свойстве Caption формы напишем

Примеры работы с подпрограммами

Наша задача: получить от пользователя вещественное число, удвоить его, и результат вывести на экран. Пользователь может ввести и целое число, но процедура обработает его как вещественное, например, если пользователь введет 3, то процедура получит 3.0. В результате вычисления получится 6.0, но FloatToStr() конечные нули не выводит, так что пользователь увидит на экране просто 6.

Ладно, сейчас нужно решить, как получить у пользователя число. Для этого используем компонент TEdit, который нам уже знаком по прошлым лекциям. Для начала установим метку TLabel с поясняющим текстом

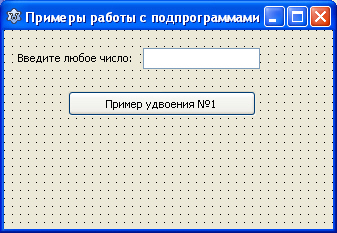
Введите любое число:

а рядом установим TEdit. Имена у TLabel и TEdit оставим по умолчанию, TEdit будет называться Edit1. Не забудьте очистить у него свойство Text.

Ниже установите простую кнопку **TButton**, в Caption которой напишите текст:

Пример удвоения №1

Разумеется, будут и другие примеры. Подровняйте компоненты, при необходимости измените их размеры. Наша форма должна выглядеть примерно так:



Пока что мы будем вынуждены доверять пользователю, что он введет в поле **Edit1** число, и ничего более. Но на прошлой лекции вам было обещано показать реализацию "защиты от дураков", так что чуть позже мы и это сделаем.

Сгенерируйте событие нажатия на кнопку, оно будет таким:

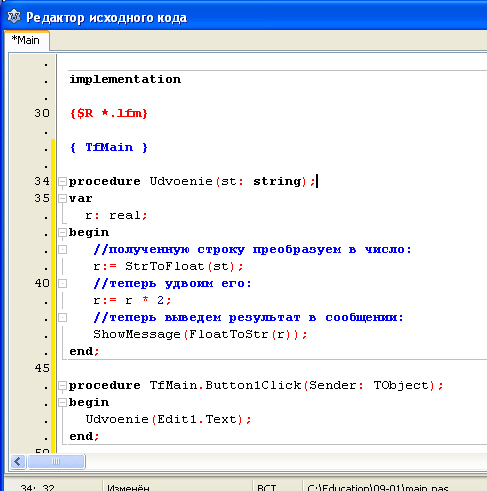
procedure TfMain.Button1Click(Sender: TObject);

begin

Udvoenie(Edit1.Text);

end;

Теперь нам нужно создать процедуру Udvoenie **выше** нашего события, сразу после комментария { TfMain }

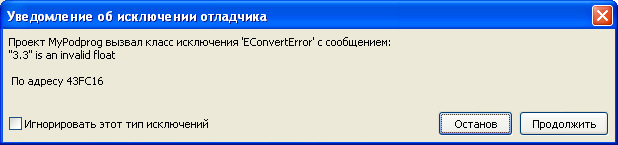


Обратите внимание, мы передаем в подпрограмму значение, которое ввел пользователь, и которое хранится в свойстве Text компонента Edit1:

Udvoenie(Edit1.Text);

Никаких дополнительных переменных в данном случае создавать не нужно. Сохраните проект и запустите его на выполнение. Попробуйте ввести целое число. Затем вещественное. Обратите внимание: если у вас установлена русская версия Windows, то в качестве разделителя вещественного числа нам нужно вводить запятую, а не точку! Помните про глобальную переменную DecimalSeparator?

Если же вы случайно или намеренно ввели точку, то выйдет сообщение об ошибке, подобное этому:



Ничего страшного, нажмите кнопку "**Останов**", затем выберите команду главного меню "**Запуск -> Сбросить отладчик**". Lazarus закроет зависший проект, и вы сможете запустить его снова. Похожая ошибка возникнет, если вы попытаетесь удвоить пустую строку. Если же вы ввели числа правильно, то программа отработает как нужно в независимости, целое это было число, или вещественное. Не закрывайте пока проект, он нам еще понадобится.

**Функции**

Функции являются такими же подпрограммами, как и процедуры, с одной разницей: они возвращают значение указанного типа. Функция начинается с ключевого слова function и имеет следующий синтаксис:

function <имя функции>(<список параметров>): <тип возвращаемого значения>;

const

<объявление констант>;

type

<объявление новых типов>;

var

<объявление переменных>;

<описание вложенных процедур и функций>;

begin

<тело процедуры>;

end;

В теле функции должен быть оператор, который присваивает возвращаемое значение стандартной переменной Result, что и приводит к возвращению функцией значения. Result объявлять не нужно, он уже есть в каждой функции. Разумеется, в Result нужно присваивать значение только указанного в заголовке типа. Пример:

function MyFunc(i: integer): integer;

begin

Result:= i \* 2;

end;

Здесь функция MyFunc принимает в качестве параметра какое-то целое число, удваивает его и присваивает результат переменной Result. Это приводит к тому, что функция возвращает этот результат. Теперь мы можем удвоить целое значение, вызвав эту функцию, например:

myperem:= 5;

myperem:= MyFunc(myperem);

Что будет в результате в переменной myperem? Если вы ответили 10, то вы правы. Здесь мы в первом шаге присваиваем переменной целого типа myperem значение 5. Во втором шаге мы вызываем функцию MyFunc, передавая ей в качестве параметра значение myperem. В третьем шаге мы присваиваем полученный от функции результат удвоения снова в переменную myperem.

Кстати, вместо системной переменной Result можно использовать имя функции:

function MyFunc(i: integer): integer;

begin

MyFunc:= i \* 2;

end;

Данный пример даст точно такой же результат. Какой способ использовать - дело выбора. Лично я предпочитаю использовать Result, это выглядит как-то более стандартно. В любом случае, вы должны знать про оба способа.

В отличие от многих других языков, в Lazarus переменной Result (или имени функции) значение можно присваивать неоднократно, если этого требует логика подпрограммы. Например, функция получает два вещественных числа. Первое нужно разделить на второе, и результат вернуть. Но на ноль делить нельзя, поэтому во избежание зависания программы, если второе число - ноль, то и вернуть нужно ноль. Такую функцию можно реализовать следующим образом:

function Delenie(r1, r2: real): real;

begin

if r2 = 0 then Result:= 0

else Result:= r1 / r2;

end;

**Практическое задание**

Сгенерируйте для второй кнопки событие OnClick, чуть выше этого события опишите функцию удвоения FuncUdvoenie. В результате у вас получится следующее:

function FuncUdvoenie(st: string): string;

var

r: real;

begin

//полученную строку сначала преобразуем в число:

r:= StrToFloat(st);

//теперь удвоим его:

r:= r \* 2;

//теперь вернем результат в виде строки:

Result:= FloatToStr(r);

end;

procedure TfMain.Button2Click(Sender: TObject);

begin

ShowMessage(FuncUdvoenie(Edit1.Text));

end;

Обратите внимание, здесь мы пошли немного другим путем. Мы создали функцию, которая лишь возвращает результирующее число в виде строки, а вывод этой строки на экран организовали при вызове функции, в строке

ShowMessage(FuncUdvoenie(Edit1.Text));

Тут в первом шаге мы вызываем функцию FuncUdvoenie, передавая ей в качестве параметра текст из Edit1. Во втором шаге отрабатывает функция - преобразует этот текст в число, удваивает его, снова преобразует в строку, и результат возвращает компилятору. А в третьем шаге с помощью функции ShowMessage() мы выводим этот результат на экран. Реализовано иначе, но работать будет точно также. Попробуйте. И пока не закрывайте проект.

**Параметры по ссылке**

Параметры по ссылке позволяют изменять сами аргументы, поскольку в процедуру или функцию передается **не копия** аргумента, а ссылка на сам аргумент. Чтобы в функцию или процедуру передать параметр по ссылке, нужно перед параметром указать ключевое слово var, например:

procedure MyProc(var myparam: integer);

Tсли вы передаете в подпрограмму параметры как по значению, так и по ссылке, то параметры по ссылке должны идти в описании последними. Вот пример объявления процедуры с множеством параметров:

procedure MyProc2(a,b: integer; c: real; var s1, s2: string);

Здесь мы объявили три параметра по значению: a и b - целые числа, c - вещественное; и два параметра по ссылке - s1 и s2, оба строкового типа. Если внутри процедуры мы изменим все эти параметры, то исходные целые числа и вещественное число не изменятся, но изменения в обеих строках будут сохранены и после выхода из процедуры. Давайте попробуем поработать с параметрами по ссылке на практике. Сделайте в проекте третью кнопку, аналогично первым двум, и расположенную ниже. В свойстве Caption этой кнопки напишем:

Пример удвоения №3

Сгенерируйте событие OnClick для нее, и чуть выше опишите процедуру, которая будет принимать и изменять параметр по ссылке, следующим образом:

procedure UdvoeniePoSsilke(var r: real);

begin

r:= r \* 2;

end;

procedure TfMain.Button3Click(Sender: TObject);

var

myReal: real;

begin

myReal:= StrToFloat(Edit1.Text);

UdvoeniePoSsilke(myReal);

ShowMessage(FloatToStr(myReal));

end;

Этот пример отличается от предыдущих, но выполняет ту же работу, и с тем же результатом. В процедуре UdvoeniePoSsilke всего одна строка

r:= r \* 2;

Эта строка удваивает не просто параметр-копию, она удваивает сам аргумент! Поскольку перед параметром r указано ключевое слово var, то этот параметр - не копия аргумента, а ссылка на него. И удваивая r, мы удваиваем аргумент. Что и было продемонстрировано следующим событием нажатия на третью кнопку. Здесь, в первом шаге мы преобразуем строковое представление числа из Edit1 в число, и результат присваиваем вещественной переменной myReal. Затем, во втором шаге, мы передаем эту переменную в процедуру UdvoeniePoSsilke, которая изменяет значение myReal. Ну и, наконец, в третьем шаге, мы выводим результат на экран, предварительно преобразовав его в строковое представление. Надеюсь, вы не запутались в параметрах и аргументах?

**Описание подпрограмм с их предварительным объявлением**

Применение процедур и функций вышеописанными способами имеет некоторые недостатки. Во-первых, мы вынуждены описывать процедуры и функции выше того места, где будем их применять. Это связано с тем, что компилятор не будет знать про эти подпрограммы, если описать их ниже, и не сможет их выполнить. Собственно, вы не сможете даже скомпилировать такой проект. Во-вторых, в таких подпрограммах мы не сможем обращаться к свойствам компонентов. Если бы мы попытались из подпрограммы обратиться к той же Edit1.Text, то не смогли бы это сделать, хотя в событии нажатия на кнопку мы делаем это без труда. Дело в том, что события принадлежат к самой нашей форме, это видно по названию события:

procedure TfMain.Button3Click(Sender: TObject);

Видите, компилятор обращается вначале к форме, и только потом к самому событию? А вот в случае подпрограмм мы обращаемся напрямую к процедуре или функции. Но есть способ объявления подпрограммы, как части формы. Такой способ предоставляет нам следующие преимущества:

1. Предварительно объявленную подпрограмму можно описывать в любом месте программы, не обязательно выше места её применения.
2. В такой подпрограмме можно обращаться к свойствам и событиям компонентов.
3. Подпрограмму можно описать как приватную, или как публичную. Приватную можно использовать только в текущем модуле (pas-файле). Публичную можно использовать в любом другом модуле, к которому подключается текущий.

Давайте посмотрим, как это делается на практике. Добавьте четвертую кнопку ниже предыдущих трех, с надписью в Caption

Пример удвоения №4

Сгенерируйте для неё событие нажатия на кнопку, в котором просто укажем вызов процедуры MyPrivat:

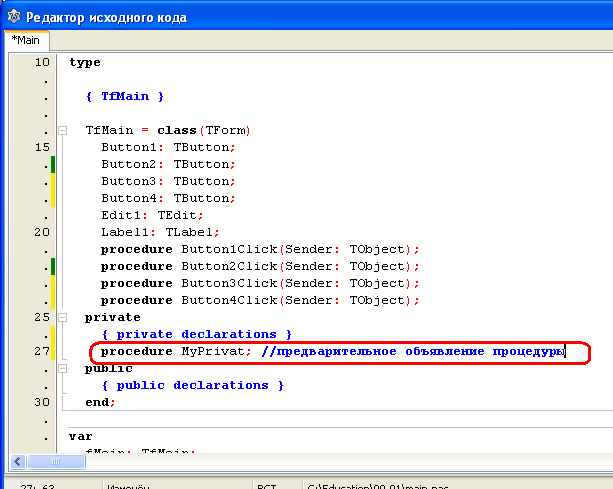
procedure TfMain.Button4Click(Sender: TObject);

begin

MyPrivat;

end;

Это ничего, что самой процедуры еще нет, сейчас мы её объявим, а затем создадим. Поднимите курсор на начало модуля. Вы увидите раздел type, в котором есть объявление класса TfMain - нашей формы. А там объявлены все компоненты и сгенерированные нами ранее события OnClick. Ниже располагаются подразделы private и public, где мы можем объявлять подпрограммы соответственно, приватные и публичные. Давайте объявим процедуру MyPrivat в подразделе private:



Теперь, не убирая курсора с этой строки, нажмите клавиши **<Ctrl + Shift + C>**. Это приведет к тому, что в самом низу модуля (но перед завершающим "end.") сгенерируется описание процедуры MyPrivat. На самом деле, она могла бы быть сгенерирована где-то и в другом месте, всё зависит от того, в каком порядке указаны объявления всех событий и других подпрограмм. Посмотрите на рисунок - там MyPrivat объявлена последней, потому и сгенерировалось описание внизу модуля.

Никаких параметров нам в данном случае не нужно, мы будем обращаться к свойству Text компонента Edit1 прямо из нашей подпрограммы. И обратите внимание, в описании перед именем процедуры указано имя формы. Отредактируйте подпрограмму следующим образом:

procedure TfMain.MyPrivat;

var

r: real;

begin

//преобразуем в число то, что ввел пользователь:

r:= StrToFloat(Edit1.Text);

//теперь удвоим его:

r:= r \* 2;

//теперь выведем результат в сообщении:

ShowMessage(FloatToStr(r));

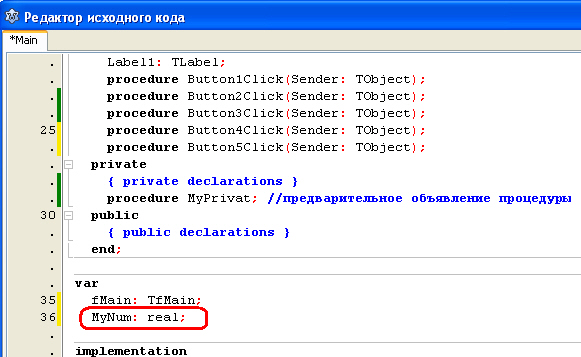
end;

Комментарии достаточно подробны, чтобы вы смогли разобраться с кодом. Подобный способ применения подпрограмм с предварительным объявлением является наиболее удобным, советую использовать именно его. Подпрограммы без предварительного объявления стоит использовать только в простейших случаях, когда, к примеру, нужно рассчитать какие-то данные, ну например, преобразовать значения температуры из шкалы в Фаренгейтах в шкалу по Цельсию. Тогда можно описать функцию подобного преобразования выше, и передавать в неё различные величины. Но даже и такую подпрограмму можно предварительно объявить!

**Область видимости переменных**

До сих пор мы не рассматривали переменные в этом аспекте. Просто объявляли их в подпрограммах, и использовали только внутри их. Такие переменные называются **локальными**, поскольку имеют локальную область видимости. Поясню. Переменная, объявленная внутри процедуры или функции, физически создается только тогда, когда компилятор обращается к данной подпрограмме. До этого переменной не существует. Только когда происходит вызов процедуры или функции, в оперативной памяти физически выделяется место для объявленных в подпрограмме переменных, констант и других объектов. Когда же подпрограмма завершает свою работу, то все эти переменные (константы и проч.) автоматически уничтожаются. К ним нельзя обратиться из других подпрограмм, их там просто не видно. Вот почему мы можем объявлять переменные с одинаковым именем в различных подпрограммах - это разные переменные, и они не мешают друг другу. Посмотрите на код нашего проекта - мы трижды объявляли переменную r вещественного типа, и каждый раз это была другая переменная.

Однако бывают моменты, когда требуется использовать **глобальные переменные** - переменные, которые видны по всему модулю, и в других модулях, если к ним подключается текущий. Такие переменные мы можем объявить либо в разделах private и public, **до** объявления подпрограмм, либо в разделе interface, после объявления переменной с именем формы, до ключевого слова implementation. Давайте объявим переменную MyNum:



Теперь добавим на форму пятую кнопку, текст в Caption будет соответствующий. Сгенерируйте для неё событие OnClick. Обратите внимание, оно создалось выше предыдущей процедуры MyPrivat, поскольку объявления различных событий располагаются выше разделов private и public. Опишем событие следующим образом:

procedure TfMain.Button5Click(Sender: TObject);

begin

MyNum:= StrToFloat(Edit1.Text);

//теперь удвоим его:

MyDouble;

//выводим результат на экран:

ShowMessage(FloatToStr(MyNum));

end;

Обратите внимание: переменную MyNum мы в событии не объявляем - она уже объявлена глобально, и ей можно пользоваться в любом месте модуля. Процедуры MyDouble ещё нет, объявим её в разделе private ниже MyPrivat:

procedure MyDouble; //удвоение глобальной переменной

Далее сгенерируем описание этой процедуры (<Ctrl + Shift + C>). Само описание будет очень простым:

procedure TfMain.MyDouble;

begin

//удвоим глобальную переменную:

MyNum:= MyNum \* 2;

end;

Как видите, здесь мы работаем с той же глобальной переменной, не объявляя её внутри процедуры, поскольку она уже объявлена.

Глобальные переменные очень удобны, однако используйте их только там, где это действительно необходимо. Ведь память для глобальной переменной выделяется, когда вы загружаете программу, и не освобождается, пока вы программу не закроете.

**Досрочный выход из подпрограмм и программы**

Иногда бывает необходимо срочно завершить процедуру или функцию. К примеру, в зависимости от каких то условий, вам требуется либо продолжать обработку данных, либо закончить подпрограмму и вывести готовый результат. Для этого существует ключевое слово Exit. Запомните: если в подпрограмме встретилось exit, подпрограмма досрочно завершает свою работу и компилятор передает управление следующему за вызовом подпрограммы оператору.

Если же встретится ключевое слово Halt, то уже вся программа досрочно завершает свою работу. Она закрывается, память, занимаемая программой, освобождается, а управление передается операционной системе. Чаще всего эту команду применяют для аварийного завершения работы.

Сейчас мы не будем применять эти операторы на практике, поскольку лекция итак получилась очень большой. При желании, вы можете опробовать их самостоятельно, в каких-то из уже сделанных подпрограмм, или в новых. В дальнейшем мы не раз будем обращаться к этим операторам, по крайней мере, к оператору Exit.