Лабораторная работа 2. Тестирование и отладка приложений в среде LAZARUS

Цель лабораторной работы

Освоить работу с встроенным отладчиком, изучить категории ошибок, способы их обнаружения и устранения.

Тестирование и отладка программы

Чем больше опыта имеет программист, тем меньше ошибок в коде он совершает. Но, хотите верьте, хотите нет, даже самый опытный программист всё же допускает ошибки. И любая современная *среда разработки* программ должна иметь собственные инструменты для отладки приложений, а также для своевременного обнаружения и исправления возможных ошибок. Программные ошибки на программистском сленге называют *багами* (англ. *bug* - жук), а программы отладки кода - *дебаггерами* (англ. *debugger* - отладчик). Lazarus, как современная *среда разработки* приложений, имеет собственный встроенный отладчик, работу с которым мы разберем на этой лекции.

Ошибки, которые может допустить программист, условно делятся на три группы:

1. Синтаксические
2. Времени выполнения (run-time errors)
3. Алгоритмические

Синтаксические ошибки

Синтаксические ошибки легче всего обнаружить и исправить - их обнаруживает *компилятор*, не давая скомпилировать и запустить программу. Причем *компилятор* устанавливает *курсор* на ошибку, или после неё, а в окне сообщений выводит соответствующее сообщение, например, такое:

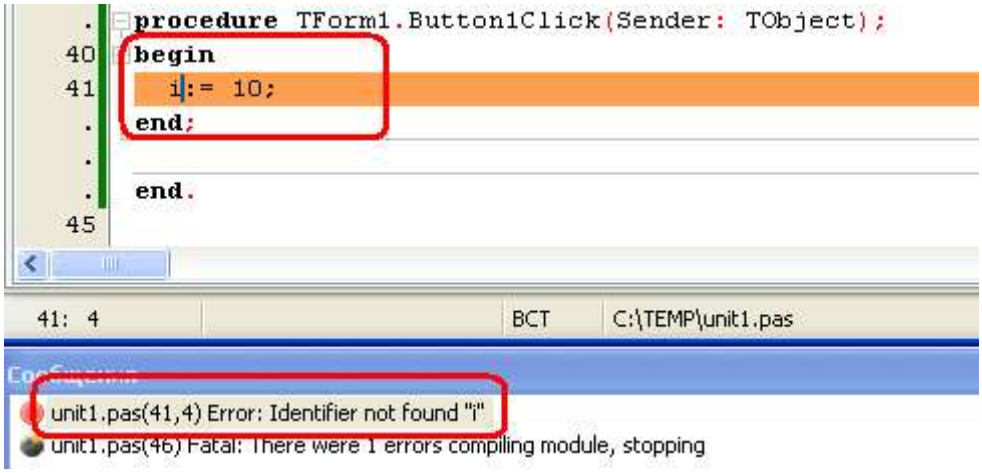

Рис. 2.1. Найденная компилятором синтаксическая ошибка - нет объявления переменной i

Подобные ошибки могут возникнуть при неправильном написании директивы или имени функции (процедуры); при попытке обратиться к переменной или константе, которую не объявляли (рис. 2.1); при попытке вызвать функцию (процедуру, переменную, константу) из модуля, который не был подключен в разделе uses; при других аналогичных недосмотрах программиста.

Как уже говорилось, *компилятор* при нахождении подобной ошибки приостанавливает процесс компиляции, выводит сообщение о найденной ошибке и устанавливает *курсор* на допущенную ошибку, или после неё. Программисту остается только внести исправления в *код программы* и выполнить повторную компиляцию.

Ошибки времени выполнения

Ошибки времени выполнения (run-time errors) тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы, или во *время тестирования*. Если такую программу запустить из среды Lazarus, то она скомпилируется, но при попытке загрузки, или в момент совершения ошибки, приостановит свою работу, выведя на экран соответствующее сообщение. Например, такое:

Рис. 2.2. Сообщение Lazarus об ошибке времени выполнения

В данном случае *программа* при загрузке должна была считать в *память* отсутствующий *текстовый файл* **MyFile.txt**. Поскольку *программа* вызвала ошибку, она не запустилась, но в среде Lazarus процесс отладки продолжается, о чем свидетельствует сообщение в скобках в заголовке главного *меню*, после названия проекта. Программисту в подобных случаях нужно сбросить отладчик командой *меню* "**Запуск -> Сбросить отладчик**", после чего можно продолжить работу над проектом.

Ошибка времени выполнения может возникнуть не только при загрузке программы, но и во время её работы. Например, если бы попытка чтения несуществующего файла была сделана не при загрузке программы, а при нажатии на кнопку, то *программа* бы нормально запустилась и работала, пока *пользователь* не нажмет на эту кнопку.

Если программу запустить из самой *Windows*, при возникновении этой ошибки появится такое же сообщение. При этом если нажать "**OK**", *программа* даже может запуститься, но корректно работать все равно не будет.

Ошибки времени выполнения бывают не только явными, но и неявными, при которых *программа* продолжает свою работу, не выводя никаких сообщений, а программист даже не догадывается о наличии ошибки. Примером неявной ошибки может служить так называемая **утечка памяти**. Утечка памяти возникает в случаях, когда программист забывает освободить выделенную под *объект память*. Например, мы объявляем переменную типа TStringList, и работаем с ней:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
end;
```

В данном примере программист допустил типичную для начинающих ошибку - не освободил *класс* TStringList. Это не приведет к сбою или аварийному завершению программы, но в итоге можно бесполезно израсходовать очень много памяти. Конечно, эта *память* будет освобождена после выгрузки программы (за этим следит *операционная система*), но утечка памяти во *время выполнения* программы тоже может привести к неприятным последствиям, потребляя все больше и больше ресурсов и излишне нагружая *процессор*. В подобных случаях после работы с объектом программисту нужно не забывать освобождать *память*:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
  ...; //работа с объектом
  MySL.Free; //освободили объект
end;
```

Однако *ошибки времени выполнения* могут случиться и во время работы с объектом. Если есть такой риск, программист должен не забывать про возможность обработки исключительных ситуаций. В данном случае вышеприведенный код правильней будет оформить таким образом:

```
begin
  try
    MySL:= TStringList.Create;
    MySL.Add('Новая строка');
    ...; //работа с объектом
  finally
    MySL.Free; //освободили объект, даже если была ошибка
  end;
end;
```

Итак, во избежание ошибок времени выполнения программист должен не забывать делать проверку на правильность ввода пользователем допустимых значений, заключать опасный код в блоки try…finally…end или try…except…end, делать проверку на существование открываемого файла функцией FileExists и вообще соблюдать предусмотрительность во всех слабых местах программы. Не полагайтесь на пользователя, ведь недаром говорят, что если в программе можно допустить ошибку, *пользователь* эту возможность непременно найдет.

Алгоритмические ошибки

Если вы не допустили ни синтаксических ошибок, ни ошибок времени выполнения, *программа* скомпилировалась, запустилась и работает нормально, то это еще не означает, что в программе нет ошибок. Убедиться в этом можно только в процессе её *тестирования*.

Тестирование - *процесс проверки работоспособности программы путем ввода в неё различных, даже намеренно ошибочных данных, и последующей контрольной проверке выводимого результата.*

Если *программа* работает правильно с одними наборами исходных данных, и неправильно с другими, то это свидетельствует о наличии алгоритмической ошибки. Алгоритмические ошибки иногда называют логическими, обычно они связаны с неверной реализацией алгоритма программы: вместо "+" ошибочно поставили "-", вместо "/" - "*", вместо деления значения на 0,01 разделили на 0,001 и т.п. Такие ошибки обычно не обнаруживаются во *время компиляции*, *программа* нормально запускается, работает, а при анализе выводимого результата выясняется, что он неверный. При этом *компилятор* не укажет программисту на ошибку - чтобы найти и устранить её, приходится анализировать код, пошагово "прокручивать" его выполнение, следя за результатом. Такой процесс называется *отладкой*.

Отладка - *процесс поиска и устранения ошибок, чаще алгоритмических. Хотя отладчик позволяет справиться и с ошибками времени выполнения, которые не обнаруживаются явно.*

Работа с отладчиком

Давайте от теории перейдем к практике. Загрузите **Lazarus** с новым проектом, установите на форму простую кнопку и сохраните проект в папку **ЛР2**. Имена проекта, формы, модуля и кнопки изменять не нужно, оставьте имена, данные по умолчанию.

Далее, сгенерируйте событие OnClick для кнопки, в котором напишите следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  st: TStringList;
begin
  //создаем список строк:
  st:= TStringList.Create;
  try
    //генерируем список:
    for i:= -3 to 3 do begin
      st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
    end;
    //выводим список на экран:
    ShowMessage(st.Text);
  finally
    //st будет освобождена даже в случае run-time ошибки:
    st.Free;
  end;
end;
```

Что мы тут делаем? Целочисленную переменную i используем в качестве счетчика для *цикла* for. Цикл производим от -3 до 3, то есть, 7 раз, включая ноль. В теле *цикла* мы делим 10 на *значение* i, результат оформляем в виде строки и добавляем к списку строк st. Выше говорилось, что подобные действия нужно заключать в блок обработки исключительных ситуаций try-finally-end, что мы и сделали.

Если вы внимательно изучали курс, то невооруженным глазом видите, что при четвертом проходе *цикла* произойдет ошибка времени выполнения - *деление* 10 на ноль. Такой очевидный пример больше всего подходит для знакомства с встроенным отладчиком, так

как вы уже знаете, где будет ошибка, и сможете проанализировать работу отладчика. Поэтому притворимся, что не подозреваем об ошибке.

Итак, программу мы написали, сохранили, пора её компилировать. Нажмите кнопку "**Запустить**" на **Панели управления** (или **<F9>**). *Программа* нормально скомпилировалась и запустилась. Нажмем кнопку **Button1**. И тут же получаем ошибку:

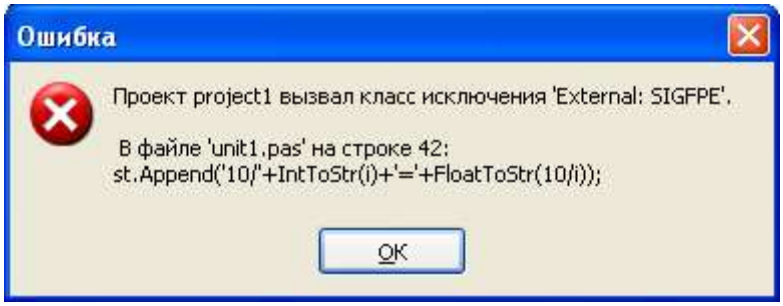

Рис. 2.3. Сообщение Lazarus об ошибке

Судя по сообщению, ошибка произошла во *время выполнения* кода 42-й строки. Ладно, нажмем "**ОК**" и командой "**Запуск -> Сбросить отладчик**" прекратим выполнение программы. Вернемся к коду и проанализируем 42-ю строку (если вы добавляли пустые строки, то у вас будет другой номер):

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

Ну что, ничего криминального тут нет, почему же произошла ошибка? Код верный и должен был нормально выполняться… Когда вы заходите в подобный *тупик*, помочь вам может *здравый смысл* и встроенный отладчик. *Здравый смысл* говорит, что ошибка произошла где-то в теле *цикла* for. А чтобы воспользоваться отладчиком, нужно приостановить выполнение программы на этом цикле, чтобы потом построчно его продолжить. Для остановки работы программы служат так называемые **точки останова** (англ. *breakpoints*).

Точки останова - *это строки, перед выполнением которых отладчик приостанавливает выполнение программы, и ждет ваших дальнейших действий.*

Вы можете установить одну такую точку или несколько, в различных частях кода. Поскольку ошибка возникает в 42-й строке, разумней будет приостановить выполнение на предыдущей, 41-й строке. Переведите *курсор* на эту строку, на любое её *место*.

Установить точку останова можно разными способами:

- Командой главного меню "**Запуск -> Добавить точку останова -> Точка останова в исходном коде**". В открывшемся окне "**Параметры точки останова**" нажать "**ОК**".
- Щелкнуть по строке правой кнопкой, и в всплывающем меню выбрать "**Отладка -> Переключить точку останова**".
- Нажать "горячую клавишу" **<F5>**.
- Щелкнуть по нужной строке в левой части **Редактора кода**, где указаны номера строк.

Последние два способа наиболее удобны, но выбирать вам. В любом случае, строка с установленной точкой останова окрасится красным цветом:

Рис. 2.4. Строка с точкой останова

Снять точку останова удобней также последними двумя способами. *Точка останова* у нас есть, снова нажимаем кнопку "**Запустить**". *Программа* начинает свою работу, нажимаем кнопку "**Button1**".

Теперь *программа* не вывела ошибку, а приостановила свою работу и вывела на передний план **Редактор кодов** с выделенной серым цветом строкой, которая в данный момент готовится к выполнению:

Рис. 2.5. Строка, которая будет выполнена далее

Тут очень важно понимать, что *программа* была остановлена ДО выполнения этой строки, а не ПОСЛЕ неё. То есть, в настоящий момент переменной i еще не присвоено *значение* -3. Далее, мы можем выполнять с отладчиком различные действия, которые собраны в разделе главного *меню* "**Запуск**". Обычно требуется *пошаговое выполнение* программы. Для этого можно использовать команду "**Запуск -> Шаг в обход**" (или **<F8>**), "**Запуск -> Шаг с входом**" (или **<F7>**) или "**Запуск -> Шаг с выходом**" (или **<Shift+F8>**). "**Шаг в обход**" означает, что если в коде будет встречен вызов какой-нибудь функции или процедуры, отладчик выполнит их и остановится на следующей после вызова строке. При выборе "**Шаг со входом**", отладчик также пошагово будет выполнять и вызываемые функции-процедуры. "**Шаг с выходом**" подразумевает, что если в строке нет вызовов функций, то остановки происходят, как при "**Шаг в обход**". Если в строке есть *выражение*, то остановка происходит вначале перед строкой, затем перед вычислением каждой функции, чтобы мы имели возможность просмотреть значения параметров, передаваемых в функцию.

У нас вызовов функций нет, поэтому мы можем воспользоваться как **<F7>**, так и **<F8>** (чаще всего используют **<F8>** - Шаг в обход).

Итак, нажмем **<F8>**, и отладчик выполнит строку с точкой останова, и выделит серым следующую строку. Снова нажмем **<F8>**, и снова будет выделена эта строка - был выполнен шаг *цикла*. Нажав несколько раз **<F8>**, мы добьемся появления на экране всё той же ошибки, которая заблокирует дальнейшее выполнение программы. Становится понятно, что цикл нормально выполняется несколько проходов, после чего всё же возникает ошибка. Включаем логику: внутри *цикла* у нас изменяется только *переменная* **i**, значит, ошибка как-то связана с ней. А как узнать, как именно?

Здесь нам на помощь приходит еще один полезный инструмент отладчика - наблюдение за значениями переменных. Сбросьте программу командой "**Запуск -> Сбросить отладчик**".

Теперь снова нажмите кнопку "**Запустить**", а потом снова кнопку "**Button1**". Отладчик снова приостановил выполнение программы на строчке с циклом, однако не спешите нажимать **<F8>**. Для начала, добавим наблюдение над переменной **i**. Делается это командой "**Запуск -> Добавить наблюдение**", которая была недоступна, пока *программа* не начала выполняться. В строке "**Выражение**" укажите переменную **i**, и нажмите "**ОК**":

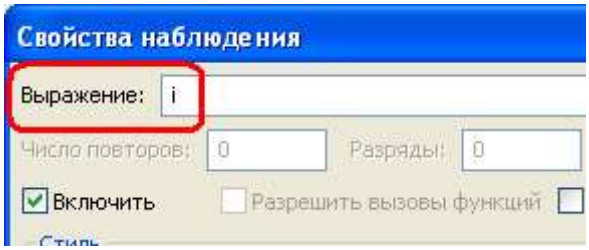

Рис. 2.6. Установка наблюдения за переменной

Теперь отладчик наблюдает за значениями переменной i, но нам от этого не легче - мы то не видим этих значений! Чтобы их увидеть, нужно вывести на экран окно **Списка наблюдений**. Делается это командой "**Вид -> Окна отладки -> Окно наблюдений**" или "горячими клавишами" **<Ctrl+Alt+W>**.

Расположите окно ниже **Редактор кода**, на *место* **Окна сообщений**. В этом окне вы сможете видеть *выражение* - нашу переменную i, и её текущее *значение*. Чтобы показать работу с отладчиком более наглядно, давайте добавим еще одно *выражение*, за которым будем наблюдать. Выберите "**Запуск -> Добавить наблюдение**" и в строке "**Выражение**" укажите не просто переменную, а *выражение*

10/i

которое у нас должно вычисляться внутри *цикла*. В окне **Списка наблюдения** вы увидите и переменную i, и *выражение*, а также их значения:

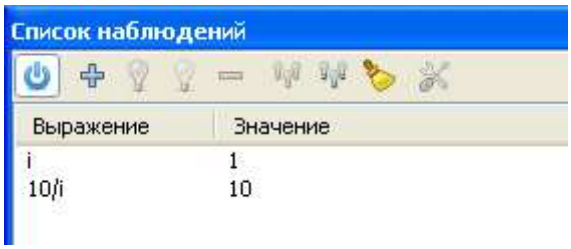

Рис. 2.7. Окно Списка наблюдений

Поскольку переменной i еще не было присвоено значения -3, то в колонке значений вы, скорее всего, увидите 1, которым по умолчанию была проинициализирована наша *переменная*. Соответственное *значение* будет и у выражения. Теперь мы готовы двигаться дальше. Нажимаем **<F8>**. В **Списке наблюдений** сразу же изменилась картина - i теперь равно -3, а *выражение* -3,3333…

Нажимаем **<F8>** ещё раз. Снова значения изменились, теперь i = -2, а *выражение* = -5. Мы понимаем, что цикл работает, и два его шага были сделаны. Нажимаем **<F8>** еще два раза. Сейчас *переменная* содержит ноль, а *значение* выражения указывает "inf". Однако строка с вычислением еще не была выполнена, не забываем об этом. Снова нажимаем **<F8>**, и снова

получаем ошибку. А в значениях переменной и выражения видим *слово* "evaluating", что переводится, как "оценка". Теперь мы наглядно видим, что в строке

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

возникает ошибка, когда *переменная* i равна нулю. И тут уже несложно догадаться, почему эта ошибка возникает - потому что происходит попытка деления 10 на 0.

Это можно проверить, пропустив выполнение вычисления, когда i=0. Закройте окно с ошибкой, сбросьте отладчик. Снова нажмите кнопку "**Запустить**", и кнопку "**Button1**". Снова выведите окно **Списка наблюдения**. Нажимайте **<F8>**, пока i не станет 0, а *выражение* - inf. Теперь, в **Окне наблюдений** щелкните правой кнопкой мыши по строке с переменной i, и в всплывающем *меню* выберите команду "**Вычислить/Изменить**". В открывшемся окне вы увидите строку "**Выражение**", где будет указана *переменная* i. В *поле* "**Результат**" будет указано *значение* 0. А в строке "**Новое значение**" нам нужно указать *значение*, которое мы желаем принудительно присвоить переменной. Тут укажем 1 и нажмем **<Enter>** или кнопку "**Изменить**". В *поле* "**Результат**" *значение* должно смениться на 1. Снова нажмем **<F8>**, *значение* i изменится на 2, ошибки не будет. Нажав **<F8>** еще несколько раз, мы доберемся до конца программы и увидим сообщение, которое она и должна была вывести по нашему замыслу:

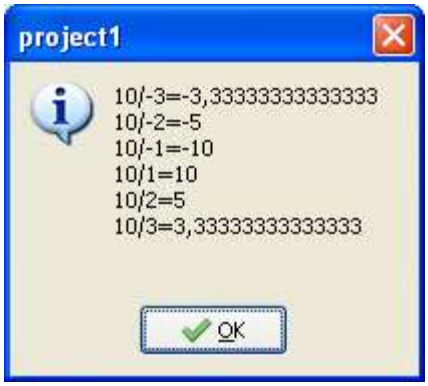

Рис. 2.8. Результирующее сообщение программы

Как видите, *вычисление*, где i равна нулю, было пропущено.

Встроенный отладчик имеет и другие инструменты, с которыми вы сами сможете со временем освоиться, экспериментируя с ними.

Контрольные вопросы и задания

1. Покажите на примерах, как в оболочке Lazarus осуществляется:
запуск и выход из оболочки,
загрузка и сохранение файла,
вызов справки, в т.ч. по ключевому слову, на которое указывает курсор,
контекстный поиск и замена текста,
компиляция и запуск программы.
2. Объясните понятия: синтаксическая ошибка, ошибка времени выполнения, логическая ошибка.
3. 4Покажите на примерах, как в оболочке Lazarus осуществляется: добавление, редактирование и удаление переменных в окне просмотра значений переменных,

пошаговое выполнение программ, в т.ч. с пошаговым выполнением операторов в вызываемых подпрограммах, выполнение программы до строки, на которую указывает курсор, завершение отладки программы, создание, редактирование и удаление точек останова программы.

4. Составьте отчёт, в котором опишите свои действия во время выполнения заданий 1 – 6, проиллюстрируйте скриншотами.