Programming – Block C

http://www.win.tue.nl/~wstomv/2ip05/

Lecture 12

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Feedback to T.Verhoeff@TUE.NL

Steps to Construct a GUI Application in Lazarus (1)

Set compiler options: Project > Compiler Options ...

Parsing Delphi mode, Include Assertion Code, Use Ansi Strings

Code Checks: I/O, Range, Overflow, Stack

Linking Debugging: Display Line Numbers in Run-time Error Backtraces

Later also: Use Heaptrc Unit;

Target OS specific options: GUI application

Steps to Construct a GUI Application in Lazarus (2)

Add GUI components to (main) form:

Standard `TMainMenu`, `TButton`, `TLabel`, `TEdit`, `TMemo`

Additional `TPaintBox`, `TStringGrid`

Common Controls `TTrackBar`, `TProgressBar`, `TUpDown`

Dialogs `TOpenDialog`, `TSaveDialog`

Misc `TFileNameEdit`

Steps to Construct a GUI Application in Lazarus (3)

Configure properties of components via Object Inspector:

- **`Name`** (at design-time: name of component in source code)
- `Left`, `Top`, `Width`, `Height` (position and size, via dragging)
- `Caption`, `Text`, `ReadOnly`, `Min`, `Max`

Lazarus adds code to the type definition of form's class, appearing in the **`interface`** part of the form's unit, and also to the corresponding `*.lfm` file.

Steps to Construct a GUI Application in Lazarus (4)

Create, configure, implement event handlers via Object Inspector:

- `FormCreate`, `OnClick`, `OnChange`, `OnPaint`
- Multiple controls can share the same event handler

Lazarus adds code to the type definition of the form's class, appearing in the **`interface`** of part the form's unit, and also to the corresponding `*.lfm` file.

The implementation is placed in the **`implementation`** part.

Steps to Construct a GUI Application in Lazarus (5)

Add local routines:

1. Add the routine heading and contract to the **`public`** section of the form's class definition, which appears in the **`interface`** part of the form's unit
2. Use code completion to create the implementation skeleton in the **`implementation`** part of the form's unit
3. Fill in the body of the routine

Steps to Construct a GUI Application in Lazarus (6)

Add given unit to project:

1. Copy or move the source code of the given unit to the project folder
2. Open the source code in Lazarus
3. Project > Add editor file to project
4. Add the given unit's name to the **`uses`** clause of the units where you want to use the given unit

   N.B. Both the **`interface`** part and the **`implementation`** part can have a **`uses`** clause

Today's Topics

- Introduction to Abstract Data Types (ADTs)

  specification (read/write), usage, implementation

  User Contract Maker
- Simple ADTs in Object Pascal

  syntax, semantics, pragmatics
- Contract styles: property-oriented vs. model-oriented
- Some standard container ADTs

(Data) Types

A (data) type is a set of values and accompanying operations

Compare to an *algebra* in mathematics. In Pascal:

- `Boolean`, `Integer`, `Real`, `Char`,
- *enumeration* `(Red, Green, Blue)`, *subrange* `'0'..'9'`,
- **`string`**, **`array`**, **`record`**, **`set`**, `TextFile`, **`file`**,

- *pointer* `^Node`, **`class`**, *procedural*

Example of Type Usage

```
1  var
2    v : T; { variable v declared of type T }
3
4  begin { memory for v allocated, value undefined }
5
6    ReadLn(...v...)
7
8  ; if ...v... then ...
9
10 ; ...v... := ...
11
12 ; WriteLn(...v...)
13
14 end { memory for v de-allocated }
```


Example of Type Usage (2)

```
1  type
2    Point = record
3      x, y: Integer;
4    end;
5
6  var
7    p : Point;
8
9  begin
10
11   p.x := 0
12 ; p.y := 0
13
14 ...
```

```
1  type
2    Axis = (x, y);
3    Point = array
4      [ Axis ] of Integer;
5
6  var
7    p : Point;
8
9  begin
10
11   p[x] := 0
12 ; p[y] := 0
13
14 ...
```


Limitations of 'Classic' Pascal Types

- Memory allocation for storage:
  - same size for all values, determined at compile time
  - tied to (textual) *scope* of variables, and block activation
    scope = subset of text where named entity 'exists',
    in Pascal, this is typically a block (can be nested)
- Syntax for operations (how to use variables of the type):
  - ad hoc, dependent on type, i.e., on *implementation decisions*
  - change of *definition* requires change of all *using occurrences*

Abstract Data Type (ADT): Definition of Concept

An Abstract Data Type is a type whose specification and usage abstracts from (i.e. does not depend on) implementation details.

The implementation of an ADT can be changed, without affecting the using occurrences, provided it adheres to the ADT's specification.

Implementation hiding is also known as encapsulation.

An ADT can serve as a module of a program.

Simple Abstract Data Type: Interface Syntax

- Type name `TStringList`
- Operations (methods): **`procedure, function constructor, destructor, property`**
  - Constructor, destructor (mem. mgmt) `Create, Free`
  - Queries (state inspection) `Count, Strings`
  - Commands (state change) `LoadFromFile, Sort`

Consult FCL docs for more information on `TStringList`

Simple Abstract Data Type: Usage Syntax

```
1 program TStringListExample;
2   { illustrates the use of the ADT TStringList }
3
4 uses
5   Classes;  { has a few ADT definitions }
6
7 var
8   v : TStringList; { a dynamic list of strings }
```


Simple Abstract Data Type: Usage Syntax (2)

```
10 begin
11   v := TStringList.Create  { create empty list }
12
13 ; v.LoadFromFile('strings.in')  { read v from text file }
14
15 ; if v.Count <> 0 then begin  { v is not empty }
16     WriteLn(v.Strings[0])  { write first string in v }
17   end
18
19 ; v.Sort  { sort v }
20
21 ; v.SaveToFile('strings.out')  { write v to text file }
22
23 ; v.Free  { destroy v and de-allocate memory }
24 end.
```

Defining Your Own Simple Abstract Data Type: Syntax

- Put it in a (separate) **unit**
- Define the type name as a **class**
- Declare the operations as **public** methods

Rectangles ADT in Object Pascal: Interface Overview

```
1  unit Rectangles;
2    { provides ADT for axis-parallel grid rectangles }
3
4  interface
5
6  type
7    TRectangle = class(TObject)  // axis-parallel grid rectangles
15   public
17     constructor Create(AXL, AYL, AXH, AYH: Integer);
22     function XL: Integer; // lower X coordinate
23     function YL: Integer; // lower Y coordinate
24     function XH: Integer; // higher X coordinate
25     function YH: Integer; // higher Y coordinate
31     function Contains(AX, AY: Integer): Boolean;
```


Rectangles ADT in Object Pascal: Interface Overview

```
34     function Intersects(const ARectangle: TRectangle): Boolean;
39     procedure SetXL(AX: Integer);
42     procedure SetYL(AY: Integer);
45     procedure SetXH(AX: Integer);
48     procedure SetYH(AY: Integer);
51     procedure Shift(AX, AY: Integer);
54     procedure Intersect(const ARectangle: TRectangle);
57     procedure Hull(const ARectangle: TRectangle);
61   end; { class TRectangle }
```


Simple Abstract Data Type: Interface Semantics = Contract

- Set of (abstract) values
- Contracts for operations
  - preconditions
  - postconditions or effects
  - invariants

Rectangles ADT in Object Pascal: Interface Contract

```
1  unit Rectangles;
2    { provides ADT for axis-parallel grid rectangles }
3
4  interface
5
6  type
7    TRectangle = class(TObject)  // axis-parallel grid rectangles
8    private (* ignore implementation details *)
```

```
15   public
16     // construction
17     constructor Create(AXL, AYL, AXH, AYH: Integer);
18       // pre: AXL <= AXH  /\  AYL <= AYH
19       // post: XL=AXL /\ YL=AYL /\ XH=AXH /\ YH=AYH
```


Rectangles ADT in Object Pascal: Interface Contract

```
20
21     // basic queries
22     function XL: Integer; // lower X coordinate
23     function YL: Integer; // lower Y coordinate
24     function XH: Integer; // higher X coordinate
25     function YH: Integer; // higher Y coordinate
26
27     // invariants
28     // WellFormed: XL <= XH  /\  YL <= YH
29
30     // derived queries
31     function Contains(AX, AY: Integer): Boolean;
32       // pre: true
33       // ret: XL <= AX <= XH  /\  YL <= AY <= YH
```


Rectangles ADT in Object Pascal: Interface Contract

```
34     function Intersects(const ARectangle: TRectangle): Boolean;
35       // pre: true
36       // ret: Self intersects ARectangle
37
38     //commands
39     procedure SetXL(AX: Integer);
40       // pre: AX <= XH
41       // effect: XL := AX
42     procedure SetYL(AY: Integer);
43       // pre: AY <= YH
44       // effect: YL := AY
45     procedure SetXH(AX: Integer);
46       // pre: XL <= AX
47       // effect: XH := AX
```


Rectangles ADT in Object Pascal: Interface Contract

```
48     procedure SetYH(AY: Integer);
49       // pre: YL <= AY
50       // effect: YH := AY
51     procedure Shift(AX, AY: Integer);
52       // pre: true
53       // effect: XL, YL, XH, YH := XL+AX, YL+AY, XH+AX, YH+AY
54     procedure Intersect(const ARectangle: TRectangle);
55       // pre: Intersects(ARectangle)
56       // effect: Self := Self intersection ARectangle
57     procedure Hull(const ARectangle: TRectangle);
58       // pre: true
59       // effect: Self := smallest rectangle enclosing
60       //                 Self and ARectangle
61   end; { class TRectangle }
```

Using Rectangles ADT in Object Pascal

```
1  program RectanglesTest;
2
3  uses
4    Rectangles;
5
6  var
7    r, s: TRectangle;
8
9  begin
10   r := TRectangle.Create(0, 0, 200, 100)
11 ; s := TRectangle.Create(100, 50, 300, 150)
12 ; r.Intersect(s)
13 ; if r.Contains(50, 50) then writeln('Huh?')
14 end.
```


Pathological Behavior with Class References: Aliasing

```
1  program Aliasing;
2
3  uses
4    Rectangles;
5
6  var
7    r, s: TRectangle;
8
9  begin
10   r := TRectangle.Create(0, 0, 200, 100)
11 ; s := r  (* r, s: two names for the same object *)
12 ; s.SetYH(200)
13 ; if r.Contains(150, 150) then writeln('Huh?')
14 end.
```


Contract Styles: Property-oriented vs. Model-oriented

- Property-oriented: Abstract values and operations (implicitly) defined by (axiomatically) postulating their joint properties

  E.g. $(\mathbb{R}; <, +, *)$ is (uniquely) defined by the properties that it is a *complete, ordered field*.

- Model-oriented: Abstract values and operations defined by a mathematical model ("example" implementation).

  E.g. a model for $(\mathbb{R}; <, +, *)$ can be based on *infinite decimal expansions*, on *Cauchy sequences in* $\mathbb{Q}$, or on *Dedekind cuts in* $\mathbb{Q}$.

Example of Property-oriented Specification

ADT F has operations

```
1  constructor M;
2    { pre: true;  post: E }
3  function E: Boolean;
4    { pre: true }
5  procedure P(i: Integer);
6    { pre: true;  post: not E }
7  function I: Integer;
8    { pre: not E }
9  procedure R;
10   { pre: not E }
```

Example of Property-oriented Specification

ADT F has the following additional properties for `v: F; i: Integer`:

```
1   { v.E } v.P(i) { v.I = i }
2   { not v.E and v.I = A } v.P(i) { v.I = A }
3   { v.E } v.P(i) ; v.R { v.E }
4   { not v.E } v.P(i) ; v.R  equivalent  { not v.E } v.R ; v.P(i)
```

This uniquely determines $(F; M, E, P, I, R)$

What is $(F; M, E, P, I, R)$?

Example of Model-oriented Specification

ADT F (Unbounded First-In First-Out Buffer) has as abstract values sequences of integers: the state of `v: F` is a sequence of integers, also known as a queue.

`M`: Constructor `MakeEmpty` yields the empty sequence.

`E`: Query `IsEmpty` returns whether the sequence is empty.

`I`: Query `Item` returns the first item of the sequence.

`P`: Command `Put(i)` appends `i` to the end of the sequence.

`R`: Command `Remove` drops the first element of the sequence.

Standard Container ADTs

Stack Last-In-First-Out Buffer

Queue First-In-First-Out Buffer

Priority Queue Minimum-Out Buffer

Deque Double-Ended Queue

List Anyway-In-Out Buffer

Dictionary Associative Buffer

What interfaces and contracts? Also see **unit** `Contnrs` in FCL.

Standard Container ADTs: Interfaces and Contracts

Stack : Create, IsEmpty, Top, Push, Pop

Queue : First, Put, RemFirst

Priority Queue : Minimum, Put, RemMinimum

Deque : First, Last, RemFirst, RemLast

List : Count, Get item on index, Insert, Delete

Dictionary : Add/Delete key-value pair, Find value for key

Unbounded versus bounded How to implement efficiently?

ADT Specification: Mathematical Model of Abstract State

For Stack:

- Abstract state: sequence of elements
- IsEmpty: sequence is empty
- Top: last item of sequence
- Push: append item to end of sequence
- Pop: drop item from end of sequence

What Lies Ahead

- How to implement ADTs
- Performance issues: memory, speed
- How to design complete ADTs from scratch