Programming – Block C

`http://www.win.tue.nl/˜wstomv/2ip05/`

Lecture 11

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Feedback to `T.Verhoeff@TUE.NL`

Today's Topics

- Link with Block A and Block B
- Motivation for Abstract Data Types
- Simple Graphical User Interfaces in Lazarus

An Easy Programming Problem: Windows 2001

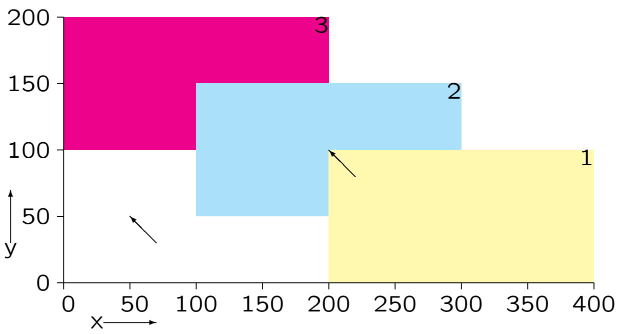
Problem G, Prelims, Dutch Programming Championship NKP 1998

Input:

```
1
3
200 0 400 100
100 50 300 150
0 100 200 200
2
50 50
200 100
```

Output:

```
Desktop
1
```



A Hacked Solution (with a mistake)

```
program Windows2001h; { Hacked version }
var r,n,m,x,y,j,k:integer;
  w:array[1..10000] of record xl,xh,yl,yh:integer end;
begin
  readln(r);
  for r:=1 to r do begin readln(n);
    for n:=1 to n do with w[n] do readln(xl,xh,yl,yh);
    readln(m);
    for m:=1 to m do begin readln(x,y);k:=1;j:=n+1;
      while k<>j do with w[k] do
        if (xl<=x)and(x<=xh)and(yl<=y)and(y<=yh) then j:=k
        else k:=k+1;
      if k<=n then writeln(k) else writeln('Desktop')
end end end.
```

A Monolithic Solution (using an array)

```
program Windows2001m1;
  { Monolithic version using an array to store the windows }

var
  r: Integer;     { number of runs (input) }
  i: Integer;     { to count off the runs }
  n: Integer;     { number of windows (input) }
  w: array [1..10000] of record
    xl, yl, xh, yh: Integer; { coordinates of a window }
    end;        { w[1..n] = list of windows (input) }
  j: Integer;     { to count off the windows }
  m: Integer;     { number of mouse clicks (input) }
  c: Integer;     { to count off the mouse clicks }
  x, y: Integer; { coordinates of a mouse click (input) }
  k: Integer;     { window number (output) }

begin
  ReadLn(r) { number of runs }
```

A Monolithic Solution (using an array)

```
; for i := 1 to r do begin

    { Read and store the windows }
    ReadLn(n) { number of windows }
  ; for j := 1 to n do with w[j] do
      ReadLn(xl, yl, xh, yh) { coordinates of the window }

    { Read and process the mouse clicks }
  ; ReadLn(m) { number of mouse clicks }
  ; for c := 1 to m do begin
      ReadLn(x, y) { coordinates of the mouse click }

      { Find first window that contains (x,y)
        using a Bounded Linear Search }
    ; k := 1 ; j := n + 1
    ; while k <> j do with w[k] do
        if (xl <= x) and (x <= xh) and
           (yl <= y) and (y <= yh) then j := k
```

A Monolithic Solution (using an array)

```
        else k := k + 1
      { k = j, hence if k = n+1 then not found,
        otherwise w[k] is first window containing (x,y) }

      { Output the result }
    ; if k = n + 1 then WriteLn(’Desktop’)
      else WriteLn(k)
    end { for c }

  end { for i }
end.
```

Quality Assessment (Red needs improvement)

- Comments: at top, variable declarations, statement groups
- Variables: sensible names, single purpose
- Layout: indentation, whitespace, empty lines; **begin ... end**
- Constants: explicitly named
- Protection: check input for validity
- Modular structure: explicit

Why Modular Structure?

- Correct by design
- Construction by a team
- Verification
- Adaptation
- Reuse

How to avoid, detect, locate, repair errors . . .

A Monolithic Solution (adapted to use a linked list)

```
program Windows2001m2;
  { Monolithic version using a linked list to store the windows }

type                                                           (*ADDED*)
  NodeP = ^Node; { pointer to a node }                         (*ADDED*)
  Node = record { node in linked list of windows }             (*ADDED*)
    xl, yl, xh, yh: Integer; { coordinates of window }         (*ADDED*)
    tail: NodeP; { pointer to next window, if not nil }        (*ADDED*)
    end;                                                       (*ADDED*)

var
  r: Integer;     { number of runs (input) }
  i: Integer;     { to count off the runs }
  n: Integer;     { number of windows (input) }
  w: NodeP;       { list of windows (input) }                (*CHANGED*)
  u, v: NodeP;    { to construct and traverse list w }         (*ADDED*)
  j: Integer;     { to count off the windows }
  m: Integer;     { number of mouse clicks (input) }
```

A Monolithic Solution (adapted to use a linked list)

```
  c: Integer;     { to count off the mouse clicks }
  x, y: Integer; { coordinates of a mouse click (input) }
  k: Integer;     { window number (output) }

begin
  ReadLn(r) { number of runs }
; for i := 1 to r do begin

    { Read and store the windows }
    ReadLn(n) { number of windows }
  ; w := nil ; u := nil                                     (*ADDED*)
    { inv: u^ is last window in list w, if u <> nil }       (*ADDED*)
  ; for j := 1 to n do begin                              (*CHANGED*)
      New(v) { create new node }                            (*ADDED*)
    ; with v^ do begin                                      (*ADDED*)
        ReadLn(xl, yl, xh, yh) { coordinates of window }
      ; tail := nil                                         (*ADDED*)
      end { with v^ }                                       (*ADDED*)
```

A Monolithic Solution (adapted to use a linked list)

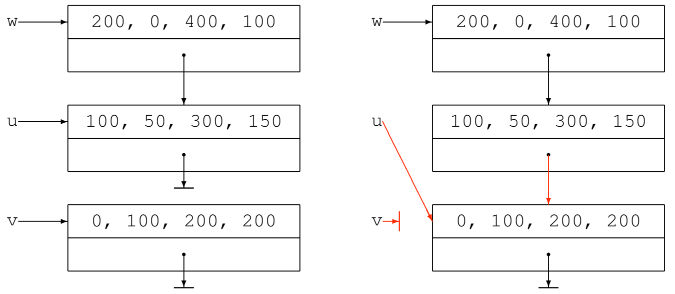
```
    ; if u = nil then w := v else u^.tail := v              (*ADDED*)
    ; u := v ; v := nil                                     (*ADDED*)
    end { for j }                                           (*ADDED*)

    { Read and process the mouse clicks }
  ; ReadLn(m) { number of mouse clicks }
  ; for c := 1 to m do begin
      ReadLn(x, y) { coordinates of the mouse click }

      { Find first window that contains (x,y)
        using a Bounded Linear Search }
    ; u := w ; v := nil ; k := 1                          (*CHANGED*)
    ; while u <> v do with u^ do                          (*CHANGED*)
        if (xl <= x) and (x <= xh) and
           (yl <= y) and (y <= yh) then v := u            (*CHANGED*)
        else begin u := tail ; k := k + 1 end             (*CHANGED*)
      { u = v, hence if u = nil then not found, }         (*CHANGED*)
      { otherwise u^ is first window containing (x,y) }(*CHANGED*)
```

A Monolithic Solution (adapted to use a linked list)

```

   { Output the result }
  ; if u = nil then WriteLn('Desktop')                   (*CHANGED*)
   else WriteLn(k)
  end { for c }

  { Deallocate all windows }                             (*ADDED*)
 ; while w <> nil do begin                               (*ADDED*)
   v := w^.tail ; Dispose(w) ; w := v                    (*ADDED*)
  end { while }                                          (*ADDED*)

 end { for i }
end.
```

Linked List with Pointers: Recursive Definition

```
NodeP = ^Node; { pointer to a node }                          (*ADDED*)
Node = record { node in linked list of windows }              (*ADDED*)
 xl, yl, xh, yh: Integer; { coordinates of window }           (*ADDED*)
 tail: NodeP; { pointer to next window, if not nil }          (*ADDED*)
 end;                                                         (*ADDED*)
```



A Solution Structured with Routines

```
program Windows2001s1;
 { Mildly structured version using an array to store the windows }

const
 MaxWindowListLength = 10000; { maximum length of a WindowList }

type
 Window = record
  xl, yl, xh, yh: Integer; { lower-left and upper-right corners }
  { invariant: xl <= xh  /\  yl <= yh }
  end;

 WindowList = record
  length: 0..MaxWindowListLength; { number of windows in the list }
  item: array [1..MaxWindowListLength] of Window;
   { item[1..length] = list of windows }
  end;

```

A Solution Structured with Routines

```
function InWindow(const w: Window; x, y: Integer): Boolean;
 { pre: true
  ret: whether w contains point (x, y) }
 begin
  with w do begin
   Result := (xl <= x) and (x <= xh) and
             (yl <= y) and (y <= yh)
  end { with w }
 end; { InWindow }

procedure ReadWindowList(out aWindowList: WindowList);
 { pre: input contains a properly formatted list of windows
  post: aWindowList has been read from input }
 { N.B.: uses global variable input }
 var
  iWindow: Integer; { to count off the windows }
 begin
  with aWindowList do begin
```

A Solution Structured with Routines

```
      ReadLn(length)

    ; for iWindow := 1 to length do begin
        with item[iWindow] do begin
          ReadLn(xl, yl, xh, yh)
        end { with }
      end { for iWindow }

    end { with aWindowList }
  end; { ReadWindowList }

function FindFirstWindow(const aWindowList: WindowList;
                         x, y: Integer): Integer;
  { pre: true
    ret: k such that
           1 <= k <= length + 1,
           for all i with 1 <= i < k: (x,y) not in item[i], and
           if k <= length then item[k] contains (x, y)
```

A Solution Structured with Routines

```
         where length and item[_] are from aWindowList }
  var
    k, u: Integer; { to traverse aWindowList }
  begin
    with aWindowList do begin
      { Bounded Linear Search }
      k := 1 ; u := length + 1

    ; while k <> u do begin
        if InWindow(item[k], x, y) then u := k
        else Inc(k)
      end { while }
      { k = u, hence if k = length+1 then not found,
        otherwise item[k] contains (x,y) }

    end { with aWindowList }
  ; Result := k
  end; { FindFirstWindow }
```

A Solution Structured with Routines

```
procedure ProcessAllMouseClicks(const aWindowList: WindowList);
  { pre: input contains a properly formatted list of mouse clicks
    post: the mouse clicks have been read from input and
          the corresponding output has been written }
  { N.B.: uses global variables input and output }
  var
    nMouseClicks: Integer; { number of mouse clicks (input) }
    iMouseClick: Integer; { to count off the mouse clicks }
    x, y: Integer; { coordinates of a mouse click (input) }
    k: Integer; { window number (output) }
  begin
    ReadLn(nMouseClicks)

  ; for iMouseClick := 1 to nMouseClicks do begin
      ReadLn(x, y)
    ; k := FindFirstWindow(aWindowList, x, y)
    ; if k <= aWindowList.length then WriteLn(k)
```

A Solution Structured with Routines

```
      else WriteLn(’Desktop’)
    end { for iMouseClick }

  end; { ProcessAllMouseClicks }

procedure ProcessOneRun;
  { pre: input contains a properly formatted run
    post: the run has been read from input and
          the corresponding output has been written }
  { N.B.: uses global variables input and output }
  var
    theWindowList: WindowList; { list of windows (input) }
  begin
    ReadWindowList(theWindowList)
  ; ProcessAllMouseClicks(theWindowList)
  end; { ProcessOneRun }

{ main program }
```

A Solution Structured with Routines

```
var
  nRuns: Integer; { number of runs (input) }
  iRun: Integer; { to count off the runs }

begin
  ReadLn(nRuns)

; for iRun := 1 to nRuns do begin
    ProcessOneRun
  end { for iRun }

end.
```

Parameter Kinds in Pascal

```
function InWindow(const w: Window; x, y: Integer): Boolean;

procedure ReadWindowList(out aWindowList: WindowList);
```

- value parameter: local variable initialized (copied) at call time
- **out** parameter: body can set variable supplied at call time
- **var** parameter: body can update variable supplied at call time
- **const** parameter: no copy; body cannot change it

Divide and Conquer through Contracts

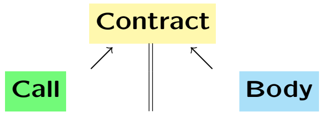
```
procedure ReadWindowList(out aWindowList: WindowList);
  { pre: input contains a properly formatted list of windows
    post: aWindowList has been read from input }
  { N.B.: uses global variable input }
```

		Contract	
		Precondition	Postcondition
	User	obligation	benefit
Party		↓	↑
	Maker	benefit →	obligation

The Contract as the (Only) Interface

```
function InWindow(const w: Window; x, y: Integer): Boolean;
  { pre: true
    ret: whether w contains point (x, y) }
  begin
    with w do begin
      Result := (xl <= x) and (x <= xh) and
                (yl <= y) and (y <= yh)
    end { with w }
  end; { InWindow }
```

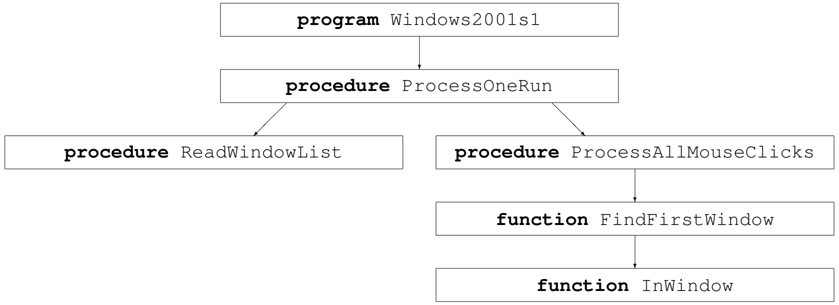
```
        if InWindow(item[k], x, y) then u := k
```



Software Architecture: Key Ingredients

- Building blocks (the units of design, construction, verification, adaptation, reuse)
  - Variables and statements
  - Routines (functions, procedures, with parameters)
  - Data types (classes, with methods)
  - Object Pascal ‘units’, components, packages, . . .
- Relationships between building blocks (dependencies, coupling)

Software Architecture: Static Call Graph



program Windows2001s1

procedure ProcessOneRun

procedure ReadWindowList

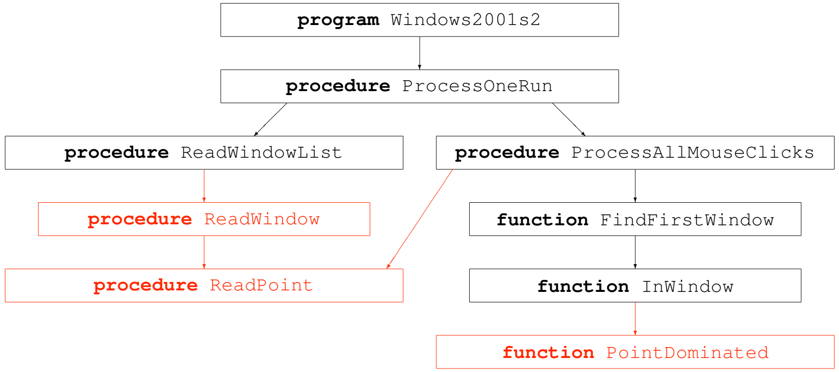
procedure ProcessAllMouseClicks

function FindFirstWindow

function InWindow

Local versus global variables

Alternative Architecture: Decomposition with Points



program Windows2001s2

procedure ProcessOneRun

procedure ReadWindowList

procedure ProcessAllMouseClicks

procedure ReadWindow

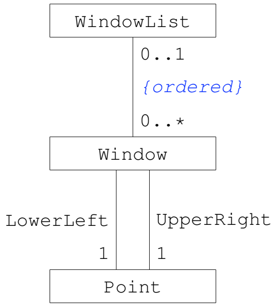
function FindFirstWindow

procedure ReadPoint

function InWindow

function PointDominated

Architecture on the Basis of Data Types



WindowList

0..1

{ordered}

0..*

Window

LowerLeft

UpperRight

1

1

Point

Interface styles

- Interface coordinates calls of routines: ‘control flow’
- Console Interface: **Program-driven** control flow

  Interactive (via prompting)

  batch (via command-line arguments and files)
- Graphical User Interface (GUI): **Event-driven** control flow

  Events: Keyboard, mouse, buttons, menus, sliders, . . .

  ‘Main event loop’: wait for ‘event’ and call its handler (routine)

GUI building blocks

Lazarus Component Library (LCL):

- Windows, . . .
- Buttons, . . .
- Text areas, . . .
- Menus, . . .
- Graphical objects, . . .

Object-oriented programming

- Class: a collection of data items and associated operations

  type for a variabele, compare to `Integer`

  fields, methods, properties
- Object: an instance of a class

  value of a variabele, compare to a concrete number, like `5`
- Hierarchy: relationships bedtween classes via inheritance

  TComponent ⟵ TControl ⟵ . . . ⟵ TButton

Constructing GUI programs

Programming versus Configuring

Visual ‘Programming’: construct a program by ‘clicking it together’

Convenient Facilities

- **procedure** *ShowMessage* (**const** *Msg*: *String*);

  Shows dialog window with *Msg*; execution waits for OK-click

- **function** *IntToStr* (*Value*: *Integer*): *String*;

  Converts *Value* to a *String*

- **function** *StrToInt* (**const** *s*: *String*): *Integer*;

  Converts *s* to an *Integer*

- **function** *TryStrToInt* (**const** *s*: *String*; **out** *i*: *Integer*): *Boolean*;

- **function** *Val* (**const** *s*: *String*; **out** *V*; **out** Code: *Word*);

Convenient Facilities

- **procedure** *Close*; { in *Form* }

  Closes the form (in main form: terminates program execution)

- To allow a menu on a Form : `BorderStyle := bsSizeable`

Files in a Lazarus GUI program

What you should save and submit of a Lazarus GUI program:

- `*.lpi`: Lazarus Project Information (XML)
- `*.lpr`: Lazarus Program (Pascal code)
- `*.pas`: Lazarus Unit (Pascal code)
- `*.lfm`: Lazarus Form (Object configuration data in text)
- `*.lrs`: Lazarus Resources (binary data)

Other files contain only secundary information

What Lies Ahead

- Abstract Data Types and object-oriented programming
- Dynamic variables and pointers
- Recursion, both in control and in data
- Contract checking through `Assert`
- Event-driven, interactive Graphical User Interfaces