Programming – Block C

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

Lecture 14

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Feedback to T.Verhoeff@TUE.NL

Today's Topics

- Dynamic variables: heap, pointer type ^T, New, Dispose

  Informatics Thriller: *My Life among the Pointers*
- Linear dynamic data structures with pointers: Linked Lists
- A first encounter with recursion

Static Memory Allocation and Deallocation

```
1 var
2   A : VeryBigTypeA;
3   B : VeryBigTypeB;
4
5 begin { A, B allocated }
6   'Use A'
7 ; 'Use B'
8 end { A, B deallocated }
```

```
1 procedure UseA;
2   var A : VeryBigTypeA;
3   begin { A allocated }
4     'Use A'
5   end; { A deallocated }
6
7 procedure UseB;
8   var B : VeryBigTypeB;
9   begin { B allocated }
10    'Use B'
11  end; { B deallocated }
12
13 begin UseA ; UseB end
```


Pascal Pointer Type (informal)

```
1 type
2   THistogram = array [ Char ] of Cardinal;
3
4 var
5   v: THistogram; { a histogram }
6   p: ^ THistogram; { a pointer to a histogram }
```

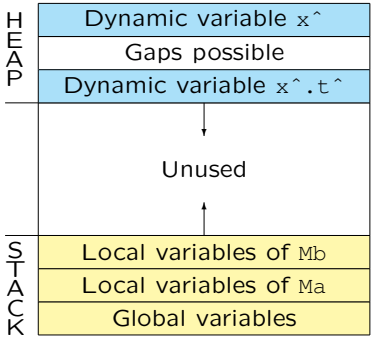
p is a pointer to (reference to, address of) a THistogram variable.

If p <> **nil**, then p^ is the THistogram variable pointed to by p.

Dynamic Memory Allocation and Deallocation

```
1  var
2    p : ^ VeryBigTypeA; { pointer }
3    q : ^ VeryBigTypeB; { pointer }
4
5  begin                 { p, q allocated }
6    New(p)              { p^ allocated }
7  ; 'Use p^'
8  ; Dispose(p)          { p^ deallocated }
9
10 ; New(q)              { q^ allocated }
11 ; 'Use q^'
12 ; Dispose(q)          { q^ deallocated }
13 end                   { p, q deallocated }
```


Memory Organization: Stack and Heap



Pascal Pointer Type

```
type
  PT = ^T; { pointer type for type T, T must be a type name }
```

Set of values for `^T`: pointers to *dynamic variables* of type `T`, including the special pointer **`nil`**, pointing to *nothing*

Dynamic variables exist on the heap, which grows toward the stack.

Memory management of the heap is up to the program (not the compiler).

N.B. Static variables exist on the stack, managed by the compiler, based on block activations.

Pascal Pointer Type Operations

Operations on `^T` (`p`, `q` expressions of type `^T`)

- `p^` (dereferencing): **pre**: `p <>` **`nil`**
  ret: the *variable* of type `T` pointed to by `p`
- `p := q` (assignment): **pre**: `q = Q`
  post: `p = Q` (N.B. aliasing!)
- `New(p)`: **pre**: true
  post: `p` points to newly allocated, uninitialized dynamic variable of type `T`
- `Dispose(p)`: **pre**: `p = P /\ P <>` **`nil`**
  post: variable `P^` has been deallocated, `p` is undefined

Example of Basic Pointer Usage: `program PointerExample;`

```
4  var
5    p, q: ^Integer; // pointers to dynamic Integer variable
6
7  begin
8    Writeln ( 'Used on heap: ', GetHeapStatus.TotalAllocated )
9  ; Writeln ( 'SizeOf ( Integer ) = ', SizeOf ( Integer ) )
10 ; New ( p ) // p^ points to a (new) dynamic Integer variable
11 ; Writeln ( 'Used on heap: ', GetHeapStatus.TotalAllocated )
12 ; q := p     (* ALIASES: different names for same thing *)
13 ; p^ := 37
14 ; Writeln ( 'p^, q^ = ', p^, ', ', q^ )
15 ; Dispose ( q )  // the dynamic variable no longer exists
16 ; Writeln ( 'Used on heap: ', GetHeapStatus.TotalAllocated )
17 end.
```


Example of Basic Pointer Usage: Output

```
Used on heap: 32
SizeOf ( Integer ) = 4
Used on heap: 48
p^, q^ = 37, 37
Used on heap: 32
```

Storage overhead for `p^`: $48 - 32 - 4 = 12$ bytes

This is needed for memory management administration.

Costs of Pointers

Memory usage for `var p: ^T`:

- `p = nil`: $\mathcal{O}(1)$, generally 4 bytes
- `p <> nil`: $\mathcal{O}(1)$ + memory usage for `p^` (except when *sharing*)

Speed of operations:

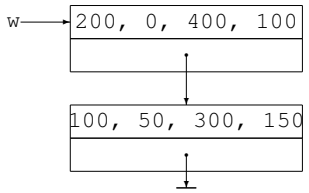
- `nil`, `p = q`, `p := q`: $\mathcal{O}(1)$
- `New(p)`, `Dispose(p)`: depends on heap manager

Embedding and Nesting with Pointers

```
1  var
2    p: ^T  { plain pointer }
3    a: array [ Char ] of ^T  { embedded pointers }
4
5  type
6    R = record p: ^T end;  { embedded pointer }
7
8  var
9    q: ^R  { nested pointers }
10
11 begin
12   ... p^ ...        { is of type T }
13   ... a['c']^ ...   { is of type T }
14   ... q^.p^  ...    { is of type T }
```

Recursion with Pointers: Linked List

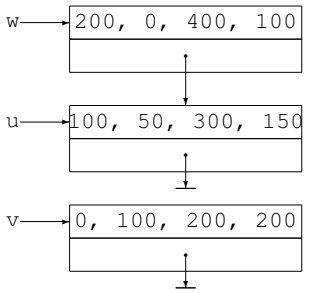
```
4  type                                                    (*ADDED*)
5    NodeP = ^Node; { pointer to a node }                  (*ADDED*)
6    Node = record { node in linked list of windows }      (*ADDED*)
7      xl, yl, xh, yh: Integer; { coordinates of window }  (*ADDED*)
8      tail: NodeP; { pointer to next window, if not nil } (*ADDED*)
9      end;                                                (*ADDED*)
10
11 var
15   w: NodeP;       { list of windows (input) }           (*CHANGED*)
```

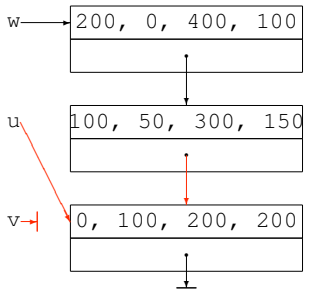



```
w^.xl = 200        w^.xh = 400
w^.yl = 0          w^.yh = 100

w^.tail^.xl = 100  ...xh = 300
w^.tail^.yl = 50   ...yh = 150

        w^.tail^.tail = nil
```


Appending to the End of a Linked List

```
15   w: NodeP;       { list of windows (input) }           (*CHANGED*)
16   u, v: NodeP;    { to construct and traverse list w }  (*ADDED*)

37     ; if u = nil then w := v else u^.tail := v          (*ADDED*)
38     ; u := v ; v := nil                                 (*ADDED*)
```





Traversing a Linked List

```
1  function Count(p: NodeP): Integer;
2    // pre: p points to nil-terminated tail-linked list
3    // ret: length of list pointed to by p
4    begin
5      Result := 0
6
7      // inv: Result + length(list(p)) = length(list(initial p))
8      //      /\  p is tail-reachable from initial p
9    ; while p <> nil do begin
10       Result := Result + 1
11     ; p := p^.tail
12     end
13     // Result = length(list((initial p))
14
15   end; { Count }
```


Traversing a Linked List (Recursively)

```
1  function Count(p: NodeP): Integer;
2    // pre: p points to nil-terminated tail-linked list
3    // ret: length of list pointed to by p
4    begin
5      if p = nil then begin
6        Result := 0
7      end
8      else begin // p <> nil
9        Result := 1 + Count ( p^.tail )
10     end
11     // Result = length of (initial p)
12   end; { Count }
```

Disadvantage: Uses one stack frame per Node counted

Find Last Node of a Non-Empty Linked List

```
1  function FindLast(p: NodeP): NodeP;
2    // pre: p <> nil points to nil-terminated tail-linked list
3    // ret: pointer to last element in list
4    begin
5      Result := p
6
7      // inv: Result <> nil  /\  Result is reachable from p
8    ; while Result^.tail <> nil do begin
9        Result := Result^.tail
10     end
11     // Result^.tail = nil  /\  Result is reachable from p
12
13   end; { FindLast }
```


Find Last Node of a Non-Empty Linked List (Recursively)

```
1  function FindLast(p: NodeP): NodeP;
2    // pre: p <> nil points to nil-terminated tail-linked list
3    // ret: pointer to last element in list
4    begin
5      if p^.tail = nil do begin
6        Result := p
7      end
8      else begin // p^.tail <> nil
9        Result := FindLast ( p^.tail )
10     end
11     // Result^.tail = nil  /\  Result is reachable from p
12   end; { FindLast }
```


Bounded Linear Search on a Linked List

```
1  function Find(p: NodeP; ...): NodeP;
2    // pre: p points to nil-terminated tail-linked list
3    // ret: pointer to first element in list satisfying ...,
4    //      if such an element occurs in the list, else nil
5    begin
6      Result := nil
7
8    ; while p <> Result do begin { p <> nil }
9        if ... p^ ... then  { p^ satisfies search condition }
10         Result := p
11       else
12         p := p^.tail
13     end
14
15   end; { Find }
```


Bounded Linear Search on a Linked List (Recursively)

```
1  function Find(p: NodeP; ...): NodeP;
2    // pre: p points to nil-terminated tail-linked list
3    // ret: pointer to first element in list satisfying ...,
4    //      if such an element occurs in the list, else nil
5    begin
6      if p = nil then begin
7        Result := nil
8      end
9      else begin // p <> nil
10       if ... p^ ...  // p^ satisfies search condition
11       then Result := p
12       else Result := Find(p^.tail, ...)
13     end
14   end; { Find }
```

‘Cons’ a New Element to a Linked List

```
1  procedure Cons(var p: NodeP; const v: TValue);
2    // pre: p points to nil-terminated tail-linked list
3    // post: p points to list with v added in front
4    var
5      q: NodeP; { to create new node }
6    begin
7      New(q)
8    ; q^.value := v
9    ; q^.tail := p
10   ; p := q
11   end; { Cons }
```


‘Cons’ a New Element to a Linked List (2)

```
1  procedure Cons(var p: NodeP; const v: TValue);
2    // pre: p points to nil-terminated tail-linked list
3    // post: p points to list with v added in front
4    var
5      q: NodeP; { to save initial value of p }
6    begin
7      q := p
8    ; New(p)
9    ; p^.value := v
10   ; p^.tail := q
11   end; { Cons }
```


Delete First Element from a Linked List

```
1  procedure DeleteFirst(var p: NodeP);
2    // pre: p <> nil points to nil-terminated tail-linked list
3    // post: p points to list minus first element
4    var
5      q: NodeP; { to save p^.tail }
6    begin
7      q := p^.tail  (* p := p^.tail may cause memory leak *)
8      { if necessary, dispose p^.value here }
9    ; Dispose(p)  (* can't do this first, because then p undefined
10   ; p := q
11   end; { DeleteFirst }
```


Delete First Element from a Linked List (2)

```
1  procedure DeleteFirst(var p: NodeP);
2    // pre: p <> nil points to nil-terminated tail-linked list
3    // post: p points to list minus first element
4    var
5      q: NodeP; { to save initial value of p }
6    begin
7      q := p
8      { if necessary, dispose p^.value here }
9    ; p := p^.tail
10   ; Dispose(q)
11   end; { DeleteFirst }
```

Complications with Pointers

Memory management: explicit burden of program

Embedding, nesting, recursion: Pointers can occur *inside* a type

Memory sharing: via aliasing (`p^ = q^`)

Memory leak: after `p := q` and `New(p)`, `(old p)^` is possibly still allocated but *unreachable*

Dangling pointers: after `Dispose(p)`, other pointers could still point to `(old p)^`

Memory fragmentation: `New(p) ; New(q) ; Dispose(p)`

Common Pointer Errors

- Confuse `p` and `p^`: e.g. `p.tail` instead of `p^.tail`
- Dereference the **`nil`** pointer: `p^` when `p = nil`
- Forget to allocate `p^` with `New(p)` before using `p^`
- Forget to deallocate `p^` with `Dispose(p)` when finished with `p^`

  E.g. when using a function result as value parameter
- Access `p^` after deallocation (possibly via an alias)
- Lose access to a dynamic variable before deallocation

The Philosopher's Stone (Direct Recursion)

Consider the following specification and implementation of `Miracle`:

```
1  procedure Miracle;
2    { pre:  True
3      post: False }
4  begin
5    { True }
6    Miracle  // OK according to the contract above
7    { False }
8  end;
```

For *all* predicates Q and R, a call to `Miracle` satisfies:

`{ Q }` `Miracle` `{ R }`

What is wrong? How to avoid this? Forbid a routine to call itself?

Holy Grail (Mutual Recursion)

```
1  procedure Holy;
2    { pre:  True
3      post: False }
4
5    procedure Grail;
6      { pre:  True
7        post: False }
8      begin
9        Holy
10     end; { Grail }
11
12   begin { Holy }
13     Grail
14   end; { Holy }
```

```
1  procedure Grail; forward;
2
3  procedure Holy;
4    { pre:  True
5      post: False }
6    begin
7      Grail
8    end; { Holy }
9
10 procedure Grail;
11   { pre:  True
12     post: False }
13   begin
14     Holy
15   end; { Grail }
```

What is wrong? How to avoid this? No routine calls itself (directly)!

Recursion

Static call graph of program P:

- Nodes: routines (incl. main program block) defined in P
- Arrows: $q \rightarrow r$ where body of q contains a call to r

Recursion: Cycle in call graph (*syntactic* phenomenon)

Direct recursion: Cycle of length 1: body contains call to itself

Indirect recursion: Cycle of length > 1

Mutual recursion: Cycle of length 2

Reasoning about Recursion

When reasoning about a program involving recursion, one has an additional obligation:

Explain why the recursion terminates.

Not every call of a recursive routine R may lead to another call of R.

Compare to induction in logic. The induction hypothesis may only be applied to smaller values. The base case(s) must be handled independently.

Typically, this is done by giving a variant function that takes on natural number values and that decreases on every recursive call.

In the case of the earlier routines working on linked lists, the variant function is the length of the list parameter.

"Degenerate" Recursion

```
1 procedure Bogus;
2   { pre:  True
3     post: True
4     vf:   0 }
5 begin
6   if False
7   then Bogus
8 end;
```

Syntactic recursion, but not when running.

```
1 procedure Abs(e: Real; out x: Real);
2  { pre:  e=E
3    post: x=|E|
4    vf:   0 if 0 <= e else 1 }
5 begin
6   if e < 0 then Abs ( -e, x )
7   else x := e
8 end
```

What is wrong with

```
if e <= 0 then Abs ( -e, x )
```


Sum Digits Recursively

```
5  function DigitSum ( n, r: Cardinal ): Cardinal;
6    // pre: 0 <= n  /\  2 <= r
7    // ret: sum of digits of n in radix r notation
8    (* vf:  n *)
9    begin
10     if n = 0 then begin
11       Result := 0
12     end
13     else begin { 2 <= r, 0 < n } (* hence, n div r < n *)
14       Result := n mod r  +  DigitSum ( n div r, r )
15     end
16   end;
```

Variant function is (very generous) upper bound on recursion depth.
N.B. Fails for `r = 1` (excluded by precondition).
Can also be implemented easily with a **while** loop.

Erroneous Recursive Array Summer (Why?)

```
4  function Sum(const a: array of Integer; i, j: Integer): Integer;
5    { pre: 0 <= i <= j <= Length(a)
6      ret: (sum k: i <= k < j: a[k])
7      vf:  j - i }
8  var
9    h: Integer;
10 begin
11   if i = j then begin
12     Result := 0
13   end
14   else begin { i < j }
15     h := (i + j) div 2  { h between i and j }
16   ; Result := Sum ( a, i, h ) + Sum ( a, h, j )
17   end
18 end;
```


Relationship between `while` Loops and (Tail) Recursion

```
while B do S
```

is equivalent to

```
1 procedure WhileDo;
2 begin
3   if B then begin
4     S
5   ; WhileDo
6   end
7 end;
```

Loops (iteration) can be eliminated by using recursion.

Recursion can be eliminated by using loops (may not be so easy), but recursive solutions can be simpler than iterative solutions.

Bit Patterns

All bit patterns of length 3:

```
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

How to generate with a program?

Bit Patterns of Length 3

```
1  procedure GenerateAll3BitPatterns;
2  var
3    b0, b1, b2: 0 .. 1; { the three bits }
4  begin
5    for b2 := 0 to 1 do begin
6      for b1 := 0 to 1 do begin
7        for b0 := 0 to 1 do begin
8          writeln ( b2, b1, b0 )
9        end { for b0 }
10     end { for b1 }
11   end { for b2 }
12 end;
```

How to generate all bit patterns of length N (parameter)?

Variable number of nested **for** loops?

Bit Patterns of Length n: Recursive Routine

```
11 procedure Generate(const s: String; n: Integer);
12   { pre: 0 <= Length(s) <= n  /\  s is a bit pattern
13     post: each n-bit pattern with prefix s has been written to stdout
14     vf: n - Length(s) }
15 var
16   b: 0 .. 1; { one bit }
17 begin
18   if Length(s) = n then begin
19     writeln ( s )
20   end
21   else begin { 0 <= Length(s) < n }
22
23     for b := 0 to 1 do begin
24       Generate ( s + IntToStr(b), n )
25     end { for b }
26
27   end
28 end;
```


Bit Patterns of Length n: Initial Call

```
Generate ( '', 5 )
```

In a way, this involves a variable number of nested **for** loops.

Runtime grows exponentially with recursion depth.
Memory overhead (in stack frames) grows linearly.

Note that the string and integer parameter are repeatedly passed on.

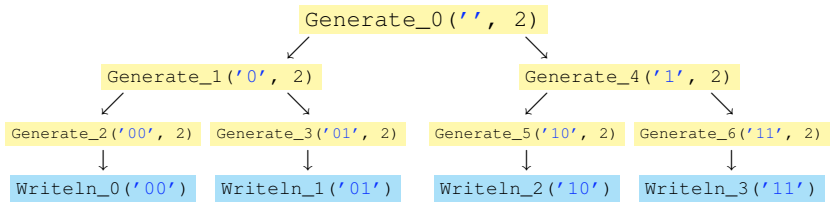
This involves a lot of superfluous copying, which can be suppressed (but that falls outside the scope of this course).

Dynamic Call Tree

Dynamic call tree of an *execution* of program P:

- Nodes: routine *calls* in execution of P.
- Arrows: $q \rightarrow r$ where execution of call q results in execution of call r.

Example for bit patterns with $n = 2$:



What Lies Ahead

- Branching structures with pointers: Binary (Search) Trees
- More recursive functions and procedures