16.070

Introduction to Computers & Programming

Ada: Access types, exception handling

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

Dynamic data structures

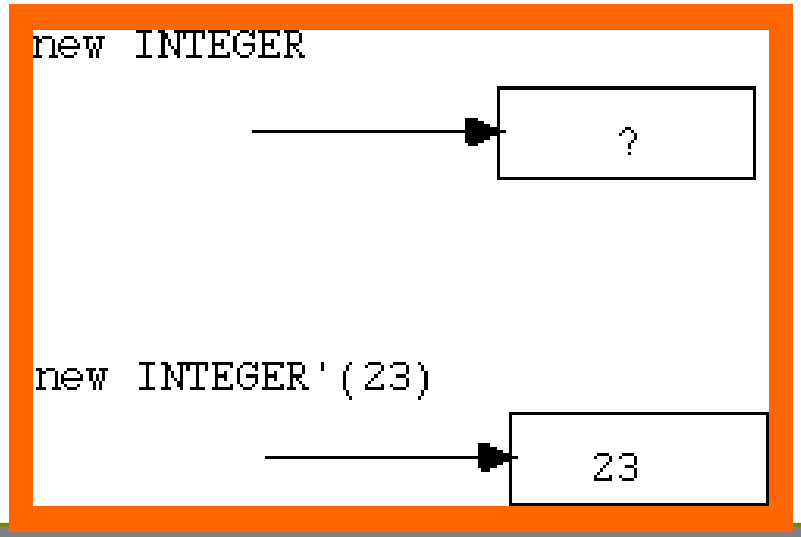
- Motivation
  - How to store:
    - Queue of items?
    - Family tree?
  - Possibilities?
    - array: space may be too much, or not enough
    - direct access file: slow
  - Needed:
    - variable size: ***dynamic*** data structures
    - pointers: ***access*** variables

Static vs. Dynamic

- Static = fixed size
  - array
  - space allocated by compiler
  - space wasted or insufficient
- Dynamic = variable size
  - size starts at zero, changes as necessary
  - space allocated by programmer
  - ***access variables*** needed

Storage allocation

- Allocator
  - new T
    - T is arbitrary data type
    - memory is *allocated* for an object of type T
    - pointer to that memory is returned
  - **new T'(value)**
    - memory allocated as above
    - initial value stored in that memory
    - pointer to the memory is returned
  - examples:

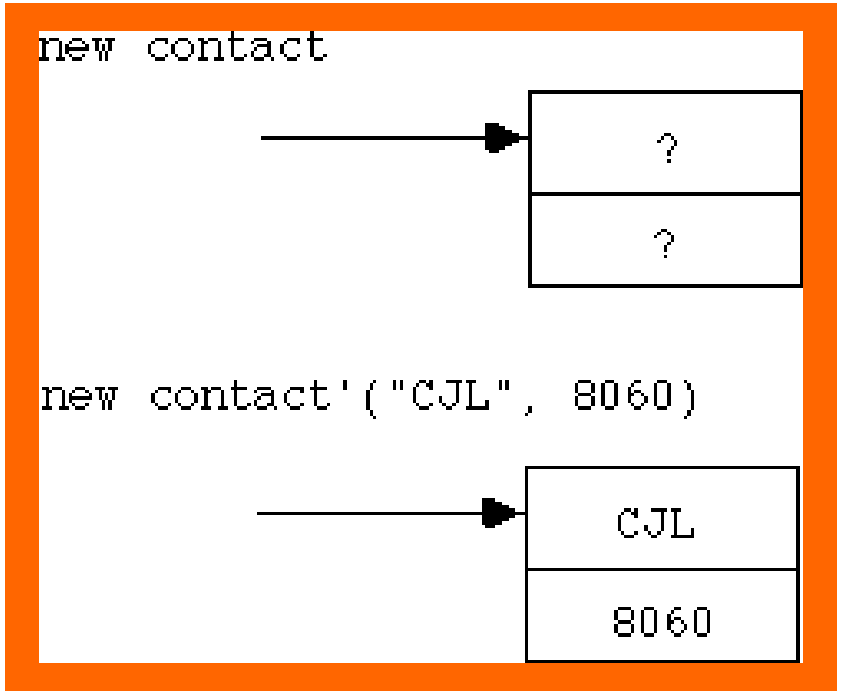


Allocation with records

- data type for dynamic data structures is usually a record

```
type contact is record
   initials  : string(1..3);
   extension : integer;
end;
```

- examples:



Access variables

- Access types
  - Can declare variables of ***access types***

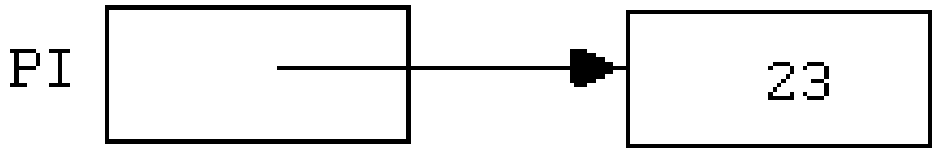
```
type int_pointer     is access INTEGER;
type contact_pointer is access contact;

PI : int_pointer;
PC : contact_pointer;
```

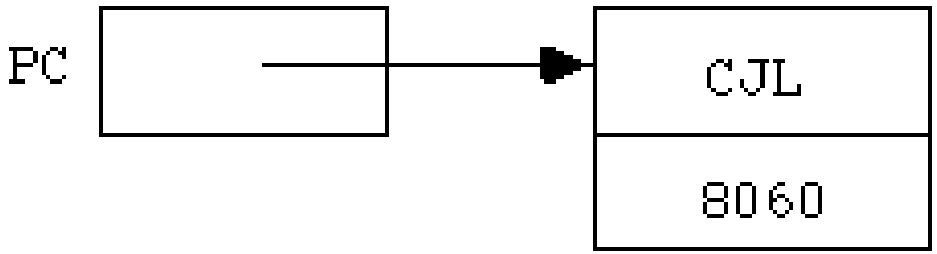
Access variables

- Access variables
  - use *access variables* to save pointers to allocated objects

```
PI := new INTEGER'(23)
```




```
PC := new contact'("CJL", 8060)
```



Access variables

- Access variables are ***pointers***
  - Do ***not*** contain data
  - Contain ***pointer*** to data
- Access variables provide ***indirect*** access to data
- Access variables are initialized by default to **null**
  - **null** = does not point to any data

Accessing an object

- to access the entire object pointed to by P
  - P.all
- (If the object is a record) to access a field:
  - P.fieldname
- (If the object is an array) to access an element:
  - P(i)
- CONSTRAINT_ERROR if P has value **null**
- P is not affected when the object is accessed

Assignment with access variables

- can assign access variables if they access the same type

```
type contact is record
   initials  : string (1..3);
   extension : integer;
end record;

type contact_pointer is access contact;

P1, P2, P3, P4 : contact_pointer;
```

- Trace the effect of these statements:

```
P1           := new contact'("CJL", 8817);
P1.extension := 8060;
P2           := new contact;
P3           := P1;
P2.all       := P1.all;
P3           := null;
P4.extension := 8060;          -- constraint error
P4           := new INTEGER; -- syntax error
```

- Next example compares how the same problem can be solved with and without access types. Thus you can see how the access types are used.

Package defining a record 1(2)

```
-- Culwin p. 468
package pers_pack is
   min_weight   : constant := 0.0;
   max_weight   : constant := 250.0;
   min_height   : constant := 0;
   max_height   : constant := 300;
   max_name_len : constant := 15;

   space : constant character := ' ';

   subtype weights is FLOAT range min_weight..max_weight;
   subtype heights is INTEGER range min_height..max_height;
   subtype name_strings is string (1..max_name_len);

   type person_records is record
      f_name : name_strings := (others => space);
      s_name : name_strings := (others => space);
      weight : weights := 0.0;
      height : heights := 0;
   end record;
```

Package defining a record 2(2)

```
   procedure get_person (a_person  : out person_records);

   procedure show_person (a_person : in person_records);

   function name_less_than
      (this_pers,that_pers : person_records) return boolean;

   function name_equal
      (this_pers,that_pers : person_records) return boolean;

end pers_pack;
```

Example WITHOUT access types 1(2)

```
with pers_pack; use pers_pack;
procedure main is
   person_1 : person_records;
   person_2 : person_records;

   procedure swap_people
      (a_person, another_person : in out person_records)
   is
      temp : person_records;
   begin
      temp := a_person;
      a_person := another_person;
      another_person := temp;
   end;
```

Example WITHOUT access types 2(2)

```
begin
   -- input two records from the user
   get_person (person_1);
   get_person (person_2);

   -- swap if necessary
   if name_less_than (person_1, person_2) then
      swap_people (person_1, person_2);
   end_if;

   -- print out the sorted records
   show_person (person_1);
   show_person (person_2);
end main;
```

Same program USING access types 1(2)

```
-- Culwin p. 469
with pers_pack; use pers_pack;
procedure main is
   type person_ptr is access pers_pack.person_records;

   person_1 : person_ptr := new pers_pack.person_records;
   person_2 : person_ptr := new pers_pack.person_records;

   procedure swap_people
      (a_person, another_person : in out person_ptr)
   is
      temp : person_ptr;
   begin
      temp           := a_person;
      a_person       := another_person;
      another_person := temp;
   end;
```

Same program USING access types 1(2)

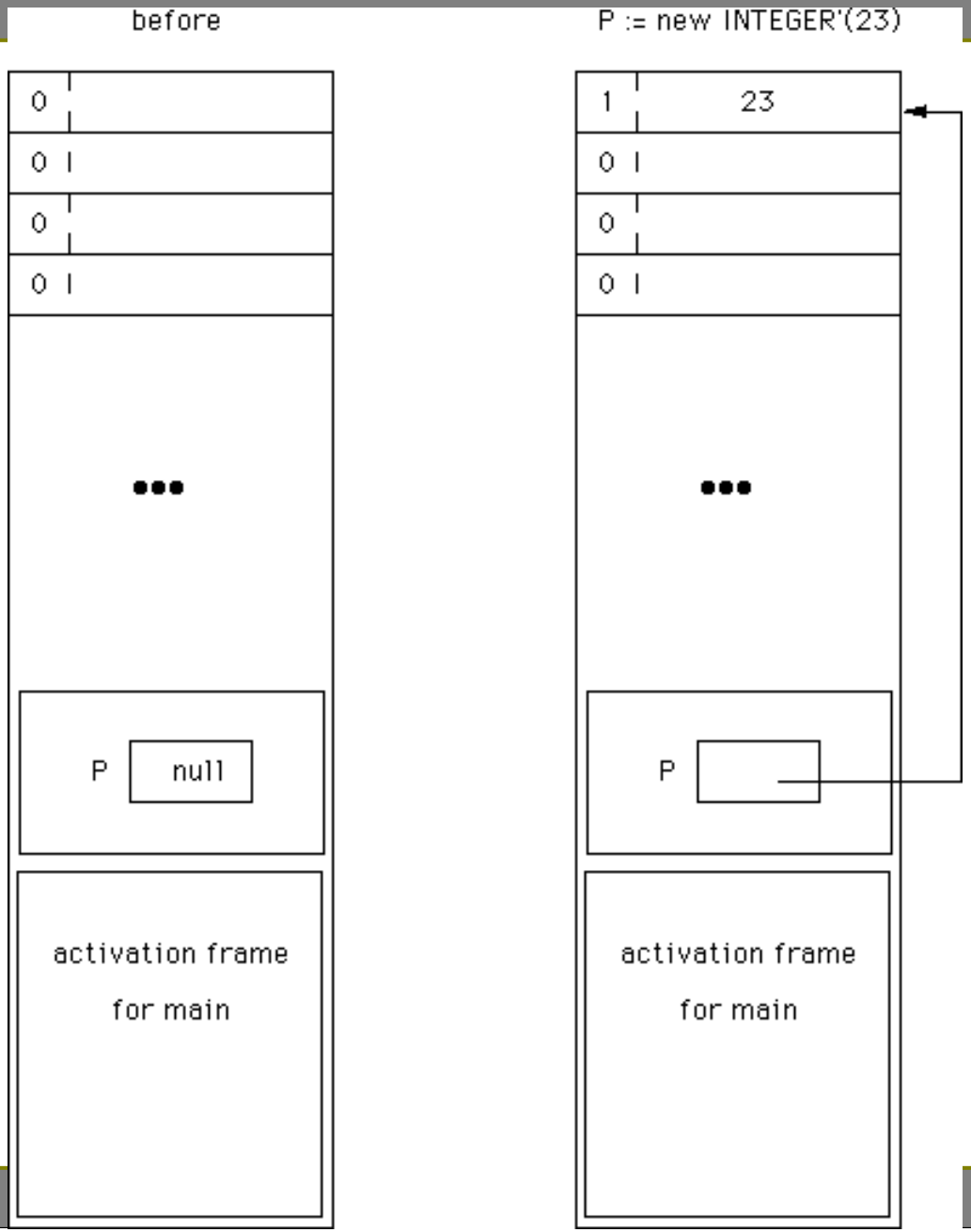
```
begin
   -- input two records from the user
   get_person (person_1.all);
   get_person (person_2.all);

   -- swap if necessary
   if name_less_than (person_1.all, person_2.all) then
      swap_people (person_1, person_2);
   end_if;

   -- print out the sorted records
   show_person (person_1.all);
   show_person (person_2.all);

end main;
```

Dynamic storage allocation

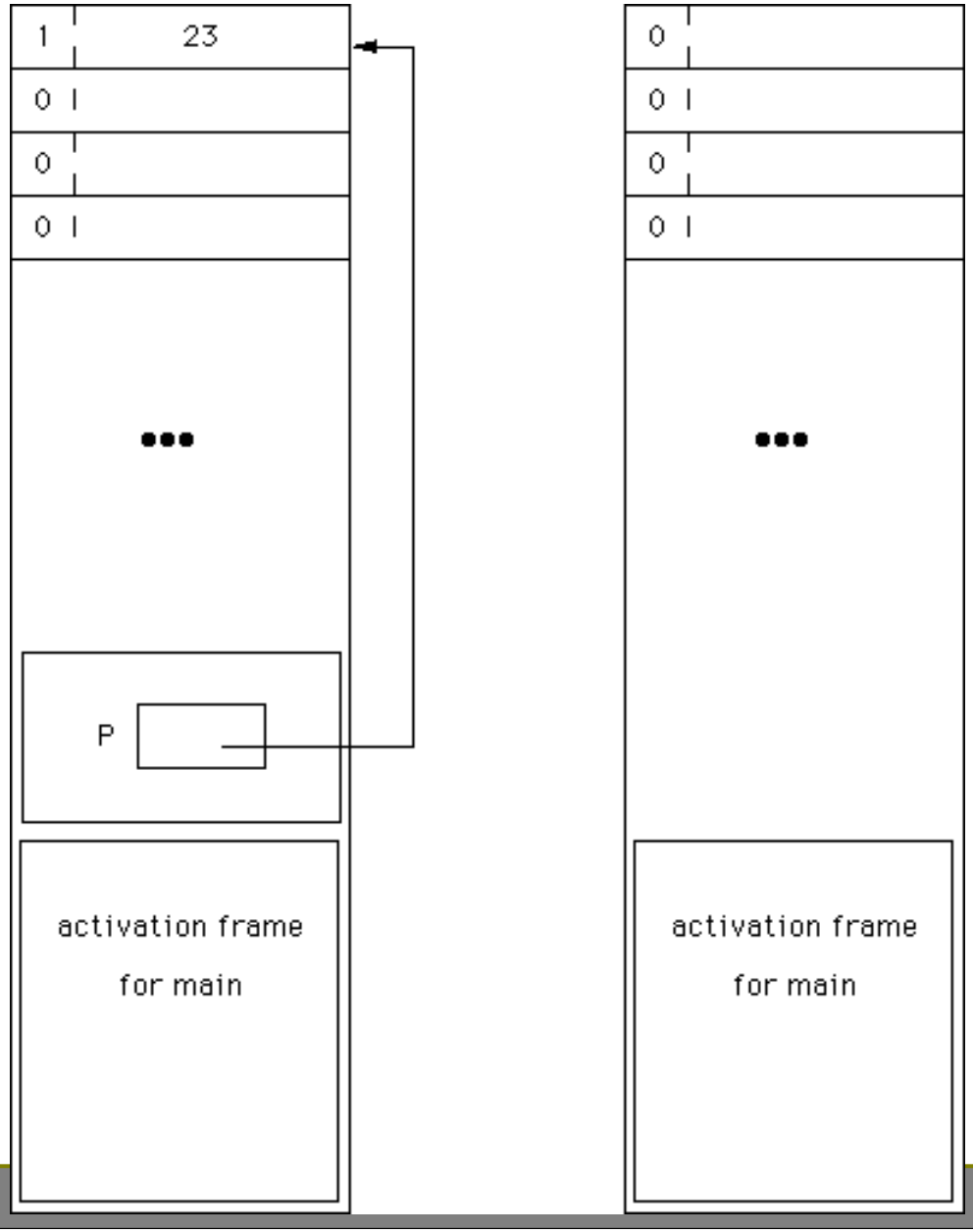


Deallocation of storage

- Storage may be deallocated (released) manually, or you can leave it to the system to do it automatically.
- Automatic on procedure exit
  - safe, simple
  - no ***dangling pointers***
- Programmer can deallocate space manually
  - UNCHECKED_DEALLOCATION
  - programmer deallocates storage
  - care: avoid dangling pointers

Automatic deallocation

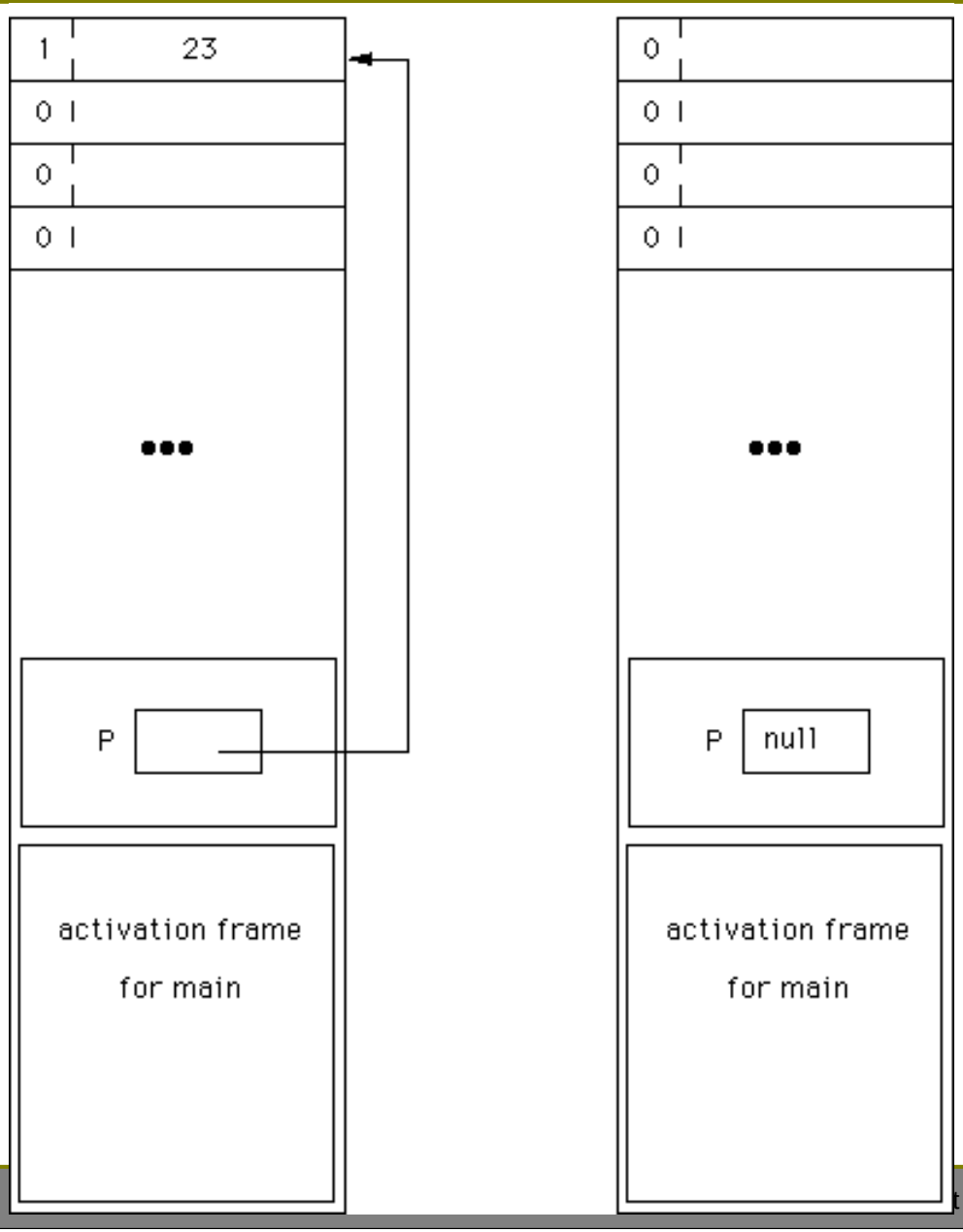
About to exit



After exit

Manual deallocation

before



deallocate_int (P);

Code for manual deallocation

```
with UNCHECKED_DEALLOCATION;
procedure main is

   type T is ... ; -- type definition
   type T_ptr is access T;

   procedure deallocate_T is new
      unchecked_deallocation (T,T_ptr);

   PT : T_ptr;

begin
     ...
   PT := new T; -- allocate from heap
     ...
   deallocate_T (PT); -- return to heap
     ...
end;
```

Linked records

- Dynamic data structures involve records:
  - One or more fields for data
  - One or more fields for access to other records

```
type node;

type link is access node;

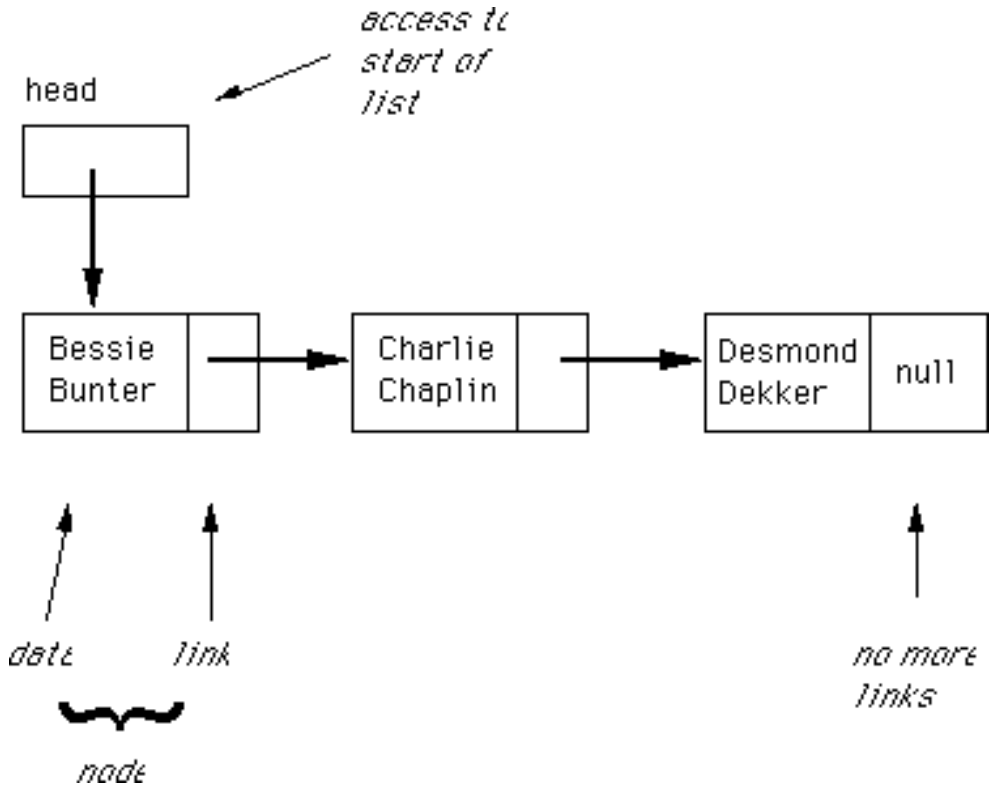
type node is record
   name : string (1 ..15);
   next : link;
end;
```

Linking records

- `P, Q, R : link;`
- Trace the effect of the following statements:
- ```
  P := new node'("Bessie Bunter ", null);
  Q := new node'("Charlie Chaplin", null);
  R := new node'("Desmond Dekker ", null);
  P.next := Q;
  Q.next := R;
  P.next.next := new node;
  ```
- Records are not usually linked using separate pointers like this. Instead, standard methods may be used to build a dynamic data structure: the **linked list**.
  ```

Linked list

- A **linked list** is a sequence of **nodes**
  - Each node is linked to the one following
  - Last one has a **null** link
  - **List head** pointer refers to start of list



Linked list operations

- Standard list operations, regardless of type of data stored
  - Initialize empty list
  - Insert node at beginning of list
  - Insert node at end of list
  - Insert on order
  - Find node that contains given value
  - Print out contents of list
  - Delete record
  - Count nodes

Operations on linked lists

initialize

```
procedure initialize (L  : out link) is
begin
   L := null;
end;
```

add at start

```
new_list := new node'(DATA,null);
new_list.next := list;
list := new_list;

list := new node'(DATA, list);

procedure prepend (DATA : in datatype ; head : in out link) is
begin
   head := new node'(DATA, head);
end;
```

Operations on linked lists

traverse a list

Use a loop:

access variable points to each node in turn
initially set to head of the list
finish when access variable is null

```
P := head;
while P /= null loop
   ... do something with this node
   P := P.next;
end loop;
```

Code like this is **very** common in real programs

Operations on linked lists

Print the list

As an example of code that involves traversing a list, consider printing the contents of the list:

```
procedure print_list (head : in link) is
   p : link;
begin
   p := head;
   while p /= null loop
      put (head.data); new_line:
      print_list (head.next);
   end if;
end;
```

An alternative implementation:

```
procedure print_list (head : in link) is
begin
   if head /= null then
      put (head.data); new_line;
      print_list (head.next);
   end if;
end;
```

Operations on linked lists

Append the list

```
procedure append (data : in datatype; head : in out link) is
   p1, p2 : link;
begin
   if head = null then
      -- list is currently empty
      -- new element becomes the entire list

      head := new node'(data,null);
   else
      -- first we need to find the end of the list
      p1 := head;
      while p1 /= null loop
         p2 := p1;
         p1 := p1.next;
      end loop;

      -- p2 now points to the last element in the list
      -- add new element after p2
      p2.next := new node'(data,null);
   end if;
end append;
```

Linked list ADT

Package specification

```
package list is
   type node;
   type link is access node;

   type node is record
      data : integer;
      next : link;
   end record;

procedure initialise (head : out link);

-- add a node at the start of the list
procedure prepend (data : in integer; head : in out link);

-- add a node at the end of the list
procedure append (data : in integer; head : in out link);

-- print the contents of the list
procedure print_list (head : in link);

-- insert a node in (ascending) sorted order
procedure insert (data : in integer; head : in out link);
-- etc ...
end list;
```

Example program

read, store, sort, display any number of items

```
with list;
with text_io;
program listdemo is

   package int_io is new text_io.integer_io(integer);
   use int_io, text_io, list;

   head : link;                  -- head of a linked list
   num  : integer;               -- a single data item

begin
   list.initialise (head);       -- create an empty list

   while not end_of_file loop
      get (num);                 -- read next data item
      skip_line;
      list.insert (head, num); -- insert in sorted order
   end loop;

   list_print_list (head);       -- print in sorted order
end;
```

Exceptions

- Exception = unexpected situation
  - Program errors:
    - Divide by zero
    - Index out of range
  - Program usage:
    - Faulty input
- Solutions
  - Debug
  - Defensive programming
  - Exception handling
- Checking for all possible errors make code hard to follow
- Ada's **exception handling** mechanism provides a clean way to handle exceptions

Example

```
with Text_Io;

procedure Main is

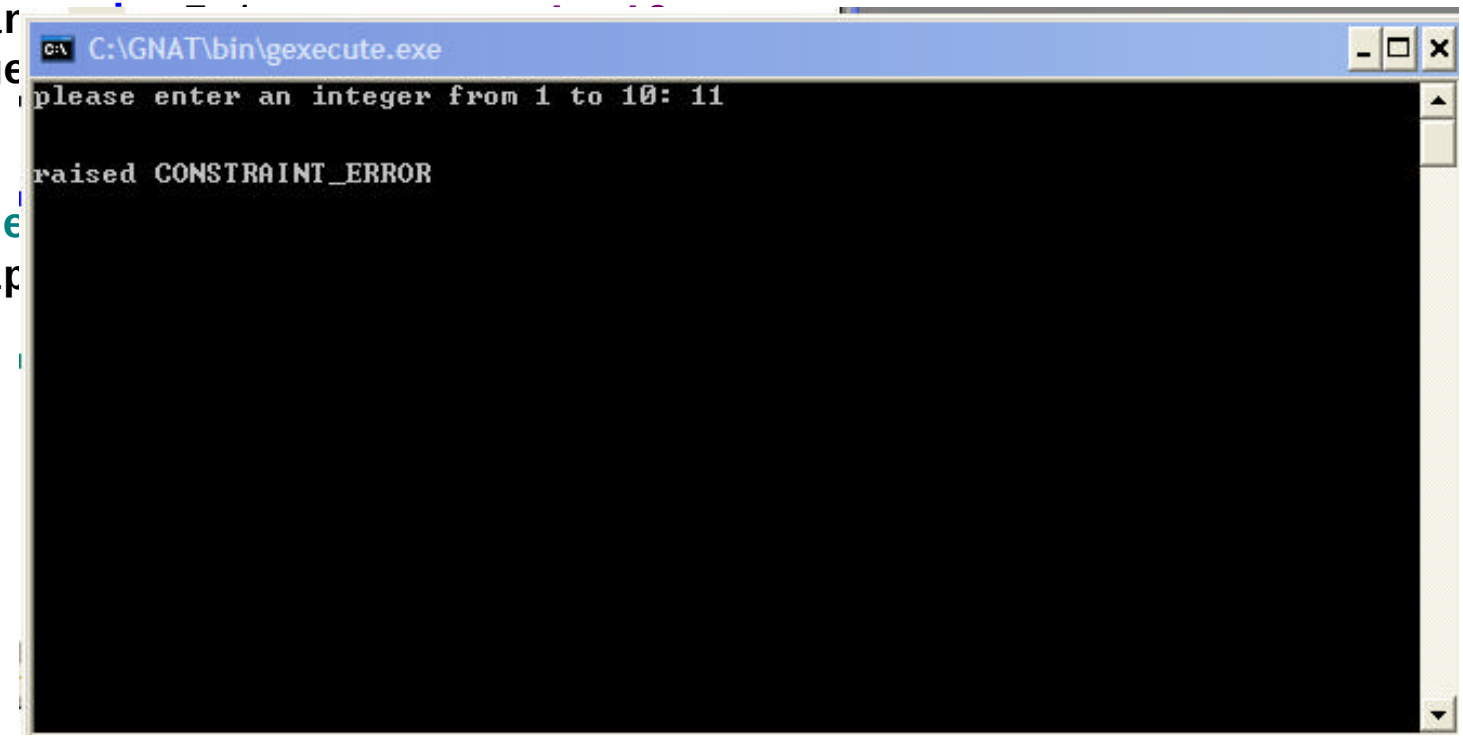
   package Int_Io is new Text_Io.Integer_Io(Integer);
   use Text_Io, Int_Io;

   subtype Numrar
   Num : Numrange

begin --main

   Put ("please e
   Get(Num); Skip

end;
```



When the program is executed, there are a couple of ways that invalid data can be entered.

Exception Handling

- Subprograms with exception handlers
  - General form:

```
subprogram specification
   declarations
begin
   statements
exception
   one or more exception handlers
end;
```

  - Operation:
    - When exception occurs, control jumps to the handler for that exception
    - When handler statements finish, subprogram terminates
    - Control **never** returns to point where exception occurred
    - If no handler, subprogram terminates and exception is passed back to its caller
      - Keep doing this until main reached with no handler (program crashes)
      - Or suitable handler found

Example

```
procedure Open_File (The_File : in out File_Type) is

   Filename : String (1..30);
   Namelen : Natural;

begin

   Put ("what file do you want to read? ");
   Get_Line (Filename, Namelen);
   Open (File => The_File,
      Name => Filename(1..Namelen),
      Mode => In_File);

exception

   when Status_Error =>
      Put_Line ("The file is already open");
   when Name_Error =>
      Put_Line ("There is no file with that name");
   when Use_Error =>
      Put_Line ("The file cannot be read");
   when others =>
      Put_Line ("Unexpected error on opening file");

end Open_File;
```

Block statement

- You can define your own block at any point in an Ada program.
- Its structure is similar to a subprogram:
  - ```
    declare
       declarations
    begin
       normal sequence of statements
    exception
       exception handlers
    end;
    ```
  - The declarations and exception handler are optional.
    ```

Declare block for local variables

```
procedure main is
   x,y : integer;
begin
   statements;

   -- time to swap two variables
   declare
      temp : integer;
   begin
      temp := x;
      x := y;
      y := temp;
   end;

   more statements
end;
```

The local declarations are only known inside the block statement.

Exception in block statements

- The other reason for defining a block statement is to enable local exception handling (especially in a loop).
- Operation:
  - when an exception occurs:
    - execution transfers straight to its exception handler
    - appropriate exception handler is executed
    - execution of the whole block statement terminates
    - execution continues with statement after the block
  - if no local exception handler:
    - block terminates immediately
    - control passes to outer block, to see if it has an appropriate exception handler
    - etc.

Example program

```
--Safe I/O
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
function Get_Day return Days is
   Local_Day  : Days;                  -- entered by user
   Good_Day   : Boolean := False:      -- loop control
begin
   while not Good_Day loop
      begin -- while block
         Put ("Enter a day name (first 3 letters) : ");
         Get (Local_Day);
         -- this point only reached if valid entry given
         Good_Day := True;
      exception -- exception handler for while block
         when Data_Error =>
            Put ("Must be first 3 letters of a day name");
            New_Line;
            Skip_Line;
      end; -- while block
   end loop;
   Skip_Line;
   return Local_Day;
end Get_Day;
```

Strategies for handling exceptions

- Three levels of ambition:
  1. Take control – try to act so program can continue
  2. Identify exception fir handling elsewhere – detect, identify, pass it on
  3. Ignore – program halts (crashes)
- Aim to control an exception in the part of the program where its effect can most sensibly be handled.

1. Take control

- Example: tan(x) may be impossible to compute or represent
- NUMERIC_ERROR exception can be detected and handled
- Handle the exception locally; caller never realises anything was wrong.
  - ```
    function TAN (X : FLOAT) return FLOAT is
    begin
       return SIN(X) / COS(X);
    exception
       when NUMERIC_ERROR =>
          if (SIN(X)>=0.0 and COS(X)>= 0.0) or
             (SIN(X)< 0.0 and COS(X)< 0.0)
          then
             return FLOAT'LAST;
          else
             return -FLOAT'LAST;
          end if;
    end TAN;
    ```
    ```

2. Pass exception back

- **raise** statement in exception handler:
  - perhaps take some action locally
  - identify exception and pass it back to caller

```
function TAN (X : FLOAT) return FLOAT is
begin
   return SIN(X) / COS(X);
exception
   when NUMERIC_ERROR =>
      PUT_LINE ("The value of tangent is too big");
      raise;
end TAN;
```

3. Ignore the exception

- No example is needed of the third level of ambition. We are all familiar with that one! It is what we have been doing all the time up to now.

Language-defined exceptions

- Several built-in exceptions are defined in Ada:

Exception	**Example**
constraint_error	Outside allowed range of values
numeric_error	Cannot return correct numeric result Underflow, overflow, divide by zero
program_error	(unusual) eg., end of function reached with no *return*
storage_error	all memory used up infinite recursion
tasking_error	parallel programs
data_error	Invalid data type (may be uncluded in constraint_error)

Exceptions in Input/Output

- TEXT_IO defines several exceptions:

Exception	**Example**
data_error	invalid data type, data has wrong form
status_error	try to open an already open file
mode_error	try to read from an output file
name_error	no such file
use_error	try to open printer for reading
end_error	EOF encountered while reading
layout_error	SET_COL beyond LINELENGTH limit
device_error	hardware failure

Reading enumeration values

```
--Safe I/O (again)
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
function Get_Day return Days is
   Local_Day  : Days;                  -- entered by user
   Good_Day   : Boolean := False:      -- loop control
begin
   while not Good_Day loop
      begin -- while block
         Put ("Enter a day name (first 3 letters) : ");
         Get (Local_Day);
         -- this point only reached if valid entry given
         Good_Day := True;
      exception -- exception handler for while block
         when Data_Error =>
            Put ("Must be first 3 letters of a day name");
            New_Line;
            Skip_Line;
      end; -- while block
   end loop;
   Skip_Line;
   return Local_Day;
end Get_Day;
```

Reading enumeration values, with range checks 1(3)

- Safe input of enumerated types
  - Using **GET** with enumerated types can cause an error, if an invalid value is entered. Ada lets you catch those errors, by providing an exception handler for the error.
  - The example on next slide shows how you can read a value representing a day of the week from the user and be guaranteed that something appropriate is typed. If something invalid is entered, the procedure reports the error and gives the user another chance.

Reading enumeration values, with range checks

```
   type week_days is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
   subtype work_days is week_days range Mon .. Fri;
   subtype weekend_days is week_days range Sat .. Sun;

procedure safe_get_day (out_day : out week_days;
   min : in week_days := week_days'FIRST;
   max : in week_days := week_days'LAST ) is

-- procedure for the safe input of enumeration values

   local_day : week_days; -- local input var
   good_day : BOOLEAN := FALSE; -- loop control
```

```
begin -- safe_get_day
   while not good_day loop
      begin -- while block
         PUT("Enter an day between ");
         PUT( min ); PUT(" and "); PUT( max ); PUT(" ");
         GET( local_day );
         -- this point is reached only when input is a day code
         if (local_day < min) or else (local_day > max) then
            raise DATA_ERROR;
         else
            good_day := TRUE;
         end if;
         -- this point is reached if input is a valid day code
         -- between min and max
      exception
         when DATA_ERROR => PUT_LINE("Invalid day!. Good days are ");
            for this_day in week_days range min .. max loop
               PUT( this_day ); PUT(" ");
            end loop;
            NEW_LINE; SKIP_LINE; -- tidy up terminal
      end; -- protected while block
   end loop;
   -- this point can only be reached when valid value input
   SKIP_LINE; -- tidy up terminal handling
   out_day := local_day; -- export input value
end safe_get_day;
```

Reading float values, with range checks 1(2)

```
--Safe float I/O
procedure gen_float_input( out_float : out FLOAT;
                          min, max  : in FLOAT) is
   local_float : FLOAT;              -- local input var
   good_one    : BOOLEAN := FALSE;   -- loop control

begin -- gen_float_input
   while not good_one loop
   begin -- protected block of code
      PUT("Enter a float in range ");
      PUT( min, EXP => 0 ); PUT( " to ");
      PUT( max, EXP => 0 ); PUT( " ");
      GET( local_float );
-- this point can only be reached if the get
-- did not raise the exception

-- now tested against limits specified
      good_one:=((local_float>=min) and (local_float<=max));

      if not good_one then
         raise DATA_ERROR;
      end if;
```

Reading float values, with range checks 2(2)

```
   exception
      when DATA_ERROR =>
         PUT_LINE("Invalid input, please try again ");
         SKIP_LINE;
   end; -- protected block of code
   end loop;

   -- this point can only be reached when valid value
input
   SKIP_LINE; -- tidy up terminal handling

   out_float := local_float; -- export input value
end gen_float_input;
```

Opening a file

```
-- safe file opening
procedure open_file (the_file : in out FILE_TYPE) is
   filename : string (1..30);
   namelen : natural;
begin
   PUT ("What file do you want to read? ");
   GET_LINE (filename, namelen);
   OPEN (FILE => the_file,
         NAME => filename(1..namelen),
         MODE => IN_FILE);
exception
   when STATUS_ERROR =>
      PUT_LINE ("The file is already open");
   when NAME_ERROR =>
      PUT_LINE ("There is no file with that name");
   when USE_ERROR =>
      PUT_LINE ("The file cannot be read");
   when others =>
      PUT_LINE ("Unexpected error on opening file");
end open_file;
```

User defined exceptions

- You can declare your own exception types

```
TAN_ERROR : exception;
```

- Example:

```
procedure main is
   X, res    : FLOAT;
   TAN_ERROR : exception;
   function tan (X : FLOAT) return FLOAT is
   begin
      return sin(x)/cos(x);
   exception
      when NUMERIC_ERROR =>
         raise TAN_ERROR;
   end;

begin
   PUT (“Enter a real number X: “); GET (X);
   res := tan(X);
   PUT (“Tan(X) is “); PUT(res); NEW_LINE;
exception
   when TAN_ERROR =>
      PUT_LINE (“The tangent is too big”);
end;
```

Declaring exceptions in packages

- NUMERIC_ERROR may arise in tan(x) function
- Perhaps too unilateral to handle it locally, so prefer to pass an exception back to the caller.
- What to pass back?
  - NUMERIC_ERROR is too general
  - more specific name desirable (eg TAN_ERROR)
- Where to declare TAN_ERROR?
  - not in function TAN (invisible outside)
  - not in calling code (belongs with TAN)
  - best is in a package, along with TAN

Example: User defined exception1

```
--package specification
package TRIGONOMETRY is
   function SIN (X : FLOAT) return FLOAT;
   function COS (X : FLOAT) return FLOAT;
   function TAN (X : FLOAT) return FLOAT;
   TAN_ERROR : exception;
end TRIGONOMETRY;

--package body
package body TRIGONOMETRY is
   function SIN (X : FLOAT) return FLOAT is
      begin ... end SIN;
   function COS (X : FLOAT) return FLOAT is
   begin ... end COS;
   function TAN (X : FLOAT) return FLOAT is
   begin return SIN(X) / COS(X);
   exception
      when NUMERIC_ERROR => raise TAN_ERROR;
   end TAN;
end TRIGONOMETRY;
```

Shows how a package specification can define an exception; the package body can raise that exception when appropriate; and a user program can recognize and handle the exception

Example: User defined exception1

```
--user program
with TEXT_IO, TRIGONOMETRY;
procedure compute_tan is
   package flt_io is new TEXT_IO.FLOAT_IO(FLOAT);
   use TEXT_IO, flt_io, TRIGONOMETRY;

   number, res : FLOAT;

   begin -- compute_tan
      loop
         begin
            PUT ("Enter a real number (or CTRL-D to quit)");
            exit when END_OF_FILE;
            GET (number); SKIP_LINE;
            res := tan(number);
            PUT("Tangent is "); PUT(res); NEW_LINE;

         exception
            when TAN_ERROR =>
               PUT_LINE ("Tangent is too big");
         end;
      end loop;
end compute_tan;
```

Example: User defined exceptions 2

A package to do with vector arithmetic. Some vector operations only make sense if they are performed with vectors of the same length. The package can define an exception LENGTH_ERROR, to be raised if an operation is invoked with vectors of different lengths.

```
--Package specification

package vector_package is
   type vector is array (integer range <>) of float;

   function add           (v1,v2  : vector) return vector;
   function scalar_product (v1,v2 : vector) return float;

   LENGTH_ERROR : exception;
end vector_package;
```

Example: User defined exceptions 2

```
--Package body
package body vector_package is
   function add (v1,v2 : vector) return vector is
   begin ... end add;
   function scalar_product (v1,v2 : vector) return float is
      sum : float := 0.0;
      temp : vector(v1'RANGE);
   begin
      if v1'LENGTH /= v2'LENGTH then
         raise LENGTH_ERROR;
      end if;
      temp := v2;
      for i in v1'RANGE loop
         sum := sum + v1(i)*temp(i);
      end loop;
      return sum;
   end scalar_product;
end vector_package;
```