1: A Crash Course in Ada

```
with Ada.Text_IO;
procedure Hello is
  use Ada.Text_IO;
begin
  Put_Line("Hello Bauhaus-Uni Weimar!");
end Hello;
```

Welcome to Adaland!

At quiet nights one can
hear C programmers debug.

– – Rainer Koschke, Uni Stuttgart

Homework Reading Task

Read:

in the **Ada Programming Wikibook**
⟨http://en.wikibooks.org/wiki/Ada_Programming⟩,

- especially **Getting Started**
- and in **Language Fatures** the parts about
  - **Expressions**,
  - **Control Structures**,
  - the **Type System**,
  - **Subprograms**, and
  - **Packages**.

Furthermore, read Sections 1–4 and 6–7 from **Ada Distilled**
http://www.adaic.org/resources/add_content/docs/distilled/adadistilled.pdf.

You will soon need that knowledge to solve your homework problems.

1.1: C-Syntax Pitfalls and Comparison to Ada

Goals of this section:

- Show some syntactical pitfalls of the C-family of languages, and how Ada avoids them
- Provide a gentle introduction to the Ada syntax
- Entertain the audience ;-)

Different philosophies of programming languages:

C, C++:	**Ada:**	**Java, C#:**
short, reduce writing if in doubt: the programmer is right	ease of reading, often more writing if in doubt: syntax error ("least surprise")	*borrow some of the ideas from Ada,* *but also inherit many of the problems from C*

What is the output of these two programs?

```ada
with Ada.Text_IO;

procedure XY is
  use Ada.Text_IO;
  X: Integer := 8265;
  Y: Integer := 0252;
begin
  Put_Line(Integer'Image(X));
  Put_Line(Integer'Image(Y));
end XY;
```

8265
252

```c
#include<stdio.h>

int main() {

  int x   = 8265;
  int y   = 0252;

  printf("%d\n", x);
  printf("%d\n", y);
}
```

8265
170 **(Why not 252?)**

But you can still use non-decimal bases in Ada:

```ada
Dec: Integer := 0170;        Oct: Integer := 8#252#;
Hex: Integer := 16#AA#;      Bin: integer := 2#1010_1010#;
```

The unexpected “coolness” of C/C++/Java

buggy C:

```
celsius = (5/9) * (fahrenheit - 32);
/* compiles */
/* But why on earth is celsius always zero? */
```

analysis:

- celsius and fahrenheit are of type float
- if operands mix float and int, integers are implicitely converted to float
- this happens to the two integers (5/9) and 32
- but 5/9=0; the conversion turns this into 0.0.

Ada:

```
Celsius := (5/9) * (Fahrenheit - 32);
-- does not compile: ''invalid operand types''
```

```
Celsius := (5.0/9.0) * (Fahrenheit - 32.0);
-- compiles
-- properly transforms Fahrenheit to Celsius
```

Ada Syntax

- "=" is comparison for equality.
- ":=" is assignment.
- If you mess up between "=" and ":=": *Syntax Error!*
- No "dangling else", because **if** is always followed by **end if**.
- No "$*$"-operator; parameter modes **in**, **out**, and **in out**.
  (Trying to write something to an **in**-parameter is a Syntax Error.)
- No "$++$"-operator.
- "$--$" begins a comment, which ends at the end of the line.
- Ada is CASE-INSENSITIVE.
  - Convention: `Use_Underscores`; keywords **in** lowercase.
  - Never use: `ComplexNameWithoutUnderscore`!
  - Example: `Example` $=$ `ExAmple` $\neq$ `Ex_Ample`!

  It is strongly recommended to always follow the convention
  ($\rightarrow$ Ada Pitfalls).

Why is this harmful?

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
err = sslRawVerify(...);
```

Apples Famous “goto fail” Bug

```
1  if ((err = SSLFreeBuffer(&hashCtx)) != 0)
2      goto fail;
3  if ((err = ReadyHash(&SHA1, &hashCtx)) != 0)
4      goto fail;
5  if ((err = SHA1.update(&hashCtx, &clientRandom)) != 0)
6      goto fail;
7  if ((err = SHA1.update(&hashCtx, &serverRandom)) != 0)
8      goto fail;
9  if ((err = SHA1.update(&hashCtx, &signedParams)) != 0)
10     goto fail;
11     goto fail;
12 if ((err = SHA1.final(&hashCtx, &hashOut)) != 0)
13     goto fail;
14 err = sslRawVerify(...);
```

- If no check has failed in lines 1, 2, 4, 7, or 9, then execution continues at “fail”, with `err=0` (“all is fine”).
- The tests in line 12 and 14 are skipped. But they are important for security!
- Without these tests, certificates are not checked properly, and a TLS connection can be attacked by “middlepersons”.
- Data can be read and modified, phishing becomes simple, …

C-Syntax versus Ada Syntax

both with the same misleading indention

```c
if (something);
  do_first();
  do_next();
```

```ada
if Something then
  null;
end if;
  Do_First;
  Do_Next;
```

```c
if (something)
  do_first();
  do_next();
```

```ada
if Something then
  Do_First;
end if;
  Do_Next;
```

```c
if (something) {
  do_first();
  do_next();
}
```

```ada
if Something then
  Do_First;
  Do_Next;
end if;
```

Can you spot the errors here?

```
enum alert_type {low, med, high, very_high};

void handle_alert(enum alert_type alert){
  switch(alert){
    case low:  activate_camera();
    case med:  send_guard();
    case high: sound_alarm();
  }
}
```

The Correct Program in Ada and C

```ada
type Alert_Type is (Low, Med, High, Very_High);

procedure Handle_Alert(Alert: Alert_Type) is
begin
  case Alert is
    when Low       => Activate_Camera;
    when Med       => Send_Guard;
    when High      => Sound_Alarm;
    when Very_High => Alert_The_Police;
  end case;
end Handle_Alert;
```

```c
enum alert_type {low, med, high, very_high};

void handle_alert(enum alert_type alert){
  switch(alert){
    case low:       activate_camera(); break;
    case med:       send_guard();      break;
    case high:      sound_alarm();     break;
    case very_high: call_the_police();
  }
}
```

More on Ada's "case" Statement

1. No fall-through! Thus, no need for a **break** statement.
2. When you forget an alternative: syntax error!
3. You can write **when others**. But think twice before doing so!

```
type Alert_Type is (Low, Med, High, Very_High);

procedure Handle_Alert(Alert: Alert_Type) is
begin
  case Alert is
    when Med       => Send_Guard;
    when High      => Sound_Alarm;
    when Very_High => Alert_The_Police;
  end case; -- the compiler will complain here!
end Handle_Alert;
```

```
  case Alert is
    ...
    when Low       => null;
    ...
  end case; -- this compiles fine!
```

What about this program?

```
typedef int Time;
typedef int Distance;
typedef int Speed;
const Speed SAFETY_SPEED = 120;

void increase_speed (Speed s){...}

void go(Time t, Distance d){
  Speed s = d/t;
  if (s < SAFETY_SPEED)
    increase_speed(t);
}

void perform_safety_checks(){
  Time     t = get_time();
  Distance d = get_distance();
  ...
  go(d,t);
}
```

Oh no!

Why did it go wrong so badly?

```c
typedef int Time;
typedef int Distance;
typedef int Speed;
const Speed SAFETY_SPEED = 120;

void increase_speed (Speed s){...}

void go(Time t, Distance d){
  Speed s = d/t;
  if (s < SAFETY_SPEED)
    increase_speed(t); /* ERROR! */
}

void perform_safety_checks(){
  Time      t = get_time();
  Distance d = get_distance();
  ...
  go(d,t); /* ERROR! */
}
```

The Same Buggy Code in Ada

```
procedure Go(T: Integer; D: Integer) is
  S: Integer := D/T;
begin
   if S < Safety_Speed then
      Increase_Speed(T); -- ERROR!
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Integer := Get_Time;
  D: Integer := Get_Distance;
begin
   ...
  Go(D,T); -- ERROR: should be Go(T,D)!
end Perform_Safety_Checks;
```

Ada: Semantic typing to your rescue!

```
type Time     is range 1 .. 9999;
type Distance is range 0 .. 9999;
type Speed    is range 0 .. 9999;
  -- Time, Distance and Speed are three different types
  -- using one, where the other one is expected, is an error
  -- or would require a formal ''type conversion''
  -- even if they are implemented identically (mostly)

Safety_Speed: constant Speed := 120;

procedure Increase_Speed(S: Speed) is
  ...
end Increase_Speed;
```

Now The Compiler Will Detect Such Bugs

```
procedure Go(T: Time; D: Distance) is
  S: Speed := D/T; -- Syntax Error!
begin
  if S < Safety_Speed then
    Increase_Speed(T); -- Syntax Error;
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Time := Get_Time;
  D: Distance := Get_Distance;
begin
  ...
  Go(D,T); -- Syntax Error;
end Perform_Safety_Checks;
```

Towards a Correct Program in Ada (1)

Mixed computations require a formal type conversion!

```
procedure Go(T: Time; D: Distance) is
  S: Speed := Speed(Integer(D)/Integer(T));
begin
  if S < Safety_Speed then
    Increase_Speed(T); -- Syntax Error;
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Time := Get_Time;
  D: Distance := Get_Distance;
begin
   ...
  Go(D,T); -- Syntax Error;
end Perform_Safety_Checks;
```

Towards a Correct Program in Ada (2)

Removing the first bug!

```
procedure Go(T: Time; D: Distance) is
  S: Speed := Speed(Integer(D)/Integer(T));
begin
   if S < Safety_Speed then
     Increase_Speed(S);
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Time := Get_Time;
  D: Distance := Get_Distance;
begin
   ...
  Go(D,T); -- Syntax Error;
end Perform_Safety_Checks;
```

Towards a Correct Program in Ada (3)

Removing the second bug!

```
procedure Go(T: Time; D: Distance) is
  S: Speed := Speed(Integer(D)/Integer(T));
begin
   if S < Safety_Speed then
     Increase_Speed(S);
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Time := Get_Time;
  D: Distance := Get_Distance;
begin
   ...
  Go(T,D);
end Perform_Safety_Checks;
```

Often Better Readable: “Named Parameters”

```
procedure Go(Deadline:      Time;
             The_Distance:  Distance) is
  S: Speed := Speed(   Integer(The_Distance)
                     / Integer(Deadline)       );
begin
  if S < Safety_Speed then
    Increase_Speed(S);
 end if;
end Go;

procedure Perform_Safety_Checks is
  T: Time := Get_Time;
  D: Distance := Get_Distance;
begin
  ...
  Go(The_Distance => D, Deadline => T);
end Perform_Safety_Checks;
```

“I called `send_bytes(...)`.

Why on earth does my program run forever?”

```
final int RADIO_Port = ...;

void open  (int port){...}
void send  (int port, byte data){...}
void close (int port){...}

void send_bytes(byte first,
                byte last,
                byte message){
  open(RADIO_Port);
  for (byte b = first; b <= last, b++){
    send(RADIO_PORT, message);
  }
  close(RADIO_PORT);
}
```

A for-loop in Ada never runs endlessly!

```
procedure Send_Bytes(First, Last: Byte;
                     Message: Byte) is
begin
  Open(Radio_Port);
  for B in First .. Last loop
    Send(Radio_Port, Message);
  end loop;
  Close(Radio_Port);
end Send_Bytes;
```

Overflow and Friends – a Comparison

Ada:

- Checks for:
  - Integer overflow,
  - Division by 0,
  - Access to an array-element out of bounds.

  Raises an **exception** whenever a check fails
- Checks can be turned off
- Special types (**mod** N) for "wrap-around" semantic

C/C++:

- Undefined

Java:

- Mostly like Ada
- Except for integer "wrap-around"
- Checks can't be turned off

C#:

- Mostly like Ada
- Two modes for integer operations: "Checked" and "Unchecked"(=wrap-around)

Programming by Contract (C)

introduce explicit `assert`-statements for a pre- and postconditions

```
typedef int Time;
typedef int Distance;

void go(Time t, Distance d){
  assert{t != 0); // dynamic check
    // aborts program and prints error if t is zero
  Speed s = d/t;
  ...
}

int square(int x){
  assert(X*X <= INT_MAX); // dynamic check
    // if X*X is too large the program will be aborted
  return(X*X);
    // without the assert, most C-compilers would return
    // (X*X) mod 2**{32} or so.
}
```

Programming by Contract (Ada)

preconditions are often implied by type definitions

```
type Time     is range 1 .. 9999;
type Distance is ...;

X: Integer;
D: Distance;

procedure Go(T: Time; D: Distance) is
  S: Speed := Speed(Integer(D)/Integer(T));
  -- precondition implied by definition of type Time
begin
   ...
Go(Time(X), D);
  -- if X in range then Go(...)
  -- else raise Constraint Error
```

Programming by Contract (Ada)

explicit preconditions in function specification

Specification:

```
function Square(X: Integer) return Integer
with Pre =>                                  -- |X| is not too large
  (if X > 1 then X <= Integer'Last/X              -- X not too large
   elsif X < -1 then X >= -Integer'First/X        -- X not too small
   else True                  -- |X| is trivially OK if X in {-1,0,1}
  );
```

(**if** X > 1 **then** ... **elsif** ... **else** ...) : "function expression" of type Boolean.

Implementation:

```
function Square(X: Integer) return Integer is
begin
  return X*X;
end Square;
```

Programming by Contract (Ada)

dynamic verification when calling the specified function

```
Y := Square(X);
```

Given the previous specification, what is this doing?

- By default, this will dynamically check $|X|$ for being too large. If so, it will raise an Assertion_Error.
- By setting a compiler switch, one can turn the check off.
- Also, one can use SPARK to *prove* that, whenever control goes to this assignment, $|X|$ will not be too large, and Assertion_Error cannot be raised.

What about Rust?

with the goal of being a “safe, concurrent, practical language”

Its syntax is superficially inpired by the C-family. Syntax and semantic also borrow from functional programming languages, such as Haskell.

Its main safety feature is its **memory safety**, preventing some of the most common bugs, such as referencing null pointers, dangling pointers and data races.

This makes Rust an interesting language for safe and secure software.

Some of the benefits of Ada over Rust:

- strong semantic typing (**type** Time **is** ...; **type** Speed **is** ...;)
- principle of least surprise (favours verbose source code over a compiler trying to infer what the programmer meant)
- separation between specification and implementation
- focus on correctness (design by contract, static verification)

1.2: Pitfalls and Issues when Programming in Ada

- Just like any other language, there are a lot of complex rules in Ada (operator overloading, name resolution, visibility, ...).
- Good: You do not need to know most of them to write correct programs. If the compiler is not sure what you mean, it will tell you by a *syntax error*.
- Also good: If your program compiles fine, it is unlikely to surpise you by doing something unexpected.
- The bad thing: In the beginning, the compiler will surprise you with *syntax error*s.
- There are still three pitfalls in Ada which make a simple program do something you may not have expected. These are related to identifier casing, array indexing, and the order of expression evaluation.
  Guess what the following program is doing!

```
with Ada.Text_IO;

procedure Pitfalls is
   Variable: String := "Global";

   procedure P(S: String) is
   begin for I in 1 .. S'Length loop
            Ada.Text_IO.Put(S(I));
         end loop;
         Ada.Text_IO.New_Line;
   end P;

   function Global_And_Local return String is
      VARIABLE: String := "LOCAL";
   begin return Variable & "␣" & VARIABLE;
   end Global_And_Local;

begin P(Global_And_Local);                          -- "Global LOCAL"?
      P(Variable);                                  -- "Global"?
      P("␣␣␣" & Variable(4 .. Variable'Last));      -- "   bal"?
      P(Variable(4 .. Variable'Last));              -- "bal"?
end Pitfalls;
```

Identifier Casing and Loop Boundaries

Identifier casing: always follow the convention described above!
Loop Boundaries: avoid immediate constants!

Bad:
```
for I in 1 .. Message'Length loop
```

Good:
```
for I in Message'First .. Message'Last loop
```

Shorter:
```
for I in Message'range loop
```

Full-length version:
```
for I in Message'Range loop
   Ada.Text_IO.Put(Message(I));
end loop;
```

Alternative (Ada 2012):
```
for Item of Message loop
   Ada.Text_IO.Put(Item);
end loop;
```

```
procedure Fool is

   procedure P(Double, Square: in out Integer) is
   begin
      Double := Double * 2;
      Square := Square * Square;
   end P;

   A: array(1..3) of Integer := (0, 3, 9);

begin
   P(A(1), A(3)); -- perfectly OK!
   P(A(2), A(2)); -- Ada 2012: illegal, before: ``erroneous''
   Ada.Text_IO.Put_Line(Integer'Image(A(2))); -- my output: 9

   for I in A'Range loop
      if A(I) /= 0 and (A(I+1) / A(I) > 1) then
         Ada.Text_IO.Put_Line(Integer'Image(A(I+1)/A(I)));
           -- raises Constraint_Error (``divide by zero''):
      end if;
   end loop;
end Fool;
```

The Order of Expression Evaluation in Ada

- Remember that in Ada the order of expression evaluation is undefined (left open for the compiler/optimizer to decide).
- In the case of logical expressions “*A* **and** *B*”, Ada requires both subexpressions to be evaluated (in whatever order).
- So, the side effects of *B* will take place even if *A* is false!
  If you do not want the side effects of *B*, write “*A* **and then** *B*”:

```
-- No exception any more!
for I in A'First .. A'Last-1 loop
   if A(I) /= 0 and then (A(I+1) / A(I) > 1) then
      Ada.Text_IO.Put_Line(Integer'Image(A(I+1)/A(I)));
   end if;
end loop;
```

- Similar for “*A* **or** *B*” resp. “*A* **or else** *B*”.

1.3: Distinguished Properties of Ada

This section will describe some basic properties of Ada with a focus on properties that distinguish Ada from other languages you are likely to know.

```
with Ada.Text_IO;

procedure Hello is
begin
  Ada.Text_IO.Put_Line("Hello!");
end Hello;
```

This is:

- a **program** (hello.adb),
- a **procedure** (Hello) and
- a **compilation unit.**

A compilation unit always starts with a **context clause** that describes the “imported” stuff from other compilation units.

In our case, this is one single package Ada.Text_IO, which we use the procedure Put_Line from.

Minor Variations of Hello

Avoiding "Fully Qualified" Names:

```
with Ada.Text_IO;
use Ada.Text_IO;
  -- use in context clause
  -- for the compilation unit
procedure Hello_2 is
begin
  Put_Line("Hello!");
end Hello_2;
```

```
with Ada.Text_IO;
procedure Hello_3 is
  use Ada.Text_IO;
  -- use locally
  -- where you need it
begin
  Put_Line("Hello!");
end Hello_3;
```

```
with Ada.Text_IO;
procedure Hello_4 is
  package TIO renames Ada.Text_IO;
  -- define a shortcut for Ada.Text_IO
begin
  TIO.Put_Line("Hello!");
end Hello_4;
```

Subprograms

Procedures

```
procedure Reverse_String(S: in out String) is
begin
  if S'Length>1 then
    Swap(S(S'First), S(S'Last));
    Reverse_String S(S'First+1 .. S'Last-1);
  end if;
end Reverse_String;
```

Subprograms

Functions

```
function Reverse_String(S: String) return String is
begin
   If S'Length=0 then
      return ""; -- empty string
   else
      return S(S'Last .. S'Last) & -- last character first
             Reverse_String(S(S'First .. S'Last-1));
             -- all other characters last
    end if;
end Reverse_String;
```

With Ada 2012, this can also be written as a “function expression”:

```
function Reverse_String(S: String) return String is
  (if S'Length=0 then ""
   else S(S'Last .. S'Last) &
        Reverse_String(S(S'First .. S'Last-1))
  );
```

Types and Subtypes

```
type Byte is range 0 .. 255;
type X_Coord is new Byte; -- not assignment-compatible to Byte
type Y_Coord is new Byte; -- not assignment-compatible to ...
...
procedure Set_Pixel(X: X_Coord, Y: Y_Coord; Col: Color) is ...
...
X: X_Coord; Y: Y_Coord;
begin -- main program
  Set_Pixel(X, Y, Green); -- is OK;
  Set_Pixel(X, Y_Coord(X), Black); -- explicit conversion, OK
  Set_Pixel(Y, Y, White); -- Syntax Error at compile time
```

```
subtype Center_X is X_Koord range 55 .. 200;
subtype Center_Y is Y_Koord range 55 .. 200;
...
Set_Center_Pixel(X: Center_X; Y: Center_Y; Col: Color);
...
X: X_Coord := 0; Y: Y_Coord := 255;
begin
  Set_Center_Pixel(X, Y, ...);
    -- raises Constraint_Error at run time
```

Example

```
type Length_Type   is range 0 ..       100;
type Volume_Type   is range 0 .. 1_000_000;
type Surface_Type  is range 0 ..    60_000;
subtype Single_Surface_Type is
        Surface_Type range 0 ..    10_000;

function Surface(X,Y: Length_Type)
            return Single_Surface_Type is
begin
  return Surface_Type(X)*Surface_Type(Y);
end Surface;

function Surface (A,B,C: Length_Type)
            return Surface_Type is
begin
  return Surface(A*B)*2 + Surface(A*C)*2
                        + Surface(B*C)*2;
end Surface;
```

Unconstrained Arrays and Strings

(<>="Box")

```
type Bit_Vector is array(Integer range <>)
  of Boolean;

type String is array (Positive range <>)
  of Character; -- String is actually predefined

type History is array (Natural range <>)
  of Year_Record;
```

- type T is *unconstrained* if it has *undetermined discriminants*
- rule of thumb: T is *unconstrained* if compiler *doesn't know the storage size* an object of type T would have
- examples for constrained types

```
type Byte is Bit_Vector(0 .. 7);
type Output_Line is String(1 .. 80);
```

Constrained Arrays

```
Recent: History(1900 .. 2011);
  -- no need to start counting by 0 or 1!
...
   Put(Recent'First);                  -- 1900
   Put(Recent'Last);                   -- 2011
   Put(Recent'Length); -- 2011-1900+1 =  112
```

- Discriminants can be determined implicitly:

```
S: String := "Constant String";
```

- Parameters of subprograms and function return values can be unconstrained – the discriminants are determined when calling the subprogram, resp. when returning:

```
function Bracket(S: String) return String is
begin
  return "[" & S & "]";
end Bracket;
T: String := Bracket(Bracket(S));
  -- now T = "[[Constant String]]";
```

Packages

Separating Specifications From Implementations

```
package String_Stuff is
  function Reverse_String(S: String) return String;
  function Extract(S: String; First, Last: Positive)
                      return String;
end String_Stuff;
```

```
package body String_Stuff is
   function Reverse_String(S: String) return String is
     ...

   function Extract(S: String; First, Last: Positive)
                      return String is
   begin
     if First < S'First or Last > S'Last then
       raise Constraint_Error;
     else
       return S(First .. Last);
     end if;
   end Extract;
end String_Stuff;
```

Pre- and Postconditions

```
package String_Stuff is
  function Reverse_String(S: String) return String;
  function Extract(S: String; First, Last: Positive)
                        return String
    with Pre => (S'First <= First and S'Last >= Last
                                  and First <= Last);
end String_Stuff;
```

```
package body String_Stuff is

   function Reverse_String(S: String) return String is ...

   function Extract(S: String; First, Last: Positive)
                         return String is
   begin
      return S(First .. Last);
   end Extract;

end String_Stuff;
```

A Package Specification (stacks.ads)

```
generic
   type Item_Type is private;
   Capacity: Positive;
package Stacks is

   type Stack is tagged private;
   subtype Count_Type is Natural range 0 .. Capacity;

   function Count(S: Stack) return Count_Type;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;

   procedure Push(S: in out Stack; Item: Item_Type);
   function Pop(S: in out Stack) return Item_Type;

private -- implementation-specific information
  ...
end Stacks;
```

Private Types

Splitting "what the client programmer has to know" from "what the compiler has to know".

```
generic
   type Item_Type is private;
   Capacity: Positive;
package Stacks is

   type Stack is tagged private;
   subtype Count_Type is Natural range 0 .. Capacity;

   ...

private -- implementation-specific information
   type Item_Array is array(1 .. Capacity) of Item_Type;
   type Stack is tagged record
      Current: Count_Type := 0;
      Content: Item_Array;
   end record;
end Stacks;
```

What shall these procedures and functions actually do?

Adding pre- and postconditions to the specification.

```
function Count(S: Stack) return Count_Type;

function Is_Empty(S: Stack) return Boolean with
  Post => Is_Empty'Result = (S.Count = 0);

function Is_Full(S: Stack) return Boolean with
  Post => Is_Full'Result = (S.Count = Capacity);

procedure Push(S: in out Stack; Item: Item_Type) with
  Pre  => not Is_Full(S),
  Post => S.Count = S'Old.Count + 1;

function Pop(S: in out Stack) return Item_Type with
  Pre  => not Is_Empty(S),
  Post => S.Count = S'Old.Count - 1;
```

The Implementation (stacks.adb, first half)

```
package body Stacks is

   function Count(S: Stack) return Count_Type is
   begin
      return S.Current;
   end Count;

   function Is_Empty(S: Stack) return Boolean is
   begin
      return S.Count=0;
   end Is_Empty;

   function Is_Full(S: Stack) return Boolean is
   begin
      return S.Count = Capacity;
   end Is_Full;
```

The Implementation (stacks.adb, second half)

```
   procedure Push(S: in out Stack; Item: Item_Type) is
   begin
      S.Current := S.Current + 1;
      S.Content(S.Current) := Item;
   end Push;

   function Pop(S: in out Stack) return Item_Type is
   begin
      S.Current := S.Current - 1;
      return S.Content(S.Current+1);
   end Pop;
end Stacks;
```

What does this program do?

```
with Ada.text_IO, Stacks;

procedure Stack_Test is

   package C_Stacks is new Stacks(Item_Type => Character,
                                  Capacity  => 10);
   S: C_Stacks.Stack;
begin
   S.Push('!'); S.Push('a');
   S.Push('d'); S.Push('A');
   S.Push('␣'); S.Push('o');
   S.Push('l'); S.Push('l');
   S.Push('e'); S.Push('H');
   if not S.Is_Full then
      raise Program_Error;
   end if;
   while not S.Is_Empty loop
      Ada.Text_IO.Put(S.Pop);
   end loop;
end Stack_Test;
```

Homework Reading Task

Read:

in the **Ada Programming Wikibook** ⟨`http://en.wikibooks.org/wiki/Ada_Programming`⟩, the parts about

- **Exceptions**,
- **Generics**,
- **Object Orientation**,
- **Containers**, and
- **Ada Programming Tips**.

Further, read sections 9, 11, and 12 from **Ada Distilled**.

1.4: Polymorphism

Wikipedia: *"… polymorphism … allows values of different data types to be handled using a uniform interface."*

Different types of polymorphism:

- Subprogram overloading
- Parametric polymorphism, (or "generic programming")
- Subtype polymorphism (inheritance or "tagged types")
- …

We have already seen some of this before.

Subprogram Overloading

Same Name For Different Subprograms

```
procedure Reverse_String(S: in out String); -- few slides before
function Reverse_String(S: String) return String; -- ditto

procedure Put(Item: Integer);
procedure Put(Item: Float);
procedure Put(Item: Character);
procedure Put(Item: String);

function "+"(Left, Right: Integer) return Integer;   -- (1)
function "+"(Left, Right: Integer) return Float;     -- (2)
function "+"(Left: Integer; Right: Float) return Float;
function "+"(Left: Float; Right: Float) return Float;
function "+"(Left: Integer; Right: Float) return Float;
...
  I: Integer := 2+3; -- uses (1)
  F: Float   := I+1; -- uses (2)
                     -- would not work in many other languages
```

use [[all] type] ... (1)

- **use** ⟨Package-Name⟩:
  makes *all* identifiers from the *entire* package spec directly visible, exceot for the **private** parts (Ada 83)
- **use type** ⟨Type-Name⟩:
  makes all "primitive" *operators* of the type directly visible (Ada 95)
- **use all type** ⟨Type-Name⟩:
  makes all "primitive" subprograms of the type directly visible, (but not the name of the type itself) (Ada 2012)

```ada
package A is
  type Number is ...

  -- two primitive subprograms
  function To_Number(X: Integer) return Number;
  procedure Put(X: Number);

  -- one primitive operator
  function "+"(X,Y: Number) return Number;
end A;
```

use [[all] type] ... (2)

```
with A;
procedure P is
   N: A.Number := A.To_Number(1);
   M: A.Number := A.To_Number(2);
begin
   A.Put(A."+"(M,N));
   -- no use: cannot write M+N
end P;
```

```
with A;
procedure 95 is
   use type A.Number;
   N: A.Number := A.To_Number(1);
   M: A.Number := A.To_Number(2);
begin
   A.Put(M+N);
end P95;
```

```
with A;
procedure P83 is
   use A;
   N: Number := To_Number(1);
   M: Number := To_Number(2);
begin
   Put(M+N);
end P83;
```

```
with A;
procedure P12 is
   use all type A.Number
   N: A.Number := To_Number(1);
   M: A.Number := To_Number(2);
begin
   Put(M+N);
end P12;
```

Parametric Polymorphism

```
package Stacks(type Item_Type) is ...
  ------> this is what we mean -- but not the Ada Syntax
```

```
generic
   type Item_Type is private;
package Stacks is ...
  ------> this is the same in Ada Syntax
```

```
generic
   type Item_Type is private;
   Capacity: Positive;
package Stacks is
  ------> This is what we actally used before
```

How To Instantiate a Generic Package

- In context-clause: **with** ⟨Generic-Name⟩
- Where you need it:
  package ⟨Name⟩ **is new** ⟨Generic-Name⟩(⟨Parameter⟩);

```
package C_Stacks is new Stacks(Item_Type => Character,
                               Capacity => 10);

package I_Stacks is new Stacks(Integer, 1024);

package A is new Approximation
                  (Float,
                   Ada.Numerics.Elementary_Functions.Sqrt,
                   Ada.Numerics.Elementary_Functions."**");
```

```
use type A.Number; -- makes operators visible
```

Abstract Messages

Subtype Polymorphism

```
package Messages is
  type Message is abstract tagged private;

  procedure Show(The_Message: Message) is abstract;

private
  type Message is abstract tagged null record;
  -- null record = record null; end record

end Messages;
```

- One abstract class `Messages.Message`
- One abstract “primitive operation” (can be overridden) `Show`
- No keyword “class”, but **tagged record** / **tagged private**

Short Messages

```
package Messages.Short is
  type SMS is new Message with private;

  overriding procedure Show(Self: SMS);

  package Create is
    procedure Message(What: String;
                      The_SMS: out SMS);
  end Create;
private
  type SMS is new Message with
    record
      What: String(1 .. 160);
      How_Long: Integer range 0 .. 160 := 0;
    end record;
end Messages.Short;
```

You create an object by calling `Create.Message;`

The Implementation

```
package body Messages.Short is
  package body Create is
    procedure Message(What: String;
                        The_SMS: out SMS) is
    begin
      if What'Length > 160 then
        raise Constraint_Error;
      end if;
      The_SMS.How_Long := What'Length;
      The_SMS.What(1 .. What'Length) := What;
    end Message;
  end Create;

  procedure Show(Self: SMS) is
  begin
    Ada.Text_IO.Put_Line
        (Self.What(1 .. Self.How_Long));
  end Show;
end Messages.Short;
```

Screen-Wide Messages

```
package Messages.Screen is
  type Screen_Message is new Message with private;

  overriding procedure Show(Self: Screen_Message);

  not overriding procedure Extend_Line
    (Self: in out Screen_Message;
     Additional_Line: in String);

  package Create is
    function Message(First_Line: String)
      return Screen_Message;
  end Create;
```

A Client Using Messages

```
procedure Test_Messages is
  SMS: Messages.Short.SMS;
  Screen: Messages.Screen.Screen_Message
    := Messages.Screen.Create.Message("lang");

  procedure Display(Msg: Messages.Message'Class) is
  begin
    Msg.Show;
  end Display;

begin
  Messages.Short.Create.Message("kurz", SMS);
  SMS.Display;
  Screen.Display;
end Test_Messages;
```

No Surprise!

```
procedure Display(Msg: Messages.Message'Class) is
begin
   Msg.Show; -- a ''dispatching call''
end Display;
```

- "`Msg.Show`" = "`Messages.Show(Msg)`" (Object.Method notation for tagged types)
- Formal class-wide parameter "`Msg`"
- The actual parameter can be of type `Messages.Message` (∗), or of a type derived from `Messages.Message`

The actual call which is performed depends on the type of the the parameter `Msg` when `Display` is called. This is called "dispatching".

(∗) Actually not, because `Messages.Message` is abstract.

Surprise!

```
procedure Display(Msg: Messages.Message'Class) is
begin
   Msg.Show; -- a ``dispatching call''
end Display;
```

- Why is `Messages.Message'Class` the type of `Msg`?

(!) If we change the type of `Msg` to `Messages.Message`, we can call `Display` only with an actual parameter of that type, not of a subtype. (But remember the footnote above.)

- In contrast to other object-oriented languages, calls are not always dispatching. The `'Class`-attribute of the type is the request to perform dispatching.

Inheritance vs. Generics

Benefits of inheritance and dispatching:

- More flexible programs
- Code reuse

Disadvantage of inheritance:

- Code is harder to analyse (due to dispatching)
- Testing can become a lot more difficult

Recommendation for safe and secure programs:

- Use inheritance if you actually need the flexibility
- Prefer parametric polymorphism when possible

Dispatching vs. “case”

Semantically, . . .

- Dispatching call is (possibly big) **case**-statement
- Adding new class is adding a **case** alternative, possibly at many places (and you may forget to update one of your **case**s)
- Common wisdom: avoid **case**-statements
- But: dispatching is more difficult to analyze
- In Ada: compiler catches missing **case**-alternatives

Ada wisdom:

- If in doubt, prefer **case** over dispatching
- But then, *please* avoid **when others**

Automatic Initialization/Finalization

Read: How the Ada Reference defines the standard package `Ada.Finalization`: `http://www.adaic.org/resources/add_content/standards/12rm/html/RM-7-6.html`

Simple Example: The Package Clean_up

```
with Ada.Finalization;

generic
   with procedure First_Steps; with procedure Last_Steps;
package Clean_Up is -- no visible content

private
   type Caretaker is new Ada.Finalization.Limited_Controlled
      with null record;
   overriding procedure Initialize(Object: in out Caretaker);
   overriding procedure Finalize  (Object: in out Caretaker);
end Clean_Up;
```

```
package body Clean_Up is
   overriding procedure Initialize(Object: in out Caretaker) is
     begin First_Steps; end Initialize;
   overriding procedure Finalize(Object: in out Caretaker) is
     begin Last_Steps; end Finalize;
   Some_Caretaker: Caretaker;
     -- calls Initialize when created
     -- and Finalize when going out of scope
end Clean_Up;
```

Simple Example: Using Clean_up

```
with Ada.Text_IO, Clean_Up;

procedure Test_Clean_Up is

   procedure Open_File is
   begin
      Ada.Text_IO.Put_Line("Open␣File");
   end Open_File;

   procedure Close_File is
   begin
      Ada.Text_IO.Put_Line("Close␣File");
   end Close_File;

   package Cu is new
     Clean_Up(First_Steps => Open_File, Last_Steps => Close_File);

begin
   ...
end Test_Clean_Up;
```

Running Test_Clean_up

. . . does the right thing!

```
begin
   Ada.Text_IO.Put_Line("Working with File");
   raise Program_Error;
end Test_Clean_Up;
```

```
Open File
Working with File
Close File

raised PROGRAM_ERROR : test_clean_up.adb:20
                       explicit raise
```

1.5: Pointers / Access Types

Ada supports four different kinds of access-types:

1. Access to data in a storage pool ("heap"), allocated by calling `new`.
2. General access types to both data in a storage pool and "normal" data allocated on the stack (marked as "aliased").
3. Subprogram access parameters.
4. Access to a subprogram.

As a rule of thumb, you should hardly ever use the first three – except for implementing a container library, or the like (see next slide).

Access to subprogram comes sometypes handy, as a weak form of polymorphism (use of callbacks).

Why to avoid Access types in Ada?

If you think you need access types (“pointers”), think again:

- There is `Ada.Containers`. Why write your own containers?
- Most Ada implementations do not support garbage collection (GC):
  - GC and real-time do not fit well
  - Experienced Ada programmers hardly ever use access types anyway.
- Programmers for other languages often use a pointer-to-unconstrained pattern (e.g. for class-wide operations). In Ada, you can just declare your parameter as “XXX’Class”.
- If you really need unconstrained objects, store it in a container. A “Holder” is a container for one single unconstrained element.

Callbacks

```
with Ada.Text_IO;
procedure Iterate_Example is

  type Do_String is not null access procedure(S: String);

  procedure Do_Wordwise(Big_String: String;
                        Do_Word: Do_String) is ...
  end Do_Wordwise;

  procedure Output_String(E: String) is
  begin
    Ada.Text_IO.Put_Line(E);
  end Output_String;

begin
  Do_Wordwise("I am a string.", Output_String'access);
           -- output 4 lines: I \\ am \\ a \\ string.
end Iterate_Example;
```

Callbacks (2)

```
procedure Do_Wordwise(Big_String: String;
                      Do_Word: Do_String) is
   I: Natural := Big_String'First;
   J: Natural := I;
begin
   while I <= Big_String'Last loop
      while I <= Big_String'Last
        and then Big_String(I)/=' ' loop
          I := I + 1;
      end loop;
      Do_Word(Big_String(J .. I-1));
      -- You would expect a dereference here: Do_Word.all(...);
      -- But if the syntax unmistakably is a subprogramm call,
      -- you don't need the dereference operator ".all".
      I := I + 1;
      J := I;
   end loop;
end Do_Wordwise;
```