Ada-95 dla programistów C/C++

Dariusz Wawrzyniak
(na podst. oprac. Simona Johnstona)

Część I

Ada Packages

Outline

Outline

Ada packages

- Ada package consists of two parts: the specification (header) and body (code).
- The specification contains an explicit list of the visible components of a package.
- The specification is a completely stand alone entity which can be compiled on its own and so must include specifications from other packages to do so.
- An Ada package body at compile time must refer to its package specification to ensure legal declarations.

The skeleton of a package

```
--file example.ads, the package specification.
package example is
    ...
end example;
```

```
--file example.adb, the package body.
package body example is
    ...
end example;
```

Include a package in another

- A C file includes a header by simply inserting the text of the header into the current compilation stream with `#include "example.h"`.
- Ada package specification has a two stage process:
  - compilation,
  - inclusion.

Include a package — example

```
-- Specification for package example
with Project_Specs;
package example is
  type My_Type is new Project_Spec.Their_Type;
end example;
```

```
-- Body for package example
with My_Specs;
package body example is
  type New_Type_1 is new My_Specs.Type_1;
  type New_Type_2 is new Project_Specs.Type_1;
end example;
```

Basic visibility rules

- If the inclusion of `Project_Specs` appears in the specification of `Example`, all the items declared in `Project_Specs` are available in the specification of `Example`, its body (implemenation) and all packages that include `Example`.
- If the inclusion of `Project_Specs` appears in the body of `Example`, the items declared in `Project_Specs` are available only in the body.

Naming rules

To avoid using packege names in naming items defined in the packege **use** can be applied:

```
with My_Specs; use My_Specs;
package body example is
   ...
end example;
```

It is usual in Ada to put the **with** and the **use** on the same line, for clarity.
There is a special form of the **use** statement which can simply include an element (types only) from a package, consider:

```
use type Ada.Calendar.Time;
```

Outline

Data encapsulation

- Data encapsulation requires, for any level of safe reuse, a level of hiding.
- The declaration of some data is deferred to a future point so that any client cannot depend on the structure of the data.
- This allows the provider to change that structure if the need arises.

Data encapsulation in C

In C this is done by presenting the 'private type' as a `void*` which means that you cannot know anything about it, but implies that no one can do any form of type checking on it.

```
typedef void* list;
list create(void);
```

Data encapsulation C++

In C++ we can forward declare classes and so provide an anonymous class type.

```
class Our_List {
public:
  Our_List(void);
private:
  class List_Rep;
  List_Rep* Representation;
};
```

The implementation of `Our_List` and its internal representation `List_Rep` gives all the advantages of type checking, but hides the details of structuring the list.

Private part of a package

In Ada the concept of data hiding is formalised into the ’private part’ of a package. This private part is used to define items which are forward declared as private.

```
package Our_List is
  type List_Rep is private;
  function Create return List_Rep;
private
  type List_Rep is
    record
       -- some data
    end record;
end Our_List;
```

Private and limited private data types

Private type

- the only operations that the client may use are `:=`, $=$ and $/=$,
- **all other** operations must be provided by functions and procedures in the package.

Limited private type

- no predefined operators available,
- **all** operations must be provided by functions and procedures in the package.

Items of a private type

You may not in the public part of the package specification declare variables of the private type as the representation is not yet known, we can declare constants of the type, but you must declare them in both places, forward reference them in the public part with no value, and then again in the private part to provide a value:

```
package Example is
  type A is private;
  B : constant A;
private
  type A is new Integer;
  B : constant A := 0;
end Example;
```

Outline

Nested packages in Ada-83

Ada allows the nesting of packages within each other, this can be useful for a number of reasons. With Ada-83 this was possible by nesting package specs and bodies physically, thus:

```
package Outer is
  package Inner_1 is
  end Inner_1;

  package Inner_2 is
  end Inner_2;
private
end Outer;
```

Nested packages in Ada-95

Ada-95 has added to this the possibility to define child packages outside the physical scope of a package, thus:

```
package Outer is
  package Inner_1 is
  end Inner_1;
end Outer;
```

```
package Outer.Inner_2 is
end Outer.Inner_2;
```

As you can see `Inner_2` is still a child of outer but can be created at some later date, by a different team.

Outline

Renaming (1)

Renaming is not a package specific topic, and it is only introduced here as the using of packages is the most common place to find a renames clause.
Renaming allows to save a lot of (re)typing.

```
with Outer;
with Outer.Inner_1;
package New_Package is
  OI_1 renames Outer.Inner_1;

  type New_type is new OI_1.A_Type;
end New_Package;
```

Renaming (2)

Renaming helps to remove ambiguity.

```
with Package1;
function Function1 return Integer
         renames Package1.Function;
with Package2;
function Function2 return Integer
         renames Package2.Function;
```

Temporal renaming

```
for device in Device_Map loop
  Device_Map(device).Device_Handler.Request_Device;
  Device_Map(device).Device_Handler.Process_Function
                            (Process_This_Request);
  Device_Map(device).Device_Handler.Relinquish_Device;
end loop;
```

```
for device in Device_Map loop
  declare
    Device_Handler : Device_Type renames
                     Device_Map(device).Device_Handler;
  begin
    Device_Handler.Request_Device;
    Device_Handler.Process_Function(Process_This_Request);
    Device_Handler.Relinquish_Device;
  end;
end loop;
```

Część II

Ada-95 Object Oriented Programming

Outline

Outline

The concept of class in C++

- C++ extends C with the concept of a **`class`**.
- A class is an extension of the existing **`struct`** construct.
- The difference with a class is that a class not only contains data (member attributes) but code as well (member functions).

An example of class

```
class A_Device {
public:
  A_Device(char*, int, int);

  char* Name(void);
  int   Major(void);
  int   Minor(void);
protected:
  char* name;
  int   major;
  int   minor;
};
```

Object creation

The code introduces a constructor, a function with the same name as the class which is called whenever the class is created. In C++ these may be overloaded and are called either by the **new** operator, or in local variable declarations as below.

```
A_Device lp1("lp1", 10, 1);

A_Device* lp1;
  lp1 = new A_Device("lp1", 10, 1);
```

Creates a new device object called `lp1` and sets up the name and major/minor numbers.

Ada-95 extensions towards OOP

Ada has also extended its equivalent of a `struct`, the **record** but does not directly attach the member functions to it.

```
package Devices is
  type Device is tagged private;
  type Device_Type is access Device;
  function Create(Name: String; Major: Integer;
           Minor: Integer) return Device_Type;
  function Name(this: Device_Type)
           return String;
  function Major(this: Device_Type)
           return Integer;
  function Minor(this: Device_Type)
           return Integer;
```

Ada-95 extensions towards OOP (cont.)

```
private
  type Device is tagged
    record
      Name  : String(1 .. 20);
      Major : Integer;
      Minor : Integer;
    end record;
end Devices;
```

The equivalent declaration of an object would be:

```
lp1 : Devices.Device_Type :=
        Devices.Create("lp1", 10, 1);
```

Outline

The tagged type

- The addition of the keyword **`tagged`** to the definition of the type `Device` makes it a class in C++ terms.
- The tagged type is simply an extension of the Ada-83 record type.
- It includes a 'tag' which can identify not only its own type but its place in the type hierarchy.

The ’Tag attribute

The tag can be accessed by the attribute `’Tag` but should only be used for comparison, i.e.

```
with Ada.Tags;
dev1, dev2 : Device’Class := ... ;
    -- initialization is required

if dev1’Tag = dev2’Tag then
```

this can identify the **isa** relationship between two objects.

The 'Class attribute

Another important attribute 'Class exists which is used in type declarations to denote the *class-wide type*, the inheritance tree rooted at that type, i.e.

```
type Device_Class is Device'Class;
-- or more normally
type Device_Class is access Device'Class;
```

The second type denotes a pointer to objects of type Device and any objects whose type has been inherited from Device.

Outline

Class member attributes

Member attributes in C++ directly map onto data members of the tagged type. So the `char* name` directly maps into `Name : String`.

Class member functions

- Non-virtual, non-const, non-static member functions map onto subprograms, within the same package as the tagged type.
- The first parameter is of that tagged type or an access to the tagged type, or who returns such a type.

Virtual member functions

- Virtual member functions map onto subprograms, within the same package as the tagged type.
- The first formal parameter is of the **tagged** (not class-wide) type, or an **anonymous access** type to the tagged (not class-wide) type, or the return value is of such a type.

Pure virtual functions

- A pure virtual function maps onto a virtual member function with the keywords **`is abstract`** before the semicolon.
- When any pure virtual member functions exist the tagged type they refer to must also be identified as abstract.
- If an abstract tagged type has been introduced which has no data, then the following shorthand can be used:

```
type Root_Type is abstract tagged
                           null record;
```

Static members

Static members map onto subprograms within the same package as the tagged type. These are no different from normal Ada-83 subprograms, it is up to the programmer when applying coding rules to identify only member functions or static functions in a package which includes a tagged type.

Constructors/Destructors for Ada

- There is no constructors and destructors in Ada.
- In the examples the constructor has been synthesised with the `Create` function which creates a new object and returns it.
- If you intend to use this method then the most important thing to remember is to use the same name throughout, `Create Copy Destroy` etc are all useful conventions.
- Ada does provide a library package `Ada.Finalization` which can provide constructor/destructor like facilities for tagged types.

Outline

Inheritance

- The most common attribute sited as the mark of a true object oriented language is support for inheritance.
- Ada-95 adds inheritance as *tagged type extension*.
- Ada does not directly support multiple inheritance.

Inheritance in C++

```
class A_Tape : public A_Device {
   public:
      A_Tape(char*, int, int);
      int Block_Size(void);

   protected:
      int block_size;
};
```

Inheritance in Ada-95

```
package Devices.Tapes is
  type Tape is new Device with private;
  type Tape_Type is access Tape;
  function Create(Name: String; Major: Integer;
                  Minor : Integer) return Tape_Type;
  function Block_Size(this: Tape_Type)
              return Integer;
private
  type Tape is new Device with
    record
      Block_Size : Integer;
    end record;
end Devices.Tapes;
```

public/protected/private

In the example at the top of this section we provided the `Device` comparison. In this example the C++ class provided a public interface and a protected one, the Ada equivalent then provided an interface in the public part and the tagged type declaration in the private part. Because of the rules for child packages a child of the `Devices` package can see the private part and so can use the definition of the `Device` tagged type.