Ada-95 dla programistów C/C++

Dariusz Wawrzyniak
(na podst. oprac. Simona Johnstona)

Part I

Ada Basics

Outline

First View — outline

1. First View
2. Ada Types
3. Ada Statements

First View

- Ada is case insensitive, so **`begin BEGIN Begin`** are all the same.
- The tick (`’`) is used to access attributes for an object.

  ```
  a : Integer := Integer’Size;
  ```

  In C: **`int`** `a =` **`sizeof`**`(`**`int`**`) * 8;`
- Another use for the tick is to access the attributes `First` and `Last` (for an integer the range of possible values is `Integer’First to Integer’Last`).
- The tick is also used for other Ada constructs as well as attributes.

Operators in C and Ada (1)

Operator	C/C++	Ada
Assignment	`=`	`:=`
Equality	`==`	`=`
NonEquality	`!=`	`/=`
Plus Equals	`+=`	
Subtract Equals	`-=`	
Multiply Equals	`*=`	
Division Equals	`/=`	
Or Equals	`|=`	
And Equals	`&=`	

Operators in C and Ada (2)

Operator	C/C++	Ada
Modulus	%	**`mod`**
Remainder		**`rem`**
Absolute Value		**`abs`**
Exponentiation		`**`
Range		`..`

Ada Types — outline

1. First View
2. Ada Types
   - C/C++ types to Ada types
   - Declaring new types and subtypes
   - Simple types
   - Arrays
   - Records
3. Ada Statements

Declarations

- Note that objects are defined in reverse order to C/C++, the object name is first, then the object type.
- As in C/C++ you can declare lists of objects by separating them with commas.

C

```c
int i;
int a, b, c;
int j = 0;
int k, l = 1;
```

Ada

```ada
i : Integer;
a, b, c : Integer;
j : Integer := 0;
k, l : Integer := 1;
```

Constants

Another difference is in defining constants.
C

```
const int days_per_week = 7;
```

Ada

```
days_per_week : constant Integer := 7;
days_per_week : constant := 7;
```

In the Ada example it is possible to define a constant without type, the compiler then chooses the most appropriate type to represent it.

Strong typing

- Ada is a strongly typed language (possibly the strongest)
- In C the use of **`typedef`** introduces a new name which can be used as a new type.
- The weak typing of C and even C++ (in comparison) means that we have only really introduced a very poor synonym.

C

```
typedef int INT;
INT a;
int b;

a = b; //works
```

Ada

```
type INT is new Integer;
a : INT;
b : Integer;

a := b; -- fails.
```

Weaker typing

Strong typing can be a problem, and so Ada provides a feature for reducing the distance between the new type and its parent.

```
subtype INT is Integer;
a : INT;
b : Integer;
a := b; -- works.
```

The most important feature of the subtype is to constrain the parent type in some way, for example to place an upper or lower boundary for an integer value.

Integer types

Integer, Long_Integer etc.

Any Ada compiler must provide the Integer type, this is a signed integer, and of implementation defined size. The compiler is also at liberty to provide `Long_Integer`, `Short_Integer`, `Long_Long_Integer` etc. as needed.

Unsigned Integers

Ada does not have a defined unsigned integer, so this can be synthesised by a range type, and Ada-95 has a defined package, `System.Unsigned_Types` which provide such a set of types.

Modular Integers

Ada-95 has added a **modular** type which specifies the maximum value, and also the feature that arithmatic is cyclic, underflow/overflow cannot occur.

```
type BYTE is mod 256;
type BYTE is mod 2**8;
```

Note: it is not required to use 2**x, you can use any value, so 10**10 is legal also.

Characters and Boolean

Character

This is very similar to the C char type, and holds the ASCII character set. However it is actually defined in the package `Standard` as an enumerated type. There is an Ada equivalent of the C set of functions in `ctype.h` which is the package `Ada.Characters.Handling`.
Ada Also defines a `Wide_Character` type for handling non ASCII character sets.

Boolean

This is also defined in the package `Standard` as an enumerated type (see below) as `(FALSE, TRUE)`.

Strings (1)

- Ada has a predefined String type (`Standard`).
- There is a good set of Ada packages for string handling, much better defined than the set provided by C.
- Ada has a & operator for string concatenation.
- Ada cannot use 'unconstrained' types in static declarations.

```
type A_Record is
  record
    illegal : String;
    legal   : String(1 .. 20);
  end record;
procedure check(legal : in String);
```

Strings (2)

- The lower bound of the size must be greater than or equal to 1.
- The C/C++ `array[4]` which defines a range `0..3` cannot be used in Ada, `1..4` must be used.
- One way to specify the size is by initialisation, for example:

  ```
  Name : String := "Simon";
  ```

  is the same as defining `Name` as a `String(1..5)` and assigning it the value `"Simon"` seperatly.
- For parameter types unconstrained types are allowed, similar to passing `int array[]` in C.
- Ada has a package `Ada.Strings.Unbounded` which implements a variable length string type.

Floating point

- Ada has two non-integer numeric types, the floating point and fixed point types.
- The predefined floating point type is `Float` and compilers may add `Long_Float`, etc.
- A new Float type may be defined in one of two ways:

```
type FloatingPoint1 is new Float;
type FloatingPoint2 is digits 5;
```

Fixed point

- Fixed point types are unusual, there is no predefined type 'Fixed' and such type must be declared in the long form:

```
type Fixed is delta 0.1 range -1.0 .. 1.0;
```

- This defines a type which ranges from -1.0 to 1.0 with an accuracy of 0.1. Each element, accuracy, low-bound and high-bound must be defined as a real number.
- There is a specific form of fixed point types (added by Ada-95) called decimal types. These add a clause **`digits`**, and the **`range`** clause becomes optional.

```
type Decimal is delta 0.01 digits 10;
```

Enumerations

- Enumerations are true sets (not at all like C/C++s enums) and the fact that the Boolean type is in fact:

  ```
  type Boolean is (FALSE, TRUE);
  ```

  should give you a feeling for the power of the type.
  You have already seen a range in use (for strings), it is expressed as `low .. high` and can be one of the most useful ways of expressing interfaces and parameter values, for example:

  ```
  type Hours   is new Integer range 1 .. 12;
  type Hours24 is range 0 .. 23;
  type Minutes is range 1 .. 60;
  ```

Subtypes for ranges

Another definition for `Hours`:

```
type Hours24 is new range 0 .. 23;
subtype Hours is Hours24 range 1 .. 12;
```

A subtype cannot extend the range beyond its parent, so `range 0 .. 25` would have been illegal.

Combining enumerations and ranges (1)

```
type All_Days is (Monday, Tuesday, Wednesday,
          Thursday, Friday, Saturday, Sunday);
subtype Week_Days is All_Days
                     range Monday .. Friday;
subtype Weekend is All_Days range
                   Saturday .. Sunday;
```

Combining enumerations and ranges (2)

We can now take a `Day`, and see if we want to go to work:

```
Day : All_Days := Today;

if Day in Week_Days then
  go_to_work;
end if;
```

Or you could use the form
`if Day in range Monday .. Friday` and we would not need the extra types.

Enumeration type handling (1)

Ada provides four useful attributes for enumeration type handling:

Succ — this attribute supplies the 'successor' to the current value, so the `'Succ` value of an object containing `Monday` is `Tuesday`.
Note: If the value of the object is `Sunday` then an exception is raised, you cannot `Succ` past the end of the enumeration.

Pred — this attribute provides the 'predecessor' of a given value, so the `'Pred` value of an object containing `Tuesday` is `Monday`.
Note: the rule above still applies `'Pred` of `Monday` is an error.

Enumeration type handling (2)

Val — this gives you the value (as a member of the enumeration) of element n in the enumeration. Thus `Val(2)` is `Wednesday`.
Note: the rule above still applies, and note also that `'Val(0)` is the same as `'First`.

Pos — this gives you the position in the enumeration of the given element name. Thus `'Pos(Wednesday)` is 2.
Note: the range rules still apply, also that `'Last` will work, and return `Sunday`.

Enumeration type handling (3)

```
All_Days'Succ(Monday) = Tuesday
All_Days'Pred(Tuesday) = Monday
All_Days'Val(0) = Monday
All_Days'First = Monday
All_Days'Val(2) = Wednesday
All_Days'Last = Sunday
All_Days'Succ(All_Days'Pred(Tuesday)) =
                                      Tuesday
```

Enumeration type handling (4)

Ada also provides a set of 4 attributes for range types, these are intimately associated with those above and are:

First — this provides the value of the first item in a range. Considering the range `0 .. 100` then `'First` will obviously be `0`.

Last — this provides the value of the last item in a range, and so considering above, `'Last` is `100`.

Length — this provides the number of items in a range, so `'Length` is actually `101`.

Range — this funnily enough returns in this case the value we gave it, but you will see when we come onto arrays how useful this feature is.

Arrays

- Arrays in Ada make use of the range syntax to define their bounds.
- Arrays can be of any type.
- Arrays can even be declared as unknown size.

Some example (C)

```
char name[31];
int  track[3];
int  dbla[3][10];
int  init[3] = { 0, 1, 2 };
typedef char[31] name_type;
track[2] = 1;
dbla[0][3] = 2;
```

Some example (Ada)

```
Name   : array (0 .. 30) of Character; -- OR
Name   : String (1 .. 30);
Track  : array (0 .. 2) of Integer;
DblA   : array (0 .. 2) of array (0 .. 9) of
                                   Integer; -- OR
DblA   : array (0 .. 2,0 .. 9) of Integer;
Init   : array (0 .. 2) of Integer := (0, 1, 2);
type Name_Type is array (0 .. 30) of Character;
track(2) := 1;
dbla(0,3) := 2;

-- Note try this in C.
a, b : Name_Type;
a := b; -- will copy all elements of b into a.
```

Indexes (1)

non-zero based ranges

Because Ada uses ranges to specify the bounds of an array then you can easily set the lower bound to anything you want, for example:

```
Example : array (-10 .. 10) of Integer;
```

Indexes (2)

non-integer ranges

The ranges above are all integer ranges, and so we did not need to use the correct form which is:

```
array(type range low .. high)
```

which would make Example above `array(Integer range -10 .. 10)`. Now you can see where we're going, take an enumerated type, `All_Days` and you can define an array:

```
Hours_Worked : array (All_Days
              range Monday .. Friday);
```

Unbounded array types (1)

One of Ada's goals is reuse, and to have to define a function to deal with a 1..10 array, and another for a 0..1000 array is silly. Therefore Ada allows you to define unbounded array types. An unbounded type can be used as a parameter type, but you cannot simply define a variable of such a type.

```
type Vector is array (Integer range <>)
               of Float;

procedure sort_vector
          (sort_this : in out Vector);
Illegal_Variable : Vector;
Legal_Variable   : Vector(1..5);
```

Unbounded array types (2)

```
subtype SmallVector is Vector(0..1);
Another_Legal     : SmallVector;
```

This does allow us great flexibility to define functions and procedures to work on arrays regardless of their size, so a call to `sort_vector` could take the `Legal_Variable` object or an object of type `SmallVector`, etc.
Note that a variable of type `Smallvector` is constrained and so can be legally created.

Array range attributes

If you are passed a type which is an unbounded array then if you want to loop through it then you need to know where it starts. So we can use the range attributes:

```
Example : array (1 .. 10) of Integer;

for i in Example'First .. Example'Last loop
for i in Example'Range loop
```

Note that if you have a multiple dimension array then the above notation implies that the returned values are for the first dimension, use the notation
`Array_Name(dimension)'attribute`
for multi-dimensional arrays.

Initialisation by range (Aggregates)

When initialising an array one can initialise a range of elements in one go:

```
Init : array (0 .. 3) of
            Integer := (0 .. 3 => 1);
Init : array (0 .. 3) of
            Integer := (0 => 1, others => 0);
```

The keyword **`others`** sets any elements not explicitly handled.

Slicing (1)

Array slicing is something usually done with memcpy in C/C++. Take a section out of one array and assign it into another.

```
Large : array (0 .. 100) of Integer;
Small : array (0 .. 3) of Integer;

-- extract section from one array
-- into another.
Small(0 .. 3) := Large(10 .. 13);

-- swap top and bottom halfs of an array.
Large := Large(51 .. 100) & Large(1..50);
```

Slicing (2)

Note: Both sides of the assignment must be of the same type, that is the same dimensions with each element the same. The following is illegal.

```
-- extract section from one array into another.
Small(0 .. 3) := Large(10 .. 33);
--                    ^^^^^^^^ range too big.
```

Records

```
struct _device {
  int   major_number;
  int   minor_number;
  char  name[20];
};
typedef struct _device Device;

type struct_device is
  record
    major_number : Integer;
    minor_number : Integer;
    name         : String(1 .. 19);
  end record;
type Device is new struct_device;
```

Initialization of members (1)

```
Device lp1 = {1, 2, "lp1"};

lp1 : Device := (1, 2, "lp1");
lp2 : Device := (major_number => 1,
                 minor_number => 3,
                 name         => "lp2");
tmp : Device := (major_number => 255,
                 name         => "tmp");
```

Initialization of members (2)

When initialising a record we use an *aggregate*, a construct which groups together the members. This facility (unlike aggregates in C) can also be used to assign members at other times as well.

```
tmp : Device;
-- some processing
tmp := (major_number => 255, name => "tmp");
```

Default values for record members

```
type struct_device is
  record
    major_number : Integer            := 0;
    minor_number : Integer            := 0;
    name : String(1 .. 19)  := "unknown";
  end record;
```

Ada Statements — outline

1. First View
2. Ada Types
3. Ada Statements
   - C/C++ statements to Ada
   - Compound Statement
   - **`if`** Statement
   - `switch` Statement
   - Ada loops
   - return
   - labels and goto
   - exception handling

Compound statement

A compound statement is also known as a block and in C allows you to define variables local to that block, in C++ variables can be defined anywhere. In Ada they must be declared as part of the block, but must appear in the declare part just before the block starts.

C

```
{
  declarations
  statements
}
```

Ada

```
declare
  declarations
begin
  statement
end;
```

IF statement

If statements are the primary selection tool available to programmers. The Ada if statement also has the 'elsif' construct (which can be used more than once in any if statement), very useful for large complex selections where a switch/case statement is not possible.

C

```
if (expression)
{
  statement
} else {

  statement
}
```

Ada

```
if expression then
  statement
elsif expression then
  statement
else
  statement
end if;
```

SWITCH statement

The switch or case statement is a very useful tool where the number of possible values is large, and the selection expression is of a constant scalar type.

C

```
switch (expression)
{
  case value: statement
  default:    statement
}
```

Ada

```
case expression is
  when value => statement
  when others => statement
end case;
```

There is a point worth noting here. In C the end of the statement block between case statements is a break statement, otherwise we drop through into the next case. In Ada this does not happen, the end of the statement is the next case.

Ranges of values in SWITCH statement

C

```
switch (value) {
case 1:
case 2:
case 3:
case 4:
  value_ok = 1;
  break;
case 5:
case 6:
case 7:
  break;
}
```

Ada

```
case value is
  when 1..4 => value_ok := 1;
  when 5 | 6 | 7 => null;
end case;
```

Loops

All Ada loops are built around the simple **`loop ... end`** construct

```
loop
  statement
end loop;
```

while Loop

The while loop is common in code and has a very direct Ada equivalent.

```
while (expression)
{
  statement
}

while expression loop
  statement
end loop;
```

do Loop

The do loop has no direct Ada equivalent, though section 1.2.4.5 will show you how to synthesize one.

```
do
{
  statement
} while (expression)

-- no direct Ada equivalent.
```

for Loop(1)

The for loop is another favourite, Ada has no direct equivalent to the C/C++ for loop (the most frighteningly overloaded statement in almost any language) but does allow you to iterate over a range, allowing you access to the most common usage of the for loop, iterating over an array.

```
for (init-statement ; expression-1 ;
                      loop-statement){
   statement
}
```

```
for ident in range loop
   statement
end loop;
```

for Loop (2)

Ada adds some nice touches to this simple statement:

- the variable ident is actually declared by its appearance in the loop, it is a new variable which exists for the scope of the loop only and takes the correct type according to the specified range.
- to loop for 1 to 10 you can write the following Ada code:

```
for i in 1 .. 10 loop
  null;
end loop;
```

- to loop for 10 to 1 you must write the following Ada code:

```
for i in reverse 1 .. 10 loop
  null;
end loop;
```

break and continue (1)

In C and C++ we have two useful statements *break* and *continue* which may be used to add fine control to loops. Consider the following C code:

```
while (expression) {
  if (expression1) {
    continue;
  }
  if (expression2) {
    break;
  }
}
```

break and continue (2)

In Ada there is no *continue*, and *break* is now *exit*.

```
while expression loop
  if expression2 then
    exit;
  end if;
end loop;
```

The Ada exit statement however can combine the expression used to decide that it is required, and so the code below is often found.

```
while expression loop
  exit when expression2;
end loop;
```

break and continue (3)

This leads us onto the *do loop*, which can now be coded as:

```
loop
  statement
  exit when expression;
end loop;
```

break and continue (4)

Another useful feature which C and C++ lack is the ability to 'break' out of nested loops, consider

```
while ( (!feof(file_handle) &&
        (!percent_found    )     ) {
    for (char_index = 0;
         buffer[char_index] != '\n';
         char_index++) {
        if (buffer[char_index] == '%') {
            percent_found = 1;
            break;
        }
        // some other code,
        // including get next line.
    }
}
```

break and continue (5)

This sort of code is quite common, an inner loop spots the termination condition and has to signal this back to the outer loop. Now consider

```
Main_Loop:
  while not End_Of_File(File_Handle) loop
    for Char_Index in Buffer'Range loop
      exit when Buffer(Char_Index) = NEW_LINE;
      exit Main_Loop when
                  Buffer(Char_Index) = PERCENT;
    end loop;
  end loop Main_Loop;
```

Part II

Sub-programs

Outline

Declarations and definitions of sub-programs

The following piece of code shows how C/C++ and Ada both declare and define a function. Declaration is the process of telling everyone that the function exists and what its type and parameters are. The definitions are where you actually write out the function itself. (In Ada terms the function spec and function body).

Declarations and definitions of sub-programs in C

```
return_type func_name(parameters);
return_type func_name(parameters)
{
  declarations
  statement
}
```

Declarations and definitions of sub-programs in Ada

```
function func_name(parameters) return return_type;
function func_name(parameters) return return_type is
  declarations
begin
  statement
end func_name
```

RETURN Statement

Here again a direct Ada equivalent, you want to return a value, then return a value,

```
return value; // C/C++ return
return value; -- Ada return
```

Procedures

Let us now consider a special kind of function, one which does not return a value. In C/C++ this is represented as a return type of void:

```
void func_name(parameters);
```

In Ada this is called a procedure:

```
procedure func_name(parameters);
```

Passing arguments

C/C++:

```
void func1(int  by_value);
void func2(int* by_address);
void func3(int& by_reference); // C++ only.
```

Ada:

```
type int      is new Integer;
type int_star is access int;
procedure func1(by_value     : in     int);
procedure func2(by_address   : in out int_star);
procedure func3(by_reference : in out int);
```

Sub-programs without arguments

```
void func_name();
void func_name(void);
int func_name(void);

procedure func_name;
function func_name return Integer;
```

Parameter passing modes

Ada provides two optional keywords to specify how parameters are passed, **in** and **out**. These are used like this:

```
procedure proc(Parameter : in     Integer);
procedure proc(Parameter :    out Integer);
procedure proc(Parameter : in out Integer);
procedure proc(Parameter :        Integer);
```

If these keywords are used then the compiler can protect you even more, so if you have an **out** parameter it will warn you if you use it before it has been set, also it will warn you if you assign to an **in** parameter.
Note that you cannot mark parameters with **out** in functions as functions are used to return values, such *side affects* are disallowed.

Default parameters

Ada (and C++) allow you to declare default values for parameters, this means that when you call the function you can leave such a parameter off the call as the compiler knows what value to use.

```
procedure Create
  (File : in out File_Type;
   Mode : in     File_Mode := Inout_File;
   Name : in     String    := "";
   Form : in     String    := "");
```

```
Create(File_Handle, Inout_File, "text.file");
```

```
Create(File => File_Handle,
       Name => "text.file");
```