Intermediate Ada

"Beyond 100,000 lines of code you should probably be coding in Ada."
P.J. Plauger

This unit will cover the following topics:

1. Subprograms
2. Arrays
3. Strings
4. Packages
5. Example: A stack

1. Subprograms

Ada has two forms of subprogram: *functions* and *procedures*. Very simple Ada programs consist of a single procedure. Functions differ from procedures in that they return a value. A *procedure call* is a statement, whilst a *function call* is part of an expression.

1.1 Functions

A function is a form of subprogram that can be called as part of an expression. Consider the following example which calculates the square root of a number:

```
function sqrt(x: float) return float is
    r: float;
begin
    -- perform the square root
    return r;
end sqrt;
```

The function begins with the reserved word **function**, followed by the *name* of the function, and the parameters in parentheses. If there is more than one parameter, they are separated by semicolons. The parameter list is followed by the keyword **return**, and the type of the return value. This is followed by **is**, a declarative part, and the body of the function. Here is another example which determines the sign of an integer:

```
function sign(x: integer) return integer is
begin
    if x > 0 then
        return 1;
    elsif x < 0 then
        return -1;
    else
        return 0;
    end if;
end sign;
```

Functions can also be specified recursively:

```
function factorial(n: integer) return integer is
begin
    if n = 0 then
        return 1;
    else
        return n * factorial(n-1);
    end if;
end factorial;
```

1.2 Procedures

The differences between a *function* and a *procedure* are: a procedure starts with **procedure**; it does not return a value; and the parameters must be of three different modes **in**, **out** or **in out**. The mode of a parameter is indicated by following the colon in the parameter declaration by **in**, **out** or **in out**. In the case of functions, the only allowed mode is **in**.

Consider the **function** sqrt converted to a procedure:

```
procedure sqrt(x: in float; r: out float) is
begin
    -- perform the square root
end sqrt;
```

For an example of **in out** consider:

```
procedure increment(x: in out integer) is
begin
    x := x + 1;
end increment;
```

1.3 Named and default parameters

With named parameters, the parameters do not have to be in order. For example:

```
increment(x => x);
f := factorial(n => 4);
sqrt(x => t+0.5, r => r);
```

In the second example, **n** => **4** implies that the value of the parameter **n** is **4**.

1.4 Subprograms and defined types

```
type month is (jan,feb,mar,apr,may,jun,jul,aug,sep,

function next_month(this_month : in month) return month is
begin
    if this_month = month'last then
        return month'first;
    else
        return month'succ(this_month);
    end if;
end next_month;
```

1.5 Nested subprograms

Ada allows subprograms to be nested. For example, in the code below, the function **to_metres** is nested within the procedure **convert**.

```
with x; use x;
procedure convert is
    yards, feet, inches : integer;
    metres : float;

    function to_metres(yds, ft,ins : in integer) return float is
        inches : integer;
        inches_to_metres : constant float := 0.0254;
    begin
        inches := yds * 36 + ft * 12 + ins;
        return float(inches) * inches_to_metres;
    end to_metres;

begin
    put(“yards?“); get(yards);
    put(“feet?“); get(feet);
    put(“inches?“); get(inches);
    metres := to_metres(yards, feet, inches);
    put(“number of metres = ”);
    put(metres, aft = > 3, exp = > 0);
    new_line;
end convert;
```

1.6 Parameters with initial values

Parameters can be given a *default* initial value. For example:

```
    procedure draw_line(width : in integer := 10)
    begin
        for i in 1 .. width
            put(‘-’);
        end loop;
        new_line;
    end draw_line;
```

This can be called in one of two ways:

```
     draw_line();   - default line of width 10
     draw_line(20); - line of width 20
```

1.7 Parameter modes

```
in        - information is only passed into the procedure
out       - to receive information from a procedure
in out    - pass in information which can be changed before
            being passed back out again.
```

⚠ A return statement (with no expression following it) can be used within a procedure to exit the procedure.

1.8 Example: yes-no

The example below calculates the power of 2 of a number input by the user. It uses a **function** *query* to determine if the user wishes to perform another calculation.

```
procedure yesno is
    n : integer;

    function query return boolean is
        ch: character;
    begin
        loop
            get(ch);
            skip_line;
            case ch is
                when 'y' | 'Y' =>
                    return true;
                when 'n' | 'N' =>
                    return false;
                when others =>
                    put("type y or n: ");
            end case;
        end loop;
    end query;

begin
    loop
        get(n);
        put(n**2);
        new_line;
        put("continue? ");
        exit when not query;
    end loop;
end yesno;
```

2. Arrays

2.1 Simple Arrays

Arrays are composite objects. A typical declaration is of the form:

```
    a: array (integer range 1..8) of float;
OR
    a: array (1..8) of float;
```

This declares **a** to be a variable with 6 elements, each of which is of type **float**. Setting each element to a value of zero would be achieved in the following manner:

```
    for i in 1..8 loop
        a(i) := 0.0;
    end loop;
```

Or this can be written concisely using an *array aggregate*.

```
    a := (0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0);
OR
    a := (1|2|3|4|5|6|7|8 => 0.0);
    a := (1..8 => 0.0);
```

Partial arrays can also be specified:

```
    a := (1..4 => 0.0);
    a := (others => 1.0);
```

The use of **others** means all positions that have not yet been explicitly mentioned.

Arrays can also be given an initial value:

```
    a: array (1..8) of float :=
                   (0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0);
```

A *slice* is a means of designating a portion of an array. For example:

```
    a(3..5) := (2.1,4.9,8.7);
```

2.2 Multidimensional Arrays

Arrays can also have two or more dimensions. For example:

```
     aa: array(integer range 1..4, integer range 1..5) of float;
OR
     aa: array(1..4, 1..5) of float;
```

Setting each element to a value of zero would be achieved in the following manner:

```
for i in 1..4 loop
    for j in 1..5 loop
        aa(i,j) := 0.0;
    end loop;
end loop;
```

Multidimensional arrays can also be given an initial value:

```
aa: array (1..4, 1..5) of float := ((0.0,0.0,0.0,0.0,0.0),
                                     (0.0,0.0,0.0,0.0,0.0),
                                     (0.0,0.0,0.0,0.0,0.0),
                                     (0.0,0.0,0.0,0.0,0.0));
```

⚠ A two-dimensional array, and an array of one dimensional arrays work slightly differently from a syntax point-of-view. For example

```
type grid is array(1..10,1..10) of integer;
x: grid;

type vector is array(1..10) of integer;
y: array(1..10) of vector;
```

The first is a 2D array, and is indexed as **`x(i,j)`**. The second is an array of arrays and is indexed as **`y(i)(j)`**. The latter declaration allows specific rows to be treated as separate arrays.

2.3 Array types

The previous arrays have anonymous types. They can also be given *explicit* type names. The previous declaration of array **a**, could have been modified to:

```
type vector is array (1..10) of float;
```

Then **a** could have been declared in the usual manner:

```
a: vector;
```

This allows whole arrays to be assigned. For example:

```
b: vector;
b := a;
```

The following, however, is illegal:

```
a: array (1..10) of float;
b: array (1..10) of float;
b := a;
```

but this is legal because they are of the same type:

```
a,b: array (1..10) of float;
b := a;
```

It is also possible to create more abstract array types, of the form:

```
type vector is array (integer range <>) of float
```

This creates a type named *vector* which is a 1D array of **float** elements with an **integer** index, however the bounds of the index are not provided. This information can be provided when the array is declared:

```
v: vector(1..5);
```

or a subtype could also be defined:

```
subtype v5 is vector(1..5);
v: v5;


type ages is array (1..100) of natural;
how_old : ages;

how_old(1) - the first element

type line is array (1..80) of character;
```

2.4 Arrays as parameters

Array objects may be passed as subprogram parameters *and* returned as function results. For example:

```
type v2 is array (1..10) of integer;
a, b: v2;
```

```
function addthem(a,b: v2) return v2 is
    c: v2;
begin
    for i in 1..10 loop
        c(i) := a(i) + b(i);
    end loop;
    return c;
end addthem;
```

which can be called in the following manner:

```
m, n: v2;
m := (1,2,3,4,5,6,7,8,9,10);
n := addthem(m,(1,1,2,3,5,8,13,21,34,55));
```

2.5 Composite arrays

Consider the following declarations:

```
type piece is (blank, Wpawn, Bpawn, Wknight, Bknight,
               Wbishop, Bbishop, Wrook, Brook, Wqueen,
               Bqueen, Wking, Bking);

type board is array (1..8, 1..8) of piece;
chessboard : board;
```

Each component can take one of the 13 values of type piece. The components are identified by two index values of type integer.

```
chessboard(3,4) := Wpawn;

Wpawn_count : natural := 0;

for row in 1 .. 8 loop
    for column in 1 .. 8 loop
        if chessboard(row, column) = Wpawn then
            Wpawn_count := Wpawn_count + 1;
        end if;
    end loop;
end loop;
```

2.6 Array attributes

Arrays have certain attributes associated with them, for example:

```
type table is array (1..3, 0..7) of float;
results : table;

results'first       lower bound of the first index
results'last        upper bound of the first index
results'first(n)    lower bound of the nth index
results'last(n)     upper bound of the nth index
results'range       the first index range
results'range(n)    the nth index range
results'length      number of values in the first index
results'length(n)   number of values in the nth index
```

2.7 Unconstrained arrays

In an *unconstrained* array, bounds are not specified when the array type is declared.

```
type any_list is array (natural range < >) of integer;
```

An array type called **any_list** which has one dimension, has components of type **integer**, and whose index values are of subtype **natural**. The index range is undefined and is represented by "< >", which is referred to as a *box*. When objects are declared, an index constraint must be given to specify the index bounds. For example:

```
numbers : any_list(1..15);
more_numbers : any_list(0..20);
```

One array type, two different array subtypes. The following example uses the array **any_list**.

```
procedure selection_sort(item : in out any_list) is
    small_pos : natural range item'range;
    smallest : integer;
begin
    -- sort the components of item
    for low in item'first .. item'last - 1 loop
        -- find the smallest remaining component
        small_pos := low;
        smallest := item(low);
        for pos in low + 1 .. item'last loop
            if smallest > item(pos) then
                small_pos := pos;
                smallest := item(pos);
            end if;
        end loop;
        -- swap smallest with item in position low
```

```
            item(small_pos) := item(low);
            item(low) := smallest;
        end loop;
    end selection_sort;
```

Which can be called as:

```
    selection_sort(numbers);
```

2.8 Array operations

The concatenation operator, **&**, works for *all* types of arrays:

```
    n1 : any_list(1..10);
    n2 : any_list(1..20);

    n1 := (1,3,5,7,9,11,13,15,17,19);
    n2 := n1 & n1;
```

This means that the contents of **n2** will be:

```
    1,3,5,7,9,11,13,15,17,19,1,3,5,7,9,11,13,15,17,19
```

The complete spectrum of operators <, <=, >, >= can also be applied to 1D arrays. For example:

```
    "aaa" < "ada"  --true
    "ZZZ" < "ada"  --true
```

3. Strings

3.1 Making strings

A string is declared behind the scenes as:

```
type string is array(positive range <>) of character;
```

... a 1D unconstrained array. String variables and constants can be declared as:

```
line : string(1..80);
greeting : constant string := "hello";
```

A line of asterisks can be declared as:

```
stars : constant string := (1..25 => '*');
```

3.2 String I/O

The standard **get** and **put** can be used with strings, or a whole line can be processed:

```
get_line(line, length)   read in a sequence of characters
put_line(line(line'first .. length));
```

With **get_line**, reading stops when either the end of line or the end of the string is encountered. The index position of the last character read is put into length. If the get function is used, in the form:

```
word : string(1..10);
get(word);
```

Then exactly 10 characters (including blanks, punctuation etc) are read. The data entry operation is not terminated by pressing the RETURN key; if only five characters are entered before the RETURN is pressed, the computer simply waits for the additional 5 characters!

3.3 String Concatenation

Two strings can be concatenated in the following manner, using the **&** operator. For example:

```
str : string(1..10);
str := "darth" & "VADER"
```

3.4 String Comparison

Two strings can be compared for equality using the = operator. For example:

```
if str = "darthVADER" then
    put(str);
end if;
```

This will print the string **str**, only if it is equivalent to "**darthVADER**" (remember case matters). Strings may be compared using any of the relational operators:

```
"=", "\=", "> =", "< =", ">", "<"
```

3.5 Wide strings

Ada95 defined a **wide_character** type to deal with 65,536 characters from other character sets, and therefore **wide_string** defines extended string types.

3.6 String and character handling

Ada defines a series of string and character handling functions in the package **ada.characters.handling**. For characters the following functions are defined:

- is_control, is_graphic, is_letter, is_lower, is_upper, is_basic, is_digit, is_decimal_digit, is_hexadecimal_digit, is_alphanumeric, is_special

For characters and strings, the following basic functions are defined: to_lower, to_upper

3.7 2D strings

There can also be arrays of composite types, for example:

```
subtype line is string(1..80);
screen : array(1..24) of line;
```

3.8 Variable length strings

If a string in Ada is 10 characters in length, then it is always 10 characters - unlike C, there is no end-of-string character. One way around this in Ada 2005 is the use of unbounded strings via the **ada.strings.unbounded** package. This allows strings of variable size to be input. For example, consider the following code:

```
with ada.Text_IO; use Ada.Text_IO;
with ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with ada.strings.unbounded; use ada.strings.unbounded;
with ada.strings.unbounded.Text_IO; use ada.strings.unbounded.Text_IO;

procedure varstr is
    type p_table is array(1..1000) of unbounded_string;
    p : p_table;
    s : unbounded_string;

begin
    for i in 1..5 loop
        get_line(s);
        p(i) := s;
        put_line(p(i));
        put(length(s));
    end loop;
end varstr;
```

Here a type **p_table** is created which is an array of 1000 unbounded strings. **p** is then created as an instance of **p_table**. **s** is a single unbounded string. In the loop within the program body, **get_line** is used to read in a string into **s**, then **s** is assigned to one of the elements of **p**. **put_line** prints out each element of **p**, and **length(s)** prints the actual length of the input string.

4. Packages

An Ada program is composed of separate units, some of which are existing library packages. Packages provide a mechanism for organizing complex or large programs. In reality they are similar to libraries in C. The initial context clause:

```
with text_io;
use text_io;
```

Consists of a **with** clause followed by a **use** clause. The **with** clause makes the contents of the package available. The **use** clause can be omitted, but the package name would have to be used when using subprograms:

```
text_io.put(“hello”);

package int_io is new integer_io(integer); use int_io;
package real_io is new float_io(float); use real_io;
```

These declarations are known as the “instantiation of a generic package”. Packages consist of a specification and a body.

A **with** clause must precede its corresponding **use** clause.

4.1 Package specification

The package *specification* is the interface into the package and is of the form:

```
package basic_drawing is
    procedure draw_line(width : in integer := 10);
end basic_drawing;
```

4.2 Package bodies

The body of a package defines the implementation details of the package. The body of package **basic_drawing** is defined as follows:

```
package body basic_drawing is
    procedure draw_line(width : in integer := 10)
    begin
        for i in 1 .. width
            put(‘-’);
        end loop;
        new_line;
    end draw_line;
end basic_drawing;
```

4.3 The use clause

Items declared in a package can be referenced using the package name in conjunction with the referenced item, e.g. **basic_drawing.draw_line**. However this is often too cumbersome to use, hence the **use** clause.

4.4 Separate compilation

A package specification is placed in a file named:

```
basic_drawing.ads
```

The corresponding body is placed in a normal **.adb** file with the same name as the package:

```
basic_drawing.adb
```

When a file using the package it compiled, the package itself is incorporated.

5. Example: A stack

The following code illustrates the creation of a stack package. The specification package is stored in **stack.ads**:

```
package stack is
    procedure push(x : in character);
    procedure pop( x : out character);
    function stack_is_empty return Boolean;
    function stack_top return character;
    procedure reset_stack;
end stack;
```

The body package is stored in **stack.adb**:

```
with Ada.Text_IO; use Ada.Text_IO;
package body stack is
    type list is array(1..100) of character;
    type char_stack is
        record
            item : list;
            top : natural := 0;
        end record;
    st : char_stack;

    procedure push(x : in character) is
    begin
        if s.top = 100 then
            put_line("stack is full");
        else
            st.top = st.top + 1;
            st.item(st.top) := x;
        end if;
    end push;

    procedure pop( x : out character) is
    begin
        if s.top = 0 then
            put_line("stack is empty");
        else
            x := st.item(st.top);
            st.top = st.top - 1;
        end if;
    end pop;
```

```
        function stack_is_empty return Boolean is
        begin
            return st.top = 0;
        end stack_is_empty;

        function stack_top return character is
        begin
            if st.top = 0 then
                put_line(“stack is empty”);
                return ‘ ‘;
            else
                return st.item(st.top);
            end if;
        end stack_top;

        procedure reset_stack is
        begin
            st.top := 0;
        end reset_stack;

    end stack;
```

As an example of how it could be used:

```
    with Ada.Text_IO; use Ada.Text_IO;
    with stack; use stack;

    procedure stack_eg is
        op : character;
    begin
        push(‘+’);
        pop(op);
        put(op);
    end stack_eg;
```

5.1 Multiple stacks in an ADT

To create an ADT for multiple stacks:

```
    package char_stack is
        type values is array(1..100) of character;
        type stack is
            record
                item : values;
                top : natural := 0;
            end record;
        procedure push(x : character; st : in out stack);
        procedure pop(x : out character; st : in out stack);
    end char_stack;
```

As an example:

```
op_stack, temporary : stack;
op : character;

push('+', op_stack);
pop(op, op_stack);
```

A possible package body to fit the specification is:

```
package body char_stack is

    procedure push(x : in character; st : in out stack) is
    begin
        if st.top = 100 then
            put_line("stack is full");
        else
            st.top = st.top + 1;
            st.item(st.top) := x;
        end if;
    end push;

    procedure pop (x : out character; st : in out stack) is
    begin
        if st.top = 0 then
            put_line("stack is empty");
        else
            x := st.item(st.top);
            st.top = st.top - 1;
        end if;
    end pop;

end char_stack;
```

5.1 Package privacy

Ada allows types to be declared to be **private**.

```
package char_stack is
    type stack is private;
```

```
        procedure push(x : character; st : in out stack);
        procedure pop(x : out character; st : in out stack);
        private
            type values is array(1 .. 100) of character;
            type stack is
                record
                    item : values;
                    top : natural := 0;
                end record;
    end char_stack;
```

The identifier **stack** is declared in the visible part of the package, but because it is declared to be **private**, no details of its structure are given. The declaration is given in the private part of the package specification. The keyword **private** is used to separate the visible parts of the package from the private declarations. There is only one occurrence of the keyword **private**.

Details of the structure of the type stack cannot be used outside the package.