Quick Ada

Email: dale@rmit.edu.au

The postscript files for the notes... (and don't i just wish they were in one file!).

1 The history of Ada

Overview

An understanding of the reasons why Ada was developed and the history of its development gives an appreciation of the language and its future.

History of Ada.

In 1974 the US Department of Defence (DoD) realised that it was spending too much time, effort and money developing and maintaining embedded computer systems (systems stuck in hardware e.g. missile guidance systems).

At this time over 450 different languages or language extensions were in use. This increased the time and costs for developing new systems and in continually retraining people to become familiar with existing systems. Maintenance was also hampered by the lack of standardisation of support tools (editors, compilers etc). All these factors led to the DoD realising it needed a single powerful language that could be used by all embedded computer suppliers.

The developement work began in 1975 with the DoD producing a list of language requirements which was widely circulated; however no existing language specified the criteria so in 1977 DoD requested proposals for a new language. Unlike committee languages such as COBOL, the new language was the subject of a competition and extensive industry and academic review.

Of numerous entries four were selected for further refinement. This was later cut down to two competing entries from which one was finally selected from the company Cii-Honeywell Bull. This language was christened Ada. The design team was led by Jean Ichbiah who had overall control over the language.

In 1983 the language became an ANSI standard ANSI/MIL-STD-1815A. It became an ISO standard the following year. The language is defined in a reference manual often referred to as the LRM. References to this manual occur often in books on the language, and in many compiler error messages. This book is recommended for any Ada site; although hard to read it is the final authority for any Ada question (an ongoing group has been formed to clarify any inconsistancies detected in the language).

The language has since undergone revision, with ISO standardisation of the new standard achieved in early 1995. This new Ada fixes many of the flaws in the original language, and extends it in many useful ways.

To prevent the proliferation of various imcompatable versions of Ada the Ada Joint Program Office (the body set up for control of the language) took a very novel position - they trademarked the name Ada. You were not allowed to market "Ada" compilers unless they have passed a compliance test. This has subsequently been relaxed, the protected term now being `Validated Ada'.
The resulting Ada validation certificate is limited in duration and has an expiry date. Once it expires the

compiler can no longer be marketed as a `Validated Ada' compiler. In this way the AJPO has ensured that all currently marketed compilers comply with the current standards.

The aim is to ensure that any Ada program can be compiled on any system - in this regard the AJPO has succeeded better than many other language groups.

Design Goals

From the Ada LRM:
"Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency"

Of note is the sentence, also from the LRM:

"Hence emphasis was placed on program readability over ease of writing".

These design goals can be seen in the language. It has strong typing and enforcable abstractions which have shown to increase reliability and ease maintenance.

It eschews cryptic syntax for a more verbose English style for the sake of readability (readability, programming as a human activity). Also almost all constructs can be efficiently implemented.

2 Sample programs

Overview

This chapter shows a few simple programs to give the 'feel' of an Ada program.

Simple programs

One of Ada's major philosophies is encapsulation. All of the standard I/O routines come presupplied in packages that can be included in a program. The following examples use the text_io package.

```
with Ada.Text_IO;       use Ada.Text_IO;
        -- a package containing the"put_line" procedure

procedure hello is  -- candidate for the "main" procedure.
begin
        put_line("hello");
end;
```

Unlike C which has a function main, and Pascal which has a program, any parameterless procedure can be a "main" routine. The procedure thus designated is chosen at link time.

```
with Ada.Text_IO;       use Ada.Text_IO;
with Hello;
        -- include our previous procedure

procedure your_name is
        name    :string(1..100); -- 100 character array
        last    :natural;        -- can only contain natural integers
begin
        put("Hello what is your name? ");
        get_line(name,last);

        for i in 1..10 loop            -- i is implicity declared
             Hello;
               put_line(" there " & name(1..last));
                                    -- & string concatenation
                                      -- name(1..last)- array slice
        end loop;                     -- control structure labelled
end;
```

Inputting numbers requires the use of a package devoted to the task. A package to do this exists with all Ada implementations. Other simpler packages are often created within a site (see package simple_io in the appendices).

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO

procedure Age is

        Age     :integer range 0..120;
```

```
begin
        Put("hello, how old are you ? ");
        Get(Age); -- might cause an exception if value entered
                  --is outside range

        if Age < 18 then
                put_line("ha! you're just a baby");
        elsif Age < 60 then                              -- note spelling of elsif
                put_line("working hard?");
        else
                put_line("Now take it easy old fella!");
        end if;

exception
        when constraint_error =>
                put_line("sorry only ages 0..120 are accepted");
end age; -- procedure name is optional at end;
```

---

to the index...

3 Lexical conventions

Overview

The lexical conventions describe the allowable character sequences that are used to create identifiers, numbers and the special values used in the language. Implementations must support 200 characters lexical elements at the least.

Identifiers

Can be of any length (but can be restricted by an implementation).

Can only contain characters, digits and underscores

Must start with a character

Are case insensitive

E.g.

```
Apple, apple, APPLE -- same identifier
Max_Velocity_Attained
Minor_Number_   -- illegal, trailing underscore
Minor__Revision  -- illegal, consecutive underscores
```

Literals

Literals are the typed representation of the values stored in the program.

Numeric literals can be broken up with non consecutive underscores as desired.

E.g. 5_101_456 is the number 5101456
3.147_252_6

Numeric literals cannot end or start with an underscore.

Exponential notation is available for floats and integers

E.g. 2E6, 9E+4

Numeric literals can be represented in different bases (2-16).

E.g. 2#1011#, 10#45#, 16#Fe23#, 2#11100.11001#

Float literals must have a digit either side of the radix point

E.g. 3.14, 100.0

Thus the numbers 100. or .034 are not valid float numbers.

Character literals are surrounded by single quotes

E.g. 'a', 'b'

String literals are surrounded by double quotes

E.g. "Ada", "literal", "embedded ""strings""!"

String literals cannot contain the tab character. String values can contain them, this can be achieved by either concatenating strings and characters together, or directly inserting the character into the string.

Comments

Comments are introduced by the -- symbol and extend to the end of the line.

Reserved Words

```
abort          else          new            return
abs              elsif         not              reverse
abstract*        end           null
accept             entry                          select
access           exception                        separate
aliased*         exit          of               subtype
all                            or
and            for           others  tagged*
array            function      out               task
at                                           terminate
               generic package then
begin              goto          pragma  type
body                             private
        if             procedure
case           in              protected*      until*
constant                 is              use
                            raise
declare                  range   when
delay              limited record  while
delta              loop          record  while
```

```
digits                                renames
do                mod               requeue*        xor
```

Reserved words followed by an asterisk have been introduced in Ada95.

---

to the index...

---

to the index...

4 Basics types of Ada

Overview

This chapter introduces some of the types available in Ada as well as the operations available on and attributes of those types.

Types and subtypes

```
    All types
           Elementary
                   Scalar
                          Discrete
                                 Universal_integer              -- all integer
    literals
                           Root_integer           -- Ada95 only
                                        Signed integer
                                        Modular integer -- Ada95 nsigned types
                          Enumeration
                                 User defined
                                 Character
                                 Boolean
                   Real
                          Universal_real                 -- all real literals
                          Root_real                              -- Ada95 only
                                 Floating point
                                 Fixed point
                                      ordinary fixed point
                                     decimal fixed point     -- Ada 95 only
                   Access
                     Access-to-object
                          Access-to-subprogram    -- Ada 95 only

           Composite
                  Array
                      String
                     Other array
                Untagged record
            Tagged record                                -- Ada95
                   Task
               Protected                                             -- Ada95
```

Scalar types

The predefined package Standard contains declarations for the standard types such as integer, float, character and boolean, as well as (notionally) defining the operations available on them.

All numeric literals belong to the class universal_integer or universal_float. Many of the attributes of the language (discussed later)

also return a universal value. These universal types are compatable with any corresponding integer, float or fixed type.

E.g.

```
Max_Customers : constant := 10_000; -- 10_000 is a "universal integer"
                                    -- It is not of type "integer"
```

Subtypes can be created in Ada by restricting an existing type, by defining a new type based on an existing type or by enumerating the possible values of the type. A discussion of how to create these new types follows a look at the predefined types and their attributes.

The following operations are defined for all scalar types.

```
=, /=           Equality, inequality
<, <=, >, >=
in, not in      Range membership text
```

Integer types

The following are examples of integer declarations. Here the standard predefined integer type is used.

```
Count                  : Integer;
X,Y,Z                 : Integer;
Amount                      : Integer := 0;
Unity                   : constant Integer := 1;
Speed_Of_Light  : constant := 300_000; -- type universal_integer
A_Month         : Integer range 1..12;

subtype Months is Integer range 1..12;    -- a restricted integer
-- subtypes are compatable with their base type (here integer)
-- i.e. variables of type month can be mixed with integer variables

type File_Id is new Integer; -- a new integer family derived
                             -- from type integer;

type result_range is new Integer range 1..20_000;
-- a derived type with a constraint

type other_result_range is range 1..100_000;
-- a type derived from root_integer
-- the compiler chooses an appropriate sized integer to suit the range.
```

The following operators are also defined for all integer types.

```
+,-,*,/
**              Exponentiation (integer exponent only)
mod              Modulus
rem             Remainder
abs            Absolute value
```

Floating point types

The following are examples of floating point declarations. Floating point numbers have a relative error.

```
x               : float;
```

```
a,b,c           : float;
pi              : constant float := 3.14_2;
Avogadro        : constant := 6.027E23; -- type universal_float

subtype temperatures is float range 0.0..100.0;
type result is new float range 0.0..20_000.0;

type Velocity is new Float;
type Height is new Float;
-- can't accidentally mix velocities and heights without an explicit
-- type conversion.

type Time is digits 6 range 0.0..10_000.0;
-- six decimal digits of accuracy required, in this range.

type Degrees is digits 2 range -20.00..100.00;
-- two decimal digits of accuracy required.
```

The following operators are also defined for all float types.

```
+,*,/,-
**              Exponentiation (integer exponent only)
abs             Absolute value
```

Fixed point types

The following are examples of fixed point declarations. Fixed point numbers have a bounded error, the absolute value of which is called the delta of the type.

```
type Volt is delta 0.125 range 0.0 .. 255.0;

type Fraction is delta System.Fine_Delta range -1.0..1.0;   -- Ada95
        -- Fraction'last = 1.0 - System.Fine_Delta

type Money is delta 0.01 digits 15; -- decimal fixed point
subtype Salary is Money digits 10;
```

The last example shows the usefulness of fixed point types - the ability to specify exactly how accurate the type should be. This allows control over facilities such as errors in rounding expressions, for example.

Enumeration types

An enumeration type is defined by listing all the possible values of the type.

```
type Computer_Language is (Assembler, Cobol, Lisp, Pascal, Ada);
type C_Letter_Languages is (Cobol, C);
```

Values of this type can be defined as follows:

```
a_language              : computer_language;
early_language  : computer_language := cobol;
first_language  : constant computer_language := assembler;
example                 : c_letter_language := cobol;
```

Note that Ada can distinguish between enumeration literals from different types in most cases by examining the context. If this is not

possible then type qualification must be used.

Enumeration types are useful to encode simple control codes used internally in a program.

There are two predefined enumerated types in the package STANDARD, the type character and the type boolean.

Booleans

The two values of boolean variables is true and false.

The following opeartors can be used with boolean types

```
and or not xor /=  =  'and then'  'or else'
```

Ada will not allow an unparenthesied expression to contain both and's and or's. This decreases the likelihood of misreading the intent of a complicated boolean expression.

E.g.

```
(a < b) and (b > c) or (d < e)    -- illegal
((a < b) and (b > c)) or (d < e)  -- ok
```

Usually when evaluating a boolean expression, the compiler is free to rearrange the evaluation of the terms as it sees fit. Both terms will be evaluated. For example in the following either term may be evaluated first.

```
if  a < b and c > d then ...
```

However in some instances we wish to evaluate the terms in a defined order, and stop evaluations as soon as the value of the expression can be determined.

For example

```
if a /= 0  and then b/a > 5.0 then . . .
```

Here we see if a is non zero before further evaluation.

The 'or else' statement is similar, only evaluation stops as soon as a term evaluates to true. This can be useful, for example, in a recursive search of a tree.

E.g.

```
return Present(Node.Left, Key) or else Present(Node.Right, Key);
```

Character

Ada83 initially had 7 bit characters. This restriction was eased before Ada95 arrived, but is still enforced by older compilers such as the Meridian Ada compiler. This creates problems when attempting to display graphic characters on a PC; generally you have to use integers to display characters above Ascii 127, using special routines supplied by the compiler vendor.

Ada95's Character type is based on Latin-1 and provides for 256 character positions. Ada95 also supports wide characters (ISO 10646 Basic Multilingual Plane (BMP)) and so all modern compilers can cope with 8 bit characters.

The 7 bit character set is described in the obsolecent package Standard.Ascii. The 8 bit character set is described in the package Standard. The package Ada.Characters.Latin_1 provides usable names for the characters.

Subtypes

We can restrict the range of values a variable can take by declaring a subtype with a restricted range of values (this corresponds to Pascal's user defined types). Any attempt to place an out-of-range value into a variable of a subtype results in an exception (program error). In this way program errors can be discovered. The syntax for a subtype declaration is

```
subtype Name is Base_Type;
subtype Name is Base_Type range lowerbound . . upperbound;
```

Examples of declaring subtypes are given below.

```
type Processors is (M68000,  i8086, i80386, M68030, Pentium, PowerPC);
subtype Old_Processors is Processors range M68000..i8086;
subtype New_Processors is Processors range Pentium..PowerPC;

subtype Data is Integer;
subtype Age is Data range 0 . . 140;
subtype Temperatures is Float range -50.0 .. 200.0;
subtype Upper_Chars is Character range 'A' .. 'Z';
```

Subtypes are compatable with their base types . They can be placed in the same place as any variable of the base type can. Also variables of different subtypes that are derived from the same base type are compatable.

```
My_Age  : Age;
Height  : Integer;

Height := My_Age;       -- silly, but never causes a problem.

My_Age := Height;       -- will cause a problem if height's
                        -- value is outside the range of
                        -- my_age (0..140), but still
                        -- compilable.
```

Derived types

When subtypes are created they are still compatable with their base type. Sometimes we may wish to create distinctly new types that are not associated with the original type at all. This concept of type is very different to that provided by Pascal.

To do this we create a derived type from a parent type using the following syntax

```
type Name is new Parent_Type;
type Name is new Parent_Type range lower bound . . upper bound;
```

A derived type is a completely new type and is incompatable with any other type, even those derived from the same parent type.

Derived types should be used when the modelling of a particular object suggests that the parent type is inappropriate, or you wish to partition the objects into distinct and unmixable classes.

```
type Employee_No is new Integer;

type Account_No is new Integer range 0..999_999;
```

Here employee_no's and account_no's are distinct and unmixable, they cannot be combined together without using explicit type conversion. Derived types inherit any operation defined on the base type. For example if a record was declared that had procedures push and pop, a derived type could be declared that would automatically have inherit the procedures.

Another important use of derived types is to produce portable code. Ada allows us to create a new level of abstraction, one level

higher than, for example, the abstraction of Integer over a series of bits.

This is specified by using derived types, without a parent type.

```
type Name is range <some range>;
```

For example,

```
type Data is range 0..2_000_000;
```

Here the compiler is responsible for choosing an appropriately sized integer type. On a PC, it would be a 32 bit size, equivalent to long_integer. On a Unix workstation it would still be a 32 bit integer, but this would be equivalent to an integer. Letting the compiler choose frees the programmer from having to choose. Compiling it on a new host does not require changing the source code.

Type conversion

Despite the usefullness of being able to create distinct types, there are still occasions where we wish to convert from one type to another. One typical instance is to convert from one integer to float, or vice versa.

```
X       : Integer:= 4;
Y       : Float;

Y := float(X);
. . .
X := Integer(Y);
```

This causes the compiler to insert the appropriate code for type conversion (if needed) as part of the translation.

Do not confuse this with unchecked conversions (covered later) which often perform no internal representation transformation.

It needs to be stressed however that types are created distinct for a reason and that attempts to subvert the compiler's checks by performing type conversions should be either discouraged or performed only when semantically meaningfull.

Type Qualification

In some situations an expression's or value's type can be ambiguous.

For example,

```
type primary is (red, green, blue);
type rainbow is (red, yellow, green, blue, violet);
...
for i in red..blue loop -- this is ambiguous
```

Here we need to specify precisely what type is required. This is done with type qualification.

```
for i in rainbow'(red)..rainbow'(blue) loop
for i in rainbow'(red)..blue loop -- only one qualification needed
for i in primary'(red)..blue loop
```

Type qualification does not change a value's type. It merely informs the compiler of what type the programmer thinks it should be.

Attributes

Ada also provides the ability to enquire about a type or object from within the code by using attributes. Some of the attributes for discrete types are

```
Integer'first                  -- the smallest Integer
Integer'last                    -- the largest integer
Processors'succ(M68000)   -- successor of the M68000
Upper_Chars'pred('C') -- the predecessor of 'C' ('B')
Integer'image(67)               -- the string " 67" -- space for a '-'
Integer'value("67")              -- the integer 67.
Processors'pos(M68030)        -- the position of M68030 in the type.
                         -- (3, position 0 is first).
```

An example of the use of an attribute is

```
subtype Positive is Integer range 1..Integer'last;
```

Here we achieve a maximal positive integer range without introducing any system dependent features.

In Ada83 non discrete types such as float, fixed and all their subtypes and derived types, the concepts of pred, succ and pos do not have meaning. In Ada95 they do. All other scalar attributes apply to the real types.

---

to the index...

---

to the index...

5 Control Structures

Overview

The control structures of Ada are similar in style to most conventional languages. However some differences remain.

As usual Ada control structures are designed for maximum readability, all control structures are clearly ended with an 'end something'.

If statements

All if statements end with an end if statement.

```
if boolean expression then
        statements
end if;

if boolean expression then
        statements
else
        other statements
end if;
```

To prevent the common sight of if's marching across the page there is the elsif structure. As many elsifs as required can used. Note the spelling of elsif carefully.

```
if boolean expression then
        statements
elsif boolean expression then
        other statements
elsif boolean expression then
        more other statements
else
        even more other statements
end if;
```

The final else is optional in this form of the if.

Case statements

The case statement must have an action for every possible value of the case item. The compiler checks that this is the case. In situations where it is impractical to list every possible value the others case label should be used.

Each choice's value can be either a single value (e.g. 5), a range (1..20) or a combination of any of these, separated by the character '|' .

Each of the case values must be a static value i.e. it must be able to be computed at compile time.

```
case expression   is
    when choices => statements
      when choices => statements
      . . .
   when others    => statements
end case;
```

Important The others option is mandatory in a case statement unless all possible values of the case selector have been enumerated in the when statements.

```
case letter is
  when 'a'..'z'| 'A'..'Z' => put ("letter");
      when '0'..'9'           => put ("digit! value is"); put (letter);
       when ''' | '"' | '`'    => put ("quote mark");
  when '&'                => put ("ampersand");
   when others             => put ("something else");
end case;
```

Each of the case values must be a static value i.e. it must be able to be computed at compile time.

Loops

All Ada looping constructs use the loop/ end loop form. Several variations exist. The exit statement can be used to break out of loops.

Simple Loops

The simple loop is an infinite loop. It is usually used in conjuction with the exit statement.

```
loop
    statements
end loop;
```

While Loops

The while loop is identical to the Pascal while loop. The test is performed before the loop is entered.

```
while boolean expression loop
        statements
end loop;
```

For Loops

The for looping contructs are similar to those in Pascal.

There are several rules that apply to the use of for statements.

l The index in the for loop must be a discrete type - floats are not allowable.

l The index is not explicity declared.

l The index cannot be modified by any statements (read only)

Note that the statements will not be executed if the lower value of the range is higher than the upper value.

Important The index used in the for loop does not need to be declared. It is implicitly declared to be of the same type as the range.

```
for index in range loop
 statements
end loop;


for  i in 1..20 loop
 put (i);
end loop;
```

To count backwards...

```
for index in reverse range loop
        statements
end loop;

for i in reverse 1..20 loop
        put(i);
end loop;
```

A type can be used as a range.

```
declare
 subtype list is integer range 1..10;
```

```
begin
       for i in list loop
              put(i);
 end loop;
end;
```

Here the type list is being used as a range. In a similar manner an enumerated type can be used.

Exit and exit when

The exit and exit when statements can be used to exit loops prematurely. Execution continues with the first statement following the loop. The two forms have identical effects. The following code segments are identical.

```
loop
    statements
      if boolean expression then
              exit;
   end if;
end loop;

loop
    statements
      exit when boolean expression;
end loop;
```

Labeled loops

An exit statement will normally only exit the inner most loop in which it is enclosed. We can label loops and modify the exit statement accordingly to allow for an escape from a series of nested loops. In all cases the instruction next executed is that following the loop exited.

```
outer_loop:
loop
 statements
      loop
            statements
              exit outer_loop when boolean_expression;
                end if;
 end loop;
end loop outer_loop;
```

Note that the end loop statement is also labelled.

Goto statement

The goto statement is provided in Ada for use in exceptional situations.

```
goto label;

<<label>>
```

The use of goto's is very restrictive and quite sensible. You cannot jump into if statements, loop statements or, unlike Pascal, out of procedures.

---

to the index...

---

to the index...

6 Arrays

Overview

Most languages provide arrays of one sort of another. Those provided by Ada are most similar to Pascal's, with the inclusion of several very handy features.

Unconstrained arrays, dynamic arrays and array attributes are some of the extras offered.

Simple arrays

Generally when declaring an array, a type definition would be created first, and an array would be declared using this definition.

```
type Stack is array (1..50) of Integer;
Calculator_Workspace : stack;

type stock_level is Integer range 0..20_000;
type pet is (dog,budgie,rabbit);
type pet_stock is array(pet) of stock_level;

store_1_stock:pet_stock;
store_2_stock:pet_stock;
```

In general the declaration of an array has this form:

```
type array_name is array (index specification) of type;
```

Some points to note:

* The index specification can be a type (e.g. pet).

* The index specification can be a range (e.g. 1..50).

Index values must be of a discrete type.

Anonymous arrays

Arrays can also be declared directly, without using a predefined type.

```
no_of_desks : array(1..no_of_divisions) of integer;
```

This is known as an anonymous array (as it has no explicit type) and is incompatable with other arrays - even those declared exactly the same. Also they cannot be used as parameters to subprograms. In general it is better to avoid them.

Accessing and setting arrays

To access an element of an array

```
if store_1_stock(dog) > 10 then ...
```

It is instructive to note that accessing an Ada array is indistinguishable from calling a function in all respects.

To store a value into an array...

```
store_2_stock(rabbit) := 200;
```

Array aggregates

The values of an array can all be assigned at once, using an aggregate. An aggregate must specify a value for every array element.

```
store_1_stock := (5,4,300);
```

The values in the aggregate are assigned in order to the values in the array.

It is also possible to use a named version of the aggregate where the individual elements of the array are named.

```
store_1_stock := (dog => 5, budgie => 4, rabbit => 300);
```

It is illegal to combine both notations in the one aggregate.

```
store_1_stock := (5,4,rabbit=>300);     -- illegal
```

Aggregates can also be used in declarations in exactly the same manner, using either notation.

```
store_1_stock:pet_stock := (5,4,300);
```

A discrete range can also be included in the aggregate.

```
store_1_stock := (dog..rabbit=>0);
```

The others option is particuarly useful in setting all elements of an array to a given value. In these situations type qualification is often required.

```
new_shop_stock:pet_stock := (others := 0);
```

Consider the following declarations:

```
declare
 type numbers1 is array(1..10) of integer;
       type numbers2 is array(1..20) of integer;
       a       :numbers1;
      b       :numbers2;
begin
 a := (1,2,3,4, others => 5);
end;
```

The Ada language doesn't like this; if you mix the others option with either positional or named association then you have to qualify it with a type mark:

```
a: = numbers1'(1,2,3,4, others => 5);
```

Constant arrays

Constant arrays can be defined. In this case all elements should be initialised through the use of an aggregate when declared.

```
type months is (jan, feb,mar,....,dec);
subtype month_days is integer range 1..31;
```

```
type month_length is array (jan..dec) of month_days;

days_in_month:constant month_length := (31,28,31,30,...,31);
```

Array attributes

Attributes associated with arrays are as follows.

```
array_name'first        -- lower bound of the array
array_name'last      -- upper bound of the array

array_name'length    -- the number of elements in the array
                  -- array_name'last-array_name'first +1

array_name'range  -- the subtype defined by
                     -- array_name'first . . array_name'last
```

If the array is multi dimensional then the index should be specified e.g. array_name'range(n) supplies the range for the nth index.

This facility is very useful for stepping through arrays

```
for i in array_name'range loop
        ...
end loop
```

This guarantees that every element will be processed.

Unconstrained array types

Unconstrained array types allow us to declare array types that are identical in all respects to normal array types except one - we don't declare how long they are.

The declaration of an unconstrained array defines a class of arrays that have the same element type, the same index type and the same number of indices.

```
subtype positive is integer range 1..integer'last;
type string is array (positive range <>) of character;
```

The <> represents a range that has to be specified when a variable of type string is declared (filling in the blank when declaring a variable).

The type string can be used to define a large class of character arrays, identical except in the number of elements in the array.

To create an actual array, we have to provide a index constraint for the type.

```
My_Name : String (1..20);
```

Here the index constraint is the range 1..20. The advantage of this is that all strings declared are of the same type, and can thus be used as parameter to subprograms. This extra level of abstraction allows for more generalised subprograms.

To process every element of an a variable that is derived from an unconstrained array type requires the use of array attributes such as a'range, a'first etc. as we cannot be sure what index values incoming arrays may have, such as below.

```
My_Name    : String (1..20);
My_Surname : String (21..50);
```

Unconstrained arrays are typically implemented with an object that stores the bounds, as well as a pointer to the actual array.

Standard array operations

There are several operations that can be applied to arrays as a whole and not just the individual components.

Assignment

An entire array can be assigned the value of another array. Both arrays must be of the same type. If they are of the same unconstrained type then they both must have the same number of elements.

```
declare
 my_name :string(1..10)  :="Dale      ";
 your_name       :string(1..10)  :="Russell   ";
 her_name        :string(21..30) :="Liz       ";
 his_name        :string(1..5)   :="Tim  ";
begin
 your_name := my_name;
   your_name := her_name;  -- legal, both have same number of
                                    -- elements
     his_name  := your_name; -- will cause an error, same type but
-- different length
end;
```

Test for equality, inequality

The tests for equality and inequality are available for (almost) every Ada type. Two arrays are equal if each element of the array is equal to the corresponding element of the other array.

```
if array1 = array2 then....
```

Concatenation

Two arrays can be concatenated using the & operator.

```
declare
        type vector is array(positive range <>) of integer;

        a       : vector (1..10);
        b       : vector (1..5):=(1,2,3,4,5);
        c       : vector (1..5):=(6,7,8,9,10);
begin
        a := b & c;
        Put_Line("hello" & " " & "world");
end;
```

Ordering operations

The operators <,<=,>,>= can all be applied to one dimensional arrays. They are of most benefit when comparing arrays of characters.

```
"hello" < "world"       -- returns true
```

Dynamic arrays

The length of an array can be determined at run time, rather than being specified by the programmer when writing the program.

```
declare
 x       : Integer := y  --y declared somewhere else
     a       : array (1..x) of integer;
begin
 for i in a'range loop
           ...
     end loop;
end;

procedure demo(item        :string) is
     copy    :string(item'first..item'last) := item;
 double:string(1..2*item'length) := item & item;
begin
    ...
```

Note that this does not easily allow the user's input to decide the size of the array, and should not be thought of as a device for achieving this. The second example is much more typical of it's use.

---

to the index...

---

to the index...

7 Records

Overview

Once again Ada provides the same facilities as other languages, plus a few extra. Records can have aggregates, discriminants which allow for variant records, variable sized records and initialized variables.

Simple records

In general the declaration of an record has this form:

```
type record name is
        record
                field name 1: type A;
                field name 2: type B;
                ...
                field name n: type N;
        end record;
```

For example:

```
type bicycle is
        record
                frame   :construction;
                maker   :manufacturer;
                front_brake     :brake_type;
                rear_brake      :brake_type;
        end record;
```

Accessing and setting fields

Accessing fields of a record is identical to that in pascal and C, i.e. the name of the variable is followed by a dot and then the field name.

```
expensive_bike  :bicycle;

expensive_bike.frame    := aluminium;
expensive_bike.manufacturer := cannondale;
expensive_bike.front_brake      := cantilever;
expensive_bike.rear_brake       := cantilever;

if expensive_bike.frame = aluminium then ...
```

As with arrays records can be assigned an aggregate, a complete set of values for all record elements.

```
expensive_bike:=(aluminium,cannondale,canitlever,cantilever);
```

Alternatively the aggregate can use positional notation where each element of the record is named.

```
expensive_bike := (
                        frame => aluminium,
```

```
                        manufacturer => cannondale,
                        front_brake => cantilever,
                        rear_brake =>   cantilever
                        );
```

Both notations can be mixed in the one aggregate with the proviso that all positional values precede named values.

Aggregates can also be used in declarations in exactly the same manner, using either notation.

```
expensive_bike :bicycle := (aluminium,cannondale,cantilever,cantilever);
```

The same value can be assigned to different fields by using the '|' character.

```
expensive_bike := (
                    frame => aluminium,
                    manufacturer => cannondale,
                    front_brake | rear_brake => cantilever
                  );
```

Default values

Fields in records can be given default values that are used whenever a record of that type is created (unless it is initialised with different values).

```
type bicycle is
       record
              frame:construction := CromeMolyebdenum;
              maker   :manufacturer;
              front_brake     :brake_type := cantilever;
              rear_brake      :brake_type := cantilever;
       end record;
```

Constant records

Just like normal variables constant records can be created. In this case all fields should be initialised through the use of an aggregate or default field values.

```
my_bicycle      :constant bicycle
                := (    hi_tensile_steel,
                        unknown,
                        front_brake => side_pull,
rear_brake      => side_pull);
```

The fields in a constant record cannot be assigned a value, nor can the record be reassigned a new value.

Discriminants

Ada allows records to contain discriminants. These extra fields help to customise the record further. They give an extra level of abstraction when modelling data; records can be of the same type yet still be different in size or the number of fields.

Variant records

One of the uses of discriminants allows for variant records. In this case the record contains some fields whose existance is dependent upon the value of the discriminant.

For example the discriminant in the following record is the variable vehicle_type

```
type vehicle is (bicycle, car, truck, scooter);

type transport (vehicle_type: vehicle:= car) is
        record
                owner   :string(1..10);
                description:    string(1..10);
                case vehicle_type is
                        when car                =>
                                                petrol_consumption:float;
                        when truck      =>
                                                diesel_consumption:float;
                                                tare:real;
                                                net:real;
                        when others     =>      null;
                end case;
        end record;
```

The discriminant can be supplied when a record is being declared.

```
my_car:transport(car);
my_bicycle:transport (vehicle_type => bicycle);
```

When the discriminant has the value car, the record contains the fields owner, description and petrol_consumption. Attempting to access fields such as tare and net is illegal and will cause a constraint error to be generated (this can be caught and handled using exceptions).

Constrained records

The records above are said to be constrained, that is the discriminant can never change. The my_bicycle record does not have the fields tare, net petrol_consumption etc. nor does the compiler allocate room for them.

We can declare the records without setting an initial value for the discriminant, in which case the record is said to be unconstrained. In this case the discriminant can take on any value throughout its life. Obviously in this case the compiler has to allocate enough room for the largest record possible.

Aggregates for records with discriminants must include the value of the discriminant as the first value.

```
        my_transport:transport;

begin
        my_transport := (car,"dale      ","escort  ",30.0);
```

Unconstrained records

We can create records that do not have a fixed discriminant, it can change throughout the life of the record. However to do this the discriminant of the record must have a default value.

```
type accounts is (cheque, savings);
type account (account_type: accounts:= savings) is
        record
                account_no      :positive;
                title           :string(1..10);
                case account_type is
                        when savings    =>      interest_rate;
                        when cheque     =>      null;
                end case;
        end record;
```

Here the account record discriminant has a default value of savings.

We can declare a record

```
household_account:account;
```

which is created as a savings account. We can change the record type later...

```
household_account:= (cheque,123_456,"household ");
```

Other uses of discriminants

As well as being used as a case selector in a record, discriminants can also be used to specify the length of arrays that are components of the record.

```
type text (length:positive:=20) is
        record
                value:string(1..length);
        end record;
```

In this case the length of the array is dependent upon the value of the discriminant. As described above the record can be declared constrained or unconstrained.

This text record is the usual implementation used for variable length strings and text processing.

---

to the index...

---

to the index...

8 Subprograms

Overview

Sub programs, covering both procedures and functions, are the basis of all programs in Ada. Ada provides features which will be new to Pascal and C programmers. Overloading, named parameters, default parameter values, new parameter modes and return values of any type all make Ada sub programs significantly different.

Procedures

Procedures in Ada are similar to those in Pascal. A procedure can contain return statements.

```
procedure Demo(x:integer; y:float) is
        declarations;
begin
        statements;
end demo;
```

Procedures are called in the normal Pascal style:

```
demo(4, 5.0);
```

Functions

Functions are very similar to procedures except that they also return a value to the calling sub program. The use of return statements is very similar to C. Functions can have as many return statements as required. A function returning a variable of a given type can be used anywhere a variable of that type can be used.

```
function Even( Number : Integer) return boolean is
begin
        if Number mod 2 = 0 then
                return true;
        else
                return false;
        end if;
end;
```

Subprograms in general

Ada has been designed with separate compilation very much in mind. To this end we can produce just a specification of a subprogram and submit this to the compiler. Once compiled and the description stored in an attribute file, it can be checked for compatibility with other procedures (and packages) when they are compiled. By producing a large number of procedure stubs we can pre test the design of a system and pick up any design errors before any more work is done.

A procedure specification for the above procedure would be:

```
procedure demo( x:integer; y:float);
```

If we wish to use a seperately compiled subprogram we can with it in another subprogram.

```
with demo;
procedure use_demo is
        ....
begin
        ...
        demo(4,5);
end use_demo;
```

A function specification may appear as:

```
function Even(Number : Integer) return Boolean;
```

Like Pascal and unlike C, a subprogram can contain nested subprograms within them that are not visible outside the subprogram.

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Display_Even_Numbers is

    -- declarations
    x:integer;

    function even (number:integer) return boolean is
    begin
        return number mod 2 = 0;
    end even;

begin
    for i in 1..10 loop
        if even(i) then
            put(i);
            new_line;
        end if;
    end loop;
```

```
end display_even_numbers;
```

Parameter modes

Ada provides three parameter modes for procedures and functions.

o in

o in out

o out

These modes do not correspond directly to any modes in other languages, they will be discussed below. The following should be noted.

o All parameters to subprograms are by default in.

The parameter passing mechanism is by copy-in, copy-out for in/out scalars. The language specifies that any other types can be passed by copy-in/copy-out, or by reference.

Ada95 mandates that limited private types (see later) are passed by reference, to avoid problems with the breaking of privacy.

In mode

Parameters supplied with this mode are like value parameters in Pascal, and normal parameters in C with the exception that they cannot be assigned a value inside the subprogram. The formal parameter (that in the sub program) is a constant and permits only reading of the value of the associated actual parameter.

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Demo(x : in integer; y : in integer) is
begin
        x := 5; -- illegal, in parameters are read only.
        put(y);
        get(y); -- also illegal
end demo;
```

In out mode

This mode corresponds directly to the var parameters of Pascal. Actual parameters can be used on either the left or the right hand side of statements in procedures. These parameters are effectively read/write.

```
procedure Demo( x : in out integer;
                y : in     integer) is
```

```
    z:constant integer:=x;

begin
    x := z*y; -- this is ok!
end demo;
```

Out mode

The formal parameter is a variable and may be assigned values, however it's initial value is not necessarily defined, and should not be relied upon.

```
procedure demo( x:out integer;
                      y:in integer) is

    z : integer := x; -- not wise, X not initialised
begin
    x := y;

end demo;
```

Caution - out mode parameters that aren't initialized!

```
procedure Location(
           target   : in     key;
           position :    out Small_Integer_Range;
           found    :    out boolean) is
```

Here if the target is not found, then if position is not assigned a value, the copy out parameter mode causes the unitialised value to be placed into the awaiting actual parameter, the associated range check may cause a constraint error.

E.g.

```
declare
    The_Position : Small_Integer_Range;
begin
    Location( Some_Key, The_Position, Result);
    ...
```

If result = false, a constraint error may be generated when the procedure returns.

Named parameters

Normally the association between the formal (defined in the sub program specification) and actual parameters (supplied in the sub program call) is on a one to one basis i.e. it is positional. The first formal parameter is associated with the first actual parameter, etc.

To enhance the readability of sub program calls (Ada is designed to be readable) we can associate the name of the formal parameter and the actual parameter. This feature makes sub program calls immensely more readable.

```
procedure demo(x:integer; y:integer);  -- procedure specification

...
demo(x=>5, y=> 3*45);                -- when calling, associate formal
                                       -- and actual parameters.
```

Lining up the parameters vertically can also be an aid to readability.

```
demo(   x => 5,
        y => 3*45);
```

Because the association is made explicit (instead of the implicit association with positional parameters) there is no need to supply them in the same order as in the sub program specification. There is no restriction on the order in which named parameters are written.

```
demo( y => 3*45, x => 5);        -- the order of named parameters
                                       -- is irrelavent
```

Mixing positional and named parameters

Positional parameters and named parameters can be mixed with the one proviso:
positional parameters must precede the named parameters.

```
procedure square(result :   out integer;
                        number  :in     integer) is
begin
        result:=number*number;
end square;
```

could be called in the following manners:

```
square(x,4);
square(x,number => 4);
square(result => x,number => 4);
square(number => 4, result => x);

square(number => 4, x)   -- illegal as positional follows named.
```

Default parameter values

A default value can be be given for any in parameters in the procedure specification. The expression syntax is the same as that for pre initialised variables and is:

```
with Ada.Text_IO;       use Ada.Text_IO;
procedure print_lines(no_of_lines: integer:=1) is

begin
        for count in 1 . . no_of_lines loop
                new_line;
        end loop;
end print_lines;
```

This assigns a value for no_of_lines if the procedure is called without a corresponding parameter (either positional or named).

E.g. the procedure could be called as

```
print_lines;    -- this prints 1 line.
print_lines(6); -- overrides the default value of 1.
```

Similarly if a procedure write_lines was defined as

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

procedure write_lines(letter    :in     char:='*';
                      no_of_lines:in integer:=1) is

begin
        for i in 1 . . no_of_lines loop
                for i in 1 . . 80 loop
                        put(letter);
                end loop;
                new_line;
        end loop;
end write_lines;
```

then it could be called as

```
write_lines;                            -- default character, default
                                                -- no. of lines.
write_lines('-');                       -- default no. of lines.
write_lines(no_of_lines => 5);  -- default character
write_lines('-',5)                      -- specifying both.
```

Local subprograms

So far the subprograms presented have all been independent compilation units. It is possible to embed subprograms in another subprogram such that only one compilation unit is constructed. These local subprograms can only be referenced from the surrounding subprogram. This is identical to the features provided by Pascal.

```
with Ada.Text_IO;       use Ada.Text_IO;
with Ada.Integer_Text_IO;       use Ada.Integer_Text_IO;

procedure ive_got_a_procedure is

        x       :integer:=6;
    y       :integer:=5;

    procedure display_values(number:integer) is
     begin
           put(number);
            new_line;
       end display_values;

begin
        display_values(x);
      display_values(y);
end ive_got_a_procedure;
```

In this example the scope of procedure display_values is limited to inside the procedure ive_got_a_procedure - it can't be 'seen' or called from anywhere else.

Separate compilation

In the previous example if any change is made to any of the code then both procedures must be resubmitted to the compiler (because they are in the one source file). We can separate the two components into seperate files while still retaining the limited scope of procedure display_values. This is a little like the #include directive in C, but the files are now independent compilation units.

In the first file...

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Ive_got_a_procedure is

        x       :integer:=6;
        y       :integer:=5;

        procedure display_values(number:integer) is separate;

begin
        display_values(x);
        display_values(y);
end ive_got_a_procedure;
```

In the second file...

```
separate(Ive_got_a_procedure)   -- note no trailing semicolon

procedure display_values(number:integer) is
begin
        put(number);
        new_line;
end display_values;
```

Apart from being in another file (and being a seperate compilation unit) the code is identical in all respects to the previous version. However if the inner subprograms change then only they have to be submitted to the compiler. This also allows a program to be broken up into several pieces, which can ease the job of understanding it.

Overloading

Finding new names for functions that do the same thing to variables of different types is always a problem. The procedure insert is a good example. To give programmers a bit more breathing space, Ada allows sub programs to have the same name, it only insists that they are distinguishable. This is called overloading.

Subprogram overloading

Two sub programs with the same name are distinguishable if their profile is different. The profile consists of the number of parameters, thier type and, if it is a function, the return type.

So long as the compiler can tell which sub program you requested by matching the profile of the call to the specifications of the sub programs you have supplied, it's happy; otherwise you'll get an ambiguous reference error.

```
procedure Insert(Item : Integer);  -- Two procedures with the
procedure Insert(Item : Float);    -- same name, but different.
                                   -- profiles.
```

The procedures Put and Get in the package Ada.Text_IO are examples of overloaded subprograms.

Operator overloading

In languages such as Pascal the + operator is overloaded. Sometimes it is used to add integers, sometimes reals, sometimes strings. It is quite obvious that this one operator is used to represent very different code.

Ada allows programmers to overload operators with thier own code. One restriction on this overloading of the operator name, is as expected, that the functions are distinguishable from the originals supplied, i.e. its profile is unique. Even this problem can be overcome by specifying the package name (see the next section).

Overloaded operators cannot be seperate compilation units. - they must be contained in another unit such as a procedure, function or package.

Consider an example where we wish to provide facilities to add two vectors together.

```
procedure add_demo is
        type Vector is array (positive range <>) of Integer;
        a : Vector(1..5);
        b : Vector(1..5);
        c : Vector(1..5);

        function "+"(left,right:vector) return vector is

                result   : Vector(left'first..left'last);
                offset   : constant Natural := right'first-1;

        begin
                if left'length /= right'length then
                        raise program_error;    -- an exception,
                                                        -- see later
                end if;

                for i in left'range loop
                        result(i):=left(i) + right(i - offset);
                end loop;
                return result;
        end "+";

begin
        a:=(1,2,3,4,5);
        b:=(1,2,3,4,5);
        c:= a + b;
end add_demo;
```

This example uses most features discussed in the last few chapters.

---

to the index...

---

to the index...

9 Packages

Overview

Although its roots are in Pascal, Ada still borrowed heavily from other languages and was influenced by the latest in software engineering techniques discussed in the 1970's. Undoubtedly the major innovation from that period is the concept of packages. Separation of the specifications from the implementation of 'objects' results in far more flexible code. The containment of the data structures behind a wall of functional interfaces results in much more maintainable systems. This chapter discusses Ada's implementation of packages.

Packages are not about how a program will run, but about how it is constructed and how it is to be understood and maintained.

Package specifications

The package specification of an Ada package describes all the subprogram specifications, variables, types, constants etc that are visible to anyone who 'withs' the package into thier own code.

The following is an example of a package specification.

```
package odd_demo is

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

        pi      :constant float:=3.14;

        x:      integer;

        type a_record is
                record
                        left:boolean;
                        right:boolean;
                end record;

        -- note that the following two subprograms are
        -- specifications only,the body of the subprograms are
        -- in the body of the package

        procedure insert(item:in integer; success:out boolean);
        function present(item:in integer) return boolean;

end odd_demo;
```

The items described in the package specification are often described as resources. We can access these values by 'with'ing the package in our code and then using the package name, a dot, and the name of the resource wanted, whether it is a type declaration, a function or a variable name.

```
with odd_demo;

procedure odder_demo is

        my_name :odd_demo.string;
        radius  :float;
        success :boolean;
begin
        radius := 3.0*odd_demo.pi;
        odd_demo.insert(4, success);
        if odd_demo.present(34) then ...
end odder_demo;
```

It should be noted that accessing resources in a package is textually identical to accessing the fields in a record.

As accessing the resources of a package using the full dot notation can be cumbersome, the use clause may be employed. When a package is 'use'd in this way the resources are available as though they were declared directly in the code.

```
with odd_demo;  use odd_demo;

procedure odder_demo is

    my_name : string(1..10);
        radius  : float;
        success : boolean;
begin
 radius:=3.0*pi;
 insert(4, success);
     if present(34) then ...

end odder_demo;
```

If two packages are with'ed and use'd in the one compilation unit (e.g. subprogram or another package) then there is the possibility of a name clash between the resources in the two packages. In this case the ambiguity can be removed by reverting to dot notation for the affected resources.

```
--****************************
package no1 is
   a,b,c:integer;
end no1;

--****************************
```

```
package no2 is
     c,d,e:integer;
end no2;

--****************************
with no1;  use no1;
with no2;       use no2;

procedure clash_demo is
begin
    a:=1;
   b:=2;
   c:=3;          -- ambiguous, are we referring to no1.c or
                     -- no2.c?
   no1.c:=3;    -- remove abiguity by reverting to dot
                      -- notation
     no2.c:=3;
end;
```

Another problem encountered is when local resources 'hide' the resources in a use'd package. In this case the ambiguity can still be removed by using dot notation.

```
package no1 is
       a:integer;
end no1;

with no1;       use no1;
procedure p is
       a:integer;
begin
       a:=4;           -- once again this is ambiguous
       p.a:= 4;        -- remove abiguity by using procedure name in
                       -- dot notation
       no1.a:=5;       -- dot notation for package
end p;
```

Package body

The package body is where all the implementation details for the services specified in the package specification are placed. As in the specification the package body can contain type declarations, object (variable) declarations, subprograms etc.

The format of a package body is

```
package body odd_demo is
```

```
        type list is array (1..10) of integer;
    storage_list    :list;
    upto    :integer;

        procedure insert(item           :in     integer;
                         success     :out    boolean) is
      begin
     ....
      end insert;

         function present(item:in integer) return boolean is

      begin
    . . . .
  end present;


begin          -- initialisation statements for the whole package
             -- These are run _before_ the main program!
       for i in storage_list'range loop
                storage_list(i):=0;
      end loop;
        upto:=0;
end odd_demo;
```

The resources in the body of the package are unavailable for use by any other package. Any attempt to reference them will result in a compiler error.

The variables declared in the package body retain their value between successive calls to the public sub procedures. As a result we can create packages that store information for later use.

The begin/end at the end of the package contain initialisation statements for the package. These are executed before the main procedure is run. This is true for all packages. The order in which different package's initialisation statements are run is not defined.

All the resources specified in the package specification are available in the package body without the use of a with clause.

Private types

So far all the types declared in the package specification have been completely visible to the user of the package.

Sometimes when creating packages we want to maintain total control over the manipulation of objects. For instance in a package that controls accounts in general ledger, we only want to provide the facilities to make withdrawals, deposits and creation of accounts. No other package needs to know or should have access to the representation details of the account object.

In Ada packages type declarations (and also constants) can be declared private.

```
package accounts is
        type account is private;        -- declaration comes later

        procedure withdraw(an_account:in out account;
                           amount    :in     money);

        procedure deposit( an_account:in out account;
                           amount    :in     money);
        function create(   initial_balance:money) return account;
        function  balance( an_account:in    account) return integer;

private         -- this part of the package specification
                -- contains the full description.
        type account is
                record
                        account_no      :positive;
                        balance :integer;
                end record;
end accounts;
```

Outside the package the only operations that can be performed on a private type are assignment, tests for equality and those operations defined by subprograms in the package specification.

Full details of the representation details are available in the package body. Any routine in the package body can access and modify the private type as though it were not private - the privacy of an object applies only outside the package.

It may seem a contradiction to put the private specifications in the package specifications - the public part of the package when you are trying to hide the representation details of an object. This is required for programs that allocate an object of that private type- the compiler then knows how much space to allocate.

Although the reader of the package specifications can see what the representation of the private type really is, there is no way he or she can make explicit use of the knowledge.

Objects can be created outside a package even if of a private type.

E.g.

```
with accounts;  use accounts;

procedure demo_accounts is
        home_account    :account;
        mortgage                :account;
        this_account            :account;
begin
        mortgage := accounts.create(initial_balance => 500.00);
        withdraw(home_account,50);
```

```
        . . .
        this_account:=mortgage; -- can assign private types.

        -- comparing private types.
        if this_account = home_account then
                        . . .
end;
```

Limited private types

A private type can be made even more private by removing the ability to compare and assign the values outside the package. You generally then have to provide a function to test equality (you can overload the equality operator =, this implicity overloads /= as well). Also you may have to create a procedure to perform assignments.

Q Why do we need private types?

A Consider the following:

```
type text (maximum_length:positive:=20) is
        record
                length: index:=0;
                value   :string(1. . maximum_length);
        end record;
```

In this record the length field determines the number of characters in the field value that have meaning. Any characters from position length+1 to maximum_length are ignored by us when using the record. However if we ask the computer to compare two records of this type, it does not know the significance of the field length; it will compare the length field and all of the value field. Clearly in this situation we need to write a comparison function.
Ada95 allows the programmer to override the equality operator for all types.

Deferred constants

In some package specifications we may wish to declare a constant of a private type. In the same manner as declaring a private type (and most forward references) we give an incomplete declaration of the constant - the compiler expects the rest to follow in the private section of the package specifications.

```
package coords is
       type coord is private;
  home: constant coord;  -- the deferred constant!

private
 type coord is record
            x      :integer;
```

```
                y       :integer;
        end record;

     home:constant coord:=(0,0);
end coords;
```

Child units (Ada95)

It was found in large system developments that a single package specification could grow extraordinarily large, with subsequent costs in recompilation of compilation units that depend on it if it were to change. To remedy this situation, the concept of child compilation units was conceived. Child units allow a logically single package to be broken into several physically distinct packages and subprograms. As well as solving the problem of large recompilations, they also provide a convenient tool for providing multiple implementations for an abstract type and for producing self contained subsystems, by using private child units.

Extending an existing package

A package that consists of a set of declarations may need to be extended at some time. For example a stack package may need the addition of a peek facility. If this package is heavily "withed" by many other units, then modifying the specification to include this extra features could result in a large recompilation despite the fact that most clients would not be using the new feature.

A child package can be declared that logically extends the package, but in a physically separate manner.

For example

```
package stacks is
       type stack is private;
  procedure push(onto:in out stack; item:integer);
        procedure pop(from :in out stack; item: out integer);
   function full(item:stack) return boolean;
       function empty(item:stack) return boolean;
private
       -- hidden implementation of stack
       ...
     -- point A
end stacks;

package stacks.more_stuff is
       function peek(item:stack) return integer;
end stacks.more_stuff;
```

The package stacks.more_stuff is a child packge of stacks. It has the visibility of all the declarations preceeding point A, that is the parent package presents the same visibility to a child package as it does to its body. The child package can see all of its parents private parts. The package bodies would be compiled

separately.

The clients who wish to use the function peek can simply have a with clause for the child package,

```
with stacks.more_stuff;

procedure demo is
        x       :stacks.stack;
begin
        stacks.push(x,5);
        if stacks.more_stuff.peek = 5 then
                ....
end;
```

'With'ing the child package automatically causes the parent package to be 'with'ed, and its parent package to be 'with'ed, and so on. However the use clause does not act in this way. Visibiliy can only be gained on a package by package basis. This is no doubt a pragmatic decision based on what most organisations would prefer (as indicated by existing use of the 'use' clause).

```
with stacks.more_stuff; use stacks; use more_stuff;

procedure demo is
        x       :stack;
begin
        push(x,5);
        if peek(x) = 5 then
                ....
end;
```

A package can have child functions and procedures. Rules for use of these are easily inferred.

Private child units

Private child units allow a child unit to be created that is only visible within the hierachy of the parent package. In this way facilities for a subsystem can be encapsulated within its a hidden package, with the compilation and visibility benefits that this entails.

Because they are private, a child package's specification is allowed to advertise, in its specification, private parts of its parents. This would not normally be allowed as it would allow the breaking of the hidden implementation of a parent's private parts.

```
private package stacks.statistics is
        procedure increment_push_count;
end;
```

The procedure stack.statistics.increment_push_count could be called from within the implementation of the stacks package; this procedure is not available to any clients external to this package hierachy.

to the index...

---

to the index...

10 Generics

Overview

Code reuse has been one of the great programming hopes for many years. Although suitable in the area of mathematical routines (where the functions are well defined and stable) trying to develop libraries in other areas has met with very limited success, due to the inevitable intertwining of process and data types in procedures. Ada has attempted to free us from this problem by producing code that does not rely as much on the specific data types used, but on their more general algorithmic properties.

As described by Naiditch, generics are like form letters; mostly they are complete letters with a few blanks requiring substitution (eg name and address information). Form letters aren't sent out until this information is filled in. Similarly generics cannot be used directly, we create a new subprogram or a package by 'instantiating' a generic and then using the instantiated compilation unit. When a generic is instantiated we have to supply information to fill in the blanks, such as type information, values or even subprograms.

Generics

In Ada a program unit (either a subprogram or a package) can be a generic unit.

This generic unit is used to create instances of the code that work with actual data types. The data type required is passed in as a parameter when the generic unit is instantiated. Generics are usually presented in two parts, the generic specification and then the generic package.

Once they are compiled the generics are stored in the Ada library and can be with'ed (but never use'd) by other compilation units. These other units, whether they be subprograms, packages or generics, can be with'ed and instantiated (the process of creating a usable subprogram or package by supplying generic parameters). The instatiated subprogram or package can be stored in the Ada library for later use.

The following is the instantiation of the generic package integer_io. Int_io is now available to be with'ed (and use'd) by any program.

```
with text_io;   use text_io;
package int_io is new integer_io(integer);
```

The package can be instantiated with other types as well.

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

package account_no_io is new integer_io(account_no);
```

Generic subprograms

The following is a compilation unit. Once it is compiled it is available to be 'with'ed (but not 'use'd) from the Ada library. It includes the keyword generic, a list of generic parameters and the procedure specification.

```
generic
 type element is private;        -- caution private here means
                                   -- element is a parameter to
                                    -- the generic sub program

procedure exchange(a,b        :in out element);
```

The body of the generic procedure is presented as a seperate compilation unit. Note that it is identical to a non-generic version.

```
procedure exchange(a,b  :in out element) is
     temp    :element;
begin
  a:=temp;
        a:=b;
   b:=temp;
end exchange;
```

The code above is simply a template for an actual procedure that can be created. It can't be called. It is equivalent to a type statement - it doesn't allocate any space, it just defines a template for the shape of things to come. To actually create a procedure:

```
procedure swap is new exchange(integer);
```

We now have a procedure swap that swaps integers. Here "integer" is called a generic actual parameter. "Element" is called a generic formal parameter.

```
procedure swap is new exchange(character);
procedure swap is new exchange(element => account); -- named association
```

You can create as many of these as you like. In this case the procedure name is overloaded and as normal the compiler can tell which one you call by the parameter type.

It can be called (and behaves) just as though it had been defined as

```
procedure swap(a,b      :in out integer) is
     temp:integer.
begin
      . . .
end;
```

Generic packages

Packages can be generics as well.

The following generic package specification is fairly standard:

```
generic
        type element is private;        -- note that this is a
                                       -- parameter to the generic
package stack is
        procedure push(e:       in element);
        procedure pop(e:        out element);
        function empty return boolean;
end stack;
```

The accompanying package body would be

```
package body stack is
   the_stack       :array(1..200) of element;
      top     :integer range 0..200:=0;

       procedure push(e:in element) is
         ....

    procedure pop(e:out element) is
         ...

     function empty return boolean is
                ...

end stack;
```

Quite simply you replace any instance of the data type to be manipulated with that of the generic type name.

How would you create a generic unit and test it?

- Create it using a specific type and translate it to use generic parameters.

Generic parameters

There are three types of parameters to generics

Type parameters

Value and object parameters

Subprogram parameters

So far we have only seen type parameters.

Type parameters

Despite the alluring appeal of generics we are still restricted by the very nature of the tasks we are attempting to accomplish. In some generics we may wish to provide a facility to sum an array of numbers. Quite clearly this is only appropriate for numbers, we can't add records together. To provide protection from someone instantitating a generic with an inappropriate type, we can specify the category of types that can be used when instantiating.

The compiler can also check that we don't perform any inappropriate actions on a variable inside the code e.g. it wont allow us to find the 'pred of a record.

A list of the various type restrictions is given below.

```
type T is private               -- very few restrictions
type T is limited private       -- fewer restrictions,
type T is (<>)          -- T has to be a discrete type
type T is range <>              -- T must be an integer type
type T is digits <>             -- T must be a floating point type
type T is delta <>              -- T must be a fixed point - not
                                -- discussed

type T is array(index_type) of element_type
        -- The component type of the actual array must match the
        -- formal array type. If it is other than scalar then
        -- they must both be either constrained or
        -- unconstrained.

type T is access X      -- T can point to a type X (X can be
                        -- a previously defined generic parameter.
```

To understand the rule governing instantiation of array type parameters consider the following generic package

```
generic
        type item       is private;
        type index      is (<>);
        type vector     is array (index range <>) of item;
        type table      is array (index) of item;
package P is . . .
```

and the types:

```
type color is (red,green.blue);
type Mix is array (color range <> ) of boolean;
type Option is array (color) of boolean;
```

then Mix can match vector and Option can match table.

```
package R is new P(     item => boolean,
```

```
                                    index => color,
                                    vector => mix,
                                    table => option);
```

Value parameters

Value parameters allow you to specify a value for a variable inside the generic:

```
generic
        type element is private;
        size: positive := 200;

package stack is

        procedure push...
        procedure pop...
        function empty return boolean;

end stack;

package body stack is

        size:integer;
        theStack :array(1..size) of element;
        . . .
```

You would instantiate the package thus:

package fred is new stack(element => integer, size => 50);

or

```
package fred is new stack(integer,1000);
```

or

```
package fred is new stack(integer);
```

Note that if the value parameter does not have a default value then one has to be provided when the generic is instantiated.

Value parameters such as string can also be included.

```
generic
        type element is private;
        file_name       :string;

package ....
```

Note that the file_name parameter type (string) is not constrained. This is identical to string parameters to subprograms.

Subprogram parameters

We can pass a subprogram as a parameter to a generic.

Why? If you have a limited private type you can pass tests for equality and assignment in as subprogram parameters.

E.g.

```
generic

        type element is limited private;
        with function "="(e1,e2:element) return boolean;
        with procedure assign(e1,e2:element);

package stuff is . . .
```

To instantiate the generic

```
package things is new stuff(person,text."=",text.assign);
```

Other forms allow for a default subprogram if none is given.

```
with procedure assign(e1,e2:element) is myAssign(e1,e2:person);
```

Or you can specify the computer makes a default selection of procedure based on the normal subprogram selection rules:

```
with function "="(e1,e2:element ) return boolean is <>;
```

If no function is supplied for the "=" function then the default equal function will be used according to the type of element (i.e. if element is integer the normal integer "=" will be used).

---

to the index...

11 Exceptions

Overview

Exceptions are described in the Ada Language Reference Manual as errors or other exceptional conditions that arise during normal program execution. Exception handling is the process of catching these errors at run time and executing appropriate code to resolve the error, either by correcting the cause of the problem or by taking some remedial action.

Predefined exceptions

There are five exceptions predefined in the language. They are described below.

The exception CONSTRAINT_ERROR is raised whenever an attempt is made to violate a range constraint.

```
procedure constraint_demo is

        x       :integer range 1..20;
        y       :integer;

begin
        put("enter a number "); get(y);
        x:=y;
        put("thank you");
end constraint_demo;
```

If the user enters a number outside the range 1..20, then x's range constraint will be violated, and a constraint_exception will occur. The exception is said to have been 'raised'. Because we have not included any code to handle this exception, the program will abort, and the Ada run time environment will report the error back to the user. The line 'put("thank you");' will not be executed either - once an exception occurs the remainder of the currently executing block is abandoned.

This error also occurs when an array index constraint is violated.

```
procedure constraint_demo2 is

   x       :array (1..5) of integer:=(1,2,3,4,5);
  y       :integer :=6;

begin
      x(y):= 37;
end constraint_demo2;
```

In this example the constraint exception will be raised when we try to access a non existant index in the array.

The exception NUMERIC_ERROR is raised when a numeric operation cannot deliver a correct result (e.g. for arithmetic overflow, division by zero, inability to deliver the required accuracy for a float operation). NUMERIC_ERROR has been redefined in Ada95 to be the same as CONSTRAINT_ERROR

```
procedure numeric_demo is

        x       :integer;
        y       :integer;

begin
        x:=integer'last;
        y:=x+x;         -- causes a numeric error
end numeric_demo;
```

The exception PROGRAM_ERROR is raised whenever the end of a function is reached (this means that no return statement was encountered). As well it can be raised when an elaboration check fails.

```
procedure program_demo is

       z       :integer;

       function y(x :integer) return integer is
        begin
           if x < 10 then
                  return x;
               elsif  x < 20 then
                      return x
                end if;
 end y;          -- if we get here, no return has been
                            -- encountered

begin
     z:= y(30);
end program_demo;
```

The exception STORAGE_ERROR is raised whenever space is exhausted, whether during a call to create a dynamic object or when calling a subprocedure (and stack space is exhausted).

The exception TASKING_ERROR is raised when exceptions arise during intertask communication; an example is task that attempts to rendezvous with a task that has aborted.

Handling exceptions

So far the code seen has not processed the exceptions. In these cases the programs are aborted by the run time code. To make exceptions useful we have to be able to write code that is run whenever an exception occurs.

The exception handler is placed at the end of a block statement, the body of a subprogram, package, task unit

or generic unit.

To handle a constraint error consider the following code.

```
declare

        x:integer range 1..20;

begin
        put("please enter a number ");
        get(x);
        put("thank you");
exception
        when constraint_error =>
                put("that number should be between 1 and 20");
        when others =>
                put("some other error occurred");
end;
```

If the user enters a number between 1 and 20 then no error occurs and the message "thank you " appears. Otherwise the message "that number ..." appears and the block terminates.

If we want the user to continue to enter in numbers until there is no constraint error then we can write the following:

```
loop
        declare
                ...
        begin
                ...
                get(x);
                exit;
        exception
                when constraint_error =>
                        put("that number ...
        end;
end loop;
```

This highlights the point the instruction executed after an exception is that following the block in which the exception was handled.

Exceptions can be raised by the programmer simply by using the raise statement.

```
raise numeric_error;
```

The exception raised is indistinguishable from a genuine numeric_error.

Exception propagation

If an exception is not handled in the subprocedure in which it was raised, the exception is propagated to the subprocedure that called it. A handler for the exception is searced for in the calling subprocedure. If no handler is found there then the exception is propagated again. This continues until either an exception handler is found, or the highest level of the current task is reached, in which case the task is aborted. If there is only one task running (typical for student projects) then the Ada runtime environment handles the exception and the program is aborted.

```
procedure exception_demo is

        ---------------------------------
        procedure level_2 is
                -- no excpetion handler here
        begin
                raise constraint_error;
        end level_2;

        ---------------------------------
        procedure level_1 is
        begin
                level_2;
        exception
                when constraint_error =>
                        put("exception caught in level_1");
        end level_1;

begin
        level_1;
exception
        when constraint_error =>
                put("exception caught in exception_demo");
end exception_demo;
```

If this program was run the only output would be "exception caught in level_1". The exception is handled here and does not propagate any further. If we want to we can place a raise statement in the exception handler - this has the effect of propagating the exception up to the calling subprogram. In this way an exception can be viewed by each subprocedure in the call hierachy, with each performing whatever action it deems necessary.

```
....
exception
        when constraint_error =>
                package_disabled:=true;
                raise;  -- re raise the current exception,
                                -- allow other procedures to have a go
                                -- at processing the exception.
end;
```

The raise statement is very useful when used in the others section of an exception handler. In this case the appropriate exception is still raised and propagated.

User defined exceptions

Ada gives the user the ability to define their own exceptions. These are placed in the declarative part of the code. They can be placed whereever a normal declaration is placed (e.g. even in package specifications). The format for the declaration of an exception is

```
my_very_own_exception :exception;
another_exception        :exception;
```

The use of exceptions is the subject of much debate about whether it is a lazy way of programming, without thinking too much about the problem and likely error conditions, or whether it is a valid form of control structure that can be used to some effect.

Problems with scope

Handling user exceptions is identical to that of the predefined exceptions except for the problem of scoping. Consider the following example.

```
with text_io; use text_io;

procedure demo is

        procedure problem_in_scope is
                cant_be_seen    :exception;
        begin
                raise cant__be_seen;
        end problem_in_scope;

begin
        problem_in_scope;
exception
        when cant_be_seen =>
                put("just handled an_exception");
end demo;
```

This example is illegal. The problem is that the scope of an_exception is limited to procedure exception_raiser. It's name is not defined outside of this procedure and thus it cannot be explicity handled in procedure demo.

The solution is to use the others clause in the outer procedure's exception handler.

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

```
procedure demo is

        procedure problem_in_scope is
                cant_be_seen    :exception;
        begin
                raise cant_be_seen;
        end problem_in_scope;

begin
        problem_in_scope;
exception
        when others =>
                put("just handled some exception");
end demo;
```

Another problem arises when one procedure's exception hides another exception due to scoping rules.

```
with text_io; use text_io;

procedure demo is
     fred    :exception;

     -------------------------------------
    procedure p1 is
 begin
           raise fred;
     end p1;

 -------------------------------------
    procedure p2 is
         fred    :exception;     -- a local exception
    begin
           p1;
     exception
               when fred =>
                    put("wow, a fred exception");
   end p2;
 -------------------------------------
begin
       p2;
exception
   when fred =>
            put("just handled a fred exception");
end demo;
```

The output of this procedure is "just handled a fred exception". The exception handled in p2 is simply a local exception. This is similar to the handling of scope with normal variables.

Procedure p2 could be rewritten as

```
-----------------------------------
procedure p2 is
        fred    :exception;
begin
                p1;
exception
        when fred =>
                -- the local exception
                put("wow, an_exception");

        when demo.fred =>
                -- the more 'global' exception
                put("handeled demo.fred exception");
end p2;
```

Suppression of exception checks

Exceptions are generated because extra code that is inserted inline detects some erroneous condition, or through some hardware checking mechanisms such as software interrupts or traps.

Accordingly it is possible to suppress the insertion of these checks into the code. The method that Ada uses is the pragma SUPPRESS. However it should be noted that through use of program analysis by the compiler, a good number of checks can automatically be removed at compile time.

The pragma can be placed in the code where the suppression is required. The suppression extends to the end of the current block (using normal scope rules).

It has a large range of options to enable suppression of various checks on either a type basis, an object basis or a functional basis. The versatility of the pragma is dependent upon the implementation. As well various implementations are free to implement (or ignore) any pragma suppress feature.

The exception CONSTRAINT_ERROR can be raised by failing several suppressable checks

```
pragma suppress (access_check);
pragma suppress (discriminant_check);
pragma suppress (index_check);
pragma suppress (length_check);
pragma suppress (range_check);
pragma suppress (division_check);
pragma suppress (overflow_check);
```

The exception PROGRAM_ERROR has only one suppressable check

```
pragma suppress (elaboration_check);
```

The exception STORAGE_ERROR has only one suppressable check

```
pragma suppress (storage_check);
```

Applying suppression of checks

We can suppress the checking of exceptions on individual objects.

```
pragma suppress (index_check, on => table);
```

It can also relate to a single type.

```
type employee_id is new integer;
pragma suppress (range_check, employee_id);
```

The use of a pragma would be as follows. In this case the scope of the pragma is till the end of the block.

```
declare
 pragma suppress(range_check);
   subtype small_integer is integer range 1..10;
   a       :small_integer;
 x       :integer:=50;
begin
      a:=x;
end;
```

This code would cause no constraint error to be generated.

---

to the index...

12 Files

Overview

Conceptually similar to most extended Pascals file i/o, Ada provides a simple yet effective collection of file packages. These can be discarded and replaced by any other file package, with of course a subsequent decrease in portability. Most of the facilities provided would not be used in high performance applications directly, but may form the basis of such a file system.

The I/O packages

So far all the I/O performed has been on directed to the standard input/output. All of the facilities provided have come from the TEXT_IO package (even the generic packages integer_io and float_io). This package provides facilities for file manipulation of textual files (files of characters) and provides facilities such as close, delete, reset,open, create etc.

Two other standard I/O packages are for storing files that consist of the one type of fixed length object, such as records, arrays, floating point numbers etc. These are the generic packages sequential_io and direct_io.

The Text_IO package

The text i/o package's main data type is the FILE_TYPE. This is the internal representation of the file. When a file is opened or created an association is made between the name and the FILE_TYPE. The object of type FILE_TYPE is used from then on to reference the file.

The procedure create creates a file and writes some data to it.

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

procedure demo_file_io is
        my_file :text_io.file_type;

begin
        create( file => my_file,
                        mode => out_file,
                        name => "data.dat");

        put(    file => my_file,
                item => "numbers and their squares");

        for i in 1..10 loop
                put(my_file,i);
                put(my_file,"    ");
                put(my_file,i*i);
                new_line(my_file);
        end loop;
        close(my_file); -- required, Ada may not close your
                                -- open files for you
end demo_file_io;
```

The following program reads data from one file and writes it to another, a character at a time. It should be noted that the concept of 'end of line' is different to that provided by Unix and Dos. In those systems there is simply a character that marks the end of the line

which is processed as a normal character; in Ada and Pascal there is the concept of a line terminator that is not a character in the file. To read pass this terminator you need to perform a skip_line. Similarly to partition the output file into lines, the command new_line has to be given.

```
-- program to read a file a character at a time, and to write
-- it out a character at a time

with text_io;      use text_io;

procedure read_write is

     input_file      :file_type;
     output_file  :file_type;
        char            :character;

begin
        open(input_file,in_file,"input.dat");
   create(output_file,out_file,"output.dat");

       while not end_of_file(input_file) loop
           while not end_of_line(input_file) loop
                   get(input_file,char);
                    put(output_file,char);
           end loop;
                skip_line(input_file);
           new_line(output_file);
  end loop;
       close(input_file);
      close(output_file);
end read_write;
```

There are various procedures to perform file manipulation. These are

Create - Creates a file with the given name and mode. Note that if the file

- has a null string, then the file is temporary and is deleted later.

Open - Opens an existing file with the given name and mode.

Delete - Deletes the appropriate file. It is an error to delete an open file.

Reset - Returns the read (or write) position to the start of the file.

As well there are functions that report on the status of the file system.

End_of_File - Returns true if we are at the end of the current file.

End_of_Line - Returns true if we are at the end of the current text line.

Is_open - Returns true if the given file is open.

Mode - Returns the mode of the given file.

Name - Returns the name (string) of the current file.

There are various other routines, it is best to examine the specifications of the package text_io (appendix B) to get a clear idea of the package.

Use of text files

Ada's text_io facilities rely on the exception mechanism to report errors in the creation and opening of files. For example attempting to create a file that already exists causes an exception, as does attempting to open a file that does not.

To get around this circularity the following procedure robust_open could be used. This attempts to open the file, if it fails due to the file not being there, it attempts to create it instead.

The code is, of course, subject to race conditions. This program could be interrupted after an attempt open the file, and before the create is attempted; a second process could conceivably create the file in this time.

```
------------------------------------------------------------
-- Attempts to open the specified file.
-- If it doesn't work then it creates it instead

with text_io;        use text_io;
procedure robust_open( the_file :in out     file_type;
                       mode     :in file_mode;
                       name     :in string) is
begin
 open(the_file,mode,name);

exception
       when name_error =>
              create(the_file,mode,name);
end robust_open;
```

Another utility, the boolean function file_exists, allows students to check if the file exists. An exception (use_error) is raised if the is already open.

```
------------------------------------------------------------
-- Returns true if the file specified in 'name' exists.
-- This works by attempting to open the file. If this
-- suceeds then the file is there.

with text_io;   use text_io;

function file_exists(name       :string) return boolean is
        the_file:text_io.file_type;
begin
        open(the_file, in_file, name);

        -- yes it worked, close the file and return true
        close( the_file);
        return true;

exception
        when name_error =>
                -- couldn't open the file, assume it's not there.
                return false;
end file_exists;
```

Ada95 Text_io enhancements

The mode append_file has been added to the allowable modes for text files. As well the concept of a standard error file (c.f. Unix/C) has been added. The procedures flush have been added to enable flushing of text file buffers.

Improvements in character handling include look ahead, and get_immediate, with various options.

Generic packages for the I/O of the new modular and decimal types have been included - their specifications conform to that of the other numeric generic I/O packages.

The sequential_io package

Most large systems do not utilise text as the basis for their files. The files are usually composed of composite objects, and typically they are records. The sequential_io generic package allows us to create files whose components are any type (they must however be constrained).

The basics of the sequential_io generic package are identical to the text_io package, except that the procedures get and put are now read and write, and the procedures deal in terms of the type the package was instantiated with. Similarly the concepts of line has dissapeared, so that the function end_of_line and the procedures skip_line and new_line have also gone.

Use of the package is demonstrated below.

```
with sequential_io;     -- generic package
with personnel_details;       use person_details;
     -- has record type 'personnel'

with produce_retirement_letter;

procedure sequential_demo is

       package person_io is new sequential_io(personnel);
      data_file       :person_io.file_type;

   a_person        :personnel;

begin
        person_io.open(data_file,in_file,"person.dat");

 while not person_io.end_of_file(data_file) loop
         person_io.read(data_file,a_person);

             if a_person.age > 100 then
                      produce_retirement_letter(a_person);
            end if;
 end loop;

       close(data_file);
end sequential_demo;
```

No direct access of the file is possible. The file is opened at the start and processed until you get to the end, you reset or close the file.

Ada95 has added the file mode append_file to the specifications of sequential_io.

The direct_io package

The direct io package builds on top of the sequential io package by providing the ability to seek directly to a given record, to determine the size of the file, to determine the current index and to be able to open the file with a new mode - inout_file (read/write).

These facilities should make it possible, in conjuction with a suitable indexing package, to provide very high level file processing packages.

The following code demonstrates the use of direct files. For brevity it assumes that the employee records are stored on the basis of their employee numbers.

```
with direct_io; -- generic package
with personnel_details; use personnel_details;
        -- has record type 'personnel'
        -- has procedure display_personnel, etc

with int_io;    use int_io;
with display_menu;

procedure direct_demo is

        package person_io is new direct_io(personnel);
        data_file       :person_io.file_type;
        a_person        :personnel;
        option  :integer;
        employee_no     :integer;

begin
        person_io.open(data_file,inout_file,"person.dat");

        loop
                display_menu;
                get_option(option);

                case option is
                        when 1 =>
                                get(employee_no);
                                set_index(positive_count(employee_no));
                                read(data_file,a_person);
                                display_person(a_person);
                        when 2 =>
                                get(employee_no);
                                set_index(positive_count(employee_no));
                                read(data_file,a_person);
                                get_new_details(a_person);
                                write(data_file,a_person);
                        when 3 =>
                                exit;
                        when others =>
                                put("not a great option!");
                end case;
        end loop;
        close(data__file);
end direct_demo;
```

---

to the index...

13 Access types

Overview

Access types (and the objects they access) are used to create dynamic data structures. It should be noted that access types are not necessarily implemented as pointers are in other languages (e.g. the address of the object in question), although in many cases they are. They can be used to create dynamic data structures such as linked lists, trees, graphs etc.

Access types

Access types allow for the dynamic referencing and allocation of objects in Ada.

To specify an access type consider the following:

```
type person is
        record
                name    :string(1..4);
                age     :0.. 150;
        end record;

fred    :person

type person_ptr is access person;
someone :person_ptr;
```

To dynamically create an object that someone will point at:

```
someone:= new person;
```

An object referred to by someone can be referenced in exactly the same way as an explicity declared object.

E.g.

```
fred.name   := "fred";
someone.name:= "jim ";
```

A special value called null is used to indicate that the access object does not reference any valid objects. An access object is always initialised to null.

Given the following declarations

```
declare
        x       :person_ptr := new person;
        y       :person_ptr := new person;

begin
        x := ("fred",27);
        y := ("anna",20);
```

```
        x := y;                  -- *
        y.all := ("sue ",34);
        put(x.name);
end;
```

This will output the string "sue ". When the line highlighted with a star is executed, the value of the access object y is assigned to x, not the object that y was accesses. Both x and y both refer to the same object, so a change to the object that y refers to implicity changes the object x refers to.

If we wish to change a field of the object refered to by x

```
x.name := y.name;
```

Access objects x and y still point to distinct objects. Both access objects are implicitly dereferenced in this example. Because of the implicit derefence, we need a special syntax to denote the entire object the access object accesses. Ada has the reserved word **all** for this purpose.

```
x.all := y.all; -- all field in object referred to by y now
                -- in the object referred to by y. They are
                -- still distinct objects.
```

A comparison of Pascal, C and Ada pointer/access syntax.

```
                      Pascal  C                      Ada
Access          a^.fieldname    *a.fieldname           a.fieldname
                                        a->fieldname

Copying pointer b := a; b = a;          b := a;

Copying accessed        b^ := a^;       *b = *a;               b.all := a.all
object
```

Ada95. Typically all access objects are allocated from a storage pool (heap) (see below). However it is possible to create an access type to other objects, so long as they are declared aliased, and the access type is declared to point at 'all' objects.

```
procedure demo is
       type ptr is access all integer;
 a :aliased integer;
        x, y :ptr;
begin
 x := a'access;
  y := new integer;
end;
```

Scope rules ensure that there cannot be any dangling references to objects. The attribute Unchecked_Access can be used to create access values in an unsafe manner.

Access types to subprograms

A feature not available in Ada83, access types to subprograms allow for functional parameterisation such as used in Fortran mathematical libraries, as well as ad-hoc late binding. As for the rest of Ada, this is type safe.

An example from the LRM (3.10).

```
type Message_Procedure is access procedure (M:in String := "Error!");
procedure Default_Message_Procedure (M :in String);
Give_Message : Message_Procedure := Default_Message_Procedure'Access;
...
procedure Other_Procedure (M :in String);
...
Give_Message := Other_Procedure'Access;
...
Give_Message.("File not found");
Give_Message.all;
```

Self referencing data structures

To define a structure that refers to itself requires normally requires making references to data structures that don't as yet exist. Ada circumvents this problem by allowing for incomplete type declarations. These simply specify the name of a type that is yet to be defined. The compiler expects that the full definition will be given before the end of the source file.

```
type element;   -- incomplete type definition.

type ptr is access element;

type element is        -- full definition of the type.
 record
          value:integer;
          next:ptr;
       end record;
```

Initialisation of objects

When an object is dynamically allocated its value can be set in the same statement as its creation. This simply uses a qualified (type name specified) aggregate.

E.g.

```
head := new element'(40,head);  -- insertion at the head
                                        -- of a linked list.
```

The same thing can be done using named aggregates.

```
head := new element'(value => 40; next => head);
```

Garbage collection

There is no language requirement for deallocated memory to be put back into use for later allocation. You should consult your compiler reference manual for details.

If you want to deallocate memory (similar to free system call in Unix), then you can instantiate the generic procedure Unchecked_Deallocation (child of package Ada in Ada95). It is called unchecked because it does no live object analysis before deallocation.

```
generic
 type Object(<>) is limited private;
     type Name is access Object;
procedure Ada.Unchecked_Deallocation(X: in out Name);
pragma Convention(Intrinsic, Ada.Unchecked_Deallocatoin);
```

This would be instantiated as follows...

```
procedure Free is new Ada.Unchecked_Deallocation(object => node,
                                                 name   => ptr);
```

Free could then be called as follows...

```
head := new node;
...
free(head);
```

Storage Pools

Ada95 allows a storage pool to be specified from which allocated memory comes from. Different access types can share a pool, typically most user defined access types share the one program wide pool. By extending the abstract type Root_Storage_Pool, defined in packge System.Storage_Pools, users can write their own storage pool type, and then associate it with an access type via the Storage_Pool attribute.

For example (from LRM, 13.11)

```
Pool_Object : Some_Storage_Pool_Type;

type T is access <something or other>;
for T'Storage_Pool use Pool_Object;
```

Storage pools allow for the possibility access types that are smaller than for a default access type that accesses a default pool.

---

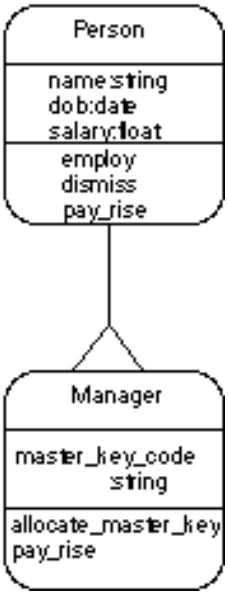
to the index...

14 Object Oriented features of Ada

Overview

The type system of Ada that existed in Ada83 has been extended to include support for object oriented programming. Rather than an extension that would be discordant with what existed, the object oriented facilities extend the existing type system to allow for type extension. These provide all the classical object oriented facilities such as inheritance, dynamic dispatching, polymorphism, with the usual Ada features of readability and safety.
Types are for many purposes similar to classes.

The following object diagram is implemented using Ada.



Inheritance (Ada83)

Inheritance can be split into two distinct concepts, inheritance of operations (methods) and inheritance and further addition of attributes (type extension). Ada 83 supported inheritance of operations. If a new type is derived from a base type, then the derived type inherits the operations available from the parent. For example,

```
type person is
        record
                name    :string(1..10);
         dob     :date;
                salary :float := 0.0; -- :-(
        end record;

procedure employ(someone:in out person);
procedure dismiss(someone:in out person);

type manager is new person;
```

```
    -- derived type manager inherits the operations of person,
    -- and can add new operations of its own.

    procedure allocate_master_key(someone:in out manager);
```

This model of inheritance does not allow for extension of types, that is we cannot add in extra attributes. The procedures (or functions) that take a parameter of a type are said to be the operations on the type and are equivalent to methods in other OO languages.

Inheritance (Ada95)

Ada95 has extended the notion of type derivation to support conventional object oriented programming. A new type, the tagged record, has been introduced to represent a type that can be extended (inherited). The syntax for a tagged type is

```
    type person is tagged
            record
                    name    :string(1..10);
             dob     :date;
                    salary :float := 0.0;
            end record;
```

A tagged record is treated like a conventional record in most situations. Field accessing, and setting is just as normal.

```
    jane    :person := ("Jane      ", (6,10,1978), 210.0);
```

Operations on the type are written just as before...

```
    procedure employ(someone:in out person);
    procedure dismiss(someone:in out person);
    procedure pay_rise(someone:in out person);
```

If we want to extend the type, we follow a notation similar to the type derivation common in Ada...

```
    type manager is new person with
            record
                    master_key_code :string(1..10);
            end record;
```

The "with record" part indicates the extension to the base type (in this case "person").

New operations for the manager can be added, just as in Ada83...

```
    procedure allocate_master_key(
                            someone:in out manager;
                            code   :in     string);
```

We may even want to override an inherited operation...

```
procedure pay_rise(someone:in out manager);
```

We can now write a different algortihm for pay_rise's for manager's compared to person's.

How do I set all of this up?

In Ada, the package concept is simply used for encapsulating the declaration of a type, and it's operations. The package is used to hide the implementation details, and as an aid to recompilation (we can split a program up into smaller chunks, which individually are quick to compile). A very common convention is to place one type and it's operations into one package.

The above example would thus be placed into two packages...

```
with dates; use dates; -- definition of type date

package persons is
        type person is tagged
                record
                        name    :string(1..10);
                 dob     :date;
                        salary  :float := 0.0;
                end record;

        procedure employ(someone:in out person);
        procedure dismiss(someone:in out person);
        procedure pay_rise(someone:in out person);
end persons;
```

The declaration of the derived type would be placed in a second package...

```
with persons;   use persons;

package managers is
        type manager is new person with
                record
                        master_key_code :string(1..10);
                end record;
        procedure allocate_master_key(
                            someone:in out manager;
                                code   :in     string);
        procedure pay_rise(someone:in out person);
end managers;
```

For tagged types, the operations on the type that are included in the same package as the type declaration (as all of the above are) have a special status - they are called the primitive operations of the type - and are all potentially dispatching calls. This will be described in further detail later.

Using tagged types

Using these types is quite simple. You just declare variables as you would have before...

```
with text_io;   use text_io;
with persons;   use persons;
with dates;     use dates;
with managers;  use managers;

procedure demo is
        me      :person;
        him     :manager;

begin
        me.name := "Santa     ";
        me.dob  := (29,11,1962);

        him.name := "Rudolph   ";       -- inherited field
        him.dob := (28, 6, 1962);       -- inherited field
        employ(him);            -- inherited operation

        -- new method for type manager
        allocate_master_key(him, code => "XYZ398");


        pay_rise(him);          -- Manager version
        pay_rise(me);           -- Person version

end;
```

Null records and extensions

A type may need to be created that has no fields (attributes), only operations (methods). This can be achieved by specify an empty record when declaring a type.

```
type root is tagged
        record
                null;
        end record;
```

Ada has a special syntax for null records being used with tagged types:

```
type root is tagged null record;
```

This can now be used as the basis of further derivations.

Operations on the type can be added as normal within a package specification...

```
procedure do_something(item     :root);
```

More typically a new type may be derived from an existing type without the need for new attributes, but with the need for new subprograms.

The syntax for such an extension would be...

```
type director is new manager with null record;

procedure pay_rise(someone :in out director);
```

A director has no new components added, but can has yet another method for allocating a pay_rise.

Abstract types and subprograms

Sometimes we wish to create an artificial type that we never expect to use, but serves as the "root" of a tree of types. These are called abstract types. They may also include abstract subprograms , which do not have a body and must be overridden by a derived type. This forces all descendants of a type to support a common functionality.

An abstract type has the following syntax...

```
package Sets is
        type Set is abstract tagged null record;

        function Empty return Set is abstract;
        function Empty(Element:Set) return boolean is abstract;
        function Union(left, right:set) return Set is abstract;
        function Intersection(left, right:set) return Set is abstract;
        procedure Insert(Element:natural; into:Set) is abstract;
end Sets;
```

This implementation of a set of natural numbers is taken from the Ada Language Reference Manual. Note that you cannot declare a variable of type Set because it is an abstract type. E.g.

```
with Sets;      use Sets;

procedure wont_compile is
        my_set  :Set;   -- illegal, abstract type
begin
        null;
end;
```

However a derived type may or may not be abstract...

```
with Sets;
```

```
package quick_sets is

        type bit_vector is array(0..255) of boolean;
        pragma pack(bit_vector);

        type quick_set is new Sets.set with
                record
                        bits:bit_vector := (others => false);
                end record;

        -- advertise concrete implementations
        function Empty return quick_set;
        function Empty(Element:quick_set) return boolean;

        etc. etc.
```

Class wide types

For any given tagged type, there is an associated class wide type, that encompasses the tagged type and all types derived from it. Ada has used the word "class" differently to most object oriented languages. It is perhaps used in a more traditional English sense, as a collection of similar types.

For example in the type family (inheritance hierachy)

```
vehicle
        motorised
                truck
                car
                train
        unmotorised
                bicycle
```

the type `vehicle'class` refers to all the types above. `Motorised'class` refers to the classes motorised, truck, car and train.

This can be used to allow for dynamic selection of the appropriate operation.

Assume that the following procedure was defined in package persons.

```
procedure reward_good_work(someone:in out person) is
begin
        print_certificate(someone.name);
        pay_rise(someone);
end;
```

If we call this with a manager as a parameter, then the procedure `persons.pay_rise` will be called, not

managers.pay_rise.

A solution to this problem is to accept as a parameter any variable in the class wide type. This is done as follows...

```
procedure reward_good_work(someone:in out person'class) is
begin
        print_certificate(someone.name);
        pay_rise(someone);
end;
```

If the following declarations are made...

```
me      :person;
him     :manager;
```

and then...

```
reward_good_work(me);   -- calls persons.pay_rise
reward_good_work(him);  -- calls managers.pay_rise
```

Here the pay_rise procedure called depends on the specific type passed in as a parameter. This is called dynamic dispatching - deciding which procedure to dispatchi to (or call) at run time (dynamically). Passing in a manager object results in a call to the manager's pay_rise routine. Passing in a person object results in a call to the person's pay_rise routine.
What pay_rise routine would be called if a Director object was passed in?

In fact all the primitive operations of type are potentially dispatching calls, given the right circumstances. The example using a class wide type is one example. Another (closely related) example is the use of class wide access (pointer) types.

What you can't do with a class wide type (such as person'class) is create an array of them. This is because each object is potentially a different size - arrays require all elements to be the same size. Similar considerations apply for any place where a declaration must be of a fixed size (constrained, in Ada parlance).

Class wide pointers

Most pointers in Ada are restricted to pointing at just a single type. For example...

```
type node;
type ptr is access node;
type node is record
                item    :integer;
                next    :ptr;
        end record;
```

Here variables of type ptr are restricted to always pointing at variables of type node.

With the concept of class wide types in Ada, comes the concept of a pointer that can point to any item within

an inheritance hierachy.
This is achived by...

```
type person_ptr is access person'class;
who      :person_ptr;
```

who' can now point at any object from the derivation tree rooted at person, i.e. person, manager, director, and quite importantly, any future types included in the inheritance hierachy.
For example we could make 'who' point at...

```
who := new person;
who := new manager;
who := new director;
```

Interestingly, we could create an array of class wide access types...

```
type person_list is array(1..10) of person_ptr;
people  :person_list;
...
people(1) := new person;
people(2) := new manager;
people(3) := new person;
people(4) := new director;

-- initialise the variables appropriately.
...

for i in people'range loop
        exit when people(i) = null;

        -- now dispatch to the appropriate routine...
        pay_rise(people(i).all);
end loop;
```

Class wide pointers can be used to build hetrogeneous data structures, that is those were the components are not all the same type (they must however be in the same class).

Private tagged types

A type can be declared private to help enforce information hiding (especially of the exact details of its representation, e.g. is it an array, or a linked list?). Similarly tagged types can be made private.

```
type person is tagged private;
```

The private section of the package must complete the declaration (provide what is called a full view of the type)...

```
type person is tagged record...
```

A type extension can also have a private extension if desired...

```
type manager is new person with private;
```

This gives sufficient flexibility for most purposes.

Generic type parameters

Tagged types can now be specified as generic type parameters (they can still be passed as actuals to the normal private or limited private type parameters). The syntax is as follows...

```
generic
        -- actual must be a tagged type
        type T is tagged private;

        -- actual must be direct descendent of S
        type T is new S;

        -- actual must be derived from S somewhere
        type T is new S with private;

package some_interesting_pacakge is...
```

Multiple Inheritance

Ada95 does not directly support MI. It provides features that can be used to build MI "by hand" with a bit of extra work. The options are generics (to extend a type with a set of new features), access discriminants (to allow a type to point at it's containing object, and therefore allow delegation) and another one that I can't quite remember now.

---

to the index...

15 Concurrency support

Overview

Ada is one of the few mainstream languages that supports concurrency from within the language. This has the benefit of stable semantics, and therefore portability between computers. The task and rendezvous mechanism of Ada83 has been enhanced through the introduction of protected objects and greater considertion for real time systems.

Tasks

An Ada program consists of at least one, and possible more tasks, which run concurrently. The tasks run independently of each other, communication/synchronisation is achieved by high level concepts such as the rendezvous or with protected objects. With an abstraction level higher than simple semaphores, the rendezvous and protected objects come supplied with a range of options for guards (as Dijkstra postulated), timeouts and (in Ada95) to selectively requeue clients and abort operations.

The death of a task does not necessarily affect the operation of other tasks, except if they attempt to communicate with it. There is no specialised task, even the main procedure of the Ada program can end, with other tasks continuing.

The rendezvous

As one mechanism for tasks to safely communicate and synchronise with each other, the rendezvous is conceptually simple. Tasks publish entry points at which they are prepared to wait for other tasks. These entry points have a similar semantics to procedures; they can have parameters of in, inout and out mode, with default parameter values. A task wishing to rendezvous with a second task simply makes a procedure-like rendezvous call with an entry; Either is held up until they can both communicate. The rendezvous is non symmetrical - one task is viewed as a server and cannot initiate a rendezvous.

```
procedure demo is
       task single_entry is
               entry handshake;
        end task;

        task body single_entry is
        begin
               delay 50.0;

               accept handshake;

               delay 1.0;
        end;
begin
        for i in 1..random(100) loop
               delay(1.0);
        end loop;
        handshake;
end;
```

When this program is run the task single_entry is started. If succeful the parent task is also activated. If 'random' returns a value less than 50, then after the initial delay the main task waits for the second task, and vice versa if it returns greater than 50. Once the two tasks have rendezvoused, they depart on their separate ways. The main task finishes, the second task continues until it too finishes. The program then terminates.

The same program can be modified to include parameters in the entry call. In this example, the two tasks swap duration times for the delay statements. However they may still not finish together as the quicker task to the rendezvous will still have to wait for the second task.

```
procedure demo is
       x :duration := duration(random(100));
   y :duration;

    task single_entry is
            entry handshake(me_wait: in duration; you_wait: out duration);
        end task;

        task body single_entry is
               a :duration := duration(random(100));
               b :duration;
        begin
               delay a;

               accept handshake(me_wait:in duration; you_wait: out duration) do
                      b := me_wait;
                  you_wait := a;
          end handshake;

               delay b;
        end;
begin
        delay(x);
        handshake(x,y);
 delay(y);
end;
```

A task can have several rendezvous'. If it waits for a rendezvous and there is no possible task that can rendezvous with, then it will abort with a tasking_error exception.

Rendezvous' can be treated in a manner very similar to procedure calls. They can be in loops, if statements, subprograms etc. Typically however they are structured in a high level loop; this allows a client architecture in which after servicing a rendezvous, and performing some processing, it goes back and waits for another rendezvous.

```
procedure demo is
       type service is (increment, decrement, quit);
   task single_entry is
               entry accept_command(which : service);
        end task;

        task body single_entry is
              command :service;
              i       :integer;
        begin
               loop
                   accept accept_command(which :service) do
                              command := which;
                      end accept_command;

                    case command is
                       when increment => i := i + 1;
```

```
                        when decrement => i := i - 1;
                        when quit => exit;
                    end case;
                end loop;
        end;

    begin
        -- stuff, depending perhaps upon user input.
    end;
```

This situation can be impoved having multiple entry points, and making use of the select statement in conjunction with the accept statement.

```
    procedure demo is
        task multiple_entry is
          entry increment;
                entry decrement;
        end multiple_entry;

        task body multiple_entry is
            i : integer;
      begin
            select
                  accept increment;
                       i := i + 1;
              or
                       accept decrement do
                              i := i - 1;
                     end decrement
            or
                       terminate;
               end select;
       end;
    begin
        -- rendezvous'
    end;
```

The select statement allows the task to wait on several different entry points. This example shows the increment entry without any associated processing. The decrement entry forces the calling task to wait while the decrement takes place. The terminate alternative causes task multiple_entry to finish if it is uncallable from any other task.

The select statement can have a selective wait, unconditional alternative, a terminate or none of these. The previous example showed the terminate. The unconditional alternative is shown below.

```
    procedure demo is
        task multiple_entry is
                entry increment;
                entry decrement;
        end multiple_entry;

        task body multiple_entry is
            i : integer;
        begin
           select
                  accept increment;
```

```
                        i := i + 1;
                or
                        accept decrement do
                                i := i - 1;
                        end decrement
                else
                        perform_some_processing;
                end select;
        end;
begin
        -- rendezvous'
end;
```

In this example, the task stops only briefly to inspect if another task is waiting to rendezvous. If not it goes off and does some processing.

The selective wait allows the task to wait for a given time, after which it gives up and continues on with other actions.

```
procedure demo is
       task multiple_entry is
          entry increment;
                entry decrement;
        end multiple_entry;

     task body multiple_entry is
             i : integer;
    begin
           select
                  accept increment;
                        i := i + 1;
                or
                        accept decrement do
                                i := i - 1;
                        end decrement
                or
                        delay 3.0;      -- not a normal delay statement!
                        perform_some_processing;
                end select;
        end;
begin
        -- rendezvous'
end;
```

Here the task will wait for at least 3.0 seconds for another task. If none comes along it will call perform_some_processing.

Similarly the client task can either wait unconditionally for a rendezvous, it can not wait at all (impatient), or it can specify a time out after which it will give up on a rendezvous.

Task types

All of the above tasks are of an anonymous type. Ada allows you to create a task type (via a special and non consistent syntax from all other type declarations) and to declare objects of the type. They can be declared dynamically (through an allocater) as well as local variables. They can be composed into other objects - you could have an array of tasks, or a record, a component of which, is a task.

Problems with the rendezvous mechanism

The rendezvous mechanism is a synchronisation as well as a communication mechanism. Tasks that should normally run asynchronously and which want to pass data between them are required to stop and "have a chat", instead of simply leaving the data for the other task. The solution to this problem is usually to creat an intermediary task that manages the data flow, and, the designer hopes, is always available to respond to a rendezvous from either task. It sits between the two tasks, acting as an emissary for both.

Protected Objects

A simpler solution can be provided in Ada95 which has a concept of a protected object or type. Protected objects provide for task safe access to data via entries and guards, without the overhead of a separate task being created. The subprograms and entries inside a protected object are executed in the context of the calling task. As with tasks types you can create arrays of protected type, and compose other objects from them.

All the examples in this section come from Introducing Ada9x, John Barnes.

An example of a protected entry follows

```
protected Variable is
   function Read return item;
      procedure Write(New_Value:item);
private
 Data :item;
end Variable;

protected body Variable is
      function Read return item is
    begin
          return Item;
    end;

    procedure Write(New_value :item) is
     begin
           data := New_Value;
      end;
end Variable;
```

Functions are only allowed read access to the data, procedures can manipulate them in any way they wish. Procedure calls are exclusive, that is only one task can access a procedure at any one time. Multiple reads can occur concurrently.

Another example is a semaphore, to provide exclusive access to resources.

```
protected type Counting_Semaphore (Start_count :Integer := 1) is
        entry Secure;
   procedure Release;
      function Count return integer;
private
   Current_count :integer := start_count;
end;

protected body counting_semaphore is
        entry Secure when Current_count > 0 is
        begin
```

```
                Current_Count := Current_count + 1;
        end;

        procedure Release is
        begin
                Current_Count := Current_count + 1;
        end;

        function Count return integer is
        begin
                return current_count;
        end;
end Counting_semaphore;
```

When called the barrier on the entry is queued. If it is false, the task is queued, pending it becoming true.

---

to the index...

1. 15 Concurrency support
2. Overview
3. Tasks
4. The rendezvous
5. Task types
6. Problems with the rendezvous mechanism
7. Protected Objects

---

to the index...

16 Language interfaces

Overview

No matter how good a new language is, there is usually a great deal of investment in existing programs. The designers of Ada realised this and provide a standard mechanism to interface with other language. The facilites provided are optional, and just which languages are supported depend on the compiler vendor. Considerably more support has been built into Ada95. Consult your compiler's reference manual for full details.

Interfacing with foreign languages (Ada83)

Assume you are on a Unix system and you wish to make use of the kill command. You should perform the following.

```
function kill(  pid     :in integer;
                        sig     :in integer) return integer;

pragma interface(C,kill);
```

The first parameter to pragma interface is the language of the called routine, the second the name the routine is known by in Ada.

Another example for a package is

```
package MATHS is
        function sqrt(x:float) return float;
        function exp (x:float) return float;
private
        pragma interface(fortran,sqrt);
        pragma interface(fortran,exp);
end MATHS;
```

The pragma interface cannot be used with generic subprograms.

Interfacing with foreign languages (Ada95)

Ada95 makes extensive use of the predefined libraries to enable data type translation between Ada and foriegn languages. Together with the pragmas import, export and convention they allow Ada systems to be easily used in a multi-language environment.

Pragma Import

The pragma Import allows the inclusion of foreign language entities within an Ada program, such as variables or procedures/functions. The code below shows an example of the use of pragma import for the Unix function read.

Pragma Import consists of three parameters,
- the language convention (only Ada and Instrinsic must be supported)
- Ada

- Instrinsic

- C

- Fortran

- Cobol

- any other implementation defined value

- the Ada name for the object
- the foreign langauge name for the object, as a string.

```
------------------------------------------------------------
procedure read( file_descriptor :in     integer;
                buffer          :in out string;
                no_bytes        :in     integer;
                no_read         :   out integer) is

        function read(  file_descriptor :integer;
                        buffer  :system.address;
                        no_bytes:integer) return integer;

        pragma import(C, read, "read");
begin
        no_read := read(file_descriptor,
                        buffer(buffer'first)'address,
                        no_bytes);
end;
```

The interface packages

The interface package hierachy consists of packages designed to ease the interfacing of Ada with other langauges. The standard suggests interfaces for C, COBOL and Fortran.
The package hierachy is

```
package Interfaces
```

```
package Interfaces.C
package Interfaces.C.Pointers
package Interfaces.C.Strings
package Interfaces.COBOL
package Interfaces.Fortran
```

The packages give sufficient power to deal with most foreign language interfacing issues.

One area in which Ada has problems interfacing to other languages is functions that contain non homogenous variable length parameter lists, such as printf. Such functions are inherently type unsafe, and there is no satisfactory way to handle such situations.

Ada can, however, handle functions where the argument types are homogenous; this is achieved through the use of unconstrained array types.

A simple example would be a C function...

```
void something(*int[]);
```

it could be accessed as follows...

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

procedure something(item :vector) is

  function C_Something(address:system.address);
   pragma import(C, C_something, "something);

begin
 if item'length = 0 then
        C_something(system.null_address);
      else
          C_something(item(item'first)'address);
  end if;
end;
```

A larger and more complex example is given below for the Unix C function execv. The added complication is caused by the necessity of translation from Ada strings to C style character arrays (and is not necessarily as good as it could be. See the LRM for more information on using interfaces.c child family).

The C function is defined as...

```
int execv(const char *path, char *const argv[]);
```

you need to declare a few things...

```
-------------------------------------------------------
type string_ptr is access all string;
type string_array is array(natural range <>) of string_ptr;
```

```
function execv( path           :string;
                arg_list       :string_array) return interfaces.c.int;

-------------------------------------------------------
-- execv replaces the current image with a new one.
-- A list of arguments is passed as the command line
-- parameters for the called program.
--
-- To call this routine you can...
--
-- option2      :aliased string := "-b";
-- option3      :aliased string := "-c";
-- option4      :string := "Cxy";
-- result       :interfaces.C.int;
-- ...
-- result := execv(path => "some_program",
--                 -- build up an array of string pointers...
--                 argv => string_array'(new string'("some_program"),
--                                       new string'("-a"),
--                                       option2'Unchecked_access,
--                                       option3'Unchecked_access,
--                                       new string'('-' & option4));
--
-- Any mixture of dynamic string allocation and
-- 'Unchecked_access to aliased variables is allowed. Note however that
-- you can't do  "some_string"'access, as Ada requires a name,
-- not a value, for the 'access attribute to be applied to.
-------------------------------------------------------
```

This function could be implemented as follows. Note that the address of the first item in the array is passed, not the address of the array. Arrays declared from unconstrained array types often have a vector which includes extra information such as the lower and upper bounds of the array.

```
function execv( path    :string;
                argv    :string_array )return interfaces.C.int is

        Package C renames Interfaces.C;
        Package Strings renames Interfaces.C.Strings;

        C_path  :constant Strings.chars_ptr(1..path'length+1)
                                        := Strings.New_string(path);

        type char_star_array is array(1..argv'length + 1) of
                                           Strings.char_array_ptr;

        C_argv  :char_star_array;
        index   :integer;
        result  :C.int;
```

```
        ------------------------------------------------------------
        function C_execv(     path    :Strings.Char_ptr;
                              C_arg_list:Strings.Char_ptr)
                                      return C.int;
        pragma import(C, C_execv, "execv");
        ------------------------------------------------------------
begin

       -- set up the array of pointers to the strings

        index := 0;
        for i in argv'range loop
                index := index + 1;
                C_argv(index) := Strings.New_String(argv(i).all));
        end loop;


        -- append C style null to the end of the array of addresses

        C_argv(C_argv'last) := Strings.Null_Ptr;

        -- pass the address of the first element of each
        -- parameter, as C expects.

        result := C_execv( C_path(1)'address, C_argv(1)'address));

        -- Free up the memory as obviously this didn't work
        for i in argv'range loop
                Strings.free(argv(i));
        end loop;

        Strings.free(C_path);

        return result;
end execv;
```

---

to the index...

17 Idioms

Overview

Using a language effectively requires more than a knowledge of it's syntax and semantics. An acquaintaince with common problems, and the various methods to solve these problems can simplify the process of writing programs, and in this respect Ada is no exception. This chapter is involved in exposing some of the more common idioms of programming in Ada.

Introduction

Abstraction is one of the most important part of programming, and it should always be considered when programming solutions. You must always make an effort to distinguish *what* service a type is providing and *how* it is implemented. You expose the what, but try to hide the how.

Similarly a knowledge of how to hide things in Ada is important. Ada provides for (predominantly) orthogonal concepts of type/behaviour, and modularity/controlling visibility. The semantics of a program (how it runs, what it does) are independent of whether you design the program carefully for modularity and robustness in the face of change. Clearly though it is more often than not worth the effort to design a system competently.

Abstraction

Abstractions allow us to present only that which we want the user to be concerned with, and not giving access to information which is irrelavent. For example in the all too often used stack example, the user should not be too concerned with whether the implementation uses arrays or linked lists.

A rough stab at an abstraction could be...

```
    package stacks is

            max     :constant  := 10;
            subtype count is integer range 0..max;
            subtype index is range 1..max;

            type list is array(index) of integer;

            type stack is
                    record
                            values : list;
                            top : count;
                    end record;

            overflow        :exception;
            underflow       :exception;
```

```
        procedure push(item :in out stack; value :in integer);
        procedure pop(item :in out stack; value : out integer);

        function full(item :stack) return boolean;
        function empty(item :stack) return boolean;

        -- return an empty initialized stack
        function init return stack;

end;
```

Here however the details of the implementation are rather obvious; the **how** tends to get in the way of the **what**. This can be changed simply by moving whatever is not relavent into the private section...

```
package stacks is

        type stack is private;
        -- This is called a partial view of the type stack
        -- It makes info about its implementation inaccessable.

        overflow        :exception;
        underflow       :exception;

        procedure push(item :in out stack; value :in integer);
        procedure pop(item :in out stack; value : out integer);

        function full(item :stack) return boolean;
        function empty(item :stack) return boolean;

        -- return an empty initialized stack
        function init return stack;

private
        -- full view of the type, along with other private details.
        max     :constant  := 10;
        subtype count is integer range 0..max;
        subtype index is range 1..max;

        type list is array(index) of integer;

        type stack is
                record
                        values : list;
                        top : count;
                end record;
end;
```

Although the programmer who uses your abstraction can see all the details if they have access to the code, they can't write their programs to rely on this information. Also the separation clearly enforces the notion of what is important for the abstaction, and what is not.

Alternatively we could change the private section to reflect a linked list implementation...

```
package stacks is

        type stack is private;
        -- make info about its implementation inaccessable.
        -- this is called a partial view of the type stack

        overflow        :exception;
        underflow       :exception;

        procedure push(item:in out stack; value:in integer);
        procedure pop(item:in out stack; value : out integer);

        function full(item:stack) return boolean;
        function empty(item:stack) return boolean;

        -- return an empty initialized stack
        function init return stack;

private
        type stack;

        type ptr is access stack;

        type stack is record
                value :integer;
                next    :ptr;
        end;
end;
```

A user program could then use either package as follows...

```
with stacks;    use stacks;

procedure demo is
        a       :stack := init;
        b       :stack := init;
        temp    :integer;
begin
        for i in 1..10 loop
                push(a,i);
        end loop;

        while not empty(a) loop
                pop(a, temp);
                push(b, temp);
        end loop;
end;
```

The concept of being able to substitute a different implementation, or more precisely, only relying on the public

interface, is a very important design principle for building robust systems.

(Note that these two implementations aren't quite identical. Consider the case where we have

```
with stacks; use stacks;

procedure not_quite_right is
        a, b:stack
begin
        push(a,1);
        push(a,2);

        b := a;
end;
```

An array implementation will work correctly when copied, but a linked list implementation will only copy the head pointer. Both a and b will then point to the same linked list. The solution to this is presented later on. For the examples presented here on, assume that this problem has been fixed).

Creating abstractions from other abstractions (code reuse)

Private inheritance

It is important to establish the difference between what services a type provides, and how it implements that service. For example a list abstraction (that has appropriate routines, and itself may be implemented as a linked list, or an array) can be used to implement a queue abstraction. The queue abstraction will have only a few operations:

```
add_to_tail
remove_from_head
full
empty
init
```

We want to ensure there is a clear distinction between the abstraction and the implementation in our program. Preferably the compiler should check and ensure that no-one makes the mistake of calling routines from the implementation, rather than the abstraction.

Below is an example package which can lead to problems.

```
package lists is

        type list is private;

        procedure add_to_head(item:in out list; value :in integer);
        procedure remove_from_head(item:in out list; value :out integer);
```

```
        procedure add_to_tail(item:in out list; value:in integer);
        procedure remove_from_tail(item:in out list; value: out integer);

        function full(item: list) return boolean;
        function empty(item:list) return boolean;

        function init return list;

private
        type list is... -- full view of type.
end;



with lists;     use lists;      -- abstraction of lists

package queues is

        type queue is new list;
        -- inherits operations from the list type
        -- i.e. the following are "automagically" (implicity) declared
        --
        -- procedure add_to_head(item:in out queue; value :in integer);
        -- procedure remove_from_head(
        --      item    :in out queue;
        --      value   :   out integer);
        -- etc.


end queues;
```

Here type queue inherits all of the operations of the list type, even those that aren't appropriate, such as remove_from_tail. With this implementation of queue, clients of the abstraction could easily break the queue, which should only allow insertion at the tail, and removal from the head of the list.

For example a client of the queues package (something that "with queues"), could easily do the following

```
with queues;    use queues;

procedure break_abstraction is
        my_Q    :queue;
begin
        add_to_head(my_Q, 5);
        add_to_tail(my_Q, 5);

        -- queues should only add at the tail, remove from the head,
        -- or vice-versa
end;
```

What we need to do is advertise a different abstraction, but reuse the list abstraction privately.

```
with lists;     -- this is only used in the private section.

package queues is
        type queue is private;

        procedure remove_from_head(item:in out queue; value :out integer);
        procedure add_to_tail(item:in out queue; value:integer);

        function full(item: queue) return boolean;
        function empty(item:queue) return boolean;

        function init return queue;

private
        type queue is new lists.list;

end;



package body queues is

-- Perform a type conversion (from queue to list), and then call
-- the appropriate list routine.

use lists;
-------------------------------------------------------------------------
procedure remove_from_head(item:in out queue; value :out integer) is

begin
        lists.remove_from_head(list(item), value);
end;


-------------------------------------------------------------------------
function full(item: queue) return boolean is

begin
        return lists.full(list(item));
end;

...

function init return queue is
begin
        return queue(lists.init);
end;

end;
```

Another example

Let's say we wish to create a stack package, but we don't want to recode all of the routines. We may already have a linked list implementation that could form the basis of the storage for the stack. The list package stores integers.

Assume the package lists is defined as follows...

```
package lists is

        type list is private;

        underflow       :exception;

        procedure insert_at_head(item:in out list; value :in integer);
        procedure remove_from_head(item:in out list; value:out integer);

        procedure insert_at_tail(item:in out list; value :in integer);
        procedure remove_from_tail(item:in out list; value: out integer);

        function full(item:list) return boolean;
        function empty(item:list) return boolean;

        -- returns an empty initialized list
        function init return list;

private
        ...
end;
```

We can then make use of Ada's ability to hide the full view of a type in the private section. Here we declare a brand new type (as far as the client is concerned), which is actually implemented as a derived type. The client can't "see" this - the "under the hood" details remain well hidden.

We implement the stack as...

```
with lists;

package stacks is

        type stack is private;

        underflow :exception;

        procedure push(item:in out stack; value:in integer);
        procedure pop(item:in out stack; value : out integer);

        function full(item:stack) return boolean;
        function empty(item:stack) return boolean;

        -- return an empty initialized stack
        function init return stack;
```

```
    private
            -- create a new type derived from an existing one.
            -- This is hidden from the clients of the type
            type stack is new lists.list;


            -- stack inherits all the operations of type list.

            -- They are _implicity_ declared as
            --
            -- procedure insert_at_head(item:in out stack; value :in integer);
            -- procedure remove_from_head(item :in out stack;
            --                            value:   out integer);
            -- ...
            -- function full(item:stack) return boolean;
            -- function empty(item:stack) return boolean;
            -- function init return stack;

    end;
```

We now have to relate the implicity declared routines to those that are advertised publically. This is done by a simple "call through" mechanism.

```
    package body stacks is

    procedure push(item:in out stack; value:in integer) is
    begin
            insert_at_head(item, value);
    end;
    -- you can make it more efficient by...
    pragma inline(push);


    procedure pop(item:in out stack; value : out integer) is
    begin
            remove_from_head(item, value);
    exception =>
            when lists.underflow =>
                    raise stacks.underflow;
    end;

    ...
    end stacks;
```

This is ok for publically advertised routines that have different names from the implicity declared routines (those inhertied from type list).

However in the package specification there are two functions full which have the profile

```
    function full(item:stack) return boolean;
```

One is explicity declared in the public section, one implicity declared in the private section. What's going on? The first function specification in the public part of the package is only a promise of things to come. It is an as-yet unrealised function. It will be completed in the package body. As well we have another (implicity) declared function which *just happens to have the same name and profile*. You still have to describe to the Ada compiler how you are going to implement the function declared in the public part of the package. The Ada compiler can see that both functions have the same name and profile, but it does not assume that public function should be implemented by the private function.

For example imagine that integer_lists.full always return false, to indicate that the linked list was never full, and could always grow. As the implementor of the stack package you may have decided that you wanted to enforce a limit on the stack, so that it would only contain 100 elements at most. You would then write the function stacks.full accordingly. It would be incorrect for the implementation of function full to default back to the linked list implementation.

So far so good. How do I write the function full?

Because of the visibility rules of Ada, the explicit public declaration completely hides the implicity declared private one, and so is not accessable at all. The only solution is to call on the function in the list package, which you have to do explicity.

```
package body stacks is


function full(item:stack) return boolean is
begin
        -- call the original full in the other package.

        return lists.full(lists.list(item));
                            -- type conversion of parameter

end;

...
end stacks;
```

This seems all terribly obtuse and annoying. Why does Ada force this upon you? The reason you are encountering this sort of problem is because of the nature of constructing programs with independent name spaces (different scopes where identical names do not clash with each other). Ada is merely providing a solution to a problem that is an inevitable consequence of name spaces. The alternative, of having to work with a language in which there can be no names the same is worse than this solution. So don't shoot the messenger just because you don't like the message :-).

This is a very important point to note. Don't go on unless you understand this point.

The full package body for stacks would be

```
use lists;

package body stacks is
```

```
procedure push(item:in out stack; value:in integer) is
begin
        insert_at_head(item, value);
end;


procedure pop(item:in out stack; value : out integer) is
begin
        remove_from_head(item, value);

exception =>
        when lists.underflow =>
                raise stacks.underflow;
end;
-- note how the exception advertised in the other package
-- is "translated" to an exception defined within this package.


function full(item:stack) return boolean is
begin
        return lists.full(list(item));
end;


function empty(item:stack) return boolean is
begin
        return lists.empty(list(item));
end;


-- return an empty unitialized stack
function init return stack is
begin
        return stack(lists.init);
end;

end stacks;
```

Creating abstractions by using generic abstractions

Often a generic package that implements a suitable data structure can be used to implement another abstraction. This is very similar to the section presented previously.

Assume the package generic_lists is defined as...

```
generic
        type element is private;

package generic_lists is
```

```
        type list is private;

        underflow      :exception;

        procedure insert_at_head(item:in out list; value :in element);
        procedure remove_from_head(item:in out list; value:out element);

        procedure insert_at_tail(item:in out list; value :in element);
        procedure remove_from_tail(item:in out list; value: out element);

        function full(item:list) return boolean;
        function empty(item:list) return boolean;

        -- returns an empty initialized list
        function init return list;

private
        ...
end;
```

We can instantiate this generic to create a new package that will store the integers for us. The question is "where do we want to instantiate it, such that it won't get in the way of the abstraction, or cause other developement problems?". The solution to this (as it is to a lot of these type of questions) is in the private section of the package.

We can then use the generic as follows...

```
with generic_lists;

package stacks is

        type stack is private;

        underflow :exception;

        procedure push(item:in out stack; value:in integer);
        procedure pop(item:in out stack; value : out integer);

        function full(item:stack) return boolean;
        function empty(item:stack) return boolean;

        -- return an empty initialized stack
        function init return stack;

private

        -- Instantiate the generic to create a new package.
        package integer_lists is new generic_lists(integer);


        type stack is new integer_lists.list;
```

```
        -- stack inherits all the operations of type list, as before.

end;
```

The package body is the same as previously described.

Here we can enforce privacy, without changing what the type does for a client.

Abstractions from composition

Abstractions can be implemented by using composition (composing new abstractions by putting existing abstractions together, typically in a record) and by forwarding subprogram calls onto the appropriate component object. (This is all pretty simple stuff!).

```
with lists;     use lists;

package queues is

        type queue is private;

        procedure remove_from_head(item:in out queue; value :out integer);
        procedure add_to_tail(item:in out queue; value:integer);

        function full(item: queue) return boolean;
        function empty(item:queue) return boolean;

        function init return queue;
private
        -- a queue is composed from the following items...

        type queue is record
                values:list;
                no_of_elements:integer;
        end record;
end;



package body queues is

procedure remove_from_head(item:in out queue; value:out integer) is
begin
        remove_from_head(item.values, value);
        item.no_of_elements := item.no_of_elements - 1;
end;


procedure add_to_tail(item:in out queue; value:integer) is
begin
```

```
        add_to_tail(item.values, value);
        item.no_of_elements := item.no_of_elements + 1;
end;


procedure reset (item: in out queue) is ...


function full(item: queue) return boolean is
begin
        return full(item.values);
end;

function empty(item:queue) return boolean is
begin
        return item.no_of_elements = 0;
end;

end;
```

In this example calls on a queue are "forwarded" onto the list component, which is responsible for implementing some aspect of the queue abstraction (in this case a major part!).

Abstracting common functionality - families of abstractions

Both implementations of the stack described above provided a common set of routines, that is push, pop etc. In Ada we can state the commonalities using a common type. The common type will describe what routines are to be provided, and their profiles, but not actually describe how to implement them. Type derived from this type are forced to implement each subprogram in its own way. This is called an abstract type. It is an abstraction abstraction!

The reason we go to all of this trouble is to tell the clients of the stack that no matter what implementation they choose, they can be sure of getting at least the functionality described by the abstract type.

```
package stacks is

        type stack is abstract tagged null record;
        -- An abstract type with no fields, and no "real" operations.
        -- Ada required the "tagged" description for abstract types

        underflow       :exception;
        overflow        :exception;

        procedure push(item:in out stack; value:in integer) is abstract;
        procedure pop(item:in out stack; value : out integer) is abstract;

        function full(item:stack) return boolean is abstract;
        function empty(item:stack) return boolean is abstract;
```

```
        function init return stack is abstract;

end;
```

Here we have made the type abstract - this package just provides a series of subprogram that must be implemented by someone else. We could also have made the type private...

```
package stacks is

        type stack is abstract tagged private;

        -- subprograms
        ...

private

        type stack is abstract tagged null record;

end;
```

Either in the same package (or more typically in another package) we can extend the type, and create a real type that implements the abstract type...

```
with lists;
with stacks;

package unbounded_stacks is

        type bounded_stack is new stacks.stack with private;
        -- bounded_stack is derived from the empty stack record,
        -- with some extra bits added on that are described
        -- in the private section.

        procedure push(item:in out unbounded_stack; value:in integer);
        procedure pop(item:in out unbounded_stack; value : out integer);

        function full(item:unbounded_stack) return boolean;
        function empty(item:unbounded_stack) return boolean;

        -- return an empty initialized unbounded_stack
        function init return unbounded_stack;

private

        -- Extend the empty stacks.stack record with one
        -- 'more' field.
        -- All calls will have to be forwarded onto the
        -- internal linked list.

        type unbounded_stack is new stacks.stack with
                record
```

```
                        values:lists.list;
                end record;

    end;
```

As described in the section on composition, you will have to forward calls onto the linked list within the unbounded_stack. The package body would therefore be...

```
    package body unbounded_stacks is

            procedure push(item:in out unbounded_stack; value:in integer) is
            begin
                    lists.insert_at_head(item.values, value);
            end;

            procedure pop(item:in out unbounded_stack; value : out integer) is
            begin
                    lists.remove_from_head(item.values, value);
            exception
                    when lists.underflow =>
                            raise stacks.underflow;
            end;

            ...
    end unbounded_stacks;
```

Packaging

In Ada, types and packages are almost completely independent of each other. This allows you to devise a solution to a problem in terms of types and operations, and later decide on how these parts should be split into pieces to meet the software engineering requirements of ease of modification, encapsulation, modularisation, reduced recompilation costs, namespace management etc.

This has been demonstrated already, in the choice to hide implementation details in the various packages above. However Ada also provides further choices that can be made in this regard.

Namespace management

In language such as C, there is only one namespace. There is no hierachy of names; no ability to 'hide' or qualify a name. A good analogy is the first version of the Mac OS, which had no directories. All filenames had to be different, and could not conflict with predefined system filenames. The solution typically adopted in C to resolve this problem is to prepend some form of semantic information in the name. For example all posix thread routines are prepended with 'pthread_'.

Ada83 solved this problem by allowing packages which contain their own namespace, in which locally defined names don't conflict with other names (the analogy is for one level of subdirectories (actually not quite accurate, as a package can contain other entities...)).

Ada95 however expands the namespace concept with the inclusion of child packages to allow for truly hierachic namespaces. Child packages are typically used to model some form of hierachy present in the problem domain. For example a unix binding may implement packages

```
unix            -- can be an empty "place holder" package
unix.file       -- file routines
unix.process    -- process operations
```

Alternatively the package hierachy can be used to reflect the inhertiance hierachy of an object oriented type. Note however the package hierachy does *not* define the inhertiance hierachy - the type declarations do that.

The example given earlier of an abstract stack, and a concrete implementation of an unbounded stack is a good example of how we could have chosen a different organisation, without affecting the meaning of the abstraction.

```
with stacks; use stacks;
with lists;

package stacks.unbounded is

        type unbounded_stack is new stack with private;
        -- partial view of the bounded_stack

        -- override the primitive operations with concrete services

        procedure push(item:in out bounded_stack; value:in integer);
        procedure pop(item:in out bounded_stack; value : out integer);

        function full(item:bounded_stack) return boolean;
        function empty(item:bounded_stack) return boolean;

        function init return unbounded_stack;

private
        ...
end;
```

We could create another type, bounded_stack, and decide to put it into the package stacks.bounded.

```
package stacks.bounded is
        type bounded_stack is new stacks.stack with private;
        ...
end;
```

Handling dynamic structures properly

As discussed earlier in the implementation of a stack with a linked list, making a copy of such a type through an assignment statement doesn't quite work. The problem is that copying often only copies a pointer to the head of a list (or tree) and does not copy the entire list. Another problem is that the space allocated to a linked list is often not recovered when an object goes out of scope.

An example which highlights these problems is...

```
with lists;     use lists;

procedure bad_voodoo is

        a, b:list;
        result :integer;

        procedure loose_memory is
                c :list;
        begin
                insert_at_head(c, 1);
                insert_at_head(c, 2);
        end; -- when you get to here, c goes out of scope
             -- and you lose the memory associated with the nodes!
begin
        insert_at_head(a, 1);
        insert_at_head(a, 2);
        insert_at_head(b, 3);
        insert_at_head(b, 4);

        b := a;
        -- b and a now point to the same list, and all of b's nodes
        -- have been lost!

        remove_from_tail(a, result);
        -- affects both a and b

end;
```

In Ada95 you can solve these problems by providing special routines that are "automagically" called when a values are being copied, when they go out of scope, or when they are initialized. For this to happen, the object in question must be derived from a special type called Controlled, defined in package Ada.Finalization.

The three routines automagically called are called Initialize, Adjust and Finalize. They are called in the following contexts...

```
with lists;     use lists;

procedure demo is
        a :list;                -- Initialize(a);
        b :list := Init;        -- Initialize not called
begin
        insert_at_head(a, 1);
        insert_at_head(a, 2);

        insert_at_head(b, 3);
        insert_at_head(b, 4);
```

```
        a --> | 2 |-->| 1 |--> null
        b --> | 4 |-->| 3 |--> null


        b := a;

        -----------------------------------------
        Finalize(b);
        Free b's nodes, before it is overwritten

        a --> | 2 |-->| 1 |--> null
        b --> null


        Copy a to b.
        Now they _both_ point at the same list

        a --> | 2 |-->| 1 |--> null
        b ----^


        Adjust(b);
        b has to duplicate the list it currently
        points at, at then point at the new list.

        a --> | 2 |-->| 1 |--> null
        b --> | 2 |-->| 1 |--> null
        -----------------------------------------


end;    Finalize(a), Finalize(b).
        Delete the memory associated with both a and b.

        a --> null
        b --> null
```

Here we finally give the full details for package List. The code for it looks like...

```
with Ada.Finalization;

package lists is

        type List is private;

        underflow       :exception;

        procedure insert_at_head(item:in out list; value :in integer);
        procedure remove_from_head(item:in out list; value:out integer);

        procedure insert_at_tail(item:in out list; value :in integer);
```

```
        procedure remove_from_tail(item:in out list; value: out integer);

        function full(item:list) return boolean;
        function empty(item:list) return boolean;

        -- returns an empty initialized list
        function init return list;

private

        -- Normal stuff for the list

        type Node;
        type Ptr is access Node;

        type Node is record
                Value   :Integer;
                Next    :ptr;
        end record;


        -- Only the head of the list is special...

        type List is new Ada.Finalization.Controlled with
        record
                Head    :Ptr;
        end record;


        -- No need for Initialize (pointers are automatically set to null)

        -- procedure Initialize(item:in out list);

        procedure Adjust(item:in out list);

        procedure Finalize(item:in out list);

end;
```

Note that the routines were declared in the private section. This prevents clients from calling them when it is inappropriate.

The package body would be implemented as follows...

```
with unchecked_deallocation;

package body lists is

-- routine for deallocating nodes
procedure free is new unchecked_deallocation(node,ptr);
```

```
------------------------------------------------------------
-- Implement all the other stuff (insert_at_head, remove_from_head...)

...

------------------------------------------------------------
-- given a ptr to a list, this will allocate a new
-- identical list

function Copy_List(item:ptr) return ptr is

begin
        if item = null then
                return null;
        else
                return new node'(item.value,Copy_List(item.next));
        end if;
end;

------------------------------------------------------------
-- In the assignment b := a, b has just been
-- overwritten by the contents of a. For a linked
-- list this means that both a and b point at the
-- same object. Now have to make a physical copy
-- of the items pointed at by b, and replace the head
-- ptr with a pointer to the copy just made

procedure Adjust(item:in out list) is

begin
        item.head := Copy_List(item.head);
end;

------------------------------------------------------------
-- delete all the memory in the about to be
-- destoyed item

procedure Finalize(item:in out list) is
        upto:ptr := item.head;
        temp :ptr;
begin
        while upto /= null loop
                temp := upto;
                upto := upto.next;
                free(temp);
        end loop;

        item.head := null;
end;

end;
```

The use of controlled types gives a good degree of control over dynamic objects. It simplifies the life of a client of the type as it relieves them of the worry of having to manage the types memory. It also allows for easy substitution of dynamic and non dynamic solutions to problems.

Typically once the 3 routines (initialize, adjust, finalize) have been written, the rest of the services can be written without regard to these features.

Enforcing initialization

In C++ you can enforce the initialization of all objects by supplying a class with a constructor, which is called when an object is created. The same effect in Ada is achieved by assigning a value the result of calling a function.

For example

```
package stacks is
        type stack is tagged private;

        function init return stack;

        -- other operations
        ...
private
        type list is array(1..10) of integer;
        type stack is record
                values :list;
                top    :integer range 0..10;
        end record;
end;

with stacks; use stacks;
procedure main is
        a : stack := init;
        b : stack;
begin
        null;
end;
```

In this example the object a is properly initialized to represent an empty stack. You can only use functions provided in the package; you can't set it any other way, because the type is private. However as can be seen, the object b has not been initialized. The language has not enforced an initialization requirement.

The only way to do this is through the use of record discriminants. The partial view of the type is declared to have discriminants, even if the full view does not. As the number of fields that a record has *may* depend on a discriminant, and because Ada likes constrained objects (one's whose size is known) the combination of discriminant and private type causes the compiler to enforce all objects of the type to be initialized.

```
package stacks is
        type stack(<>) is private;
        -- declare that stack _may_ have discriminant...
```

```
        function init return stack;

        -- other operations
        ...
private
        type list is array(1..10) of integer;

        -- ...even if you have no intention of it having such
        type stack is record
                values :list;
                top    :integer range 0..10;
        end record;
end;


with stacks; use stacks;
procedure main is
        a : stack := init;
        b : stack;
        -- now this is illegal. You _must_ call a function
        -- to initialize it.
begin
        null;
end;
```

This is all quite non intuitive, and rather counter to the Ada notion of readability (IMHO). Certainly C++ manages this is a more programmer friendly way.

Mutually Recursive types

Sometimes there is a need for two (or more) types that are mutually recursive, that is they have pointers that point at each other. An example is a doctor and a patient, who wish to maintain pointers at each other. Unless you wish to place them together in the same package, Ada does not properly support you. The best you can do if you want to retain separate compilation, is to define two base types, then join them together later in other packages.

```
package doctors_and_patients is
        type doctor is abstract tagged null record;
        type doctor_ptr is access all doctor'class;


        type patient is abstract tagged null record;
        type patient_ptr is access all patient'class;
end;

with doctors_and_patients; use doctors_and_patients;
package doctors is
        type doc is new doctor with private;
```

```
        -- subprograms
        ...
private
        type doc is new doctor with
                record
                        pat:patient_ptr;
                        ...
                end record;
end;

-- similarly for patient type
```

If you can forsee all operations required, or at least those required by each other, you can place them in package doctors_and_patients. Futher operations can be added along with subsequent type declarations (for example type doc).

It's probably worth point out here that package bodies can see into package specs, so...

```
with doctors; use doctors;
package body patients is

        -- The patients body can access the doctor's services
        -- declared in the doctor's spec

        ...

end patients;

with patients; use patients;

package body doctors is

        -- The doctors body can access the patients services
        -- declared in the patient's spec

end doctors;
```

---

to the index...

---

to the index...

18 Packages and program design

When a program starts to get large it needs to broken up into smaller pieces, to make it easier to understand, distribute the work, quicker to fix, isolate changes etc.

But how to do this? Typical introductory programming courses teach programming with a large main procedure, and perhaps a couple of packages.

So how do you progress from this initial stage to a more mature view of system developement? One way is to show a simple mapping (or transformation) of a program to a functionally equivalent program with a number of packages.

The following attempts to show how an existing Ada program may be broken up into smaller pieces.

Consider the following program...

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is

   -- constants-------------------------------------
   Max_Bookings    : constant := 20;
   Max_Name_Length : constant := 40;
   No_Cars         : constant := 10;

   -- types-----------------------------------------

   subtype Name_Array is String(1..Max_Name_Length);
   subtype Registration is String(1..6);

   type Date is
      record
         Day   : Positive;
         Month : Positive;
         Year  : Positive;
      end record;

   type Booking is
      record
         On    : Date;
         Name  : Name_Array;
         Car   : Registration;
      end record;

   type Booking_Array is array (1..Max_Bookings) of Booking;

   -- Variables-------------------------------------

   Bookings : Booking_Array;
   ...other variables...

   -- Procedures & Functions-----------------------
```

```
   -- Part of booking system for after a date
   function After(A, B : Date) return Boolean is...

   -- display a date in the format dd/mm/yy
   procedure Display(Item : in Date) is...

   procedure Search_For_Empty_Booking(
      Bookings : in out Booking_Array;
      On       : in     Date;
      Position :    out Positive) is...

   procedure Make_Booking(
      Bookings : in out Booking_Array;
      On       : in     Date;
      Name     : in     Name_Array;
      Success  :    out Boolean) is...

   procedure Menu( Option : out positive) is...

begin
   loop
     ...
end;
```

Figure 1. Sample program.

Here the code makes use of different types, and has numerous procedures. We can imagine, for example, that the procedure Make_Booking calls Search_For_Empty_Booking, which in turn calls the function After to check if the requested booking is after another booking.

When we look at the code there does not seem to be much cohesion - declarations of types, constants and subprograms are in their own little sections. This is one way to split a program up into pieces, but it is not a very good way.

Another way to split it up is to note that we can see that somethings "belong" together. For example the type date is closely related to the function After and procedure Display. What we can do is group these routines more closely together.

(Imagine that there are others programs being developed, and that they also had a need of these routines. It would be silly to have programmers recode these routines. Reuse of the code would be cheaper - but it is not easy in this situation. If someone copies the code by doing "cut and paste" then any bug fixes made to the date routines will only occur in one program. It would be better if we made these routines available in one place for all programs).

The first thing we should think about is grouping these items together...

```
with Ada.Text IO;         use Ada.Text IO;
with Ada.Integer Text IO; use Ada.Integer Text IO;

procedure Main is
```

Date code

```
type Date is
   record
      Day   : Positive;
      Month : Positive;
      Year  : Positive;
   end record;

-- Part of booking system for after a date
function After(A, B : Date) return Boolean is...

-- display a date in the format dd/mm/yy
procedure Display(Item : in Date) is...
```

Booking code

```
Max Bookings    : constant := 20;
Max Name Length : constant := 40;
No Cars         : constant := 10;

subtype Name Array is String(1..Max Name Length);
subtype Registration is String(1..6);

type Booking is
   record
      On    : Date;
      Name  : Name Array;
      Car   : Registration;
   end record;

type Booking Array is
   array (1..Max Bookings) of Booking;

procedure Search For Empty Booking(
   Bookings : in out Booking Array;
   On       : in     Date;
   Position :    out Positive) is...

procedure Make Booking(
   Bookings : in out Booking Array;
   On       : in     Date;
   Name     : in     Name Array;
   Success  :    out Boolean) is...

Bookings : Booking Array;

--Procedures-not-easily-classified---------------
procedure Menu( Option : out positive) is...
```

Figure 2. Like types and procedures grouped together.

This does not change how the program runs, only how it is layed out.

You can see that the bookings require the definition of a date to be available, in fact they have to preceede the definition of the booking. The date definitions do not require any previous declarations.

```
with Ada.Text IO;         use Ada.Text IO;
with Ada.Integer Text IO; use Ada.Integer Text IO;

procedure Main is

   Date declarations

   Booking declaration
      (which require dates to be defined)

   Bookings : Booking Array;

   --Procedures-not-easily-classified---------------
   procedure Menu( Option : out positive) is...


begin
   loop
     ...
end;
```

Figure 3. Overview of program structure.

From this we can see that it is possible to take these two sections out, and place them into separate files. However we have to be careful to maintain Bookings visibility of the Date declarations. Likewise the main program has to be able to "see" all of the booking declarations, and all of the date declarations.

The bookings section only has to be able to "see" the date declarations.

The date section doesn't have to be able to "see" any other declarations.

We are now in a position to place them into separate packages...

```
package Dates is

   type Date is
      record
         Day   : Positive;
         Month : Positive;
         Year  : Positive;
      end record;

   -- Part of booking system for after a date
   function After(A, B : Date) return Boolean;

   -- display a date in the format dd/mm/yy
   procedure Display(Item : in Date);

end Dates;
```

Figure 4. Date package

Note that instead of writing out the subprograms in full, we just write them as specifications. The full description of them (with all the code) is placed in the package body.

```
with Dates; use Dates; -- this is how we let Bookings
                       -- "see" the Date definitions

package Bookings is

   Max Bookings    : constant := 20;
   Max Name Length : constant := 40;
   No Cars         : constant := 10;

   subtype Name Array is String(1..Max Name Length);
   subtype Registration is String(1..6);

   type Booking is
      record
         On    : Date;
         Name  : Name Array;
         Car   : Registration;
      end record;

   type Booking Array
      is array (1..Max Bookings) of Booking;

   procedure Search For Empty Booking(
      Bookings : in out Booking Array;
      On       : in     Date;
      Position :    out Positive);

   procedure Make Booking(
      Bookings : in out Booking Array;
      On       : in     Date;
      Name     : in     Name Array;
      Success  :    out Boolean);

end Bookings;
```

Figure 5. Bookings package

The main procedure can now be...

```
with Dates;                use Dates;
with Bookings;             use Bookings;
with Ada.Text_IO;          use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Main is

   Bookings : Booking_Array;

   --Procedures-not-easily-classified---------------
   procedure Menu( Option : out positive) is...

begin
   loop
     ...
end;

Figure 6. Revised program.
```

When this program is compiled and run, it will execute *exactly* the same as the very first example. All that is happened is we have moved code into different places, so as to make it easier to write and maintain. These are software engineering, or program construction concerns, not issues relating to how the program runs.

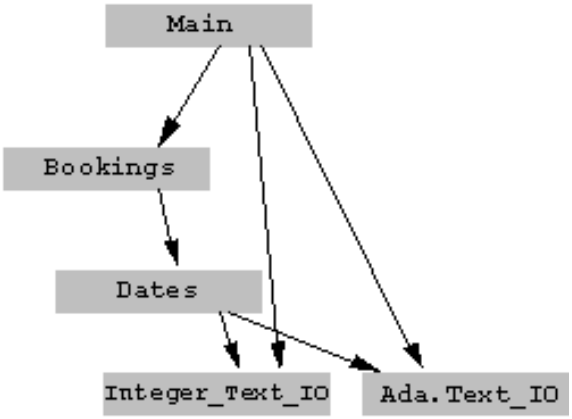
However in separating items out, we have had to be very aware of what declarations are dependent on what other declarations. If we had dates declared after bookings, bookings would not compile. This is called the dependency relationship and it is a very useful piece of information that tells us a lot about how a program is constructed.

For example if we found out that a routine in the date package was incorrect, we would need to look at all the routines that had something to do with dates to see if they were affected, and should also be fixed. In the first example, this may not have been easy. With the dependency relationships explicity described, it makes it very easy to search through large programs and find what may be affected by a bug and what isn't.

Also when a maintenance programmer has to change a program, they always have to be aware of the ripple effect - the possibility that a change in one part of a program may have consequences in other parts that are not anticipated. Dependency relationships (which are explicitly stated in the with clause) help the programmer understand how a large program is pieced together.

A "Layered" view of the program

When we think about the dependency relationships in a system, we can see that some packages don't have any dependencies. Others depend on one or two other packages, and some depend on many. In general we can depict this in a layered diagram...



Here the procedure main is in the hightest layer, and it depends on services provided by lower layer packages. An item at a lower layer however, never depends on higher layer items. Designing systems in layers makes the process of abstraction, and the notion of providing services a fairly natural one.

Interestingly by turning the diagram upside down, you get the same program strucutre as in Figure 2.

Look at putting the pieces into package specs...

When a function such as

```
-- Part of booking system for after a date
function After(A, B : Date) return Boolean;
```

is called from, say, procedure

```
procedure Make_Booking(
   Bookings : in out Booking_Array;
   On       : in     Date;
   Name     : in     Name_Array;
   Success  :    out Boolean);
```

the procedure doesn't really care how the function is written - what internal variables it has, or the order of if statements etc, so long as it produces the correct result. All it really cares about is giving two dates, and getting a boolean result. It only really depends on the specification of the function, and not the body of it.

Ada enforces this distinction when we put things into a package, by allowing only subprogram specifications in the pacakge

spec, and the full subprogram in the package body.

Look at putting pieces into a package body

When we examine our main program, we may find that subprograms such as

```
procedure Search_For_Empty_Booking(
   Bookings : in out Booking_Array;
   On       : in     Date;
   Position :    out Positive);
```

are never called from any routine other than the other booking related subprograms. For this reason there is not much point in making it publically available. We can place it soley in the package body for bookings, and not have any adverse impact on the system at all.

Designing programs with Direct_IO/Sequential_IO

Many students desiging programs with Direct_IO, or indeed any generic package, often have trouble figuring out where the instantiation should be placed.

Generally a program can be structured using child packages; the definition of an item is placed in one package, and child packages are created to contain the I/O facilities for it.

Consider, for example...

```
----------------
package Blahs is
    type Blah is...

end;
```

If we want a package to perform I/O on this type we can instantiate direct_io as a child package...

```
----------------
with Ada.Direct_IO;
with Blahs;

package Blahs.Direct_IO is new Ada.Direct_IO(Blahs.Blah);
```

However direct_io (and sequential_io) are extremely low level packages. They offer very little in terms of the functionality you would really like to have, such as the ability to search for an item, or even delete an item.

If you want to write and retrieve binary data to a file, consider the Direct_IO generic as simply a building block, used to create more sophisticated services.

(We can make an analogy between direct_io and arrays. Both are very low level concepts, and are generally used to construct higher level concepts such as hash tables).

In the example below, we create a higher level file abstraction, that supports searching and deletion of items in the file.

```
----------------
with Keys; use Keys;
```

```
package Blahs is
    type Blah is...


    -- routines to help "find" records
    function "="(Left : Blah; Right : Key_Type) return Boolean;
    function "="(Left : Key_Type; Right : Blah) return Boolean;

end;

----------------
with Ada.Direct_IO;

package Blahs.IO is
    type File_Type is limited private;

    ------------------------------------------------------
    -- Open a file in r/w mode (you may decide to include a
    -- mode parameter). Also create the file if needed.

    procedure Open(
        File : in out File_Type;
        Name : in     String);

    ----------------------------------------
    procedure Close(File : in out File_Type);


private

    -- build in support for "deleted" cells in the file
    type Component is
        record
            Item    : Blah;
            Deleted : Boolean := true;
        end record;


    package Blahs_Direct_IO is new Ada.Direct_IO(Component);


    -- provide a shorter renaming for the package
    package BDIO renames Blahs_Direct_IO;


    -- Completion of the private type advertised above
    type File_Type is new BDIO.File_Type;

end;
```

At this point, we have a file type that can be used in further child packages to build several different I/O facilities. The package has been instantiated in the private section for two reasons.

1. It prevents packages outside the Blahs.IO hierachy from accessing the low level routines in Blahs_Direct_IO, that are of no concern to them. We can force them to use the high level routines we will provide.

2 It allows child packages to 'see' the Blahs_Direct_IO package, and therefore to be able to call on these routines.

The package body would look like...

```
package body Blahs.IO is

    ------------------------------------------------------
    -- Forward the call to the blah_direct_io package

    procedure Open(
        File : in out File_Type;
        Name : in     String) is
    begin
        BDIO.Open(
            File => BDIO.File_Type(File),
            Mode => BDIO.inout_file,
            Name => Name);

    exception
        when BDIO.Name_Error =>
            BDIO.Create(
                File => BDIO.File_Type(File),
                Mode => BDIO.inout_file,
                Name => Name);
    end;


    ----------------------------------------
    procedure Close(File : in out File_Type) is
    begin
        BDIO.Close( BDIO.File_Type(File));
    end;

end;
```

For example you may want to produce a package with facilities for reading, deleting, searching etc, while another package could be used to consolidate a file.

```
    ------------------------------------
    with Keys; use Keys;

    package Blahs.IO.Advanced_Features is

        procedure Read(
            File  : in out File_Type;
            Key   : in     Key_Type;
            Item  :    out Blah;
            Found :    out Boolean);

        procedure Delete(
            File : in out File_Type;
```

```
        Key  : in     Key_Type;
        Item :    out Blah);

    etc.

end;

---------------------------------------
package Blahs.IO.Utility_Routines is

    -- Rewrite the file to remove empty cells.
    -- This presumes that free space is not managed in some
    -- more sophisticated manner
    procedure Compact_File(File : in out File_Type);

end;
```

The package body for these packages can be roughly sketched out as follows. Note that it makes a simplifying assumption as to how it searches for a key value.

```
---------------------------------------
with Keys; use Keys;

package body Blahs.IO.Advanced_Features is

    -- Call on the Read facility from package BDIO
    procedure Read(
        File  : in out File_Type;
        Key   : in     Key_Type;
        Item  :    out Blah;
        Found :    out Boolean) is

        Data : Component;
        use BDIO;
    begin
        for i in 1..BDIO.Size(BDIO.File_Type(File)) loop

            BDIO.Read(BDIO.File_Type(File), Data);

            -- use the "=" operator defined in Blahs to
            -- compare what we have read
            if (not Data.Deleted) and then
               (Data.Item = Key) then

                Found := True;
                Item  := Data.Item;
                return;
            end if;
        end loop;

        -- Searched-for data not found.

        return False;
    end;
```

```
    etc.

end;
```

It is informative to note that if we examine the non private interfaces in this package hierachy, we will see no references to the generic Direct_IO at all.

```
----------------
package Blahs is
    type Blah is...
end;

----------------
package Blahs.IO is
    type File_Type is limited private;
    procedure Open(...
    procedure Close(...
end

----------------
package Blahs.IO.Advanced_Features is
    procedure Read(...
    procedure Delete(...
end;

----------------
package Blahs.IO.Utility_Routines is
    procedure Compact_File(...
end;
```

To use these routines you may have code as follows...

```
with Keys;                      use Keys;
with Blahs;                     use Blahs;
with Blahs.IO;                  use Blahs.IO;
with Blahs.IO.Advanced_Features;use Blahs.IO.Advanced_Features;

procedure Demo is
    Data  : Blah;
    File  : Blahs.IO.File_Type;
    Key   : Key_Type;
    Found : Boolean;

begin
    -- Read some data from the user..
    get(Key);

    Open(File, "my_file.dat");
    Read(File, Key, Data, Found);

    if Found then
```

```
        -- do whatever
    else
        -- do whatever
    end if;
end;
```

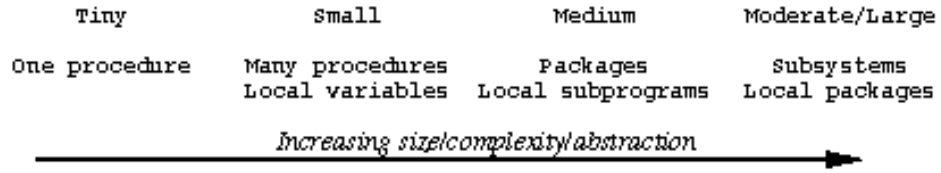
These child packages can also be written as generic packages, so that this structure can be replicated for different file types.

Private packages

When we develop systems we find that new issues appear as we tackle larger and larger programs. Initially splitting a program into subprograms makes developement easier. We can place variables that are only used by one subprogram inside it - hiding it from outside view and interference.

As the number of subprograms grows, however, we find a need to split them up into another level of grouping - the package. This gives us the opportunity to hide entire subprograms inside package bodies - subprograms that are not meant for other routines to use.

After the number of packages starts to grow, we turn to child packages to create subsystems, in which a family of logically related types and functionality is packaged together. Each subsystem provides services to all clients by advertising them in it's client specifications. Once again however, we find at this high level of abstraction some of the services that are offered should only be available to those within the subsystem.



Private packages allow you to structure your program with local packages, and prevent offering services to anyone who wants them. Banks are, of course, more secure if the internal procedures they make use of to deliver services to customers, are not made available to those same clients. We definately want the same level of security for our subsystems!

How do I know when to use private packages?

Armies during war time generally run on a "needs to know" basis. You only tell people what they need to know. Similarly when you progressed from to each new level in the diagram above, you dealt with the issue of what to hide on a "needs to know" basis. If a client doesn't need to know the details of a service then they are hidden from view. This split is based on the difference between what service is offered, and how it is implemented. You simply need to apply the same knowledge, only at a higher level.

This may come about from a hierachical object decomposition, where the services provided at a high level analysis provide the subsystem level interfaces. Further elaboration of the design results in objects which may only be needed to implement the subsystem services already offered.

A case study

A temporal assertions package (used for making assertions such as "this event must happen within 5 seconds of that event", or "this event must never occur before that event") has been developed at RMIT. Several concepts emerged from the analysis of the requirements.

```
Events  Something that occurs at an instant in time
Intervals       A duration, marked by a beginning and end event
Predicates      A statement involving events, intervals, boolean
                connectives (and, or, not, xor) and special
```

```
                conditions, such a "must occur before", and
                "must occur at least once"

    TriState Logic
                A special form of boolean logic consisting of three
    states (false, true, don't know)
```

As well as these types/concepts discovered at the analysis stage, other types were found at the design stage, such as data structures to hold the events and predicates that would be declared.

The packages developed for this were...

```
    package Assertion
            declared major types

    private package Assertion.Events_Table
            maintains data structure for storing events

    package Tri_State
            3 valued boolean logic
```

Assertion contains most of the code of the program. Package Tri_State, although not involved in any other part of the system, was felt to be a useful sort of package, and was therefore not declared private.

Package Assertion.Events_Table maintains the data structure for storing the events that clients have declared. As an alternative it could have been included in the package body of Assertion in a number of different ways.

One technique would be to dump all of the code, data structures and variables in the package body. This would not be satisfactory as it would make the package body more cluttered.

Another technique would be to place a package inside the package body...

```
    package body Assertion is

        -------------------------------
        package Events_Table is

            type Event_Table is private;
            procedure Insert(
                 Into : in out Event_Table;
                 Item : in     Event);
            ...
        end Events;

        package body Events_Table is
            procedure Insert(....) is
            ....

        end;
        -------------------------------

        Stored_Events : Events_Table.Event_Table;

        -- rest of Assertion package
```

```
end Assertion;
```

Although it fixes the problem of cluttering, it still causes the package to be longer than it needs to be (increased compilation times), harder to develop (it is harder to make calls on it). As well the structure of the program is harder to understand (the program structure is easiest to see when we can easily see the packages that make up the program).

For this reason the package was made a private child package of Assertion.

A package because

We need to hide away the low level details of how the table is implemented

A child package because

It needs to see the declarations of type Event in the spec of Assertion

It's name clearly links it into the Assertion subsystem of a program

A Private child package because

No other part of a program, apart from the Assertion subsystem, needs to know the internal details of how events are stored away.

Appendix A Text_IO package

```
with Ada.IO_Exceptions;
with System;
with System.Parameters;

package Ada.Text_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File, Append_File);

   type Count is range 0 .. System.Parameters.Count_Max;

   subtype Positive_Count is Count range 1 .. Count'Last;

   Unbounded : constant Count := 0;
   --  Line and page length

   subtype Field is Integer range 0 .. System.Parameters.Field_Max;

   subtype Number_Base is Integer range 2 .. 16;

   type Type_Set is (Lower_Case, Upper_Case);

   ---------------------
   -- File Management --
   ---------------------

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

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

   procedure Close  (File : in out File_Type);
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);

   function Mode (File : in File_Type) return File_Mode;
   function Name (File : in File_Type) return String;
   function Form (File : in File_Type) return String;

   function Is_Open (File : in File_Type) return Boolean;

   -----------------------------------------------------
   -- Control of default input, output and error files --
   -----------------------------------------------------

   procedure Set_Input  (File : in File_Type);
   procedure Set_Output (File : in File_Type);
   procedure Set_Error  (File : in File_Type);

   function Standard_Input  return File_Type;
   function Standard_Output return File_Type;
```

```
   function Standard_Error  return File_Type;

   function Current_Input  return File_Type;
   function Current_Output return File_Type;
   function Current_Error  return File_Type;

   type File_Access is access constant File_Type;

   function Standard_Input  return File_Access;
   function Standard_Output return File_Access;
   function Standard_Error  return File_Access;

   function Current_Input  return File_Access;
   function Current_Output return File_Access;
   function Current_Error  return File_Access;

   --------------------
   -- Buffer control --
   --------------------

   procedure Flush (File : in out File_Type);
   procedure Flush;

   -----------------------------------------
   -- Specification of line and page lengths --
   -----------------------------------------

   procedure Set_Line_Length (File : in File_Type; To : in Count);
   procedure Set_Line_Length (To : in Count);

   procedure Set_Page_Length (File : in File_Type; To : in Count);
   procedure Set_Page_Length (To : in Count);

   function Line_Length (File : in File_Type) return Count;
   function Line_Length return Count;

   function Page_Length (File : in File_Type) return Count;
   function Page_Length return Count;

   ------------------------------------
   -- Column, Line, and Page Control --
   ------------------------------------

   procedure New_Line (File : in File_Type; Spacing : in
Positive_Count := 1);
   procedure New_Line (Spacing : in Positive_Count := 1);

   procedure Skip_Line (File : in File_Type; Spacing : in
Positive_Count := 1);
   procedure Skip_Line (Spacing : in Positive_Count := 1);

   function End_Of_Line (File : in File_Type) return Boolean;
   function End_Of_Line return Boolean;

   procedure New_Page (File : in File_Type);
   procedure New_Page;

   procedure Skip_Page (File : in File_Type);
   procedure Skip_Page;

   function End_Of_Page (File : in File_Type) return Boolean;
```

```
   function End_Of_Page return Boolean;

   function End_Of_File (File : in File_Type) return Boolean;
   function End_Of_File return Boolean;

   procedure Set_Col (File : in File_Type;  To : in
Positive_Count);
   procedure Set_Col (To : in Positive_Count);

   procedure Set_Line (File : in File_Type; To : in
Positive_Count);
   procedure Set_Line (To : in Positive_Count);

   function Col (File : in File_Type) return Positive_Count;
   function Col return Positive_Count;

   function Line (File : in File_Type) return Positive_Count;
   function Line return Positive_Count;

   function Page (File : in File_Type) return Positive_Count;
   function Page return Positive_Count;

   ----------------------------
   -- Characters Input-Output --
   ----------------------------

   procedure Get (File : in File_Type; Item : out Character);
   procedure Get (Item : out Character);
   procedure Put (File : in File_Type; Item : in Character);
   procedure Put (Item : in Character);

   procedure Look_Ahead
     (File        : in File_Type;
      Item        : out Character;
      End_Of_Line : out Boolean);

   procedure Look_Ahead
     (Item        : out Character;
      End_Of_Line : out Boolean);

   procedure Get_Immediate
     (File : in File_Type;
      Item : out Character);

   procedure Get_Immediate
     (Item : out Character);

   procedure Get_Immediate
     (File      : in File_Type;
      Item      : out Character;
      Available : out Boolean);

   procedure Get_Immediate
     (Item      : out Character;
      Available : out Boolean);

   --------------------------
   -- Strings Input-Output --
   --------------------------

   procedure Get (File : in File_Type; Item : out String);
```

```
  procedure Get (Item : out String);
  procedure Put (File : in File_Type; Item : in String);
  procedure Put (Item : in String);

  procedure Get_Line
    (File : in File_Type;
     Item : out String;
     Last : out Natural);

  procedure Get_Line
    (Item : out String;
     Last : out Natural);

  procedure Put_Line
    (File : in File_Type;
     Item : in String);

  procedure Put_Line
    (Item : in String);

  -------------------------------------------------------
  -- Generic packages for Input-Output of Integer Types --
  -------------------------------------------------------

  generic
     type Num is range <>;

  package Integer_Io is

     Default_Width : Field := Num'Width;
     Default_Base  : Number_Base := 10;

     procedure Get
       (File  : in File_Type;
        Item  : out Num;
        Width : in Field := 0);

     procedure Get
       (Item  : out Num;
        Width : in Field := 0);

     procedure Put
       (File  : in File_Type;
        Item  : in Num;
        Width : in Field := Default_Width;
        Base  : in Number_Base := Default_Base);

     procedure Put
       (Item  : in Num;
        Width : in Field := Default_Width;
        Base  : in Number_Base := Default_Base);

     procedure Get
       (From : in String;
        Item : out Num;
        Last : out Positive);

     procedure Put
       (To   : out String;
        Item : in Num;
        Base : in Number_Base := Default_Base);
```

```
end Integer_Io;

----------------------------------
-- Input-Output of Modular Types --
----------------------------------

generic
   type Num is mod <>;

package Modular_IO is

   Default_Width : Field := Num'Width;
   Default_Base  : Number_Base := 10;

   procedure Get
     (File  : in  File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put
     (File  : in File_Type;
      Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Put
     (Item  : in Num;
      Width : in Field := Default_Width;
      Base  : in Number_Base := Default_Base);

   procedure Get
     (From : in  String;
      Item : out Num;
      Last : out Positive);

   procedure Put
     (To   : out String;
      Item : in Num;
      Base : in Number_Base := Default_Base);

end Modular_IO;

--------------------------------
-- Input-Output of Real Types --
--------------------------------

generic
   type Num is digits <>;

package Float_Io is

   Default_Fore : Field := 2;
   Default_Aft  : Field := Num'Digits - 1;
   Default_Exp  : Field := 3;

   procedure Get
```

```
      (File  : in File_Type;
       Item  : out Num;
       Width : in Field := 0);

   procedure Get
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Put
     (Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Get
     (From : in String;
      Item : out Num;
      Last : out Positive);

   procedure Put
     (To   : out String;
      Item : in Num;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

end Float_Io;

generic
   type Num is delta <>;

package Fixed_Io is
   Default_Fore : Field := Num'Fore;
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;

   procedure Get
     (File  : in File_Type;
      Item  : out Num;
      Width : in Field := 0);

   procedure Get
     (Item  : out Num;
      Width : in Field := 0);

   procedure Put
     (File : in File_Type;
      Item : in Num;
      Fore : in Field := Default_Fore;
      Aft  : in Field := Default_Aft;
      Exp  : in Field := Default_Exp);

   procedure Put
     (Item : in Num;
      Fore : in Field := Default_Fore;
```

```
       Aft  : in Field := Default_Aft;
       Exp  : in Field := Default_Exp);

    procedure Get
      (From : in String;
       Item : out Num; Last : out Positive);

    procedure Put
      (To   : out String;
       Item : in Num;
       Aft  : in Field := Default_Aft;
       Exp  : in Field := Default_Exp);

  end Fixed_Io;


  generic
       type Num is delta <> digits <>;

    package Decimal_IO is

       Default_Fore : Field := Num'Fore;
       Default_Aft  : Field := Num'Aft;
       Default_Exp  : Field := 0;

       procedure Get
         (File  : in File_Type;
          Item  : out Num;
          Width : in Field := 0);

       procedure Get
         (Item  : out Num;
          Width : in Field := 0);

       procedure Put
         (File : in File_Type;
          Item : in Num;
          Fore : in Field := Default_Fore;
          Aft  : in Field := Default_Aft;
          Exp  : in Field := Default_Exp);

       procedure Put
         (Item : in Num;
          Fore : in Field := Default_Fore;
          Aft  : in Field := Default_Aft;
          Exp  : in Field := Default_Exp);

       procedure Get
         (From : in  String;
          Item : out Num;
          Last : out Positive);

       procedure Put
         (To   : out String;
          Item : in Num;
          Aft  : in Field := Default_Aft;
          Exp  : in Field := Default_Exp);

    end Decimal_IO;

  ---------------------------------------
```

```
   -- Input-Output of Enumeration Types --
   ---------------------------------------

   generic
      type Enum is (<>);

   package Enumeration_Io is

      Default_Width : Field := 0;
      Default_Setting : Type_Set := Upper_Case;

      procedure Get (File : in File_Type; Item : out Enum);
      procedure Get (Item : out Enum);

      procedure Put
        (File  : in File_Type;
         Item  : in Enum;
         Width : in Field := Default_Width;
         Set   : in Type_Set := Default_Setting);

      procedure Put
        (Item  : in Enum;
         Width : in Field := Default_Width;
         Set   : in Type_Set := Default_Setting);

      procedure Get
        (From : in String;
         Item : out Enum;
         Last : out positive);

      procedure Put
        (To   : out String;
         Item : in Enum;
         Set  : in Type_Set := Default_Setting);

   end Enumeration_Io;

   --  Exceptions

   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;
   Layout_Error : exception renames IO_Exceptions.Layout_Error;

private
       -- Implementation defined...
end Ada.Text_IO;
```

Appendix B Sequential_IO package

```
with IO_exceptions;

generic
        type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

        type file_type is limited private;
        type file_mode is (in_file,out_file);

-- File management

procedure CREATE        (file   :in out file_type;
                         mode   :in     file_mode:=out_file;
                         name   :in     string := "";
                         form   :in     string := "");

procedure OPEN  (       file    :in out file_type;
                         mode   :in     file_mode;
                         name   :in     string;
                         form   :in     string := "");

procedure CLOSE (file   :in out file_type);
procedure DELETE        (file   :in out file_type);
procedure RESET (file   :in out file_type;
                          mode   :in file_mode);
procedure RESET (file   :in out file_type);

function MODE   (file   :in file_type) return file_mode;
function NAME   (file   :in file_type) return string;
function FORM   (file   :in file_type) return string;

function IS_OPEN(file   :in file_type) return boolean;

-- Input and output operations

procedure READ( file    :in file_type;
                         item    :out element_type);
procedure WRITE(        file    :in file_type;
                         item    :in element_type);

function END_OF_FILE(file:in file_type) return boolean;

-- Exceptions
status_error    :exception renames IO_EXCEPTIONS.status_error;
mode_error      :exception renames IO_EXCEPTIONS.mode_error;
name_error      :exception renames IO_EXCEPTIONS.name_error;
use_error       :exception renames IO_EXCEPTIONS.use_error;
device_error    :exception renames IO_EXCEPTIONS.device_error;
end_error       :exception renames IO_EXCEPTIONS.end_error;
data_error      :exception renames IO_EXCEPTIONS.data_error;

private
-- implementation dependent
end sequential_IO
```

Appendix C Direct_IO package

```
with IO_exceptions;

generic

        type ELEMENT_TYPE is private;
package DIRECT_IO is
type file_type is limited private;

package direct_io is

type file_mode is (in_file, in_out_file, out_file);
type count is range 0.. implementation_defined;
subtype positive_count is range 1..count'last;

-- File management

procedure CREATE(       file    :in out file_type;
                mode    :in     file_mode := out_file;
                name    :in     string := "";
                form    :in     string := "");

procedure OPEN( file    :in out file_type;
                mode    :in     file_mode;
                name    :in     string;
                form    :in     string := "");

procedure CLOSE (file   :in out file_type);
procedure DELETE        (file   :in out file_type);
procedure RESET (file   :in out file_type;
                mode    :in     file_mode);
procedure RESET (file   :in out file_type);

function MODE   (file   :in     file_type) return file_mode;
function NAME   (file   :in     file_type) return string;
function FORM   (file   :in     file_type) return string;

function IS_OPEN(       file    :in     file_type) return boolean;

-- Input and output operations

procedure READ( file    :in     file_type;
                item    :   out element_type);
procedure READ( file    :in     file_type;
                item    :   out element_type;
                from    :in     positive_count);

procedure WRITE(        file    :in     file_type;
                item    :in     element_type;
                from    :in     positive_count);
procedure WRITE(        file    :in     file_type;
                item    :in     element_type);

procedure SET_INDEX(    file    :in     file_type;
                to      :in     positive_count);

function INDEX( file    :in     file_type) return positive_count;

function SIZE(  file    :in     file_type) return count;
```

```
function END_OF_FILE(file:in file_type) return boolean;

-- Exceptions
status_error    :exception renames IO_EXCEPTIONS.status_error;
mode_error              :exception renames IO_EXCEPTIONS.mode_error;
name_error              :exception renames IO_EXCEPTIONS.name_error;
use_error               :exception renames IO_EXCEPTIONS.use_error;
device_error    :exception renames IO_EXCEPTIONS.device_error;
end_error               :exception renames IO_EXCEPTIONS.end_error;

private
--implementation depenedent
end DIRECT_IO;
```

Appendix D Text_package package

The text_package package is used by first year students for variable length string manipulation. It is loosely based on that provided in the LRM.
Equality can be tested by simply using the standard "=" operator because padding is maintained as spaces in the unused portion of the record.
Ada95 allows the overriding of the "=" operator for all types, and allows the operands to be of different types, so such padding would not be required.

```
with text_io;

package text_package is

        max_chars       :constant integer := 256;
        subtype length_range is integer range 0..max_chars;
        subtype index_range is length_range range
1..length_range'last;


        type text is
                record
                        value   :string(index_range);
                        length  :length_range:=0;
                end record;


-------------------------------------------------------------------
-------------
-- read a whole line of characters, throw away anything that
doesn't fit.

procedure get_line(     item    :in out text);

procedure get_line(file :in out text_io.file_type;
                   item :in out text
                        );
-------------------------------------------------------------------
-------------
-- put the characters in the_text to the screen. do not include a
new_line at
-- the end

procedure put(item      :in     text);

procedure put(file      :in     text_io.file_type;
                item    :in     text);
-------------------------------------------------------------------
-------------
-- Put the characters in the_text to the screen. Do include a
new_line at the
-- end.

procedure put_line(item :in     text);

procedure put_line(file :in     text_io.file_type;
                        item    :in     text);

-------------------------------------------------------------------
-------------
-- get the string inside the text item.
```

```
function to_string(item :in     text) return string;


-------------------------------------------------------------------
-------------
-- return the length of the string

function length(item    :in     text) return length_range;


-------------------------------------------------------------------
-------------
-- return true if the string is empty

function empty(item     :in     text) return boolean;


-------------------------------------------------------------------
-------------
-- set the value of "the_text" to "a_string". Disregard any
characters that
-- don't fit into the record

 function to_text(item  :in     string) return text;


-------------------------------------------------------------------
-------------
-- append a character onto the end of the text
-- disregard anything that doesn't fit in

procedure append(       item   :in out text;
                        add_on :in     character);

-------------------------------------------------------------------
-------------
-- append a string onto the end of the text
-- disregard anything that doesn't fit in

procedure append(       item   :in out text;
                        add_on :in       string);

-------------------------------------------------------------------
-------------
-- append the two text items together
-- disregard anything that doesn't fit in

procedure append(       item    :in out text;
                        add_on :in     text);

-------------------------------------------------------------------
-------------
-- returns the first n characters of the string, the string is
padded out with
-- spaces if item is not long enough
--

function first_n_chars(item :in   text;
                       n          :in   natural) return string;
```

```
end text_package;
```

Appendix E Simple_io package

The simple_io package is the simplified I/O package used by the first year students. It attempts to provide facilities similar to that found in the Turbo Pascal environment.

```
with text_io;

package simple_io is

        data_error: exception renames text_io.data_error;

        procedure get_line( item :   out integer);
        procedure get_line( file :in out text_io.file_type;
                                 item :   out integer);

        procedure get_line( item :   out float);
        procedure get_line( file :in out text_io.file_type;
                                 item :   out float);


        procedure get_line( item :   out string);
        procedure get_line( file :in out text_io.file_type;
                                 item :   out string);

        procedure get_line( item :   out character);
        procedure get_line( file :in out text_io.file_type;
                                 item :   out character);

        procedure get( item :   out character)
                                 renames text_io.get;

        procedure skip_line(spacing :in text_io.positive_count:=1)
                                 renames text_io.skip_line;


        function end_of_file return boolean
                                 renames text_io.end_of_file;

        function end_of_line return boolean
                                 renames text_io.end_of_line;

        procedure put( item :in integer;
                       width :in natural := 0);

        procedure put( file :in out text_io.file_type;
                       item :in integer;
                       width :in natural := 0);


        default_fore :natural := 6;
        default_aft :natural := 2;
        default_exp :natural := 0;

        procedure put( item :in float;
                       fore :in natural := default_fore;
                      aft :in natural := default_aft;
                      exp :in natural := default_exp);

        procedure put( file :in out text_io.file_type;
                       item :in float;
```

```
                        fore :in natural := default_fore;
                        aft :in natural := default_aft;
                        exp :in natural := default_exp);


        procedure put( item :in string;
                       width :in natural := 0);

        procedure put( item :in character);

        procedure new_line(line_count:in positive := 1);
        ------------------------------------------------------------
        -- The following subprograms are used for screen and
keyboard control
        --
        subtype row_range is integer range 0..23;
        subtype column_range is integer range 0..79;

        procedure move( row :in row_range;
                        col :in column_range);
        procedure home;
        -- moves the cursor to row_range'first, column_range'first

        procedure clear_screen;
        -- clears the screen and sends the cursor home
        subtype color_range is integer range 30..37;
        black   :constant color_range := 30;
        red     :constant color_range := 31;
        green   :constant color_range := 32;
        yellow  :constant color_range := 33;
        blue    :constant color_range := 34;
        magenta :constant color_range := 35;
        cyan    :constant color_range := 36;
        white   :constant color_range := 37;

        procedure text_color(color :color_range);

        function current_text_color return color_range;
        -- what is the current color used for displaying text?

        procedure get_cursor_pos(row :   out row_range;
                                 col :   out column_range);
        -- where is the cursor currently positioned?

        procedure wait_for_keypress;
        -- waits until the user hits a single key

        function read_key return character;
        -- returns the key the user types without needing
        -- the user to press the return key.

end simple_io;
```

Appendix F GNAT Ada

The Gnat (GNU Ada Translator) project has developed a fully fledged Ada compiler that will
work with the GNU linker. This will allow the compiler to be ported to any target that the GNU compiler/linker system is currently available for. The compiler is implementing the 1995 revision to the language, and so provides support for object oriented programming, enhanced concurrency support as well as numerous other features such as child and private child packages.

The Gnat Ada compiler does away with the traditional library implementation of most Ada systems and presents a more traditional environment. Source code is compiled into a .o and a .ali file. The .o represents the translation of the instructions, the .ali file contains information about the relationship of the source to other components in the system.

Gnat Ada source code must be in a file that has the same name as the compilation unit name, with either a .ads (for specifications) or a .adb (for bodies).

Packages with child units (such as unix.file) should have the dot replaced with a '-', and the usual .ads or .adb suffix appended.

For example the following program must be placed into a file called demo.adb.

```
with simple_io; use simple_io;
procedure demo is
begin
        for i in 1..10 loop
                put("hello world");
                new_line;
        end loop;
end;
```

To compile this you type in the following...

```
        gcc -c fred.adb
```

Typical questions now asked at this stage...

Do you mean that the C compiler actually compiles this program?
Does this mean that the code is compiled to C?

The answer to both of these questions is NO!

gcc is not a compiler, it is a program that inspects the file suffix (e.g. .c, .adb, .ads,
.c++) and then invokes the appropriate compiler. The gnat (GNu Ada Translator) compiles the program into the standard GNU intermediate code representation, which is then taken by the code generator specific for the computer you are running, and produces a normal '.o' object file.
Now that that's out of the way...

To link you program...

```
gnatbl  demo.ali
```

It doesn't matter how many other procedures/packages you with, you don't need to specify them in the link option.
However if you want to link in some C routines, then you may have to do the following (this example is taken from an example using the C curses library)....

```
gnatbl demo.ali -lcurses
```

Sharing software

The Gnat compiler looks for the environment variables ADA_INCLUDE_PATH
and ADA_OBJECT_PATH. Any directory in these path variables will be searched
for source code and object code, respectively.
In this way a group can share some commonly developed routines.

Compiler source code (and source to all Ada standard packages)

Currently on Arcadia/Yallara the source for the compiler and all the Ada standard
packages are in the directory

/opt/gnu/adainclude

Some of these names have been 'crunched' (i.e. reduces to 8.3 DOS filename
conventions) so it is not always immediately obvious where a package spec.
resides. For example the text_io package can be found in the file

a-textio.ads (ada.text_io package specification)

Other packages of interest are...

```
interfac.ads            package Interface
i-c.ads         package Interface.C
i-cpoint.ads            package Interface.C.Pointers
i-cstrin.ads            pacakge Interface.C.Strings

a-calend.ads            package Calendar
a-finali.ads            package Ada.Finalization (for destructors
etc)

a-string.ads            package Ada.Strings
a-strbou.ads            package Ada.Strings.Bounded
```

Appendix G RMIT Ada resources

RMIT currently has the Gnat Ada compiler installed on Arcadia and Yallara Sun computers.

A public directory

```
/public/ada
```

contains various Ada resources. These include

ada-lref.ps	quick reference sheets on Ada syntax, attributes, library packages etc.
ada-syntax.ps	
sources	directory included in all search paths with some common code
sources/unix	Unix "mini" binding, giving Ada access to selected Unix services
sources/unix-file	
sources/unix-process	

The directory

```
/opt/local/gnu/adainclude
```

contains the Gnat implementation of all the standard Ada services, such as package Calendar, Direct_IO, Text_IO etc.

The Gnat Ada compiler for other computers can be found in the /public/ada directory on Yallara. Ports to OS/2, Linux and DOS are present.

RMIT's PC network also has the GNAT Ada compiler in the directory

```
g:\pub\cs100\gnat
```

In this directory are all the files you should need for running Gnat on a DOS computer, and is better maintained than the DOS distribution on Yallara.

Go to the Tutorial Allocation System

Go to the Tutorial Allocation Admin System (for Lecturers)

Dale Stanbrough

Email: dale@rmit.edu.au

Phone: 9925 6130 (Bundoora) Phone: 9925 3266 (City)

Room 251.2.29, Bundoora East Campus
Room 10.7.36, City Campus

Consultation times:

City	Monday, 3.30 - 5.20pm
Bundoora	Friday 9 - 10.30

Academic:

1st semester subjects

- CS280 Software Engineering I
- CS582 Software Engineering 2

2nd semester subjects

- CS280 Software Engineering I
- CS504 Concurrent Computing
- CS825 Software Engineering for Grad. Dips.
- AV530 Ada programming

How to write unmaintainable code. Definately everthing every poor programmer does :-)

Older Subjects

Ada resources

My family