Esta web está en construcción. Última modificación: 30/09/2002

Guía de referencia básica de Ada 95

Tabla de contenidos

Enviar sugerencias a: zhernandez@dis.ulpgc.es

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Introducción

Objetivos.

En el primer curso de las Ingenierías en Informática de la ULPGC se usa Ada como primer lenguaje de programación. La realización de esta guía responde a la necesidad de disponer , durante la realización de las prácticas, de una fuente de consulta en línea que resulte más asequible a los alumnos principiantes que el "Ada 95 Reference Manual" que se utiliza como ayuda en los entornos de programación que se emplean. Esta mayor accesibilidad se consigue a costa de un menor rigor y exactitud; esta guía esta lejos de poder (ni mucho menos lo pretende) sustituir al mencionado "Ada 95 Reference Manual" que debe ser la referencia fundamental de un programador en Ada; su finalidad es sólo facilitar el camino en las etapas iniciales del aprendizaje. Tampoco debe obviarse la lectura y estudio de libros sobre Ada que suelen contener numerosos ejemplos y explicaciones de las que esta guía prescinde muchas veces en busca de una mayor simplicidad.

En la estructura de la guía tampoco se ha seguido exactamente la del "Ada 95 Reference Manual", se ha pretendido buscar una visión más general que pueda mantenerse al mayor nivel posible en la realización de guías para otros lenguajes de forma que el conjunto facilite la transición entre ellos.

Autores.

Esta guía ha sido diseñada por:

Zenón J. Hernández Figueroa.

La realización ha estado a cargo de:

Zenón J. Hernández Figueroa.
José R. Pérez Aguiar.
José Daniel González Domínguez

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Elementos léxicos básicos

Conjunto de caracteres.

Ada 95 utiliza el "Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set" más algunos caracteres de formato y otros de control en los comentarios. Los caracteres usados se clasifican como: *letras mayúsculas*, *letras minúsculas*, *dígitos*, *el carácter espacio*, *caracteres de control de formato* (character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF) y form feed (FF)), *otros caracteres de control* (distintos de los de formato y que se permiten dentro de los comentarios, dependen de la implementación de Ada 95) y *caracteres especiales* (los que no encajan en ninguna de las anteriores categorías).

El paquete Characters.Latin_1 declara constantes para nombrar los caracteres del conjunto BMP ISO 10646-1.

Elementos léxicos, separadores y delimitadores.

El código fuente (texto) de un programa en Ada se distribuye en un conjunto de unidades de compilación, cada una de las cuales está formada por una secuencia de *elementos léxicos*. Cada elemento léxico está formado por una secuencia de caracteres y puede ser: un *identificador*, un *delimitador*, una *palabra reservada*, un *literal numérico*, un *literal carácter*, un *literal ristra* o un *comentario*. Todas las implementaciones de Ada garantizan que una línea puede tener hasta 200 caracteres y, en consecuencia, que se pueden tener elementos léxicos de 200 caracteres, estos valores pueden ser mayores en algunas implementaciones.

Entre dos elementos léxicos consecutivos puede haber cualquier número de *separadores*, pero debe haber al menos uno si estos elementos léxicos son identificadores, palabras reservadas o literales numéricos. El papel de separador lo cumplen: el carácter espacio (salvo dentro de un comentario), el carácter de tabulación (salvo dentro de un comentario) y el final de línea. Al principio y al final de una unidad de compilación puede haber cualquier número de separadores.

Un delimitador es uno de los siguientes caracteres:

```
& ' ( ) * + , - . / : ; < = > |
```

También son delimitadores (compuestos) las siguientes parejas de caracteres:

```
=> .. ** := /= >= <= << >> <>
```

Identificadores.

Los identificadores son nombres que designan entidades del programa (variables, constantes, subprogramas,...). En Ada un identificador se forma comenzando con una letra y siguiendo con una combinación de letras, dígitos y el carácter guión bajo ("_"). No se establece diferencia entre las letras mayúsculas y minúsculas, por lo que dos secuencias de caracteres que sólo se diferenciaran en ese aspecto, representan el mismo identificador. Son ejemplos de identificadores:

```
Nombre_Empleado
Raíz_Cuadrada
Año
Persona_1
```

Como en el ejemplo, es usual que los identificadores compuestos de varias palabras se formen juntando las palabras con guión bajo y con las letras iniciales en mayúscula.

Palabras reservadas.

Las palabras reservadas son aquellas que tienen definida una significación especial en el lenguaje y no pueden usarse para otro propósito (por ejemplo, no sirven como identificadores).

abort
abs
abstract
accept
access
aliased
all
and
array
at

begin
body

case
constant

declare
delay
delta
digits
do

else
elsif
end
entry
exception
exit

for
function

generic
goto

if
in
is

limited
loop

mod

new
not
null

of
or
others
out

package
pragma
private
procedure
protected

raise
range
record
rem
renames
requeue

return
reverse

select
separate
subtype

tagged
task
terminate
then
type

until
use

when
while
with

xor

Literales.

Un literal es la representación de un valor mediante alguna notación adecuada. En Ada se distinguen: literales *numéricos*, literales *carácter*, literales *ristra* y el literal *null*, que es un valor específico de los tipos "access" (punteros).

Los literales numéricos pueden ser enteros (integer literal) o reales (real literal). La forma más básica de un literal numérico es una secuencia de dígitos, si es entero, o una secunecia de dígitos que incluye un punto, si es real.

```
12 --entero 12
12.0 --real 12.0
```

Existe además la posibilidad de incluir el caráter guión_bajo, añadir un exponente que especifique una potencia de la base de representación por la que hay que multiplicar el número para obtener el valor que representa o expresar valores en bases distintas de la decimal.

```
1_987 --entero 1987
1E3 --entero 1000
5.0E-1 --real 0.5
2#1111000# --entero 120, en base 2
```

Un literal carácter se forma poniendo un carácter entre apóstrofes:

```
'a', 'b', 'c', ...
```

Un literal ristra (string literal) se forma con una secuencia de caracteres encerrados entre comillas.

```
"ejemplo de ristra"
```

Dos comillas sin ningún carácter enmedio representan una *ristra nula*.

Comentarios.

Un comentario se puede poner en cualquier punto del programa, empieza con dos guiones ("--") y termina con el final de la línea en que se encuentre.

```
--esto es un comentario
end Un_Procedimiento; --esto es otro
```


Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

La librería Characters.Latin_1

La librería Characters.Latin_1 contiene las siguientes declaraciones:

```
(3)
      package Ada.Characters.Latin_1 is
          pragma Pure(Latin_1);
(4)
      -- Caracteres de control:
(5)
          NUL                    : constant Character := Character'Val(0);
          SOH                    : constant Character := Character'Val(1);
          STX                    : constant Character := Character'Val(2);
          ETX                    : constant Character := Character'Val(3);
          EOT                    : constant Character := Character'Val(4);
          ENQ                    : constant Character := Character'Val(5);
          ACK                    : constant Character := Character'Val(6);
          BEL                    : constant Character := Character'Val(7);
          BS                     : constant Character := Character'Val(8);
          HT                     : constant Character := Character'Val(9);
          LF                     : constant Character := Character'Val(10);
          VT                     : constant Character := Character'Val(11);
          FF                     : constant Character := Character'Val(12);
          CR                     : constant Character := Character'Val(13);
          SO                     : constant Character := Character'Val(14);
          SI                     : constant Character := Character'Val(15);
(6)
          DLE                    : constant Character := Character'Val(16);
          DC1                    : constant Character := Character'Val(17);
          DC2                    : constant Character := Character'Val(18);
          DC3                    : constant Character := Character'Val(19);
          DC4                    : constant Character := Character'Val(20);
          NAK                    : constant Character := Character'Val(21);
          SYN                    : constant Character := Character'Val(22);
          ETB                    : constant Character := Character'Val(23);
          CAN                    : constant Character := Character'Val(24);
          EM                     : constant Character := Character'Val(25);
          SUB                    : constant Character := Character'Val(26);
          ESC                    : constant Character := Character'Val(27);
          FS                     : constant Character := Character'Val(28);
          GS                     : constant Character := Character'Val(29);
          RS                     : constant Character := Character'Val(30);
          US                     : constant Character := Character'Val(31);
(7)
      -- Caracteres gráficos ISO 646:
(8)
          Space                  : constant Character := ' ';  -- Character'Val(32)
          Exclamation            : constant Character := '!';  -- Character'Val(33)
          Quotation              : constant Character := '"';  -- Character'Val(34)
          Number_Sign            : constant Character := '#';  -- Character'Val(35)
          Dollar_Sign            : constant Character := '$';  -- Character'Val(36)
          Percent_Sign           : constant Character := '%';  -- Character'Val(37)
          Ampersand              : constant Character := '&';  -- Character'Val(38)
```

```
            Apostrophe           : constant Character := ''';  -- Character'Val(39)
            Left_Parenthesis     : constant Character := '(';  -- Character'Val(40)
            Right_Parenthesis    : constant Character := ')';  -- Character'Val(41)
            Asterisk             : constant Character := '*';  -- Character'Val(42)
            Plus_Sign            : constant Character := '+';  -- Character'Val(43)
            Comma                : constant Character := ',';  -- Character'Val(44)
            Hyphen               : constant Character := '-';  -- Character'Val(45)
            Minus_Sign           : Character renames Hyphen;
            Full_Stop            : constant Character := '.';  -- Character'Val(46)
            Solidus              : constant Character := '/';  -- Character'Val(47)
(9)
            -- Los dígitos decimales'0' a '9' ocupan las posiciones 48 a 57
(10)
            Colon                : constant Character := ':';  -- Character'Val(58)
            Semicolon            : constant Character := ';';  -- Character'Val(59)
            Less_Than_Sign       : constant Character := '<';  -- Character'Val(60)
            Equals_Sign          : constant Character := '=';  -- Character'Val(61)
            Greater_Than_Sign    : constant Character := '>';  -- Character'Val(62)
            Question             : constant Character := '?';  -- Character'Val(63)
            Commercial_At        : constant Character := '@';  -- Character'Val(64)
(11)
            -- Las letras 'A' a 'Z' ocupan las posicines 65 a 90
(12)
            Left_Square_Bracket  : constant Character := '[';  -- Character'Val(91)
            Reverse_Solidus      : constant Character := '\';  -- Character'Val(92)
            Right_Square_Bracket : constant Character := ']';  -- Character'Val(93)
            Circumflex           : constant Character := '^';  -- Character'Val(94)
            Low_Line             : constant Character := '_';  -- Character'Val(95)
(13)
            Grave                : constant Character := '`';  -- Character'Val(96)
            LC_A                 : constant Character := 'a';  -- Character'Val(97)
            LC_B                 : constant Character := 'b';  -- Character'Val(98)
            LC_C                 : constant Character := 'c';  -- Character'Val(99)
            LC_D                 : constant Character := 'd';  -- Character'Val(100)
            LC_E                 : constant Character := 'e';  -- Character'Val(101)
            LC_F                 : constant Character := 'f';  -- Character'Val(102)
            LC_G                 : constant Character := 'g';  -- Character'Val(103)
            LC_H                 : constant Character := 'h';  -- Character'Val(104)
            LC_I                 : constant Character := 'i';  -- Character'Val(105)
            LC_J                 : constant Character := 'j';  -- Character'Val(106)
            LC_K                 : constant Character := 'k';  -- Character'Val(107)
            LC_L                 : constant Character := 'l';  -- Character'Val(108)
            LC_M                 : constant Character := 'm';  -- Character'Val(109)
            LC_N                 : constant Character := 'n';  -- Character'Val(110)
            LC_O                 : constant Character := 'o';  -- Character'Val(111)
(14)
            LC_P                 : constant Character := 'p';  -- Character'Val(112)
            LC_Q                 : constant Character := 'q';  -- Character'Val(113)
            LC_R                 : constant Character := 'r';  -- Character'Val(114)
            LC_S                 : constant Character := 's';  -- Character'Val(115)
            LC_T                 : constant Character := 't';  -- Character'Val(116)
            LC_U                 : constant Character := 'u';  -- Character'Val(117)
            LC_V                 : constant Character := 'v';  -- Character'Val(118)
            LC_W                 : constant Character := 'w';  -- Character'Val(119)
            LC_X                 : constant Character := 'x';  -- Character'Val(120)
            LC_Y                 : constant Character := 'y';  -- Character'Val(121)
            LC_Z                 : constant Character := 'z';  -- Character'Val(122)
```

```
          Left_Curly_Bracket   : constant Character := '{';  -- Character'Val(123)
          Vertical_Line        : constant Character := '|';  -- Character'Val(124)
          Right_Curly_Bracket  : constant Character := '}';  -- Character'Val(125)
          Tilde                : constant Character := '~';  -- Character'Val(126)
          DEL                  : constant Character := Character'Val(127);
(15)
       -- Caracteres de control ISO 6429:
(16)
          IS4                  : Character renames FS;
          IS3                  : Character renames GS;
          IS2                  : Character renames RS;
          IS1                  : Character renames US;
(17)
          Reserved_128         : constant Character := Character'Val(128);
          Reserved_129         : constant Character := Character'Val(129);
          BPH                  : constant Character := Character'Val(130);
          NBH                  : constant Character := Character'Val(131);
          Reserved_132         : constant Character := Character'Val(132);
          NEL                  : constant Character := Character'Val(133);
          SSA                  : constant Character := Character'Val(134);
          ESA                  : constant Character := Character'Val(135);
          HTS                  : constant Character := Character'Val(136);
          HTJ                  : constant Character := Character'Val(137);
          VTS                  : constant Character := Character'Val(138);
          PLD                  : constant Character := Character'Val(139);
          PLU                  : constant Character := Character'Val(140);
          RI                   : constant Character := Character'Val(141);
          SS2                  : constant Character := Character'Val(142);
          SS3                  : constant Character := Character'Val(143);
(18)
          DCS                  : constant Character := Character'Val(144);
          PU1                  : constant Character := Character'Val(145);
          PU2                  : constant Character := Character'Val(146);
          STS                  : constant Character := Character'Val(147);
          CCH                  : constant Character := Character'Val(148);
          MW                   : constant Character := Character'Val(149);
          SPA                  : constant Character := Character'Val(150);
          EPA                  : constant Character := Character'Val(151);
(19)
          SOS                  : constant Character := Character'Val(152);
          Reserved_153         : constant Character := Character'Val(153);
          SCI                  : constant Character := Character'Val(154);
          CSI                  : constant Character := Character'Val(155);
          ST                   : constant Character := Character'Val(156);
          OSC                  : constant Character := Character'Val(157);
          PM                   : constant Character := Character'Val(158);
          APC                  : constant Character := Character'Val(159);
(20)
       -- Otros caracteres gráficos:
(21)
       -- Posiciones 160 (16#A0#) .. 175 (16#AF#):
          No_Break_Space             : constant Character := ' ';  --
Character'Val(160)
          NBSP                       : Character renames No_Break_Space;
          Inverted_Exclamation       : constant Character := Character'Val(161);
          Cent_Sign                  : constant Character := Character'Val(162);
          Pound_Sign                 : constant Character := Character'Val(163);
```

```
        Currency_Sign               : constant Character := Character'Val(164);
        Yen_Sign                    : constant Character := Character'Val(165);
        Broken_Bar                  : constant Character := Character'Val(166);
        Section_Sign                : constant Character := Character'Val(167);
        Diaeresis                   : constant Character := Character'Val(168);
        Copyright_Sign              : constant Character := Character'Val(169);
        Feminine_Ordinal_Indicator  : constant Character := Character'Val(170);
        Left_Angle_Quotation        : constant Character := Character'Val(171);
        Not_Sign                    : constant Character := Character'Val(172);
        Soft_Hyphen                 : constant Character := Character'Val(173);
        Registered_Trade_Mark_Sign  : constant Character := Character'Val(174);
        Macron                      : constant Character := Character'Val(175);
(22)
    -- Posiciones 176 (16#B0#) .. 191 (16#BF#):
        Degree_Sign                 : constant Character := Character'Val(176);
        Ring_Above                  : Character renames Degree_Sign;
        Plus_Minus_Sign             : constant Character := Character'Val(177);
        Superscript_Two             : constant Character := Character'Val(178);
        Superscript_Three           : constant Character := Character'Val(179);
        Acute                       : constant Character := Character'Val(180);
        Micro_Sign                  : constant Character := Character'Val(181);
        Pilcrow_Sign                : constant Character := Character'Val(182);
        Paragraph_Sign              : Character renames Pilcrow_Sign;
        Middle_Dot                  : constant Character := Character'Val(183);
        Cedilla                     : constant Character := Character'Val(184);
        Superscript_One             : constant Character := Character'Val(185);
        Masculine_Ordinal_Indicator : constant Character := Character'Val(186);
        Right_Angle_Quotation       : constant Character := Character'Val(187);
        Fraction_One_Quarter        : constant Character := Character'Val(188);
        Fraction_One_Half           : constant Character := Character'Val(189);
        Fraction_Three_Quarters     : constant Character := Character'Val(190);
        Inverted_Question           : constant Character := Character'Val(191);
(23)
    -- Pociciones 192 (16#C0#) .. 207 (16#CF#):
        UC_A_Grave                  : constant Character := Character'Val(192);
        UC_A_Acute                  : constant Character := Character'Val(193);
        UC_A_Circumflex             : constant Character := Character'Val(194);
        UC_A_Tilde                  : constant Character := Character'Val(195);
        UC_A_Diaeresis              : constant Character := Character'Val(196);
        UC_A_Ring                   : constant Character := Character'Val(197);
        UC_AE_Diphthong             : constant Character := Character'Val(198);
        UC_C_Cedilla                : constant Character := Character'Val(199);
        UC_E_Grave                  : constant Character := Character'Val(200);
        UC_E_Acute                  : constant Character := Character'Val(201);
        UC_E_Circumflex             : constant Character := Character'Val(202);
        UC_E_Diaeresis              : constant Character := Character'Val(203);
        UC_I_Grave                  : constant Character := Character'Val(204);
        UC_I_Acute                  : constant Character := Character'Val(205);
        UC_I_Circumflex             : constant Character := Character'Val(206);
        UC_I_Diaeresis              : constant Character := Character'Val(207);
(24)
    -- Posiciones 208 (16#D0#) .. 223 (16#DF#):
        UC_Icelandic_Eth            : constant Character := Character'Val(208);
        UC_N_Tilde                  : constant Character := Character'Val(209);
        UC_O_Grave                  : constant Character := Character'Val(210);
        UC_O_Acute                  : constant Character := Character'Val(211);
        UC_O_Circumflex             : constant Character := Character'Val(212);
```

```
         UC_O_Tilde                 : constant Character := Character'Val(213);
         UC_O_Diaeresis             : constant Character := Character'Val(214);
         Multiplication_Sign        : constant Character := Character'Val(215);
         UC_O_Oblique_Stroke        : constant Character := Character'Val(216);
         UC_U_Grave                 : constant Character := Character'Val(217);
         UC_U_Acute                 : constant Character := Character'Val(218);
         UC_U_Circumflex            : constant Character := Character'Val(219);
         UC_U_Diaeresis             : constant Character := Character'Val(220);
         UC_Y_Acute                 : constant Character := Character'Val(221);
         UC_Icelandic_Thorn         : constant Character := Character'Val(222);
         LC_German_Sharp_S          : constant Character := Character'Val(223);
(25)
      -- Posiciones 224 (16#E0#) .. 239 (16#EF#):
         LC_A_Grave                 : constant Character := Character'Val(224);
         LC_A_Acute                 : constant Character := Character'Val(225);
         LC_A_Circumflex            : constant Character := Character'Val(226);
         LC_A_Tilde                 : constant Character := Character'Val(227);
         LC_A_Diaeresis             : constant Character := Character'Val(228);
         LC_A_Ring                  : constant Character := Character'Val(229);
         LC_AE_Diphthong            : constant Character := Character'Val(230);
         LC_C_Cedilla               : constant Character := Character'Val(231);
         LC_E_Grave                 : constant Character := Character'Val(232);
         LC_E_Acute                 : constant Character := Character'Val(233);
         LC_E_Circumflex            : constant Character := Character'Val(234);
         LC_E_Diaeresis             : constant Character := Character'Val(235);
         LC_I_Grave                 : constant Character := Character'Val(236);
         LC_I_Acute                 : constant Character := Character'Val(237);
         LC_I_Circumflex            : constant Character := Character'Val(238);
         LC_I_Diaeresis             : constant Character := Character'Val(239);
(26)
      -- Posiciones 240 (16#F0#) .. 255 (16#FF#):
         LC_Icelandic_Eth           : constant Character := Character'Val(240);
         LC_N_Tilde                 : constant Character := Character'Val(241);
         LC_O_Grave                 : constant Character := Character'Val(242);
         LC_O_Acute                 : constant Character := Character'Val(243);
         LC_O_Circumflex            : constant Character := Character'Val(244);
         LC_O_Tilde                 : constant Character := Character'Val(245);
         LC_O_Diaeresis             : constant Character := Character'Val(246);
         Division_Sign              : constant Character := Character'Val(247);
         LC_O_Oblique_Stroke        : constant Character := Character'Val(248);
         LC_U_Grave                 : constant Character := Character'Val(249);
         LC_U_Acute                 : constant Character := Character'Val(250);
         LC_U_Circumflex            : constant Character := Character'Val(251);
         LC_U_Diaeresis             : constant Character := Character'Val(252);
         LC_Y_Acute                 : constant Character := Character'Val(253);
         LC_Icelandic_Thorn         : constant Character := Character'Val(254);
         LC_Y_Diaeresis             : constant Character := Character'Val(255);
      end Ada.Characters.Latin_1;
```

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Variables y constantes

Objetos de datos estáticos.

Una *variable* es un objeto de datos con un *nombre*, un *tipo* y un *valor* asociado que puede modificarse libremente durante la ejecución de la parte de un programa en la que es visible (ámbito). Una declaración de variables consta de: (1) una lista de nombres de las variables que se declaran separadas por comas (",") y terminada con dos puntos (":"), (2) el tipo de las variables y, opcionalmente, (3) la asignación de un valor inicial.

```
X: Float;
Y, Z: Integer := 0;
```

Una *constante* es un objeto de datos con un *nombre*, un *tipo* y un *valor* asociado que no puede modificarse una vez definido. Una declaración de constantes es igual que una declaración de variables excepto en que se precede al tipo con la palabra **constant** .

```
C, K: constant Integer := 0;
```

Se puede hacer una declaración de constantes sin especificar el valor (deferred declaration) siempre que más adelante (en la misma región del programa) se haga una nueva declaración para definirlo.

Ada contempla también una clase de constantes que sirve para asociar un nombre para identificar un valor numérico y que no requiere la especificación de un tipo.

```
Pi: constant := 3.1416;
```

Variables dinámicas.

Los objetos de datos creados mediante declaraciones escritas en el código son estáticas en el sentido de que se crean cuando se elabora la declaración al iniciarse la ejecución de la región donde se encuentra y se destruyen cuando se termina dicha ejecución. Se pueden crear variables de forma dinámica en cualquier momento de la ejecución de un programa usando la palabra reservada **new** acompañada del tipo de la variable que se pretende crear; como resultado se devuelve la dirección del bloque de memoria asociado a la variable que se crea, esta dirección puede almacenarse en una variable puntero (access) para poder acceder posteriormente a la variable dinámica.

```
P := new Integer; --P debe ser un puntero (access) a variables de
tipo Integer.
```

Como las estáticas, las variables dinámicas se pueden crear con un valor inicial.

```
P := new Integer'(35); --la variable dinámica se crea con el valor
35.
```

En Ada, una variable dinámica se libera automáticamente cuando finaliza la ejecución de la parte del programa donde está definido el tipo puntero que permite referenciarla. Para realizar un control más eficiente de la memoria es necesario emplear operaciones de *liberación explícita* de las variables creadas dinámicamente cuando ya no sean necesarias; se necesita instanciar el procedimiento genérico "Unchecked_Deallocation" para lo que se debe incluir en la cláusula de contexto. La instanciación posterior tendrá que indicar el tipo de las variables a liberar y el tipo de puntero que se usa para referenciarlas.

```
--se incluye "Unchecked_Deallocation" en la cláusula de contexto
with Unchecked_Deallocation;
...
--dentro de la unidad de programa se instancia
--se supone que P_Integer ha sido declarado como "access Integer"
procedure Liberar_Integer is new
Unchecked_Deallocation(Integer,P_Integer);
...
--ahora se puede usar Liberar_Integer para liberar variables
dinámicas Integer
--Suponiendo que P es de tipo P_Integer y contiene la dirección de
una
--variable Integer dinámica creada previamente:
Liberar_Integer(P); --Libera la variable dinámica referenciada por
P
```


Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Punteros

Declaración.

Los punteros (access) constituyen el tipo de datos con el que se represetan direcciones de memoria; su uso más habitual es para construir estructuras de datos complejas utilizando variables ubicadas dinámicamente. Como Ada es un lenguaje fuertemente tipado la declaración de una variable puntero se hace utilizando la palabra reservada **access** y requiere que se especifique el tipo de datos que el mismo va a poder referenciar.

```
type T_Persona is record
  Nombre: string(1..30);
  D_N_I: string(1..9);
end record;
--tipo puntero a variables dinámicas de tipo Integer
type P_Int is access Integer;
--tipo puntero a variables dinámicas de tipo T_Persona
type P_Persona is access T_Persona;
--tipo puntero a un puntero dinámico
type P_Punt is access P_Int;
--variables puntero a variables dinámicas de tipo Integer
Punt1, Punt2: P_Int;
--variables puntero a variables dinámicas de tipo T_Persona
Punt3, Punt4: P_Persona;
--variable puntero estático a un puntero dinámico
Punt5: P_Punt;
```

Si se quiere que un puntero pueda referenciar también variables estáticas, hay que especificarlo en su definición usando la palabra **all**.

```
type P_Tot_Int is access all Integer;
--Las variables de tipo P_Tot_Int pueden referenciar
--variables de tipo Integer, tanto estáticas como dinámicas
Punt6 : P_Tot_Int;
```

Usar punteros para referenciar variables estáticas es peligroso ya que supone la creación de un alias, por ello sólo se puede asignar a una variable puntero la dirección de una variable estática que lo permita explícitamente (incluyendo la palabra **aliased** en su declaración).

```
I: Integer; --la dirección de I no se puede asignar a un puntero
J: aliased Integer;--la dirección de J se puede asignar a un
puntero;
```

El problema de los alias se puede minimizar declarando el puntero de forma que aquello a lo que apunta se considere constante, no puede modificarse a través del puntero; ello, además, permite que a un puntero se le puedan asignar direcciones de constantes, no sólo de variables.

```
type P_Int_Const is access constant Integer;
```

Asignación de valores.

La forma más habitual de asignar un valor válido a un puntero es dándole la dirección de una variable dinámica creada con la operación **new**.

```
Punt1 := new Integer;
```

También se le puede asignar una dirección válida almacenada en otro puntero (aunque ello crea un alias).

```
Punt2 := Punt1;
```

Si ha sido declarado "all" se le puede también asignar la dirección de una variable estática "aliased", usando el atributo "access" propio de éstas.

```
Punt6 := J'access; --correcto;
Punt6 := I'access; --incorrecto;
```

Si ha sido declarado "constant" se le podrá asignar además la dirección de una constante.

En cualquier caso a todo puntero se le puede asignar el valor **null** que representa "ninguna dirección"; en Ada los punteros se inicializan con este valor cuando se declaran, a menos que dicha declaración incluya una inicialización diferente.

Acceso a las variables referenciadas por punteros.

Para acceder al contenido de una variable puntero se utiliza su nombre, como con cualquier otro tipo de variables; para acceder a la variable representada por la dirección contenida en el puntero se utiliza el cualificador .**all**.

```
Punt1 := Punt2; --asigna puntero a puntero
J := Punt1.all; --asigna Integer a Integer
Punt6 := J'access; --asigna dirección estática a puntero
```

```
Punt6.all := I; --asigna integer a integer
```

Si la variable referenciada es de tipo record se puede acceder a cada campo con el cualificador .**all**.*nombre_campo* o abreviado . *nombre_campo*.

```
Punt3.all.D_N_I := "12345678D";
Punt3.D_N_I := "12345678D";
```

Si es un array se pueden aplicar los índices al cualificador . **all** o, directamente, para acceder a los elementos individuales.

```
type T_Vec is array (Integer range <>) of Integer;
type P_Vec is access T_Vec;
V : P_Vec := new T_Vec'(1,2,3,4,5);
...
I := V(3);
```

Estructuras recursivas con punteros.

Una aplicación frecuente de los punteros es la creación de estructuras de datos dinámicas, por ejemplo, listas encadenadas. Este tipo de estructuras están formadas por un conjunto de nodos conectados mediante campos de enlace que son punteros a nodos del mismo tipo. Para estos nodos sería necesaria una definición de tipo tal como:

```
type Nodo_Lista is record
   Info: Integer;
   Siguiente: P_Nodo_Lista;
end record;
```

donde P_Nodo_Lista es el tipo de los punteros a objetos de tipo Nodo_Lista. Esto significa que debería existir una definición para P_Nodo_Lista:

```
type P_Nodo_Lista is access Nodo_Lista;
```

Esta definición debería estar antes que la de Nodo_Lista, para que sea conocida cuando se declara el campo siguiente de éste; pero entonces Nodo_Lista no se conocería cuando se intenta definir P_Nodo_Lista, creándose un círculo vicioso. La solución consiste en empezar con una declaración incompleta de Nodo_Lista que establezca que tal tipo va a existir, pero sin concretarlo, cosa que debe hacerse más adelante en la misma región del programa.:

```
type Nodo_Lista;
type P_Nodo_Lista is access Nodo_Lista;
type Nodo_Lista is record
   Info: Integer;
```

```
   Siguiente: P_Nodo_Lista;
end record;
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Subprogramas

Definición, declaración y uso.

Un subprograma es un procedimiento (**procedure**) o una función (**function**). La diferencia entre un procedimiento y una función es que el primero sólo representa la ejecución de una secuencia de instrucciones, en función de unos parámetros, mientras que la segunda representa un valor que se genera como resultado de su ejecución. Un subprograma constituye una unidad susceptible de ser compilada por separado, o bien puede anidarse dentro de otras unidades. Se pueden usar funciones para sobrecargar los operadores del lenguaje, otorgándoles nuevos significados.

Cuando sea conveniente, es posible escribir una declación de un subprograma separada de su definición. La declaración de un subprograma es una especificación terminada en ";" que fija (1) el nombre y clase del subprograma, (2) el número, tipo y clase de sus parámetros formales y (3) el tipo del resultado que devuelve, si se trata de una función.

```
procedure Intercambia(A,B: in out Integer);
function Media(A,B: Float) return Float;
```

La definición de un subprograma consta de tres elementos: (1) cabecera, que es una especificación igual que la anterior, pero terminada con la palabra "**is**", significando que a continuación se desarrolla, (2) declaraciones locales, de cualquier elemento declarable, incluyendo la definición anidada de otros subprogramas y (3) el cuerpo, compuesto por las sentencias ejecutables del subprograma y delimitado por las palabras reservadas "**begin**" y "**end**" (esta última puede ir acompañada del nombre del subprograma). En el cuerpo del subprograma debe incluirse al menos una sentencia de retorno ("**return**") que devuelva el control a quien lo llamó; si la ejecución de un subprograma alcanza el final del mismo sin encontrar una sentencia de retorno, se produce una excepción *"Program_Error"*, si es una función, dado que la sentencia de retorno se utiliza en las funciones para especificar el valor a devolver. En los procedimientos puede omitirse la sentencia de retorno cuando el único punto de retorno se encuentra al final del cuerpo.

```
procedure Intercambia(A,B: in out Integer) is
   C: integer;
begin
   C := A;
   A := B;
   B := C;
   return;
end Intercambia;
```

```
function Media(A,B: Float) return Float is
begin
   return (A + B) / 2.0;
end Media;
```

El ámbito de una declaración local abarca al subprograma en el que está y cualquier otro anidado en éste que se halle declarado con posterioridad a la misma.

La declaración y/o definición de un subprograma puede hacerse en cualquier sección de declaraciones (parte de declaraciones locales de un subprograma, sección de declaraciones de un bloque, package body,...). En un mismo ámbito se pueden tener varios subprogramas con el mismo nombre, siempre que se diferencien en los parámetros o en el tipo del resultado si son funciones.

La llamada a un procedimiento se hace mediante una sentencia que tiene el aspecto de una instrucción más del lenguaje, mientras que la llamada a una función sólo puede hacerse como parte de una expresión:

```
...
Intercambia(X,Y);
Z := Media(L,M);
...
```

Parámetros.

Listas de parámetros formales.

Si se precisan, el número, tipo y clase de los parámetros formales se declara en una *lista de parámetros formales* que se pone justo después del nombre del subprograma en la especificación del mismo. La lista de parámetros formales se delimita por paréntesis y se compone de una o varias sublistas de parámetros de la misma clase y tipo separadas por punto y coma (";"). Cada sublista está formada por una secuencia de nombres separados por comas (","), seguida de dos puntos (":") a continuación de los cuales se pone la clase y tipo de los parámetros de la secuencia:

```
procedure Muchos_Parámetros(A,B: in Integer; C: out Float; L, M: in
out Integer);
```

Clases de parámetros.

En función del sentido en el que se transfiere la información, los parámetos pueden ser:

1. De entrada (**in**).
2. De salida (**out**).
3. De entrada/salida (**in out**).

Si un parámetro es de entrada no hace falta indicarlo (se considera la clase por defecto). Las funciones sólo admiten parámetros de entrada. Los parámetros de entrada no se pueden modificar en el subprograma que los considera constantes.

Correspondencia entre parámetros reales y formales.

Cuando se llama a un subprograma, hay que especificar los parámetros reales sobre los que se quiere que actúe. Los parámetros reales ocuparán en la ejecución el lugar de los formales correspondientes. La correspondencia entre parámetros reales y formales se establece normalmente por posición, lo que quiere decir que el primer parámetro real corresponde al primero formal, el segundo al segundo y así sucesivamente. En el siguiente ejemplo se llama al procedimiento *"Intercambia"* haciendo corresponder el parámetro real X al formal A, y el real Y al formal B:

```
...
Intercambia(X,Y);
...
```

La correspondencia entre parámetros formales y reales también puede hacerse por nombre: especificando los nombres del parámetro formal y del real, relacionándolos con una flecha ("=>"):

```
...
Intercambia(A => X,B => Y);
...
```

A un parámetro formal de entrada se le puede hacer corresponder como parámetro real cualquier expresión del tipo adecuado; a los parámetros formales de salida o de entrada/salida sólo les pueden corresponder parámetros reales que sean variables.

Subprogramas genéricos.

En muchas situaciones un mismo problema se plantea en contextos diferentes, de forma que el algoritmo que lo resuelve es el mismo, salvo por los detalles propios de cada contexto; por ejemplo si se quiere ordenar un array, las acciones que habrá que realizar son las mismas independinetemente del tipo de los elementos del array (siempre que sean ordenables), de su número, e incluso de la operación que se utilice para compararlos. Un subprograma genérico es uno que se escribe dejando todo este tipo de detalles como parámetros formales genéricos a concretar cuando haga falta, de tal manera que se podrá luego utilizar en distintas situaciones sin tener que reescribir cada vez el algoritmo completo. El siguiente ejemplo ilustra la forma de hacerlo.

```
generic
    type Tipo is private;
    with function "<"(X,Y: Tipo) return Boolean;
    type Indice is (<>);
    type Lista is array(Indice range <>) of Tipo;
```

```
procedure Ordenar(L: in out Lista);

procedure Ordenar(L: in out Lista) is
    Menor: Indice;
    Aux : Tipo;
begin
    for I in L'First..Indice'Pred(L'Last) loop
       Menor := I;
       for J in Indice'Succ(L'First)..L'Last loop
          if L(J) < L(Menor) then
             Menor := J;
          end if;
       end loop;
       if Menor /= I then
          Aux := L(I);
          L(I) := L(Menor);
          L(Menor) := Aux;
       end if;
    end loop;
    return;
end Ordenar;
```

Una vez se tiene escrito el subprograma genérico, se pueden declarar instancias del mismo con diferentes parámetros reales genérico s, lo que hará que el compilador cree los subprogramas correspondientes. Para ilustrarlo, se proporciona un procedimiento llamado *"Prueba"* en cuya cláusula de contexto se incluye la unidad *"Ordenar"* . El procedimiento *"Prueba"* declara dos instancias del procedimiento *"Ordenar"*, aplicadas a arrays de elementos de tipo *Integer* y rango *Integer* (o *Docena*), que se diferencian sólo en la operación de ordenación. Cuando el compilador elabore estas declaraciones, se dispondrá de dos procedimientos no genéricos (*"Ordena_Vector_Ascendente"* y *"Ordena_Vector_Descendente"*) que podrán ser utilizados, normalmente, igual que si se hubiesen escrito explícitamente.

```
with Text_Io,Ordenar;
use Text_Io;
procedure Prueba is
    subtype Docena is Integer range 1..12;
    type Vector is array(Integer range <>) of Integer;
    procedure Ordena_Vector_Ascendente is new
Ordenar(Integer,"<",Integer,Vector);
    procedure Ordena_Vector_Descendente is new
Ordenar(Integer,">",Docena,Vector);
    V: Vector(Docena);
    ...
begin
     ...
     Ordena_Vector_Ascendente(V);
```

```
    ...
end Prueba;
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Expresiones y operadores

Expresiones.

Una expresión es una combinación de operadores y operandos de cuya evaluación se obtiene un valor. Los operandos pueden ser nombres que denoten objetos variables o constantes, funciones, literales de cualquier tipo adecuado de acuerdo con los operadores u otras expresiones más simples. La evaluación de una expresión da lugar a un valor de algún tipo, una expresión se dice que es del tipo de su resultado. Ejemplos de expresiones:

```
a + 5*b
(a >= 0) and ((b+5) > 10)
a
-a * 2 + b
-b + sqrt(b**2 - 4*a*c)
length(s) > 0
```

Las expresiones se evalúan de acuerdo con la precedencia de los operadores. Ante una secuencia de operadores de igual precedencia, la evaluación se realiza según el orden de escritura, de izquierda a derecha. El orden de evaluación puede modificarse usando paréntesis.

Operadores.

Ada agrupa los operadores en 6 categorías, de menor a mayor precedencia. Los operadores binarios se usan en formato infijo (*<operando_izquierdo> <operador> <operando_derecho>*), como en *"a + b"*. Los operadores unarios se usan en formato prefijo (*<operador> <operando>*), como en *"-5"*.

Operadores lógicos.

Están predefinidos para cualquier tipo, *T*, que designe un booleano, modular o un array monodimensional de componentes booleanos:

```
function "and"(Left, Right : T) return T
function "or" (Left, Right : T) return T
function "xor"(Left, Right : T) return T
```

Su significado es el convencional (para los tipos modulares son operaciones bit a bit):

A	B	(A and B)	(A or B)	(A xor B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Para los tipos booleanos, existen versiones de los operadores *"and"* y *"or"*, llamadas *"and then"* y *"or else"* que tienen el mismo significado, pero realizan una "evaluación en cortocircuito" consistente en que evalúan siempre primero el operando izquierdo y, si el valor de éste es suficiente para determinar el resultado, no evalúan el operador derecho.

La evaluación en cortocircuito resulta muy útil si la correcta evaluación del operador derecho depende del valor del operador izquierdo, como en el siguiente ejemplo:
if i <= Vec'last and then Vec(i) > 0 then ...
(Evaluar Vec(i) produciría un error si i tiene un valor superior al límite máximo del vector).

Operadores relacionales.

Los operadores de igualdad están predefinidos para todos los tipos no limitados, sea *T* un tipo con estas características:

```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) return Boolean
```

Los operadores de ordenación están predefinidos para todos los tipos escalares y los arrays de elementos discretos, sea *T* un tipo con estas características:

```
function "<" (Left, Right : T) return Boolean
function "<="(Left, Right : T) return Boolean
function ">" (Left, Right : T) return Boolean
function ">="(Left, Right : T) return Boolean
```

Existe también un operador de pertenencia ("in", "not in") que determina si un valor pertenece a un rango o a un subtipo: **if** i **in** 1..10 **then** ...

Todos los operadores relacionales devuelven un resultado de tipo Boolean.

Operadores binarios de adición.

Los operadores de adición predefinidos para cualquier tipo numérico, *T*, son:

```
function "+"(Left, Right : T) return T
function "-"(Left, Right : T) return T
```

También pertenecen a esta categoría los operadores de concatenación, predefinidos para cualquier tipo de array monodimensional no limitado, *T*, de elementos de tipo *C*:

```
function "&"(Left : T; Right : T) return T
function "&"(Left : T; Right : C) return T
function "&"(Left : C; Right : T) return T
function "&"(Left : C; Right : C) return T
```

Operadores unarios de adición.

Los operadores unarios de adición predefinidos para cualquier tipo número, *T*, son la identidad y la negación:

```
function "+"(Right : T) return T
function "–"(Right : T) return T
```

Cuando se aplica a un tipo modular, el operador de negación (*"-"*) tiene el efecto de restar el valor del operando, si es distinto de cero, al módulo. Si el valor del operando es cero, el resultado es cero.

Operadores multiplicativos.

Los operadores de multiplicación y división están predefinidos entre diversas combinaciones de enteros y reales:

```
function "*" (Left, Right : T) return T
function "/" (Left, Right : T) return T
function "*"(Left : T; Right : Integer) return T
function "*"(Left : Integer; Right : T) return T
function "/"(Left : T; Right : Integer) return T
function "*"(Left, Right : root_real) return root_real
function "/"(Left, Right : root_real) return root_real
function "*"(Left : root_real; Right : root_integer) return
root_real
function "*"(Left : root_integer; Right : root_real) return
root_real
function "/"(Left : root_real; Right : root_integer) return
root_real
function "*"(Left, Right : universal_fixed) return universal_fixed
function "/"(Left, Right : universal_fixed) return universal_fixed
```

Los operadores módulo y resto están definidos para cualquier tipo entero, *T*:

```
function "mod"(Left, Right : T) return T
function "rem"(Left, Right : T) return T
```

La relación entre el resto y la división entera viene dada por la expresión: A = (A/B)*B + (A **rem** B), donde (A **rem** B) tiene el mismo signo que A, y es menor que B en valor absoluto.

El operador módulo se define de manera que (A **mod** B) tiene el mismo signo que B, un valor absoluto menor, y existe un número entero, N, tal que: A = B*N + (A **mod** B).

Operadores de máxima prioridad.

Los operadores de máxima prioridad son: el operador de cálculo del valor absoluto, definido para cualquier tipo numérico, *T*, el operador de negación lógica, definido para cualquier tipo booleano, modular o array monodimensional de componentes booleanos, *T* , y el operador de exponenciación, definido para cualquier tipo entero, *T*, o para cualquier tipo real en coma flotante, *T*. Cada uno, de acuerdo con las siguientes especificaciones:

```
function "abs"(Right : T) return T
function "not"(Right : T) return T
function "**"(Left : T; Right : Natural) return T
function "**"(Left : T; Right : Integer'Base) return T
```

Sobrecarga de operadores.

Ada permite que el programador sobrecargue los operadores del lenguaje, esto es, que pueda redefinirlos dándoles nuevos significados. Para sobrecargar un operador, simplemente hay que definir una función cuyo nombre sea el operador entre comillas y que tenga los parámetros adecuados. Por ejemplo, dado el siguiente tipo:

```
type Complejo is record
    PReal, PImag: float;
end Complejo;
```

podemos sobrecargar el operador de suma ("+") para utilizarlo con el fin de sumar números complejos:

```
function "+"(A, B : in Complejo) return Complejo is
    Suma: Complejo;
begin
    Suma.PReal := A.PReal + B.PReal;
    Suma.PImag := A.PImag + B.PImag;
    return Suma;
end "+";
```

Una vez definida esta función, se puede aplicar el operador entre variables del tipo definido en los parámetros, devolviendo un valor del tipo definido como resultado.

```
...
S, X, Y: Complejo;
...
S := X + Y;
```

Sólo se pueden redefinir los operadores del lenguaje (no se pueden inventar operadores) y manteniendo siempre su cardinalidad (los binarios se redefinirán con funciones de dos parámetros y los unarios con funciones de un parámetro).

El operador de desigualdad ("/=") devuelve siempre el complementario de la igualdad ("="), por lo que, en caso necesario, sólo hay que sobrecargar este último. De hecho, el operador de desigualdad sólo se puede sobrecargar si se le atribuye un resultado que no sea de tipo Boolean.

Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Tipos de datos.

Tipos y subtipos.

Un tipo de datos define un conjunto de valores y operaciones primitivas aplicables a los mismos. El lenguaje proporciona tipos predefinidos junto con mecanismos para que el usuario pueda definir nuevos tipos y subtipos. Los tipos se agrupan en clases formando una jerarquía .

La declaración de un tipo requiere la especificación de un nombre y la definición (descripción) del mismo; la forma general de una declaración de tipo es: **type** *Nombre_Tipo* **is** *Definición_tipo*;. La forma de la definición depende de la clase de tipo que se esté definiendo. Un tipo se dice que es derivado si en su definición se utiliza la palabra **new** en referencia a otro tipo (como los tipos Entero y Entero_Corto del siguiente ejemplo); un tipo derivado obtiene las características del que deriva (su "tipo padre").

```
type Octeto is range 0..255;
type Entero is new Integer;
type Entero_Corto is new Entero range 0..1000;
type Vector is array(1..20) of Float;
```

A veces (Ej. cuando se crean estructuras de datos dinámicas) es conveniente hacer una declaración incompleta de tipo que permita disponer de un nombre de tipo que en realidad se definirá con posterioridad en la misma región del programa.

```
type Un_Tipo; --sólo se declara un nombre
              --la definición del tipo requiere
              --una nueva declaración
```

Una declaración de tipo define los valores del tipo y un conjunto de operaciones predefinidas en función de su clase; el usuario puede adjuntar su propia definición de operaciones primitivas adicionales para el tipo que está definiendo (simplemente tiene que adjuntar subprogramas que tengan entre sus parámetros objetos del tipo).

Un subtipo define un subconjunto restringido de valores de un tipo base con atributos específicos. Se declara utilizando la palabra **subtype** en vez de **type**

```
subtype Byte is Integer range 0..255;
```

Compatibilidad y conversión de tipos.

Ada es un lenguaje fuertemente tipado: en general no se pueden mezclar valores de tipos diferentes aún cuando pudieran parecer equivalentes (ni siquiera de un tipo derivado y su tipo padre). Esta regla no incluye a los subtipos: los valores de un subtipo son valores del tipo base.

Se pueden realizar conversiones explícitas entre tipos estrechamente relacionados tal como se establece en la sección 4.6 del "Ada 95 Reference Manual" ; para ello se utiliza el tipo destino como si fuera el nombre de una función con un parámetro que es una expresión del tipo origen de la conversión.

```
X: Float;
Y: Integer := 5;
...
X := Float(Y); --se asigna a X el valor real 5.0
```

Una conversión de tipo cuyo parámetro es una variable no cambia el tipo de la variable, lo que hace es devolver el valor que en el tipo destino corresponde al valor del tipo origen asociado a la variable según las reglas de conversión.

Los tipos escalares ofrecen atributos (image, value) para obtener la representación de un valor como ristra de caracteres (string) y viceversa.

Otros tipos (Ej. ristras de tamaño limitado y ristras de tamaño libre) ofrecen, entre sus operaciones primitivas, funciones de conversión a tipos relacionados.

Tipos limitados.

Cualquier tipo heterogéneo o privado puede declararse limitado usando la palabra **limited**. Un *tipo limitado* es uno para el que no está definida la asignación ni los operadores relacionales de igualdad.

```
type T1 is limited record
   ...
end record;
type T2 is limited private;
```

Tipos privados.

Una declaración de *tipo privado* es una en la que como descripción se utiliza la palabra **private**. La declaración proporciona una vista privada de un tipo que está definido en otro sitio.

```
type T_Privado is private;
```

En el ámbito de su declaración un tipo privado sólo admite asignación y operadores relacionales de igualdad (a menos que además sea limitado).

Una declaración de tipo privado se puede hacer en la sección pública de la especificación de un paquete, tal tipo debe redeclararse en la sección privada (private) de dicha especificación, donde se definirán sus características "ocultas" que sólo podrán usarse dentro del paquete.

También puede declararse privado un parámetro formal genérico en cuyo caso dentro de la unidad genérica sólo se le prodrán aplicar las operaciones antes mencionadas, independientemente del tipo que actúe como parámetro real.

Jerarquía de clases de tipos.

La jerarquía de clases de tipos de Ada 95, tal como se recoge en la seción 3.2 del "Ada 95 Reference Manual", es:

all types
- elementary
  - scalar
    - discrete
      - enumeration
        - character
        - boolean
        - other enumeration
      - integer
        - signed integer
        - modular integer
    - real
      - floating point
      - fixed point
        - ordinary fixed point
        - decimal fixed point
  - access
    - access-to-object
    - access-to-subprogram
- composite
  - array
    - string
    - other array
  - untagged record
  - tagged
  - task
  - protected

Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Estructura de un programa. Unidades

Unidades de programa, compilación y librería.

Un programa en Ada se estructura como un conjunto de *unidades de programa*, pueden ser:

1. **subprogramas**, hay dos clases (procedimientos y funciones) y sirven para definir algoritmos.
2. **paquetes**, sirven para agrupar conjuntos de entidades relacionadas.
3. **tareas**, definen acciones que pueden ejecutarse en paralelo con otras.
4. **unidades protegidas**, sirven para coordinar la compartición de datos entre unidades que se ejecutan en concurrencia.
5. **unidades genéricas**, son subprogramas o paquetes parametrizados que definen componentes reusables.

Las unidades de programa pueden anidarse unas dentro de otras, pero sólo los *subprogramas* , *paquetes* (especificación e implementación) y *unidades genéricas* pueden ocupar el nivel más externo de anidamiento y formar una *unidad de compilación* . Una unidad de compilación es un componente de un programa que puede compilarse por separado. Se distinguen dos clases de unidades de compilación: las *subunidades*, que se encuentran logicamente incluidas en otro componente y se introducen siempre con la palabra **separate**, y las *unidades de librería*, que son componentes independientes que pueden usarse en distintos programas.

Un *subprograma principal* **inicia y gobierna la ejecución de un programa y desde él se desencadena la ejecución del resto de unidades que lo componen. Generalmente el subprograma principal es un procedimiento sin parámetros. Implementaciones particulares del lenguaje pueden admitir otro tipo de subprogramas como subprograma principal.**

Una unidad de compilación típica consta de dos partes: (1) una *cláusula de contexto* que sirve para especificar las unidades de librería que se necesitan (no hace falta si no se necesita ninguna) y (2) una unidad de programa (subprograma, paquete o unidad genérica). El siguiente ejemplo muestra una unidad de compilación que proporciona un procedimiento sin parámetros llamado *"Hola"* que puede servir como subprograma principal de un programa o como unidad de librería utilizable como parte de un programa más complejo.

```
with Text_Io; --Claúsula de contexto
use Text_Io;

procedure Hola is --Elemento de librería (subprograma)
begin
    Put_Line("Hola mundo");
```

```
end Hola;
```

La cláusula de contexto, si aparece, consta obligatoriamente de una cláusula *with* que es la que especifica, separadas por comas (","), las unidades de librería que se necesitan. En el ejemplo, la cláusula *with* especifica que se va a necesitar la unidad de librería *"Text_Io"* que es la que define el procedimiento *"Put_Line"* usado en el procedimiento *"Hola"* para escribir un saludo en la salida estándar.

Opcionalmente, se puede emplear además una cláusula *use* para especificar unidades de librería que se van a usar con frecuencia, de manera que no sea necesario indicar el nombre de la unidad cada vez que se usa un elemento de la misma. En el ejemplo, de no haberse incluido la cláusula *use*, tendría que haberse escrito:

```
with Text_Io; --Cláusula de contexto

procedure Hola is --Elemento de librería (subprograma)
begin
    Text_Io.Put_Line("Hola mundo");
end Hola;
```

La cláusula *use* ahorra trabajo y hace que los programas parezcan menos densos. La cláusula *use* puede ponerse también en cualquier lugar entre las declaraciones de la unidad de programa, aunque ello reduce el alcance de su efecto.

En el caso de que la unidad de librería incluida en el *with* sea un subprograma, no tiene sentido ponerla en una cláusula *use* ya que carece de elementos que puedan ser referenciados individualmente.

```
with Hola; --Cláusula de contexto

procedure Hola_2 is --Elemento de librería (subprograma)
begin
    Hola; --Llamada al procedimiento "Hola" incluido como
          --unidad de librería en la cláusula de contexto
end Hola;
```

Unidades genéricas.

Ada permite crear unidades genéricas, es decir, con parámetros que se pueden concretar para diferentes instancias de la unidad, consiguiendo de esta manera la reutilización de un código que se escribe una sola vez. Para ello basta anteceder la declaración de la unidad correspondiente con la palabra *"generic"*; los parámetros formales de la unidad se sitúan en la zona comprendida entre la palabra *"generic"* y el comienzo de la declaración de la unidad. Sólo los subprogramas y los paquetes admiten genericidad.

```
generic
<parámentros formales genéricos>
<unidad genérica>
```

Parámetros formales genéricos.

Los parámetros formales genéricos pueden ser: objetos, tipos o subprogramas. Al igual que los parámetros formales de los subprogramas, algunas clases de parámetros formales genéricos pueden tener un valor por defecto. Los parámetros formales genéricos se declaran de distinta forma según su naturaleza:

1. **Parámetros por valor.** Se declaran como: *<nombre>: <tipo> := <valor por defecto>*, siendo el valor por defecto opcional.

```
Valor1: integer;
Valor2: positive := 1;
```

2. **Tipos privados:** "private", "limited private", "tagged private",... La definición del tipo privado corresponde, en este caso, al llamador (no va a aparecer en la parte *private*, si estamos desarrollando un paquete).

```
type Tipo1 is private;
type Tipo2 is limited private;
type Tipo3 is tagged private;
type Tipo4 is tagged limited private;
type Tipo5 is new OtroTipoTagged with private ;
...
```

3. **Tipos escalares:**
    1. *Discretos*. Se utiliza el símbolo "(<>)".
    2. *Enteros con signo*. Se utiliza "range <>".
    3. *Modulares*. Se utiliza "mod <>".
    4. *Reales en coma flotante*. Se utiliza "digits <>".
    5. *Reales en coma fija*. Se utiliza "delta <>".
    6. *Decimales*. Se utiliza "delta <> digits <>".

```
type TDiscreto is (<>);
type TEntero is range <>;
type TModular is mod <>;
type TRealFlot is digits <>;
type TRealfijo is delta <>;
type TDecimal is delta <> digits <>;
```

4. **Arrays.** Hay que especificar el tipo de los índices y el tipo de los elementos que pueden ser a su vez parámetros formales genéricos declarados previamente:

```
type TElemento is private;
```

```
type Índice is (<>);
type Vector is array (Índice range <>) of TElemento;
```

5. **Punteros.** Hay que especificar el tipo apuntado que puede ser un parámetro formal genérico declarado previamente:

```
type TNodo is private;
type TP_Nodo is access TNodo;
```

6. **Subprogramas.** Se utiliza la palabra "with" precediendo al protocolo del subprograma que se espera:

```
type TElemento is private;
with procedure Acción(X : in TElemento);
```

Se puede especificar un nombre por defecto para el subprograma pasado por parámetro, utilizando *"is <nombre>"*:

```
with procedure Acción (X : in TElemento) is Escribir;
```

Si se quiere que el nombre por defecto sea el mismo que el del parámetro formal, se pondrá como nombre "<>":

```
with procedure Acción (X : in TElemento) is <>;
```

En caso de especificar un nombre por defecto se podrá omitir el parámetro al instanciar la unidad.

7. **Paquetes.** Se utiliza la palabra *"with"* precediendo a la instanciación del paquete que queremos como parámetro formal.

```
with package <Nombre_formal> is new <Nombre_genérico> (<>);
```

Instanciación de unidades genéricas.

Las unidades genéricas representan una plantilla mediante la cual se indica al compilador como poder crear (en distintas situaciones configuradas por los parámentros) unidades no genéricas que responden al mismo funcionamiento general. De esta forma, las unidades genéricas no se usan directamente, sino sólo para crear instancias no genéricas a partir de ellas. Para crear una instancia de una unidad genérica hay que especificar un nombre para la unidad no génerica que el compilador va a construir y los parámetros reales genéricos a utilizar en el lugar de los formales:

```
<tipo unidad> <nombre de la instancia> is new <nombre unidad genérica> (<parámetros reales genéricos>);
```

Los parámetros reales genéricos se corresponden con los formales por posición o por nombre si se utiliza el simbolo "=>". No es necesario proporcionar parámetros reales para los formales que tengan un valor por defecto (cuando es adecuado para el propósito de la instancia).

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Paquetes.

Definición y uso.

Un paquete (*package*) es una unidad que agrupa un conjunto de entidades relacionadas. Su aplicación más corriente es para realizar la declaración de un tipo de datos junto con las operaciones primitivas para su manejo, de forma que éstas puedan ser usadas desde el exterior pero se mantengan ocultos sus detalles de funcionamiento. En otras palabras, un paquete es un mecanismo de encapsulamiento y ocultación de información especialmente útil para definir *tipos abstractos de datos*. Un paquete se divide generalmente en dos partes: *especificación* e *implementación* que se sitúan en ficheros diferentes con las extensiones respectivas *"*.ads"* y *"*.adb"*.

Especificación.

La estructura de la especificación de un paquete es la siguiente:

```
package <nombre_de_paquete> is
    <elementos públicos>
private
    <elementos privados>
end <nombre_de_paquete>;
```

La parte anterior a la palabra *"private"* constituye la interfaz de los servicios ofertados por el paquete (son accesibles a quien lo use). La parte que sigue constituye la parte privada de la especificación, tales como la estructura de datos con que se implementa un tipo ofrecido en la intefaz u otros elementos similares. La palabra *"private"* puede omitirse, en cuyo caso se entiende que la parte privada de la especificación es nula. La siguiente especificación es de un paquete llamado *"Pila_De_enteros"* que ofrece un tipo llamado "*Pila"* y cuatro operaciones primitivas para manejar pilas de valores de tipo *Integer*.

```
package Pila_De_Enteros is
   type Pila is limited private;
   procedure Apilar(P: in out Pila;E: in Integer);
   procedure Desapilar(P:in out Pila);
   function Cima(P: Pila) return Integer;
   function Es_Vacía(P:Pila) return Boolean;
private
   type Nodo is record
      Info: Integer;
      Sig : Pila;
```

```
   end record;
   type Pila is access Nodo;
end Pila_De_Enteros;
```

Implementación.

La especificación de un paquete requiere una implementación si contiene declaraciones que requieran completarse (por ejemplo, la declaración de una función requiere su desarrollo). La estructura de un *"package body"* es:

```
package body <nombre_de_paquete> is
   <desarrollo del paquete>
end <nombre_de_paquete>;
```

Todas las declaraciones hechas en la especificación del paquete se pueden usar en la implementación (body), aunque estén en la parte "private". Las declaraciones hechas en el "package body" sólo se pueden usar en el mismo. A continuación se muestra el *"package body"* del paquete *"Pila_de_enteros"*:

```
with Unchecked_Deallocation;
package body Pila_De_Enteros is
   procedure Libera is new Unchecked_Deallocation(Nodo,Pila);
   procedure Apilar(P: in out Pila;E: in Integer) is
   begin
      P := new Nodo'(E,P);
   end Apilar;
   procedure Desapilar(P:in out Pila) is
      Aux: Pila := P;
   begin
      P := P.Sig;
      Libera(Aux);
   end Desapilar;
   function Cima(P: Pila) return Integer is
   begin
      return P.Info;
   end Cima;
   function Es_Vacía(P:Pila) return Boolean is
   begin
      return P = null;
   end Es_Vacía;
end Pila_De_Enteros;
```

Utilización.

Una vez compiladas la especificación y la implementación de un paquete, se puede hacer uso del mismo en otras unidades, simplemente incluyendo el nombre del paquete en sus cláusulas de contexto.

```
with Text_Io,
     Ada.Integer_Text_Io,
     Pila_De_Enteros;
use  Text_Io,
     Ada.Integer_Text_Io,
     Pila_De_Enteros;
procedure Prueba is
   P: Pila;
begin
   for I in 1..5 loop
      Apilar(P,I);
   end loop;
   while not Es_Vacía(P) loop
      Put(Cima(P));
      New_Line;
      Desapilar(P);
   end loop;
end Prueba;
```

Paquetes genéricos.

Un paquete genérico es el que posee parámetros que, al ser fijados, permite crear instancias no genéricas aplicables en situaciones diferentes, reutilizando el código sin tener que reescribirlo. Por ejemplo, en el caso de las pilas de enteros del ejemplo previo, es evidente que tanto la estructura de datos del tipo Pila, como los algoritmos de manipulación que la acompañan no dependen en realidad de que los elementos apilados sean de tipo Integer. En consecuencia, se puede escribir un paquete genérico parametrizando el tipo de los elementos, de forma que luego se puedan crear instancias de pilas que acomoden distintos tipos de elementos.

Especificación: se añade la lista de parámetros genéricos formales y se sustituyen todas las apariciones de *Integer* por *T_Elemento*, el párametro formal del paquete:

```
generic
   type T_Elemento is private;
package Pilas is
   type Pila is limited private;
   procedure Apilar(P: in out Pila;E: in T_Elemento);
   procedure Desapilar(P:in out Pila);
   function Cima(P: Pila) return T_Elemento;
   function Es_Vacía(P:Pila) return Boolean;
private
   type Nodo is record
      Info: T_Elemento;
      Sig : Pila;
```

```
   end record;
   type Pila is access Nodo;
end Pilas;
```

Implementación: se sustituyen todas las apariciones de *Integer* por *T_Elemento*:

```
with Unchecked_Deallocation;
package body Pilas is
   procedure Libera is new Unchecked_Deallocation(Nodo,Pila);
   procedure Apilar(P: in out Pila;E: in T_Elemento) is
   begin
      P := new Nodo'(E,P);
   end Apilar;
   procedure Desapilar(P:in out Pila) i s
      Aux: Pila := P;
   begin
      P := P.Sig;
      Libera(Aux);
   end Desapilar;
   function Cima(P: Pila) return T_Elemento is
   begin
      return P.Info;
   end Cima;
   function Es_Vacía(P:Pila) return Boolean is
   begin
      return P = null;
   end Es_Vacía;
end Pilas;
```

Utilización: el nuevo paquete, *"Pilas"*, aparece en la sentencia *with* de la cláusula de contexto, pero no así en la *use* puesto que al ser genérico es sólo una plantilla a partir de la cual definir instancias, pero no es en sí mismo una unidad utilizable directamente. La definición de una instancia se hace especificando un nombre para el paquete no genérico al que dará lugar y los valores reales correspondientes a los parámetros (al menos de aquellos que no tengan un valor por defecto):

```
with Text_Io,
     Ada.Integer_Text_Io,
     Pilas;
use Text_Io,
    Ada.Integer_Text_Io;
procedure Prueba is
   package Pila_De_Enteros is new Pilas(Integer);
   use Pila_De_Enteros;
   package Pila_De_Reales is new Pilas(Float);
   use Pila_De_Reales;
```

```
   P: Pila_De_Enteros.Pila;
   Q: Pila_De_Reales.Pila;
begin
   for I in 1..5 loop
      Apilar(P,I);
   end loop;
   while not Es_Vacía(P) loop
      Put(Cima(P));
      New_Line;
      Desapilar(P);
   end loop;
end Prueba;
```

Herencia de paquetes.

Se pueden derivar paquetes a partir de otros existentes, creando lo que se llama un paquete "hijo". Normalmente, lo que se pretende conseguir es añadir una nueva funcionalidad sin tener que modificar el paquete original. El siguiente ejemplo crea un paquete hijo de *"Pilas"* con el fin de proporcionar una operación que permita obtener una copia de una pila. Además, se define una constante para representar la pila vacía que, combinada con la operación de copia, permite vaciar una pila con una sola sentencia:

```
generic
package Pilas.Copiables is
   Vacía: constant Pila;
   procedure Copia(Pe: in Pila;Ps: out Pila);
private
   Vacía: constant Pila := null;
end Pilas.Copiables;
```

Un paquete derivado de otro genérico tiene obligatoriamente que ser genérico (aunque no precise parámetros, los precisa su padre). De un paquete no genérico se puede derivar uno genérico.

```
package body Pilas.Copiables is
   procedure Vaciar(P:in out Pila) is
   begin
      while not Es_Vacía(P) loop
         Desapilar(P);
      end loop;
   end Vaciar;
   function Copia(P:Pila) return Pila is
   begin
      if Es_Vacía(P) then
         return Vacía;
      else
         return new Nodo'(Cima(P),Copia(P.Sig));
```

```
      end if;
   end Copia;
   procedure Copia(Pe: in Pila;Ps: out Pila) is
      Aux: Pila := Copia(Pe);
   begin
      Vaciar(Ps);
      Ps := Aux;
   end Copia;
end Pilas.Copiables;
```

Obsérvese que ni en la especificación ni en la implementación aparece el paquete original en una cláusula de contexto; no es necesario dado que es una extensión del paquete y como tal tiene acceso a todo lo declarado en la especificación del original, incluida la parte privada, aunque no así a lo desarrollado en el *"package body"* .

Una unidad que requiera toda la funcionalidad de los dos paquetes necesitará incluirlos a ambos.

```
with Text_Io, Ada.Integer_Text_Io, Pilas, Pilas.Copiables;
use  Text_Io, Ada.Integer_Text_Io;
procedure Prueba is
   package Pila_De_Enteros is new Pilas(Integer);
   use Pila_De_Enteros;
   package Pilas_C is new Pila_De_Enteros.Copiables;
   use Pilas_C;
   P,Q: Pila;
begin
   for I in 1..5 loop
      Apilar(P,I);
   end loop;
   Copia(P,Q);
   Copia(Vacía,P);
   while not Es_Vacía(Q) loop
      Put(Cima(Q));
      New_Line;
      Desapilar(Q);
   end loop;
end Prueba;
```

Los paquete derivados y sus ancestros forman una jerarquía que se conoce como familia o sistema de paquetes.

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Tipos simples o escalares.

Introducción.

Los tipos escalares se clasifican en: tipos *enumerados*, tipos *enteros* y tipos *reales*. Los tipos enumerados y los tipos enteros pertenecen a la categoría de tipos *discretos* u *ordinales* que se caracterizan porque a cada valor le corresponde una posición que se representa por un número entero. Los tipos enteros y reales se clasifican también como tipos *numéricos*. Todos los tipos escalares admiten los operadores relacionales, los tipos numéricos admiten diversos operadores aritméticos .

Tipos enumerados.

Los tipos enumerados se definen enumerando la lista de valores literales del tipo (un valor literal enumerado puede ser un carácter literal o una secuencia de caracteres válida como identificador).

```
type Días is
(Lunes,Martes,Miércoles,Jueves,Viernes,Sabado,Domingo);
type Color is (Rojo,Azul,Amarillo);
```

El orden y la posición de los valores de un tipo enumerado vienen determinados por el orden en que se enumeran (al primer valor de la lista le corresponde el valor cero). Tipos enumerados distintos pueden definirse con literales iguales, cuando el uso de estos literales pueda resultar ambiguo se deberán cualificar con el nombre del tipo:

```
type Semaforo is (Rojo,Amarillo,Verde);
...
for A in Color'(Rojo)..Color'(Amarillo) loop
...
```

Se pueden definir subtipos de un tipo enumerado especificando un *rango* de valores:

```
subtype Laborables is Días range Lunes..Viernes;
```

La entrada/salida de valores de un tipo enumerado se realiza con el paquete Enumeration_IO.

Tipos carácter

Un tipo carácter es cualquier tipo enumerado que tenga al menos un valor literal que sea un carácter

literal. Está predefinido el tipo *Character* que comprende los 256 caracteres del conjunto de caracteres del lenguaje.

Tipos booleanos.

Existe predefinido en el paquete estándar un tipo enumerado llamado *Boolean* con los valores *True* y *False*.

Tipos enteros.

Los tipos enteros se dividen en *enteros con signo* y *modulares* , cuya aritmética está definida en función de un módulo (si A y B son variables de un tipo modular con módulo M, A + B significa (A + B) **mod** M). Está predefinido el tipo *Integer*, con dos subtipos: *Natural* y *Positive* que abarcan todo el rango positivo de los enteros, incluyendo o no el cero, respectivamente.

Un entero con signo se define especificando un rango de valores enteros, mientras que un entero modular requiere un módulo, que debe ser un valor entero positivo.

```
type Sig_Byte is range 0..255; --tipo entero con signo (de 0 a 255)
type Mod_Byte is mod 256; --tipo entero modular (con módulo 256)
```

En ambos casos se pueden definir subtipos especificando un rango de valores comprendido en los valores del tipo. Los valores de un tipo modular van desde cero hasta el módulo menos uno.

La entrada/salida de valores de un entero con signo se realiza con el paquete Integer_IO. La entrada/salida de valores de un entero modular se realiza con el paquete Modular_IO. Se puede realizar entrada/salida de valores del tipo predefinido *integer* usando el aquete no génerico Ada.Integer_Text_IO.

Tipos reales.

Los tipos reales se dividen en reales en *coma flotante* y reales en *coma fija*; los reales en coma fija pueden a su vez ser *ordinarios* o *decimales*. En todos los casos representan aproximaciones de los números reales matemáticos, con un límite de error relativo en el caso de coma flotante y absoluto en el caso de coma fija. Está predefinido el tipo *Float* que designa números reales en coma flotante.

Se pueden definir tipos en coma flotante especificando la precisión relativa deseada por medio de un valor entero que indique el número de digitos requeridos. También se puede especificar un rango real para los valores del tipo.

```
type Real is digits 8;
type Real_Corto is digits 5 range -1.0 .. 1.0;
```

Para los tipos en coma fija se debe especificar el límite absoluto de error (delta), que debe ser potencia de 10 en el caso de los decimales (para los que también se debe especificar el número total de dígitos a representar), y si se desea un rango real.

```
type Real_Fijo is delta 0.125 range -100000.0 .. 100000.0; --tipo
fijo ordinario
type Tipo_Dec is delta 0.01 digits 6 range -1000.00..1000.00; --
tipo fijo decimal
```

La entrada/salida de valores de un tipo real en coma flotante se realiza con el paquete Float_IO. La entrada/salida de valores de un tipo real en coma fija ordinario se realiza con el paquete Fixed_IO. La entrada/salida de valores de un tipo real en coma fija decimal se realiza con el paquete Decimal_IO. Se puede realizar entrada/salida de valores del tipo predefinido *float* usando el paquete no génerico Ada.Float_Text_IO.

Atributos de los tipos escalares.

Llamamos atributos de los tipos escalares a determinadas operaciones especiales que permiten averiguar propiedades del tipo y que se invocan cualificadas por el nombre del tipo mediante un apóstrofe (si T es un tipo escalar y Op una operación con un parámetro, X, para ejecutar Op se escribe T'Op(X)); las hay aplicables a cualquier tipo escalar y otras que dependen de la categoría a la que pertenezca el tipo.

Atributos generales de los escalares.

A cualquier tipo escalar, T, le son aplicables, entre otros, los siguientes atributos:

T'First	Devuelve el valor más pequeño del tipo T
T'Last	Devuelve el valor más grande del tipo T
T'Range	Devuelve el rango T'First..T'Last
T'Succ(V)	Siendo V un valor de tipo T, devuelve el valor que le sigue (en los tipos reales depende de la representación). T'Succ(T'Last) no pertenece al tipo T.
T'Pred(V)	Siendo V un valor de tipo T, devuelve el valor que le precede (en los tipos reales depende de la representación). T'Pred(T'First) no pertenece al tipo T.
T'Image(V)	Siendo V un valor de tipo T, devuelve una ristra con la representación literal de V.
T'Width	Devuelve el número máximo de caracteres que requerirá un literal de tipo T devuelto por T'Image(V).

T'Value(S)	Siendo S una ristra con la representación literal de un valor de tipo T, devuelve el valor correspondiente.

Atributos de los tipos discretos.

A cualquier tipo discreto, T, le son aplicables los siguientes atributos:

T'Pos(V)	Siendo V un valor de tipo T, devuelve la posición de V en el rango de valores de T.
T'Val(P)	Siendo P un valor entero, devuelve el valor de T que le corresponde (de no existir se produce un Constraint_Error).

Atributos de los tipos reales.

Permiten conocer datos relacionados con la precisión del tipo (dígitos, delta, escala, etc.).

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Entrada/salida por terminal.

La librería Text_IO.

Las operaciones básicas de entrada/salida de Ada se encuentran definidas en la librería estándar *Text_IO* que habrá que incluir en la *cláusula de contexto* de una unidad de programa que pretenda usarlas:

```
with Text_Io;
use Text_Io;
```

Una vez hecho esto, se pueden emplear las operaciones básicas de entrada salida sobre elementos de tipo *character* y de tipo *string* (ristras de tamaño fijo).

También se puede realizar entrada/salida de valores del tipo predefinido *Integer* y sus subtipos usando el paquete *Ada.Integer_Text_IO*; y del tipo predefinido *Float* y sus subtipos usando el paquete *Ada.Float_Text_Io* .

```
with Ada.Integer_Text_Io, Ada.Float_Text_Io;
use Ada.Integer_Text_Io, Ada.Float_Text_Io;
```

Para poder realizar operaciones de entrada/salida con elementos de tipos numéricos distintos del *Integer* y el *Float* y sus subtipos, o de tipos enumerados, se deberá instanciar el que corresponda de entre los seis paquetes genéricos de entrada/salida contenidos en la librería Tex_IO, usando como parámetro el nombre del tipo.

```
package Día_IO is new Enumeration_IO(Día);      --Enumeration_IO
sirve para tipos enumerados
package MiEntero_IO is new Integer_IO(MiEntero);-- Integer_IO sirve
para enteros con signo
package Natural_IO is new Modular_IO(Natural);  -- Modular_IO sirve
para enteros sin signo
package rf_IO is newFloat_IO(rf);               -- Float_IO sirve
para reales en coma flotante
package rfija1_IO is new Fixed_IO(rfija1);      --Fixed_IO sirve
para reales en coma fija ordinarios
package rfija2_IO is new Decimal_IO(rfija1);    -- Decimal_IO sirve
para reales en coma fija decimales
```

Si en un programa se utilizan subtipos, basta instanciar el paquete adecuado para el tipo base de los mismos, lo que permitirá usar las operaciones de entrada/salida tanto con éste como con todos sus subtipos.

Operaciones de entrada/salida.

Estructura de la entrada.

La comunicación entre un programa y la terminal no es directa: las teclas pulsadas por el usuario van acumulándose en un "buffer" de entrada formando una secuencia jalonada por marcas de fin de línea (que se generan cuando se pulsa la tecla de retorno) entre otras marcas. Cuando se ejecuta una operación de entrada, ésta toma los caracteres del principio de dicha secuencia, hasta completar un valor del tipo requerido.Un usuario puede teclear antes de que el programa ejecute una instrucción de entrada. Una operación de entrada puede no consumir toda la entrada pendiente, en consecuencia la siguiente no tiene porqué esperar a que el usuario teclee nuevos datos. El programa tan sólo se detendrá en una operación de entrada cuando el "buffer" de entrada esté vacío o no tenga suficientes caracteres para completar el valor requerido, según el tipo de operación.

```
                          ____________entrada_____________
Operaciones de entrada << A 11<eol>pedro<eol> xY23<eol> << Teclado
(dispositivo de entrada)
                          -----------------------------
```

Operaciones básicas de entrada.

Las operaciones básicas de entrada son *Get* y *Get_Line*, para leer información, y *skip_line* para saltar finales de línea.

```
get(c);
```

Si c es de tipo *character* (1), el primer carácter disponible en la entrada se carga en c, a menos que se trate de un fin de línea o fin de página, en cuyo caso se saltarán todas las marcas que se encuentren hasta encontrar un carácter válido.

Si c esde tipo *string* (2), se realizan tantas llamadas a la versión de *get* para caracteres como sea necesario para completar el tamaño de c.

Si c es de un tipo númerico (3), (4), (5), (6), (7), se saltan los caracteres de control y los espacios que haya en la entrada, a partir de ahí se lee en c mientras la secuencia de caracteres encontrados pueda interpretarse como un valor numérico del tipo apropiado. Si lo que se encuentra, tras saltar los caracteres de control y espacios, no se puede interpretar de forma adecuada se produce un error.

Si c es de tipo enumerado (8), se actúa igual que en el caso anterior, sólo que lo que se encuentre ha de ser una palabra que corresponda con uno de los valores enumerados del tipo (si bien no importa que

esté en mayúsculas o minúsculas).

```
get_line(s,l);
```

Donde s es de tipo *string* y l es de tipo *natural* (9). Lee la string s e informa en l de qué posición ocupa en s el último carácter leído. La lectura termina cuando se alcanza un final de línea o cuando se han leído tantos caracteres como el tamaño definido para s en su declaración. Si se encuentra un final de línea antes de haber completado el tamaño de s, se termina la lectura, se avanza la entrada hasta el principio de la siguiente línea, l tendrá un valor menor que el límite superior del rango de índices de s (acorde con los carácteres leídos) y la subristra de s comprendida entre la posición siguiente a l y el final conservará los valores que tenía antes de la ejecución del *get_line* . Si se completa el tamañode s, l tomará un valor igual al límite superior del rango de índices de s y no se produce ningún avance a la siguiente línea. Si no se lee ningún carácter (el primer carácter de la entrada es un final de línea), l tomara el valor inmediatamente menor al límite inferior del rango de índices de s y la entrada avanza hasta la el comienzo de la siguiente línea sin que el valor de s varíe.

```
skip_line;
```

Sirve para avanzar la entrada hasta el comienzo de la siguiente línea, desechando el resto de la actual (10).

La operación skip_line admite un parámetro que permite indicar cuantas líneas hay que saltar. Así "skip_line(5);" significa repetir 5 veces "skip_line;".

Cuando se ejecuta un *get* **que consume los caracteres válidos de una línea y la siguiente instrucción de entrada/salida en ejecutarse es un** *get_line,* **esta última no leerá nada, ya que encontrará directamente el final de línea. El problema se soluciona poniendo un** *skip_line* **detrás del** *get***.**

Ocurre un problema parecido cuando se ejecuta un *get_line* **en el que se consigue leer todo el tamaño de la ristra implicada, no pasándose por tanto a la siguiente, y a continuación se va a ejecutar otro** *get_line* **. Si se quiere que el segundo** *get_line* **empiece a leer en la siguiente línea habrá que ejecutar un** *skip_line* **, pero sólo después de verificar el resultado del primero.**

Operaciones básicas de salida.

Las operaciones básicas de salida son *put* (11), (12), (12), (13), (14), (15), (16), (17) y *put_line* (18), para escribir información, y *new_line* (19) para cambiar de línea.

```
put(x);
```

Para x de cualquier tipo escalar o de tipo *string* muestra una secuencia de caracteres que representa el valor de x.

```
new_line;
```

Genera un salto de línea. La salida continúa al principio de la siguiente línea.

Al igual que *skip_line***, admite un parámetro que permite saltar varias líneas.**

```
put_line(s);
```

Exclusivamente para s de tipo *string*, equivale a: "*put(s); new_line;* ".

Opciones adicionales

Parámetros adicionales en las operaciones get.

La operación *get* admite, para los tipos numéricos (20), (21), (22), (23), (24), un segundo parámetro que permite definir el "ancho de campo" que se va a considerar a la hora de leer el valor de la entrada. Esto significa que no se continuará leyendo mientras la secuencia de caracteres encontrada pueda interpretarse como un valor del tipo adecuado, sino sólo, y exactamente, hasta que se hayan leído tantos caracteres como indique ese parámetro.

```
entrada=123456789<eol>, x : integer, get(x) => x = 123456789,
entrada=<eol>
entrada=123456789<eol>, x : integer, get(x,4) => x = 1234,
entrada=56789<eol>
```

Otras operaciones de entrada.

Si se quiere saber cúal es el siguiente carácter que toca leer, pero sin quitarlo del buffer de entrada, se puede utilizar la operación *look_ahead* (25).

```
look_ahead(c,fin)
```

Si el siguiente carácter es un final de línea: fin tiene el valor *true* y c está indefinido; en caso contrario: fin tiene el valor *false* y c el del primer carácter del buffer de entrada.

Las operaciones de entrada corrientes toman los caracteres del buffer de entrada y los caracteres tecleados sólo pasan al mismo cuando se pulsa la tecla de retorno, dando así la oportunidad de hacer correcciones, pero en algunas aplicaciones es necesario leer directamentela la tecla pulsada, sin esperar la confirmación que supone la de retorno. Esto puede hacerse con la operación *get_inmediate* (26), que tiene dos versiones.

```
get_inmediate(c)
```

Pone en c el carácter correspondiente a la primera tecla que se pulse.

```
get_inmediate(c,e)
```

Si hay una tecla pulsada: pone en c el carácter correspondiente y e toma el valor *true*; en caso contrario: e toma el valor *false* y c queda indefinido.

Parámetros adicionales en la operación put.

Para los tipos enteros, *put* admite un segundo parámetro, *width*, que especifica el ancho del campo (número de caracteres) utilizado para escribir el número. Si el valor de este parámetro es mayor que el número de caracteres necesario para escribir el valor, se añadirán espacios por la izquierda. Si es menor se utilizará justo la cantidad que se necesite.

Para los tipos reales, se pueden añadir a put tres parámetros: *fore*, *ant* y *exp*. El primero determina el número de caracteres a usar para la parte entera (incluyendo el punto decimal y el signo si es negativo), el segundo fija el número de caracteres de la parte decimal y el tercero el del exponente. Si el valor de *fore* es mayor de lo requerido, se insertan espacios por la izquierda, y si es menor se usa lo necesario. Un valor de *exp* igual a cero causa que la salida se muestre en notación decimal sin exponente.Varias posibles combinaciones se muestran en el siguiente ejemplo.

```
R: Float:=Ada.Numerics.Pi;
Put(R);                                  3.14159E+00
Put(R,Fore => 5);                           3.14159E+00
Put(R,Fore => 5,Aft => 2);                  3.14E+00
Put(R,Exp => 1);                         3.14159E+0
Put(R,Fore => 5,Exp => 0);                  3.14159
Put(R,Fore => 5,Aft => 2, Exp => 0);        3.14
Put(R,Fore => 5,Aft => 0, Exp => 0);        3.1
Put(-R,Fore => 0,Aft => 0, Exp => 0); -3.1
Put(-R,Fore => 0,Aft => 0);              -3.1E+00
Put(-R,Fore => 0,Aft => 10);             -3.1415927410E+00
```


Anterior | Superior | Siguiente

Guía básica de referencia de Ada 95

La librería Text_IO

La librería Text_IO contiene las siguientes declaraciones:

```
with Ada.IO_Exceptions;

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 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
   Unbounded : constant Count := 0; -- Longitud de línea y página
   subtype Field       is Integer range 0 .. implementation-defined;
   subtype Number_Base is Integer range 2 .. 16;
   type Type_Set is (Lower_Case, Upper_Case);
   -- Manejo de ficheros
   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 de los ficheros de entrada y salida por defecto
   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;
-- Control de buffer
procedure Flush (File : in out File_Type);
procedure Flush;
-- Especificación de longitudes de línea y de página
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;
-- Control de columna, línea y página
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;
-- Entrada/salida de caracteres
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);
   -- Entrada/salida de strings
   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);
  -- Paquetes genéricos para entrada/salida de tipos enteros
   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;
   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;
   -- Paquetes genéricos para entrada/salida de tipos reales
   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;
   -- Paquete genérico para entrada/salida de tipos enumerados
```

```
      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;
   -- Excepciones
      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
      ... -- no especificado por el lenguaje
   end Ada.Text_IO;
```

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Tipos estructurados.

Introducción.

Un tipo estructurado define una agrupación de componentes de un tipo más simple; normalmente se distingue entre tipos estructurados homogéneos y tipos estructurados heterogéneos.

Estructuras homogéneas.

Definición.

Los *arrays* son estructuras homogéneas constituidas por un conjunto ordenado de elementos del mismo tipo a los que se puede acceder individualmente indicando su posición en el array, la cual se expresa como un conjunto de índices discretos. Un array puede ser mono o multidimensional; determinar la posición de un elemento requiere un índice por cada dimensión. Cuando se declara un array es necesario indicar el tipo de los elementos y el rango de variación de los índices de cada dimensión; si el array tiene más de una dimensión, se declara separando los rangos de cada dimensión mediante comas (",").

```
type TV is array(0..4) of Integer;
V: TV; --elementos V(0), V(1), V(2), V(3) y V(4) de tipo Integer
M:array(1..2,'A'..'B') of Float; --elementos(M(1,'A'), M(1,'B'),
M(2,'A') y M(2,'B') de tipo Float
VM: array(1..2) of TV; --array de arrays
```

Se pueden declarar arrays no restringidos (*unconstraint arrays*) especificando el tipo de los índices pero no su rango:

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

No se pueden declarar variables de un tipo no restringido: se debe definir primero un subtipo no restringido o especificar directamente los rangos de los índices en la propia declaración de una variable (se hace una excepción en el caso de los parámetros formales de un subprograma que sólo declaran una vista de los parámetros reales, para que se pueda aplicar a arrays de diferentes tamaños).

```
subtype TRestringido is TLibre(1..50);
A: TRestringido;
B: TLibre(1..10);
...
procedure OrdenarVector(V: in out TLibre);
```

```
...
OrdenarVector(A);
OrdenarVector(B);
```

Atributos.

Todos los array tienen los atributos: First, Last, Length y Range; si A es un array y N un valor entero positivo menor o igual que el número de dimensiones del array, se tiene que:

```
A'First es el límite inferior del rango del primer índice de A.
A'First(N) es el límite inferior del rango del N-ésimo índice de A.
(A'First(1) = A'First).
A'Last es el límite superior del rango del primer índice de A.
A'Last(N) es el límite superior del rango del N-ésimo índice de A.
(A'Last(1) = A'Last).
A'Range es equivalente al rango A'First .. A'Last.
A'Range(N) es equivalente al rango A'First(N) .. A'Last(N).
(A'Range(1) = A'Range).
A'Length es el número de valores del rango del primer índice.
A'Length(N) es el número de valores del rango del N-ésimo índice.
(A'Length(1) = A'Length).
```

El uso de los atributos permite escribir algoritmos aplicables independientemente del rango de las distintas dimensiones de un array.

```
procedure OrdenarVector(V: in out TLibre) is
   E: Float;
   P: Integer;
begin
   for I in V'First..Integer'Pred(V'Last) loop
      P := I;
      for J in Integer'Succ(I)..V'Last loop
         if V(J) < V(P) then
            P := J;
         end if ;
      end loop;
      if P /= I then
         E := V(I);
         V(I) := V(P);
         V(P) := E;
      end if;
   end loop;
end OrdenarVector;
```

Literales array.

Son agregados de valores del tipo de los elementos del array que se pueden utilizar para inicializar variables o constantes de tipo array, asignarlos a variables de tipo array o emplearlos en otras operaciones que involucren arrays. Un agregado posicional de una array monodimensional se forma enumerando los valores según su posición:

```
V: array(1..5) of integer := (12,3,45,6,8); --V(1) = 12, V(2) = 3,
V(3) = 45, V(4) = 6 y V(5) = 8.
```

Se puede utilizar una cláusula others para dar un mismo valor a las últimas posiciones del array:

```
V := (12,3,45, others => 0); --V(1) = 12, V(2) = 3, V(3) = 45, V(4)
= 0 y V(5) = 0.
V := (others => 0); --todas las posiciones a cero.
```

Para un array multidimensional se enumeran los elementos agrupándolos por dimensiones:

```
M: array(1..3,1..2) of character :=
(('a','b'),('c','d'),('e','f'));
...
M := (('a','b'),('c', others => 'd'),others => ('e','f'));
```

También se pueden definir agregados por asociación nominal, indicando para cada valor la posición a que corresponde, en cuyo caso se pueden enumerar salteados y desordenados:

```
V := (4, 3 => 15, 5 => 8, others => 0); --V(1) = 4, V(2) = 0, V(3)
= 15, V(4) = 0 y V(5) = 8.
M := (3 => ('e','f'), 1 => ('a','b'), others => ('c','d'));
```

Rodajas (slices).

En un array **monodimensional** se pueden referenciar *slices* : son subconjuntos contiguos de elementos; sólo hay que especificar el rango del slice.

```
V(1..3) := V(3..5);
```

Un slice es un array con el rango de indices especificado; si, por ejemplo, se quiere asignar a una variable simple el primer elemento del slice que va de las posiciones 3 a la 5 de un vector, hay que tener en cuenta que el índice de dicho elemento es 3:

```
A := V(3..5)(3); --correcto
A := V(3..5)(1); --incorrecto
```

Estructuras heterogéneas.

Definición.

Los *record* son estructuras heterogéneas: agregados de elementos (*campos*) del mismo o distintos tipos que se pueden acceder individualmente mediante su nombre. Un record se define con la palabra "record", seguida de la declaración de los campos del record y "end record".

```
type Complejo is record
    Real, Imag: Float;
end record;
```

Se pueden especificar valores iniciales para los campos de un record en la propia definición del tipo.

```
type Complejo is record
    Real, Imag: Float := 0.0;
end record;
```

El acceso a los campos individuales de una variable record se consigue poniendo un punto (".") detrás del nombre de la variable y a continuación el nombre del campo al que se quiere acceder.

```
X, Y, Z: Complejo;
...
X.Real := 1.0;
X.Imag := 1.0;
Y.Real := X.Real;
Y.Imag := X.Imag;
Z := Y; --X, Y, Z tienen el mismo valor
```

Literales record.

Se pueden formar literales record para usarlos en inicializaciones, asignaciones y otras operaciones de record de dos formas:

1. como agregado posicional, especificando los valores de todos los campos en el orden adecuado y entre paréntesis: X := (3.5, 7.1);
2. como agregado nominal, especificando los nombres de los campos junto con los valores: X := (real => 3.5, imag => 7.1);

Cuando se usa un agregado nominal los campos se pueden enumerar en cualquier orden, pero siempre hay que enumerarlos todos.

Record variante.

Se pueden definir tipos record que puedan tener campos diferentes en función del valor de un

discriminate discreto .

```
type Figura is (Rectángulo, Círculo);
type Poligono(Forma:Figura) is record --Forma es el discriminante
    Pos_X, Pos_Y: Float;
    case Forma is
        when Rectángulo => Base,Altura: Float;
        when Círculo => Radio: Float;
    end case;
end record;
```

Cuando se declaren variables hay que especificar obligatoriamente el valor del campo discriminante (excepto en la declaración de parámetros formales); una vez hecha la declaración no se puede cambiar.

```
R: Poligono(Rectángulo);
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Sentencias simples.

Introducción.

Sentencias simples son las que no contienen ninguna otra sentencia.

Asignación.

La asignación se expresa mediante el simbolo ":=" según la sintáxis:

```
Nombre_de_variable := Expresión ;
```

El primer paso en la ejecución de una asignación es la evaluación de la expresión (r-value) cuyo resultado ha de ser compatible con el tipo de la variable de la izquierda (l-value).

La operación de asignación proporciona un valor nuevo a una variable y puede realizarse en formas distintas a la sentencia de asignación, por ejemplo, pasando un parámetro por copia a un subprograma o con una operación de entrada.

Sentencias que cambian el flujo de ejecución.

Se agrupan bajo este epigráfe las sentencias que permiten abandonar la ejecución de bucles, "saltar" a punto concreto de un programa, regresar desde un subprograma o acabar la ejecución de una tarea concurrente.

Una sentencia *exit* permite abandonar la ejecución de un bucle, ello puede hacerse de manera incondicional, o en función de una condición booleana asociada al *exit* por una cláusula *wh*en.

```
loop
   ...
   exit;
   ...
end loop;

loop
     ...
     exit when Condición;
     ...
end loop;
```

Existe la posibilidad de especificar en la sentencia *exit* un identificador asociado previamente con uno de los bucles que la engloban, ello permite salir de un conjunto de bucles anidados usando una única sentencia.

```
Bucle_Externo:
loop
   loop
     ...
     exit Bucle_Externo when Condición ;
     ...
   end loop;
end loop;
```

Una sentencia de salto incondicional (*goto*) continúa la ejecución en un punto especificado por una etiqueta. Una etiqueta se forma con un identificador encerrado entre los simbolos "<<" y ">>".

```
<<Etiqueta_1>>
...
goto Etiqueta_1;
```

Una sentencia de salto incondicional sólo puede transferir la ejecución a una etiqueta situada en la misma secuencia de sentencias (no puede salir de un subprograma, ni ir a una sección de tratamiento de excepciones, ni entrar dentro de una estructura alternativa,...).

Una sentencia de retorno (*return*) sirve para terminar la ejecución de un subprograma, un *entry_body* o un *accept_statement*. En el caso de una función, debe ir acompañada de una expresión que especifique el valor a devolver por la misma.

```
return; --return de un procedimiento
return Resultado; --return de una función.
```

La ejecución de una función siempre debe terminar con una sentencia *return* (salvo casos excepcionales relacionados con la inserción de código máquina).

Otras sentencias simples.

La lista completa de clases de sentencias simples de Ada 95 es la siguiente:

```
null_statement           null;
assignment_statement     Nombre_de_variable := Expresión;
exit_statement           exit [Nombre_bucle] [when condition];
goto_statement           goto Nombre_Etiqueta;
procedure_call_statement Nombre_procedimiento;
```

```
Nombre_procedimiento(Lista_de_parámetros_reales);
return_statemen          return Expresión;
entry_call_statement     Nombre_entry [Parámetros_reales];
requeue_statement        requeue Nombre_entry [with abort]
delay_statement          delay until Expresión_espera; | delay
Expresión_espera ;
abort_statement          abort Nombre_tarea { , Nombre_tarea};
raise_statement          raise [Nombre_excepción];
code_statement           Expresión_cualificada;
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Declaraciones. Reglas de ámbito.

Una declaración asocia un nombre con (una vista de) una entidad del programa. La declaración define: (1) el nombre de la entidad que generalmente es un identificador, (2) una vista de la entidad que fija el modo en que ésta se puede usar y (3) posiblemente la propia entidad, aunque se pueden hacer declaraciones que definan vistas de entidades definidas en otra parte del programa.

Toda declaración tiene un *ámbito* (conjunto de sentencias) en el que es visible el nombre declarado y se puede usar para referenciar la vista de la entidad asociada y tener acceso a la misma. Generalmente el ámbito de una declaración se extiende hacía adelante y hacía adentro: una declaración hecha en un subprograma o en un bloque es visible en todo el resto del mismo subprograma o bloque a partir del punto en que aparece, incluyendo los subprogramas o bloques anidados en él, salvo que incluyan una declaración con el mismo nombre (una declaración oculta cualquier otra del mismo nombre hecha en un bloque más externo).

```
declare
  ...
  I,J,K : Float;
  ...
begin
  --ámbito de I,J,K de tipo Float
  ...
  declare
    ...
    I,L: Integer;
    ...
  begin
    --ámbito de J,K de tipo Float e I,L de tipo Integer
    ...
  end;
  --ámbito de I,J,K de tipo Float
  ...
end;
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Sentencias compuestas.

Introducción.

Sentencias compuestas son las que contienen otras sentencias simples o compuestas.

Bloques.

Un bloque es una sentencia compuesta formada por una secuencia de sentencias agrupadas mediante las palabras delimitadoras "begin", "end" y posiblemente con algunas declaraciones locales.

```
[declare
   declaraciones locales]
begin
   sentencias
end;
```

Un bloque puede ponerse en cualquier sitio donde pueda ponerse una sentencia simple. Ejemplos de bloques:

```
--bloque sin declaraciones locales
begin
   Put_Line("Hola");
end;
--bloque con declaraciones locales
declare
   Aux : integer; --la variable Aux sólo existe dentro del bloque
begin
   Aux := i; --i, j se suponen declaradas en un ámbito más externo
   i := j;
   j := Aux;
end;
```

Sentencias de control.

Las sentencias de control son sentencias compuestas que controlan la ejecución de un grupo de sentencias.

Selección entre dos alternativas.

La selección entre dos posibles alternativas de ejecución se realiza mediante la evaluación de una condición formulada por una expresión booleana.

```
if condición_booleana then
   sentencias
end if;
```

```
if condición_booleana then
   sentencias
else
   sentencias
end if;
```

Selección entre múltiples alternativas.

La selección entre múltiples alternativas de ejecución se realiza mediante la evaluación de un selector.

```
case selector is
   when alternativa => sentencias
   when alternativa => sentencias
   ...
   when others => sentencias
end case;
```

El selector debe ser una expresión discreta de tipos integer o enumerados (tipo ordinal). Las alternativas pueden ser uno o varios valores, o rangos, del tipo del selector separados por "|" (equivale al operador OR). Los valores no pueden repetirse entre dos cláusulas "when". En el caso de que las cláusulas "when" no cubran todos los posibles valores del tipo del selector, es necesario incluir la cláusula "others" para los valores no contemplados.

```
case Mes is
   when 1 .. 2 | 12 => Put("Invierno");
   when 3 .. 5 => Put("Primavera");
   when 6 .. 8 => Put("Verano");
   when 9 .. 11 => Put("Otoño");
   when others => Put("¿En qué planeta estás?");
end case;
```

Repetición controlada por contador.

Hace que un conjunto de sentencias se ejecute un número de veces determinado a priori; la cuenta de las ejecuciones realizadas se lleva a cabo mediante un *parámetro de control del bucle* que va tomando los sucesivos valores de un rango ordinal.

```
for Num in 1..5 loop
    Put(Num); -- escribe 1 2 3 4 5
end loop;
```

El parámetro de control no se declara, adopta automáticamente el tipo del rango que debe recorrer, es local al bucle (oculta mientras se ejecuta cualquier otra declaración con el mismo nombre) y no puede modificarse.

El rango puede recorrerse en orden inverso utilizando la palabra "reverse".

```
for Num in reverse 1..5 loop
    Put(Num); -- escribe 5 4 3 2 1
end loop;
```

La forma general de un bucle controlado por contador en Ada es:

```
for parámetro_de_control in [reverse] rango_ordinal loop
    sentencias
end loop;
```

Repetición controlada por condición lógica.

Ada sólo ofrece explícitamente la versión con control previo.

```
while condición_booleana loop
    ...
end loop;
```

Sin embargo, ofrece un bucle sin esquema de iteración preestablecido que puede simular cualquier esquema. Su forma básica es un bucle infinito como el siguiente:

```
loop
    ...
end loop;
```

De un bucle como este (de hecho de cualquier bucle) se puede salir usando una sentencia *exit* donde se desee.

```
loop
    ...
    exit when condición_booleana;
    ...
end loop;
```

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Ficheros de texto.

Estructura.

La estructura de un fichero de texto está formada por una secuencia de caracteres terminada por una marca de fin de fichero y jalonada con marcas que señalan los finales de línea y página. Al igual que en la entrada/salida por terminal, los componentes de esta secuencia pueden ser interpretados como literales de diversos tipos, no sólo *Character* o *String*.

La librería Text_IO.

Las operaciones básicas para manejar ficheros de texto se encuentran definidas en la librería estándar *Text_IO*, que habrá que incluir en la cláusula de contexto de un programa que pretenda usarlas, como se decribe para la entrada/salida por terminal , incluyendo la instanciación de los paquetes adecuados si se pretende escribir o leer de un fichero elementos que no sean de tipo *Character* o *String*.

Declaración.

Para poder usar ficheros de texto se deben declarar en el programa variables de tipo *File_Type*. Estas variables se conocen como ficheros lógicos, o simplemente ficheros, en oposición a los ficheros externos a los que hacen referencia. El programa manipula los ficheros externos a través de los ficheros lógicos.

```
Fichero1 : File_Type;
Fichero2 : Text_IO.File_Type;
```

En este caso, se declaran dos ficheros de texto. La forma usada para Fichero2 sólo es necesaria, con el fin de evitar confusiones, si en el mismo ámbito se están usando otros tipos de ficheros además de los de texto.

Creación, apertura y cierre.

Antes de poder hacer transacciones con una variable fichero debe asociarse con un fichero externo, esto se puede hacer con los procedimientos *Create* y *Open*. El primero sirve para crear un fichero y el segundo para abrir un fichero que ya existe. Ambas operaciones admiten hasta cuatro parámetros:

File es la variable fichero (fichero lógico).
Mode es el modo de apertura (puede tomar los valores: *In_File* , *Out_File* o *Append_File*).

In_File abre el fichero en modo lectura.
Out_File abre el fichero en modo escritura (si el fichero ya existe, se vacía). Es el modo por defecto en Create.
Append_File abre el fichero en modo escritura para añadir texto al final de un fichero existente.
Name es una ristra de caracteres (String) que representa el nombre del fichero externo.
Form generalmente no se usa, es un parámetro que depende del sistema y puede servir para cosas como proteger el fichero con una password.

```
Create(Fichero1, Name => S1);
Open(Fichero2, In_File, "c:\temp\ejemplo.txt");
```

La primera línea del ejemplo crea un fichero físico tomando como nombre el valor de la string S1 y lo abre en modo escritura asociándolo a la variable Fichero1; la segunda línea abre en modo lectura un fichero existente llamado "c:\temp\ejemplo.txt" y lo asocia a Fichero2.

Los errores que pudieran producirse al intentar abrir un fichero pueden controlarse mediante excepciones.

Si en Create se da como nombre del fichero físico la ristra vacía, se crea un fichero temporal que se destruye al acabar la ejecución del programa.

Las funciones *Is_Open* **(de tipo** *Boolean*)**,** *Mode* **(de tipo** *File_Mode*) **y** *Name* **(de tipo** *String*) **sirven para saber, respectivamente, si un fichero está abierto, en qué modo y con qué fichero externo está asociado; todas ellas requieren un parámetro de tipo** *File_Type*.

Una vez que se ha terminado de trabajar con un fichero se debe cerrar con la operación *Close* a la que se pasa como parámetro una variable de tipo *File_Type* .

```
Close(Fichero1);
```

Operaciones de entrada/salida.

A un fichero de texto abierto le son aplicables en toda su extensión las mismas operaciones que a la entrada/salida por terminal, con la salvedad de que se debe incluir como primer parámetro el fichero, la variable que representa el fichero lógico (tipo File_Type), al que hace referencia la operación; por ejemplo, para escribir el valor de la variable V en el fichero asociado con Fichero1, la instrucción es "Put(Fichero1,V)" en vez de "Put(V)".

Operaciones relacionadas con la estructura de los ficheros.

Existen operaciones para controlar las marcas del fichero en relación a:

- Líneas: *New_Line* , *Skip_Line* y *End_Of_Line*.

```
New_Line(Fichero1)   --Inicia una nueva línea en Fichero1
Skip_Line(Fichero1)  --Salta hasta después de la próxima marca
de fin de línea
End_Of_Line(Fichero1)--Es una función que devuelve True si se ha
alcanzado el fin
                     --de la línea, y False en caso contrario
```

- Páginas: *New_Page* , *Skip_Page* y *End_Of_Page*.

```
New_Page(Fichero1)   --Inicia una nueva página en Fichero1
Skip_Page(Fichero1)  --Salta hasta después de la próxima marca
de fin de página
End_Of_Page(Fichero1)--Es una función que devuelve True si se ha
alcanzado el final
                     --de la página y False en caso contrario
```

- Ficheros: *End_Of_File*.

```
End_Of_File(Fichero1) --es una función que devuelve True si se
ha alcanzado
                        -- el final de fichero1 y False en caso
contrario.
```

Existen operaciones para configurar el tamaño de la línea y el tamaño de la página.

Otras operaciones con ficheros.

La eliminación de un fichero se realiza mediante el procedimiento *Delete*, que borra el fichero externo asociado al parámetro *File_type* que se le pase. Para otras operaciones, consultar la librería Text_Io .

```
Delete(Fichero1);
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Excepciones en la entrada/salida.

La librería IO_Exceptions define (y las librerías Text_Io, Sequential_Io, Direct_Io y Stream_Io redefinen) las siguientes excepciones relacionadas con las operaciones de entrada/salida:

Excepción	**Ocurre cuando...**
Status_Error	se intenta acceder a un fichero que no está abierto, o abrir un fichero que ya está abierto.
Mode_Error	se intenta leer de un fichero que está abierto para escritura o escribir en un fichero que está abierto para lectura.
Name_Error	se intenta abrir o crear un fichero y el nombre externo es incorrecto (Ej.: porque el formato de la ristra no es válido como nombre de fichero o no existe un fichero físico cuando se intenta abrir).
Use_Error	se intenta abrir un fichero para un uso ilegal (Ej.: si se intenta crear un fichero con un nombre externo que ya existe y no está permitido sobreescribirlo o se intenta abrir un fichero para escritura sobre un dispositivo que es sólo de lectura).
Device_Error	se produce un fallo técnico en un dispositivo de entrada/salida.
End_Error	se intenta leer de un fichero en el que se ha alcanzado la marca de fin de fichero.
Data_Error	se intenta leer un valor entero, real o enumerado y los datos de entrada tienen un formato incorrecto.

La librería IO_Exceptions también define (y redefine la Text_Io) la excepción *Layout_Error* que indica cuando se producen errores en la estructura definida (filas, columnas, páginas) para un fichero de texto.

Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Excepciones.

Concepto.

En la ejecución de un programa pueden darse muchas situaciones inesperadas: dispositivos que fallan, datos que se entran incorrectamente, valores fuera del rango esperado, ... Incorporar todas las posibilidades de fallo y casos extraños en la lógica de un algoritmo puede entorpecer su legibilidad y su eficiencia; no tenerlas en cuenta produce que los programas aborten de forma incontrolada y puedan generarse daños en la información manejada. Los mecánismos de control de excepciones permiten contemplar este tipo de problemas separándolos de lo que es el funcionamiento "normal" del algoritmo; una *excepción* en Ada 95 es un objeto que identifica la ocurrencia de una situación anormal.

En el estándar de Ada están definidas las siguientes excepciones básicas:

Constraint_error	Ocurre cuando se intenta asignar a una variable un valor no válido o cuando se intenta acceder a una posición de un array fuera del rango permitido.
Program_error	Ocurre en situaciones extrañas cuando parte de un programa no es accesible o cuando se alcanza el "end" de una función sin encontrar un "return".
Storage_error	Ocurre cuando se agota la memoria disponible.
Tasking_error	Está relacionado con errores en programas que utilicen programación concurrente.

Además en otras librerías (Ej. las relacionadas con entrada/salida) se definen excepciones específicas.

Declaración.

El programador puede declarar sus propias excepciones para identificar problemas particulares que puedan surgir en sus programas. Las declaración está sujeta a las reglas generales de declaración y consta de: una lista de identificadores, dos puntos (":") y la palabra **exception**.

```
Error1, Error2: exception; --se declaran dos excepciones
```

Lanzamiento.

Al hecho de activar una excepción cuando ocurre la situación para la que está prevista se le conoce como "lanzar" la excepción; básicamente esto consiste en interrumpir la ejecución "normal" del programa e intentar localizar una sección de código capaz de hacerse cargo de la excepción y resolverla. Las excepciones predefinidas se lanzan automáticamente cuando ocurre la situación para la que se han

previsto; se pueden también lanzar las excepciones predefinidas o más probablemente las declaradas en el programa usando una sentencia **raise**.

```
raise Error1; --lanza la excepción Error1;
```

Manejo.

Un *manejador de excepciones* es un conjunto de instrucciones destinadas a ejecutarse en respuesta al lanzamiento de una excepción. Los manejadores de excepciones empiezan con una cláusula **when** seguida de la lista de excepciones a las que responde unidas por el operador "|" (or) y a continuación las instruciones a ejecutar.

```
when Error1 | Error2 =>
   --aquí se ponen las instrucciones a ejecutar
   --cuando ocurra la excepción Error1 o la Error2
   ...
```

Los manejadores se agrupan en una sección situada al final de cualquier bloque **begin**...**end** que contenga el algoritmo cuyas excepciones se quiere controlar; esta sección se abre con la palabra **exception** .

```
begin
   --Instrucciones "normales" en las que puede haber problemas
   ...
exception
   --manejadores de excepciones
   ...
end;
```

Durante la ejecución de un algoritmo, cuando se lanza una excepción el control es transferido a la sección de excepciones situada al final del bloque **begin**...**end** en el que se engloba y allí se busca un manejador que tenga en su lista la excepción lanzada y se ejecuta. Si no se encuentra ningún manejador que tenga a la excepción en su lista o si el bloque **begin**...**end** no tiene sección de tratamiento de excepciones la búsqueda del manejador continúa en el siguiente bloque activo más externo (aquel que inició la ejecución del actual). Mientras no se encuentre un manejador apropiado la búsqueda continúa hasta alcanzar el procedimiento principal; si en éste tampoco se encuentra un manejador acorde a la excepción lanzada, se aborta la ejecución del programa. Cuando un programa aborta se imprime en pantalla información acerca de qué excepción a ocurrido y de dónde se ha producido, por si pudiera resultar de alguna utilidad.

Ejemplo.

```
with Text_IO;
use Text_IO;
```

```
procedure Excepciones is
  package Entero_IO is new Integer_IO(Integer);
  package Real_IO is new Float_IO(Float);
  use Entero_IO, Real_IO;

  Divide_por_cero: exception; --declaración de una excepción
  --llamada División_por_cero
  procedure Leer(x,y: out integer) is
  --el procedimiento Leer ilustra como se controla una
  --excepción de forma que el programa se recupere y
  --continue operando
  begin
    --el bucle permite reintentar la lectura si falla
    loop
      --es necesario abrir un bloque para poder manejar
      --las excepciones dentro del bucle
      begin
        --el algoritmo que puede producir la excepción
        --son las siguientes tres líneas
        put("Deme dos números enteros: ");
        get(x);
        get(y);
        exit; --si se llega a este punto el algoritmo
              --se ha ejecutado sin problemas y se
              --abandona el bucle
      exception
        --Data_Error se produce si no se teclea un literal entero
        --valido. se captura y se hacen las operaciones
        --oportunas para poder reintentar la lectura
        when Data_error => --se captura la excepción
          --se informa al usuario de su error
          put_line("Por favor, teclee correctamente");
          skip_line; --se vacía el buffer de entrada
                     --que contiene un valor incorrecto
      end;
    end loop; --se deja que el bucle cicle
              --para reintentar la lectura
  end Leer;

  function Divide1(x,y: in integer) return float is
  --La función Divide1 ilustra como tras capturar una
  --excepción y hacer algunas operaciones, se
  --puede relanzar para que un bloque más externo
  --continue su manejo
    r: float;
  begin
```

```
    r := float(x) / float(y);
    return r;
  exception
    when Constraint_error =>
      put_line("El divisor es cero");
      raise; --se relanza Constraint_error
  end Divide1;

  function Divide2(x,y: in integer) return float is
  --La función Divide2 ilustra como se lanza
  --una excepción
    r: float;
  begin
    if y = 0 then
      raise Divide_por_cero;
    else
      r := float(x) / float(y);
      return r;
    end if;
  end Divide2;

  a, b: integer;
begin
  Leer(a,b);
  put("El cociente es: ");
  if a > b then
    put(Divide1(a,b));new_line;
  else
    put(Divide2(a,b));new_line;
  end if;
exception
  when Constraint_error|Divide_por_cero =>
    new_line;
    put_line("No se puede dividir por cero");
  when others =>
    new_line;
    put_Line(" Error desconocido");
end Excepciones;
```


Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Ficheros uniformes.

Estructura.

Un fichero uniforme o de componentes homogéneos es una secuencia arbitrariamente larga de elementos del mismo tipo. Por la forma en que se puede acceder a sus componentes se distinguen *ficheros secuenciales* y ficheros de *acceso directo*.

Librerías para manejar ficheros uniformes.

Los elementos necesarios para manejar ficheros de componentes homogéneos de forma secuencial se encuentran en el paquete genérico "Sequential_Io"; para manejarlos con acceso directo se emplea el paquete genérico "Direct_Io".

```
with Text_Io,Sequential_Io,Direct_Io;
```

Ambos paquetes son genéricos por lo que para poder usar ficheros de componentes uniformes hace falta especificar el tipo de los componentes; en consecuencia, no se pueden incluir en una cláusula *use*.

Declaración.

Para poder usar ficheros uniformes se ha de definir previamente el tipo de sus componentes (a menos que sea de un tipo predefinido). Supongamos que existe un tipo llamado TPersona. Una vez definido el tipo, se han de definir instancias de los paquetes de manejo de ficheros, según el tipo de acceso que se vaya a utilizar.

```
package FichTPersona_Sec is new Sequential_Io(TPersona); -- Acceso secuencial
package FichTPersona_Dir is new Direct_Io(TPersona);     -- Acceso directo
use FichTPersona_Sec, FichTPersona_Dir;
```

Ahora se pueden declarar variables de tipo fichero directo o secuencial:

```
Fichero1 : FichTPersona_Sec.File_Type;
Fichero2 : FichTPersona_Dir.File_Type;
```

Creación, apertura y cierre.

Antes de poder hacer transacciones con una variable fichero debe asociarse con un fichero externo, esto se puede hacer con los procedimientos *Create* y *Open*. El primero sirve para crear un fichero y el segundo para abrir un fichero que ya existe.

```
procedure Create(File : in out File_Type;
                 Mode : in File_Mode := Out_File; --Inout_File para ficheros de
acceso directo
                 Name : in String := "";
                 Form : in String := "");
```

```
procedure Open  (File : in out File_Type;
                 Mode : in File_Mode;
```

```
                Name : in String;
                Form : in String := "");
```

Ambas operaciones admiten hasta cuatro parámetros:

File es la variable fichero (fichero lógico).
Mode es el modo de apertura del fichero (puede tomar los valores: *In_File* , *Out_File* y *Append_File* o *Inout_File* , según sea, secuencial o directo).
In_File abre el fichero en modo lectura.
Out_File abre el fichero en modo escritura (si el fichero ya existe, se vacía). Es el modo por defecto en Create para ficheros secuenciales.
Append_File abre un fichero de acceso secuencial en modo escritura para añadir información al final de un fichero existente.
Inout_File abre un fichero de acceso directo y permite hacer lecturas y escrituras. Es el modo por defecto en Create para ficheros directos.
Name es una ristra de caracteres (String) que representa el nombre del fichero externo.
Form generalmente no se usa, es un parámetro que depende del sistema y puede servir para cosas como proteger el fichero con una palabra clave.

```
Create(Fichero1, Name => "c:\temp\unfichero.dat");
Open(Fichero2, In_File, "c:\temp\otrofichero.dat");
```

La primera línea del ejemplo crea un fichero físico tomando como nombre "c:\temp\unfichero.dat" y lo abre en modo escritura asociándolo a la variable Fichero1; la segunda línea abre en modo lectura un fichero existente llamado "c:\temp\otrofichero.dat" y lo asocia a Fichero2.

Los errores que pudieran producirse al intentar abrir un fichero pueden controlarse mediante excepciones.

Si en Create se da como nombre del fichero físico la ristra vacía, se crea un fichero temporal que se destruye al acabar la ejecución del programa.

Las funciones *Is_Open* **(***Boolean***),** *Mode* **(de tipo** *File_Mode***) y** *Name* **(de tipo** *String***) sirven para saber, respectivamente, si un fichero está abierto, en qué modo y con qué fichero externo está asociado; todas ellas requieren un parámetro de tipo** *File_Type***.**

Una vez que se ha terminado de trabajar con un fichero se debe cerrar con la operación *Close* a la que se pasa como parámetro una variable de tipo *File_Type*.

Operaciones de entrada/salida.

Para realizar lecturas desde ficheros y escrituras hacia ficheros se han de declarar previamente variables del tipo de los componentes del fichero.

```
Persona_1,Persona_2,Persona_3 : TPersona;
```

Las operaciones de entrada/salida se realizan con las operaciones *Read* y *Write*.

```
procedure Read(File : in out File_Type; Item : out Element_Type; From : in
Positive_Count);
procedure Read(File : in out File_Type; Item : out Element_Type);
```

Read lee un elemento de un fichero y avanza una posición:

- *File* es la variable fichero.
- *Item* es la variable, del tipo de los componentes del fichero, en la que se lee el elemento.
- *From* es la posición en el fichero desde donde se quiere leer el dato. Sólo es aplicable a ficheros de acceso directo. Si no se especifica se lee de la posición actual en el fichero.

```
procedure Write(File : in out File_Type; Item : in Element_Type; To : in
Positive_Count);
procedure Write(File : in out File_Type; Item : in Element_Type);
```

Write escribe un elemento en un fichero y avanza una posición:

- *File* es la variable fichero.
- *Item* es la variable, del tipo de los componentes del fichero, cuyo contenido se escribe en el fichero
- *To* es la posición en el fichero donde se quiere escribir el dato. Sólo es aplicable a ficheros de acceso directo. Si no se especifica se escribe en la posición actual en el fichero.

Operaciones relacionadas con la estructura de los ficheros.

Existe una operación para controlar la marca de fin de fichero:

```
End_Of_File(Fichero1) --es una función que devuelve True si se ha alcanzado
                      --el final de fichero1 y False en caso contrario.
```

El tamaño, en número de componentes, de un fichero de acceso directo se obtiene mediante la función "Size", que devuelve un número comprendido entre 0 y el máximo tamaño posible definido en la implementación del compilador de Ada que se esté usando.

```
function Size(File : in File_Type) return Count;
```

Se puede saltar a una posición específica de un fichero utilizando el procedimiento "Set_Index".

```
procedure Set_Index(File : in out File_Type; To : in Positive_Count);
```

La posición actual en el fichero se puede averiguar mediante la función "Index", que devuelve un valor comprendido entre 1 y el máximo tamaño posible definido en la implementación.

```
function Index(File : in File_Type) return Positive_Count;
```

Otras operaciones con ficheros.

La eliminación de un fichero se realiza mediante el procedimiento *Delete*, que borra el fichero externo asociado al parámetro *File_type* que se le pase.

```
Delete(fichero1);
```


Anterior | Superior | Siguiente

Guía de referencia básica de Ada 95

Ristras de caracteres.

Clasificación.

Ada ofrece tres clases de ristras de carácteres:

1. Ristras de tamaño fijo (fixed strings).
2. Ristras de tamaño limitado (bounded strings).
3. Ristras de tamaño dinámico (unbounded strings).

La libreria "Ada.Strings" define elementos de carácter general que se pueden usar con las distintas clases de ristras; entre otros:

```
Space : constant Character  := ' ';
type Alignment  is (Left, Right, Center);
type Truncation is (Left, Right, Error);
```

Cuando alguno de estos elementos vaya a ser usado directamente en un programa se tiene que incluir la librería "Ada.Strings" en la cláusula de contexto.

Ristras de tamaño fijo. [clasificación de ristras]

Declaración.

El paquete estándar de Ada contiene la definición del tipo *String* como un array no restringido de caracteres con rango positivo:

```
type String is array(Positive range <>) of Character;
```

Ello significa que, sin más, se pueden declarar ristras de caracteres como versiones restringidas del tipo String:

```
S1, S2: String(1..10);
S3, S4: String(1..20);
```

Las variables de tipo String son arrays y se les puede aplicar cualquier operación general de array, incluyendo la posibilidad de acceder a los elementos individuales mediante un índice (los elementos individuales son caracteres, no ristras):

```
C := S1(3); --C debe ser de tipo Character
S2(4) := 'a';
```

Asignación.

A una variable String se le puede asignar una ristra literal siempre que tenga el tamaño adecuado:

```
S1 := "ABCDEFGHIJ";
S3 := "012345678901234567890";
```

Para asignar ristras de tamaños diferentes es necesario incluir en la cláusula de contexto las librerias "Ada.Strings" y "Ada.Strings.Fixed"; la segunda ofrece un procedimiento llamado Move que permite esta operación:

```
procedure Move (Source  : in  String;
                Target  : out String;
                Drop    : in  Truncation := Error;
                Justify : in  Alignment  := Left;
                Pad     : in  Character  := Space );
```

El significado de los parámetros de Move es el siguiente: *Source* es la ristra a asignar, *Target* es la ristra donde se va a asignar, *Drop* indica qué hacer si *Source* tiene mayor longitud que Target (producir un error o cortar la ristra por uno de sus extremos, *left*, *rigth*, asignando sólo los caracteres que caben), *Justify* indica como alinear *Source* dentro de *Target* si tiene menor longitud y *Pad* es el carácter con que se rellenará el resto de *Target* en ese caso.

```
Move(S1,S4);                    --S4 = "ABCDEFGHIJ          "
Move(S1,S4,Pad =>'0');          --S4 = "ABCDEFGHIJ0000000000"
Move(S3,S2);                    --¡ERROR!
Move(S3,S2,Drop => Left);       --S2 = "0123456789"
Move(S1,S4,Justify => Rigth); --S4 = "          ABCDEFGHIJ"
```

Otras operaciones aplicables a las ristras de tamaño fijo.

Las ristras de tamaño fijo son las únicas a las que les son aplicables operaciones de entrada/salida , también se les pueden aplicar operadores relacionales ; la concatenación de ristras se hace usando el operador "&" que en Ada está definido para cualquier array monodimensional.

```
S1 := "ABCDE" & "12345"; --S1="ABCDE12345"
```

Para la *localización* de una subristra dentro de una ristra se puede usar la función Index definida en la librería "Ada.Strings.Fixed" con el siguietne protocolo:

```
function Index (Source  : in String;
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping
                             := Maps.Identity)
return Natural;
```

Donde *Source* es la ristra donde se va a buscar, *Pattern* es el patrón de búsqueda, *Going* es la dirección de búsqueda (el tipo *Direction* está definido en Ada.Strings con los valores Forward y Backward) y *Mapping* especifica como se van a corresponder los caracteres.

```
S4 := "012345678901234567890";
I := Index(S4, "345");           --I = 4
J := Index(S4, "345", Backward); --J = 14
```

La *extracción* de una subristra se puede hacer especificando un slice :

```
S2 := S4(3..12);     --S2 = "2345678901"
S2(1..5) := "ABCDE"; --S2 = "ABCDE78901"
```

Ristras de tamaño limitado. [clasificación de ristras]

Declaración.

Las ristras de tamaño limitado pueden tomar cualquier tamaño entre cero (ristra nula) y el número máximo de caracteres que se especifique en cada caso. La declaración de ristras de tamaño limitado requiere tres pasos:

Primero hay que incluir en la cáusula de contexto la librería genérica "Ada.Strings.Bounded".

```
with Ada.Strings.Bounded;
```

A continuación, dentro de la unidad de compilación, hay que instanciar la librería para los tamaños que se desee.

```
package Str10 is new
Ada.Strings.Bounded.Generic_Bounded_Length(10);
package Str20 is new
Ada.Strings.Bounded.Generic_Bounded_Length(20);
use Str10, Str20;
```

Sólo entonces se pueden declarar variables de tipo *Bounded_String* , cualificado con la instancia que corresponda al tamaño deseado.

```
SL1, SL2: Str10.Bounded_String;
SL3, SL4: Str20.Bounded_String;
```

Asignación y conversión.

Se pueden asignar entre si ristras de tamaño limitado del mismo tipo, pero no se puede asignar a una ristra de tamaño limitado una de tamaño fijo (ni al revés) ni una ristra literal (son de tamaño fijo), así como no se pueden emplear en operaciones de entrada/salida. Afortunadamente, la librería "Ada.Strings.Bounded" ofrece operaciones de conversión entre ristras de tamaño fijo y ristras de tamaño limitado.

```
function To_Bounded_String (Source : in String;
                            Drop   : in Truncation := Error)
return Bounded_String;

function To_String (Source : in Bounded_String) return String;
```

El parámetro *Drop* es necesario cuando se convierte una ristra de tamaño fijo a tamaño limitado porque la primera podría tener un tamaño mayor que el límite de la segunda y hay que especificar lo que se hace en ese caso; al revés no es necesario: la función crea una ristra de tamaño fijo exactamente igual a la longitud de *Source*.

```
get_line(S1,Lon);
SL1 := To_Bounded_String(S1(1..Lon));
put_line(To_String(SL1));
```

Recuerde que las funciones de conversión no cambian el tipo de la variable convertida (que es de entrada), sólo crean su valor equivalente en el nuevo tipo.

El valor *Null_Bounded_String* representa la ristra nula (equivale a *To_Bounded_String("")*).

Otras operaciones aplicables a las ristras de tamaño limitado.

Entre ristras de tamaño limitado son aplicables los operadores relacionales; la operación de concatenación se puede realizar tanto usando el operador "&" como la función *Append*:

```
function Append (Left, Right : in Bounded_String;
                 Drop        : in Truncation  := Error )
return Bounded_String;
```

La función *Append* devuelve la concatenación de *Left* y *Right*; el parámetro *Drop* indica que hacer en caso de que el resultado tenga una longitud mayor que el límite establecido.

```
SL1 := To_Bounded_String("Alfa"); --SL1 = "Alfa"
SL2 := To_Bounded_String("beto"); --SL2 = "beto"
SL1 := Append(SL1,SL2);           --SL1 = "Alfabeto"
SL2 := Append(SL1,SL2);           --¡ERROR!
SL2 := Append(SL1,SL2,Rigth);     --SL2 = "Alfabetobe"
```

Uno de los parámetros de *Append* **puede no ser de tipo** *Unbounded_String* **(puede ser una ristra de tamaño fijo, dinámico o un cáracter).**

La *longitud* actual de una ristra de tamaño limitado se obtiene con la función *Length*.

```
X := Length(SL1); --X = 8
```

La *localización* de una subristra requiere una función *Index* similar a las de las ristras de tamaño fijo.

```
function Index (Source  : in Bounded_String;
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping
                             := Maps.Identity)
return Natural;
```

Para la operación de extracción la sintaxis de slices (que ya no se puede utilizar puesto que las ristras de tamaño limitado no son arrays) es sustituida por una función *Slice* .

```
function Slice (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
return String;
```

La función *Slice* devuelve la subristra de *Source* que empieza en la posición *Low* y acaba en la posición *High* (ambas incluidas); el resultado es de tamaño fijo.

```
SL3 := To_Bounded_String(Slice(SL1,3,6)); -- SL3 = "fabe"
```

Ristras de tamaño dinámico. [clasificación de ristras]

Declaración.

Las ristras de tamaño dinámico pueden tomar cualquier tamaño sin necesidad de una reserva previa. La declaración de ristras de tamaño dinámico requiere, en primer lugar, incluir en la cáusula de contexto la librería "Ada.Strings.Unbounded".

```
with Ada.Strings.Unbounded;
```

```
use Ada.Strings.Unbounded;  --se puede incluir en el use porque no es genérica
```

Entonces se pueden declarar variables de tipo *Unbounded_String*.

```
SD1, SD2, SD3, SD4: Unbounded_String;
```

Asignación y conversión.

Se pueden asignar entre si ristras de tamaño dinámico, pero no se puede asignar a una ristra de tamaño dinámico una de tamaño fijo o limitado (ni al revés) ni una ristra literal (son de tamaño fijo), así como no se pueden emplear en operaciones de entrada/salida. Afortunadamente, la librería "Ada.Strings.Unbounded" ofrece operaciones de conversión entre ristras de tamaño fijo y ristras de tamaño dinámico.

```
function To_Unbounded_String (Source : in String)
return Unbounded_String;

function To_String (Source : in Unbounded_String) return String;
```

La función *To_Unbounded_String* crea una ristra de tamaño dinámico a partir de una de tamaño fijo y la operación *To_String* hace lo contrario.

```
get_line(S1,Lon);
SD1 := To_UnBounded_String(S1(1..Lon));
put_line(To_String(SD1));
```

Recuerde que las funciones de conversión no cambian el tipo de la variable convertida (que es de entrada), sólo crean su valor equivalente en el nuevo tipo.

El valor *Null_Unbounded_String* representa la ristra nula (equivale a *To_Unbounded_String("")*).

Otras operaciones aplicables a las ristras de tamaño dinámico.

Entre ristras de tamaño dinámico son aplicables los operadores relacionales; la operación de concatenación se puede realizar tanto usando el operador "&" como el procedimiento *Append* .

```
procedure Append (Source   : in out Unbounded_String;
                 New_Item : in Unbounded_String);
```

El procedimiento *Append* crea una ristra concatendo *Source* con *New_item* y pone el resultado en *Source* (que queda modificada).

```
SD1 := To_Unbounded_String("Alfa"); --SD1 = "Alfa"
```

```
SD2 := To_Unbounded_String("beto"); --SD2 = "beto"
Append(SD1,SD2);                    --SD1 = "Alfabeto"
```

El segundo parámetro de *Append* **puede no ser de tipo** *Unbounded_String* **(puede ser una ristra de tamaño fijo, dinámico o un cáracter).**

La *longitud* actual de una ristra de tamaño dinámico se obtiene con la función *Length*.

```
X := Length(SD1); --X = 8
```

La *localización* de una subristra requiere una función *Index* similar a las de las ristras de tamaño fijo.

```
function Index(Source  : in Unbounded_String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping
                            := Maps.Identity)
return Natural;
```

Para la operación de extracción la sintáxis de slices (que ya no se puede utilizar puesto que las ristras de tamaño limitado no son arrays) es sustituida por una función *Slice*.

```
function Slice (Source : in UnBounded_String;
                Low    : in Positive;
                High   : in Natural)
return String;
```

La función *Slice* devuelve la subristra de *Source* que empieza en la posición *Low* y acaba en la posición *High* (ambas incluidas); el resultado es de tamaño fijo.

```
SD3 := To_Unbounded_String(Slice(SD1,3,6)); -- SD3 = "fabe"
```


Anterior | Superior | Siguiente

Anterior | Superior | Siguiente

Funciones numéricas elementales

Ada.Numerics.Generic_Elementary_Functions ofrece funciones matemáticas.

Contenido.

funciones:

```
 function Sqrt    (X           : Float_Type'Base)        return
Float_Type'Base;
 function Log     (X           : Float_Type'Base)        return
Float_Type'Base;
 function Log     (X, Base     : Float_Type'Base)        return
Float_Type'Base;
 function Exp     (X           : Float_Type'Base)        return
Float_Type'Base;
 function "**"    (Left, Right : Float_Type'Base)        return
Float_Type'Base;

 function Sin     (X           : Float_Type'Base)        return
Float_Type'Base;
 function Sin     (X, Cycle    : Float_Type'Base)        return
Float_Type'Base;
 function Cos     (X           : Float_Type'Base)        return
Float_Type'Base;
 function Cos     (X, Cycle    : Float_Type'Base)        return
Float_Type'Base;
 function Tan     (X           : Float_Type'Base)        return
Float_Type'Base;
 function Tan     (X, Cycle    : Float_Type'Base)        return
Float_Type'Base;
 function Cot     (X           : Float_Type'Base)        return
Float_Type'Base;
 function Cot     (X, Cycle    : Float_Type'Base)        return
Float_Type'Base;

 function Arcsin  (X           : Float_Type'Base)        return
Float_Type'Base;
 function Arcsin  (X, Cycle    : Float_Type'Base)        return
Float_Type'Base;
 function Arccos  (X           : Float_Type'Base)        return
```

```
Float_Type'Base;
 function Arccos  (X, Cycle    : Float_Type'Base)       return
Float_Type'Base;
 function Arctan  (Y           : Float_Type'Base;
                   X           : Float_Type'Base := 1.0) return
Float_Type'Base;
 function Arctan  (Y           : Float_Type'Base;
                   X           : Float_Type'Base := 1.0;
                   Cycle       : Float_Type'Base)       return
Float_Type'Base;
 function Arccot  (X           : Float_Type'Base;
                   Y           : Float_Type'Base := 1.0) return
Float_Type'Base;
 function Arccot  (X           : Float_Type'Base;
                   Y           : Float_Type'Base := 1.0;
                   Cycle       : Float_Type'Base)       return
Float_Type'Base;

 function Sinh    (X           : Float_Type'Base)       return
Float_Type'Base;
 function Cosh    (X           : Float_Type'Base)       return
Float_Type'Base;
 function Tanh    (X           : Float_Type'Base)       return
Float_Type'Base;
 function Coth    (X           : Float_Type'Base)       return
Float_Type'Base;
 function Arcsinh (X           : Float_Type'Base)       return
Float_Type'Base;
 function Arccosh (X           : Float_Type'Base)       return
Float_Type'Base;
 function Arctanh (X           : Float_Type'Base)       return
Float_Type'Base;
 function Arccoth (X           : Float_Type'Base)       return
Float_Type'Base;
```

(*cycle* es las unidades que determinan una circunferencia completa. Si no se especifica se suponen radianes

Significado

function Sqrt	raíz cuadrada	function Sinh	seno hiperbolico
function Log	logaritmo	function Cosh	coseno hiperbolico
function Exp	Exponencial	function Tanh	tangente hiperbolica
function "**"	potencia	function Coth	cotangente hiperbolica
function Sin	seno	function Arcsinh	arcoseno hiperbolico
function Sin	coseno	function Arccosh	arcocoseno hiperbolico
function Tan	tangente	function Arctanh	arcotangente hiperbolica
function Cot	cotangente	function Arccoth	arcocotangente hiperbolica
function Arcsin	arcoseno		
function Arccos	arcocoseno		
function Arctan	arcotangente		
function Arccot	arcocotangente		

Forma de usarlas.

1) Cláusulas de contexto.

- Incluir en una cláusula with: "Ada.Numerics.Generic_Elementary_Functions"
- Incluir en una cláusula use: "Ada.Numerics"

2) Declaraciones.

- Instanciar el paquete "Generic_Elementary_Functions" con el tipo real adecuado e incluir la instancia en una cláusula use.

Ejemplo.

```
with Text_IO,
Ada.Numerics.Generic_Elementary_Functions;
use TExt_IO, Ada.Numerics;

procedure UsaMat is
    x, y : float;
    package MiFloat_IO is new Float_IO(float);
```

```
    package Funciones is new
Generic_Elementary_Functions(float);
    use MiFloat_IO, Funciones;
begin
    put("Dame un número real: ");
    get(x);
    y := sqrt(x);
    put("raíz cuadrada: ");
    put(y);
end UsaMat;
```


Anterior | Superior | Siguiente