Introducción al lenguaje de programación Ada

Informática II

Departamento de Teoría de la Señal y Comunicación y Sistemas Telemáticos y Computación
(GSyC)

Universidad Rey Juan Carlos

Septiembre 2019

©2019 Grupo de Sistemas y Comunicaciones.
Algunos derechos reservados.
Este trabajo se distribuye bajo la licencia
Creative Commons Attribution Share-Alike
disponible en http://creativecommons.org/licenses/by-sa/3.0/es

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Lady Ada Lovelace

- El nombre del lenguaje hace referencia a Ada Augusta Byron, condesa de Lovelace (1815–1852)
- Hija del poeta romántico inglés Lord Byron.
- Matemática, ayudante del inventor Charles Babbage.
- Considerada la primera programadora de la historia, al inventar el primer *pseudo-lenguaje de programación* para la máquina analítica.

Historia

- 1983: Primera versión del lenguaje Ada: Ada 83
- 1995: Revisión completa del lenguaje: Ada 95
- 2005: Nueva revisión (pequeña): Ada 2005
- 2012: Nueva revisión (muy pequeña): Ada 2012

¿Quién utiliza Ada hoy?

- Industrias aeronáutica y espacial: Eurofighter, Boeing, Airbus, Ariane...
- Industria ferroviaria: AVE, metro de Nueva York y París...
- Otros: BNP, BMW...

¿Por qué Ada?

Costes de desarrollo de software:

- Diseño: 15 %
- Codificación: 20 %
- Depuración: 65 %

El problema del sofware son las erratas en el código. El lenguaje Ada está diseñado para que el programador cometa las menores erratas involuntarias posibles.

Ada es un lenguaje:

- estrictamente tipado
- estáticamente tipado

Lo que hace que la mayoría de los errores puedan detectarse en tiempo de compilación, antes de empezar a ejecutar del programa.

Aunque Ada es un lenguaje muy adecuado para aprender a programar, tiene muchas características avanzadas que permiten utilizarlo para:

- Programación concurrente
- Programación orientada a objetos
- Programación de sistemas (a bajo nivel)
- Programación de sistemas de tiempo real
- Programación de sistemas distribuidos

Contenidos

1 Introducción
2 El primer programa
3 Tipos
4 Punteros
5 Estructuras de control
6 Subprogramas
7 Paquetes
8 Ficheros de texto
9 Excepciones
10 Ejemplo Final

Código en Ada

```
with Ada.Text_IO;

-- Programa que muestras las tablas de multiplicar
procedure Tablas_De_Multiplicar is
   Resultado: Integer;

begin

   Ada.Text_IO.Put_Line("Tablas de Multiplicar");
   Ada.Text_IO.Put_Line("=====================");
   Ada.Text_IO.New_Line(2);

   for Fila in 1..10 loop
      for Columna in 1..10 loop
         Resultado := Fila * Columna;
         Ada.Text_IO.Put (Integer'Image(Fila) & "*" &
                          Integer'Image(Columna) & "=" &
                          Integer'Image(Resultado));
         Ada.Text_IO.New_Line;
      end loop;
   end loop;

end Tablas_De_Multiplicar;
```

Mayúsculas y minúsculas

- Ada no distingue entre mayúsculas y minúsculas.
- El programador, sin embargo, debe intentar usarlas de forma consistente.
- Convenios habituales en Ada:
  - **Palabras reservadas:** todo en minúsculas

```
procedure   with   begin   end   for   loop
```

  - **Identificadores:** Palabras separadas por el carácter (_), cada palabra empieza con mayúscula y sigue en minúsculas

```
Tablas_De_Multiplicar
Num_Filas
Fila
Columna
```

Comentarios

- Los comentarios en Ada empiezan por dos guiones -- y duran hasta el final de la línea
- Si se quiere un comentario que abarque varias líneas es necesario empezar cada línea por --:

```
-- este es un comentario
-- de varias lineas
-- juntas
X := X + 1;
```

- Es conveniente poner un comentario delante de:
  - todos los subprogramas
  - declaraciones de variables cuya utilidad no se deduzca de su nombre
  - trozos de código difíciles de entender
  - trozos de código sobre los que se duda de su corrección
  - trozos de código que son provisionales y tendrán necesariamente que modificarse o eliminarse
- Los comentarios se ponen según se va escribiendo el código, no a posteriori.

Legibilidad del código

- Un programa se escribe una vez y se lee muchas, por lo que es imprescindible que sea lo más legible posible.
- Ayuda mucho a la legibilidad:
  - nombres sensatos de los identificadores (no importa que sean largos si es necesario para entender qué representan)
  - comentarios en los sitios adecuados
  - un sangrado correcto (recomendado: 3 caracteres de ancho de tabulación)
  - separar con líneas en blanco partes del código que hacen cosas diferentes
  - separar los operadores con un espacio por delante y por detrás
- Cuando un programador escribe un trozo de código no debe hacerlo de forma que muestre «lo listo e ingenioso que es», sino preguntándose constantemente si quien lo lea lo podrá entender (quizás sólo lo lea él, pero mucho tiempo después).

Uso de paquetes

- Para poder utilizar los elementos (subprogramas, tipos y variables) definidos en un paquete es necesario poner al principio la cláusula `with`.
- Al utilizar los elementos de un paquete del que se ha hecho `with` es necesario cualificar el nombre del elemento con el nombre del paquete:

```
with Ada.Text_IO;
...
Ada.Text_IO.Put_Line("Hola");
```

- Si se utiliza la cláusula `use` ya no hay que cualificar:

```
with Ada.Text_IO;
use Ada.Text_IO;
...
Put_Line("Hola");
```

No debe emplearse la cláusula use

- Si un programa utiliza muchos paquetes, y para todos se emplea la cláusula use, se hace difícil saber a simple vista de qué paquete es cada subprograma que se invoca.
- Por eso está fuertemente desaconsejado utilizar use.
- En su lugar puede utilizarse package ... renames para acortar nombres de paquetes que se van a utilizar mucho:

```
with Ada.Text_IO;

procedure Tablas_De_Multiplicar is
   package T_IO renames Ada.Text_IO;
...
begin
   T_IO.Put_Line("Hola");
   ...
end Tablas_De_Multiplicar;
```

Entrada/Salida de texto

- El paquete Ada.Text_IO se utiliza para entrada/salida de cadenas de caracteres (Strings).
- Hay paquetes que permiten entrada/salida de otros tipos de datos: Integer, Float, Boolean...
- En general resulta más cómodo usar siempre Ada.Text_IO y convertir otros tipos de datos a String mediante 'Image.
- También resulta cómodo usar el operador de concatenación de Strings: &.
- Ejemplo:

```
Resultado: Integer;
...
Ada.Text_IO.Put_Line("Resultado: " & Integer'Image(Resultado));
```

- 'Image es un atributo. En Ada los tipos y las variables tienen algunos atributos predefinidos. Su sintaxis es especial pero funcionan como si fueran funciones o procedimientos.
- 'Image aplicado a un tipo devuelve su representación en forma de String.

Compilador de Ada: GNAT

- GNAT es el compilador de referencia en el mundo Ada.
- GNAT se empezó a desarrollar en la NYU como proyecto de SW libre como parte de proceso de estandarización de Ada 95.
- AdaCore es una empresa fundada por los desarrolladores originales de GNAT que ofrece:
  - una versión comercial del compilador: GNAT Pro
  - una versión libre del compilador, que modifica anualmente: GNAT GPL
  - múltiples herramientas comerciales y libres relacionadas con el desarrollo de SW profesional en el mundo Ada
- GNAT GPL es fácilmente instalable en cualquier sistema operativo: GNU/Linux, MacOS, Windows. Nosotros utilizaremos la versión de GNU/Linux.
- En las versiones actuales de las distribuciones Linux basadas en Ubuntu, Mint o Debian para instalar GNAT hay que ejecutar en una ventana de terminal:

```
sudo apt-get install gnat
```

Compilación, enlazado y ejecución

- Un programa en Ada es un procedimiento principal, que puede invocar otros procedimientos y funciones anidados, y procedimientos y funciones de paquetes.
- El nombre del fichero que contiene el procedimiento principal debe ser igual que el nombre del procedimiento, pero todo en minúsculas y terminado en `.adb`: `tabla_de_multiplicar.adb`
- Compilar un fichero fuente es traducirlo a código máquina obteniendo un fichero objeto.
- Enlazar un programa es unir todos los ficheros objeto que forman parte del programa (incluyendo los ficheros objeto de los paquetes que se usan), obteniendo un fichero binario ejecutable.
- En Ada, con GNAT, la compilación y el enlazado de todos los ficheros que forman parte de un programa se realiza en un solo paso, invocando la orden `gnatmake` sobre el fichero del procedimiento principal:

```
gnatmake tabla_de_multiplicar.adb
```

- Por defecto, las versiones actuales de GNAT compilan según Ada 2012, se puede compilar para Ada 2005, Ada 95 o incluso Ada 93 añadiendo a `gnatmake` la opción adecuada: `-gnat2005`, `-gnat95`, `-gnat83`.
- El fichero ejecutable resultante tiene en Linux el mismo nombre que el programa principal, pero sin extensión: `tabla_de_multiplicar`
- Para ejecutar un fichero ejecutable en una ventana de terminal de Linux, hay que escribir:

```
./tabla_de_multiplicar
```

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Tipos Básicos (I)

```
procedure Tipos_Básicos is
   I: Integer := 10;
   J, K: Integer;
   M: Integer :=  2 * I;
   F: Float;
   Pi: constant Float := 3.14159_26536;
begin
   J := 2 * J;            -- correcto, pero de resultado indeterminado
                          -- porque J no está inicializado
                          -- dará un Aviso (Warning) al compilar
   F := 2 * I;            -- error de compilación: tipos incompatibles
   F := 2.0 * Float(I);   -- Correcto
end Tipos_Básicos;
```

- Si una variable está sin inicializar tendrá un valor cualquiera, que puede ser diferente cada vez que se ejecute el programa.
- Conviene inicializar todas las variables. El compilador no siempre puede detectar cuándo se usan variables sin inicializar y los problemas que pueden producirse son muy difíciles de detectar.
- No se pueden mezclar tipos distintos si no se hace una conversión.
- Los literales numéricos tienen tipo: `2` es un `Integer`, `2.0` es un `Float`.

Tipos Básicos (II)

```
procedure Tipos_Básicos_2 is

   N: Natural;
   P: Positive;
   Flag: Boolean;

begin

   N := 4;
   Flag := True;

end Tipos_Básicos_2;
```

- El tipo `Natural` incluye a los enteros positivos y al 0.
- El tipo `Positive` incluye a los enteros positivos sin el 0.
- El tipo `Boolean` solo incluye dos valores: `True` y `False`.

Tipos y Subtipos

```
procedure Tipos_Subtipos is

   subtype Día_Mes is Integer range 1..31;
   type Nuevo_Día_Mes is new Integer range 1..31;
   I: Integer;
   D: Día_Mes;
   ND: Nuevo_Día_Mes;

begin
   I := 12;
   D := I;                     -- Correcto
   D := I * 3;                 -- Correcto pero elevará una excepción en
                               -- tiempo de ejecución: Constraint_Error
   ND := I;                    -- Error de compilación: tipos incompatibles
   ND := Nuevo_Día_Mes(I);     -- Correcto: la conversión EXPLÍCITA entre
                               -- tipos numéricos incompatibles
                               -- es siempre posible
end Tipos_Subtipos;
```

- Un tipo y sus subtipos son compatibles, por lo que pueden mezclarse en expresiones. Dos tipos diferentes siempre son incompatibles.
- `Natural` y `Positive` no son tipos propiamente, sino subtipos de `Integer`
- Al mezclar tipos y subtipos puede ocurrir que un resultado se salga del rango de valores permitidos para un subtipo, lo que provoca un error en tiempo de ejecución: `Constraint_Error`
- Siempre es posible realizar una conversión explícita entre tipos numéricos.

Enumerados

```
with Ada.Text_IO;

procedure Enumerados is

   type Colores is (Rojo, Amarillo, Verde);
   C: Colores;

begin

   C := "Rojo";      -- No compila, "Rojo" es un String
   C := Rojo;        -- Correcto

   C := C + 1;       -- No compila: los enumerados no son enteros

   Ada.Text_IO.Put_Line(Colores'Image(C)); -- Muestra: ROJO

end Enumerados;
```

- No hay que confundir los literales de tipos enumerados con `Strings`.
- Los tipos enumerados también tienen atributo `'Image`.
- El tipo predefinido `Boolean` es un enumerado que contiene los valores `True` y `False`.

Arrays (I)

```
procedure Arrays is

   A: array (1..6) of Integer;               -- A es de un tipo anónimo
   B: array (1..6) of Integer;               -- B es de un tipo anónimo
   AA: array (0..2, 0..6) of Integer;        -- A es de un tipo anónimo

   type Vector is array (1..6) of Integer;
   C: Vector;                         -- C es de un tipo con nombre: Vector
   D: Vector;                         -- D es de un tipo con nombre: Vector

begin
   A := (3, 4, 5, 2, -1, 3);
   A(1) :=  4;
   AA(0,3) := 5;

   B := A;                       -- no compila: tipos incompatibles
   C := A;                       -- no compila: tipos incompatibles

   C := (2, 3, 4, 5, 6, 7);
   D := C;                       -- correcto

end Arrays;
```

- Una variable array puede declararse de un tipo anónimo o de un tipo con nombre.
- Todas las variables de tipos anónimos son incompatibles entre sí.
- Variables array del mismo tipo con nombre sí son compatibles.

Arrays (II)

```
procedure Arrays is

   A: array (1..6) of Integer;
   B: array (0..5) of Integer;
   C: array (-1..1) of Integer;
   D: array (1..100000) of Float;

begin
   A := (3, 4, 5, 2, -1, 3);                           -- Agregado posicional

   A := (1=>5, 2=>6, 6=>10, 5=>9, 3=>8, 4=>7); -- Agregado por nombre
   A := (3=>5, others=>0);

   D := (others => 0.0);

end Arrays;
```

- En Ada es costumbre que la primera posición de un array tenga índice 1, pero puede ser cualquiera.
- Al asignar valores a las posiciones de un array pueden utilizarse agregados posicionales o agregados por nombre.
- En los agregados por nombre cada valor es precedido por el índice en que se coloca.
- Los agregados por nombre sólo se usan cuando se quiere usar `others` para dar el mismo valor a muchas o todas las posiciones.
- Otra forma de dar valores a todo un array es a través de bucles.

Arrays (III): Arrays Irrestringidos

```
procedure Arrays_Irrestringidos is

   A: array (Integer range <>) of Integer; -- no compila: tamaño desconocido

   type Vector_I is array (Integer range <>) of Integer; -- correcto

   B: Vector_I;                    -- no compila: B tiene tamaño desconocido
   C: Vector_I (1..6);             -- correcto: se restringe el tamaño al declarar
   D: Vector_I := (3, 5, 6);       -- correcto: toma tamaño del valor inicial

begin

   D := (3, 5, 6, 7); -- no compila: tipos incompatibles pues D ya
                      -- siempre será un array de 3 posiciones
   C(1..3) := D;      -- correcto, se trabaja sólo sobre
                      -- una *rodaja* de C, no sobre el array completo.

end Arrays_Irrestrigidos;
```

- En Ada NO hay arrays dinámicos (es decir, que varíen de tamaño durante la ejecución).
- En Ada al declarar cualquier variable debe conocerse su tamaño, que ya no cambiará durante la ejecución del programa.
- Al hacer asignaciones entre arrays se necesita que, además de ser del mismo tipo, los tamaños de los arrays sean iguales.
- Una rodaja (o slice) de un array es otro array: un trozo del array original.

Arrays (IV): Arrays Irrestringidos como parámetros

```
...
type Vector_I is array (Integer range <>) of Integer;
...
procedure Suma (A: Vector_I; B:Vector_I) is
begin
   ...
   A(7) := B(9) + 1;
   ...
end Suma;
...
C, D: Vector_I(1..10);
E, F: Vector_I(1..6);
...
Suma(C, D); -- correcto
Suma(E, F); -- compila, pero provocará un Constraint_Error
...         -- al ejecutarse
```

- Los arrays irrestringidos son útiles para ser utilizados como tipo de un parámetro formal en la declaración de un subprograma.
- En la llamada al subprograma podrá pasarse un array de cualquier tamaño.
- Si dentro del código del subprograma se intenta acceder a una posición que no existe en el array que ha recibido como parámetro se producirá un `Constraint_Error`.

Strings

```
procedure Cadenas is

   C  : Character := 'A';
   S1 : String;           -- no compila: tamaño desconocido
   S2 : String := "hola"; -- correcto: S2 toma tamaño del valor inicial
   S3 : String(1..10);    -- correcto: S3 siempre tendrá tamaño 10

begin

   S2 := "buenas";        -- no compila: tamaños distintos
   S3 := "hola";          -- no compila: tamaños distintos
   S3 (1..4) := "hola";   -- correcto, usando una rodaja de S3

end Cadenas;
```

- El tipo `String` es en realidad un array irrestrigido de `Character`s, por lo que sufre todas sus limitaciones.
- Una variable `String` tiene que tener fijado su tamaño al declarse, y dicho tamaño ya no puede cambiarse durante la ejecución (¡no puede ni siquiera disminuir!).
- Declarar variables tipo `String` es, en general, una mala idea.

Strings Ilimitados (I)

- En Ada no resulta práctico utilizar el tipo predefinido `String` para casi nada.
- Es más conveniente usar el tipo `Unbounded_String`, aunque no es un tipo predefinido. Está definido en el paquete `Ada.Strings.Unbounded`.
- Los `Unbounded_String` no tienen tamaño predefinido, y pueden cambiar su tamaño dinámicamente sin problemas.
- El único inconveniente para usarlos es el tener que convertir hacia/desde el tipo `String`, mediante las funciones:
  - `To_String`: Pasa un `Unbounded_String` a `String`.
  - `To_Unbounded_String`: Pasa un `String` a `Unbounded_String`.

Strings Ilimitados (II)

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

procedure Strings_Ilimitados is

   package ASU renames Ada.Strings.Unbounded;

   US: ASU.Unbounded_String;

begin

   US := ASU.To_Unbounded_String ("hola");
   US := ASU.To_Unbounded_String ("cómo estamos"); -- correcto
   US := ASU.To_Unbounded_String (ASU.To_String(US) & " todos");
   Ada.Text_IO.Put_Line (ASU.To_String(US));

end Strings_Ilimitados;
```

- El programa muestra en pantalla: `cómo estamos todos`
- En este ejemplo para concatenar dos cadenas hemos convertido todo a `Strings` sin variables intermedias, concatenado con `&`, y hemos vuelto a convertir el resultado a `Unbounded_String`.
- `Ada.Text_IO.Put_Line` escribe `Strings`, pero no `Unbouded_Strings` si no se convierten con `To_String`.
- También puede usarse `Ada.Strings.Unbounded.Text_IO.Put_Line`, que escribe `Unbounded_Strings` directamente.

Strings Ilimitados (III)

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

procedure Strings_Ilimitados_2 is

   package ASU renames Ada.Strings.Unbounded;
   use type ASU.Unbounded_String; -- para tener visibilidad de los operadores del tipo

   US: ASU.Unbounded_String;

begin

   Ada.Text_IO.Put("Di algo: ");
   US := ASU.To_Unbounded_String(Ada.Text_IO.Get_Line);

   if US = "gracias" then
      Ada.text_IO.Put_Line("gracias a ti");
   else
      US := "Has dicho: " & US;
      Ada.Text_IO.Put_Line(ASU.To_String(US));
   end if;

end Strings_Ilimitados_2;
```

- La sentencia `use type` permite tener visibilidad de los operadores `&`, `=`, `<`, `<=`, `>`, `>=` que define el paquete `Ada.Strings.Unbounded` para dos operandos `Unbounded_String` o para un `Unbounded_String` y un `String`.
- `Ada.Text_IO.Get_Line` lee una línea que escriba el usuario y la devuelve como `String`. Hay que convertirla con `To_Unbounded_String` para poder almacenarla en una variable.

Registros

```
procedure Registros is

   type Persona is record
      Nombre: ASU.Unbounded_String := ASU.To_Unbouded_String("");
      Edad: Integer := 20;
   end record;

   P: Persona;
   Q: Persona;

begin

   P.Edad := 15;
   P := (ASU.To_Unbounded_String("Luis"), 25);
   P := (Nombre => ASU.To_Unbounded_String("Pepe"), Edad => 10);

end Registros;
```

- Los campos de un registro pueden declarse con un valor por defecto: `Q.Edad` tiene valor 20.
- También pueden inicializarse los campos de un registro con agregados por nombre.

Arrays de Registros

```
procedure Array_De_Registros is

   type Persona is record
      Nombre: ASU.Unbounded_String :=  ASU.To_Unbounded_String("");
      Edad: Integer := 20;
   end record;

   type Grupo is array (1..10) of Persona;

   L: Grupo;

begin

   L(1).Edad := 15;
   L(1) := (ASU.To_Unbounded_String("Luis"), 25);

end Array_De_Registros;
```

- Un array de registros no es más que un array en el que en cada posición hay un registro.

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Punteros

- Los punteros son un mecanismo presente en la mayoría de los lenguajes de programación para gestionar espacio en memoria de una forma indirecta.
- El uso de punteros es una herramienta potente pero peligrosa: cuando los punteros se usan incorrectamente se pueden producir erratas muy difíciles de detectar.
- Algunos lenguajes de programación llaman a los punteros con otros nombres. Así en Ada a los punteros se les llama accesos.
- Los punteros tienen dos usos principales:
  - Acceder a variables normales de forma indirecta (a través de un "alias").
  - Gestionar memoria dinámica (memoria no asociada directa y estáticamente a una variable).

Punteros como alias de otras variables (1)

Dirección	Valor
1203	
1204	
1205	
1206	
1207	
1208	
1209	
1210	
1211	
1212	
1213	
1214	
1215	
1216	
1217	
1218	
1219	
1220	
1221	
1222	
1223	

```
procedure Punteros_1 is
   …
   I: Integer;
   J: Integer;
   S: String(1..5);


   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Al declarar una variable se le reserva espacio en memoria y se etiqueta dicho espacio con su nombre.

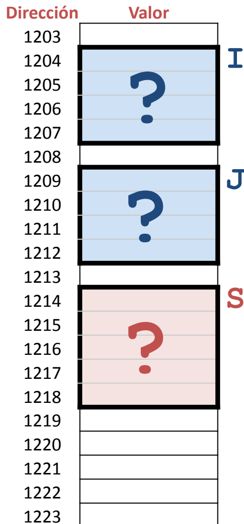
Punteros como alias de otras variables (2)

Dirección	Valor	
1203		
1204	?	I
1205		
1206		
1207		
1208		
1209	?	J
1210		
1211		
1212		
1213		
1214	?	S
1215		
1216		
1217		
1218		
1219		
1220		
1221		
1222		
1223		

```
procedure Punteros_1 is
    …
    I: Integer;
    J: Integer;
    S: String(1..5);


    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Si las variables están sin inicializar, tendrán un valor indeterminado.

Punteros como alias de otras variables (3)




```
procedure Punteros_1 is
   …
   I: Integer;
   J: Integer;
   S: String(1..5);


   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Al inicializar una variable. . .

Punteros como alias de otras variables (4)

Dirección	Valor	
1203		
1204	5	I
1205		
1206		
1207		
1208		
1209	-4	J
1210		
1211		
1212		
1213		
1214	t	S
1215	o	
1216	d	
1217	o	
1218	s	
1219		
1220		
1221		
1222		
1223		

```
procedure Punteros_1 is
   …
   I: Integer;
   J: Integer;
   S: String(1..5);


   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Al inicializar una variable se coloca su valor en la posición de memoria asociada a su nombre.

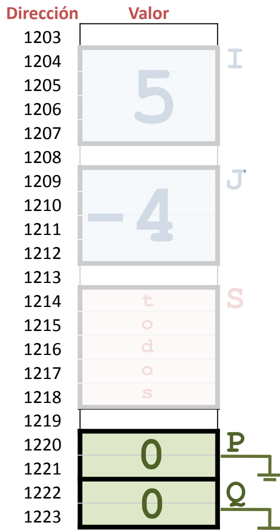
Punteros como alias de otras variables (5)

Dirección	Valor
1203	
1204	5 (I)
1205	
1206	
1207	
1208	
1209	−4 (J)
1210	
1211	
1212	
1213	
1214	t (S)
1215	o
1216	d
1217	o
1218	s
1219	
1220	
1221	
1222	
1223	

```
procedure Punteros_1 is
    …
    I: Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Una variable puntero almacena en su zona de memoria una dirección. Cuando va a utilizarse como alias de otras variables se declara con `access all`.

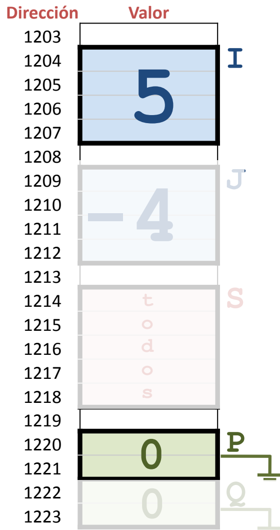
Punteros como alias de otras variables (6)




```
procedure Punteros_1 is
   …
   I: Integer;
   J: Integer;
   S: String(1..5);
   P: access all Integer;
   Q: access all Integer;
   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Una variable puntero se inicializa automáticamente a un valor nulo (es lo mismo que asignarle el valor `null`).

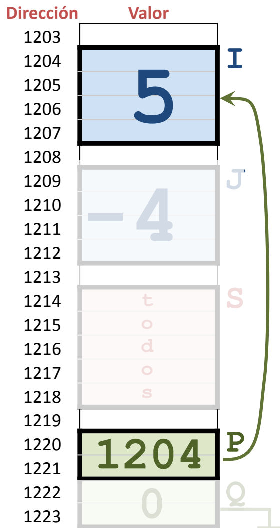
Punteros como alias de otras variables (7)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- El atributo `'Access` devuelve la dirección de una variable. Esa variable necesariamente debe estar declarada con `aliased` delante del nombre de su tipo.

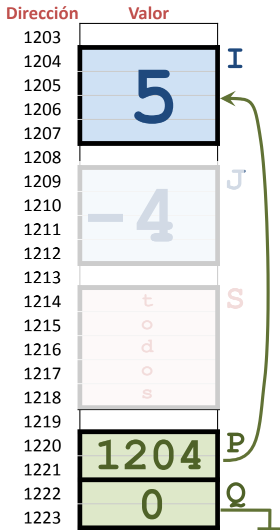
Punteros como alias de otras variables (8)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Así, el puntero P almacena la dirección de la variable I: 1204.

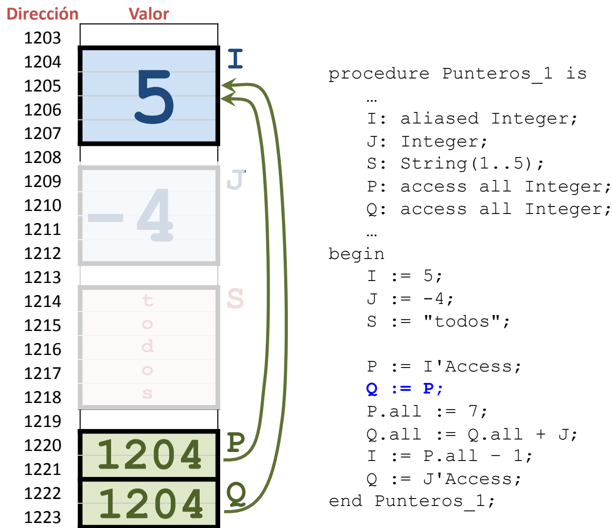
Punteros como alias de otras variables (9)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

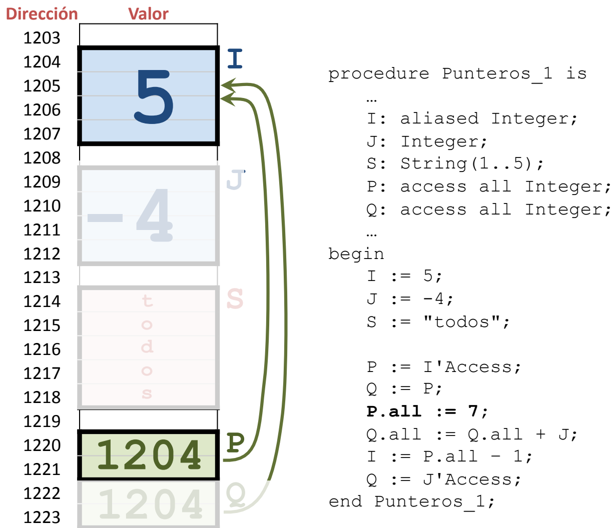
- Una asignación entre variables puntero hace que ambas terminen con la misma dirección.

Punteros como alias de otras variables (10)



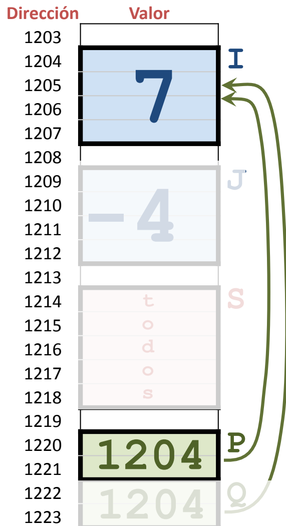
- Es decir, las dos variables terminan "apuntando" a la misma posición de memoria a la que apuntaba la variable de la derecha de la asignación.

Punteros como alias de otras variables (11)



- `P.all` hace referencia al contenido la zona de memoria apuntada por `P`.

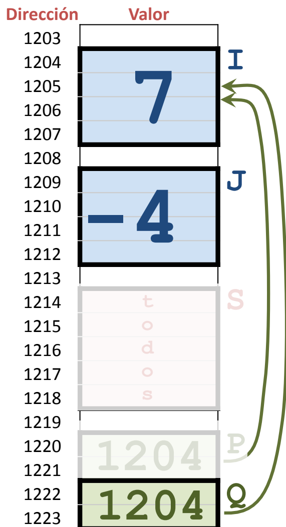
Punteros como alias de otras variables (12)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Es lo mismo utilizar en una expresión `P.all` que `I`. Por eso se dice que `P` "es un alias" de `I`.

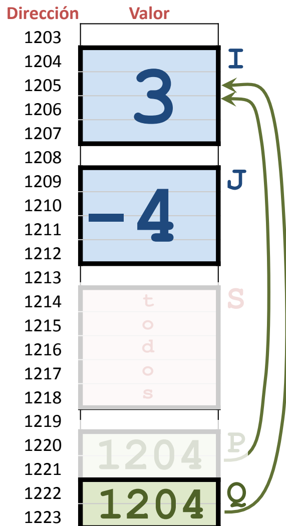
Punteros como alias de otras variables (13)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- A través de `Q.all` también se accede al contenido de `I`. `Q` es otro “alias” de `I`.

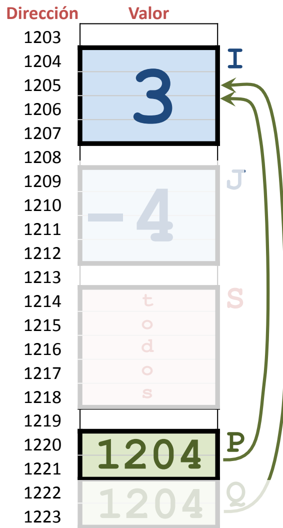
Punteros como alias de otras variables (14)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- `Q.all` es de tipo `Integer`, por eso se puede mezclar en expresiones con otros `Integer`.

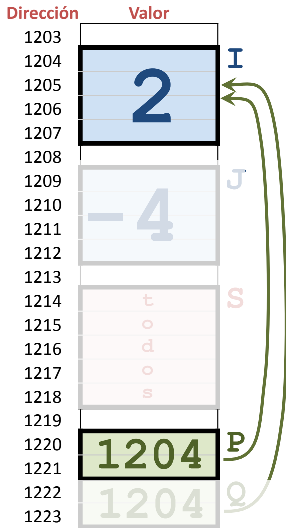
Punteros como alias de otras variables (15)




```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Se pueden mezclar en una expresión distintas formas de nombrar la misma zona de memoria.

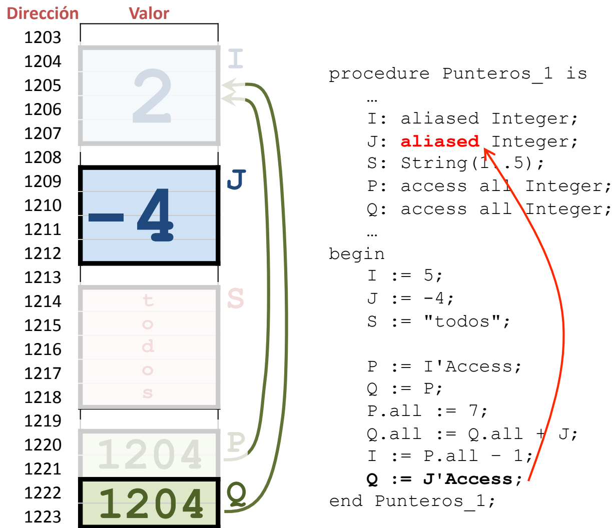
Punteros como alias de otras variables (16)




```
procedure Punteros_1 is
   …
   I: aliased Integer;
   J: Integer;
   S: String(1..5);
   P: access all Integer;
   Q: access all Integer;
   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Por ello al usar punteros hay que tener en cuenta que el código se vuelve más difícil de leer.

Punteros como alias de otras variables (17)



- Al asignarle a `Q` la dirección de `J` es necesario declarar `J` como `aliased`.

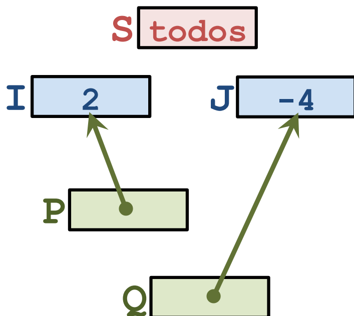
Punteros como alias de otras variables (18)

Dirección	Valor	
1203		
1204	2	I
1205		
1206		
1207		
1208		
1209	-4	J
1210		
1211		
1212		
1213		
1214	t	S
1215	o	
1216	d	
1217	o	
1218	s	
1219		
1220	1204	P
1221		
1222	1209	Q
1223		

```
procedure Punteros_1 is
   …
   I: aliased Integer;
   J: aliased Integer;
   S: String(1..5);
   P: access all Integer;
   Q: access all Integer;
   …
begin
   I := 5;
   J := -4;
   S := "todos";

   P := I'Access;
   Q := P;
   P.all := 7;
   Q.all := Q.all + J;
   I := P.all - 1;
   Q := J'Access;
end Punteros_1;
```

- Ahora `Q` ha dejado de ser un alias de `I` y se ha convertido en uno de `J`.

Punteros como alias de otras variables (19)

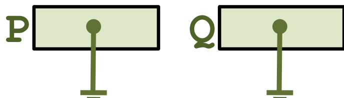



```
procedure Punteros_1 is
    …
    I: aliased Integer;
    J: aliased Integer;
    S: String(1..5);
    P: access all Integer;
    Q: access all Integer;
    …
begin
    I := 5;
    J := -4;
    S := "todos";

    P := I'Access;
    Q := P;
    P.all := 7;
    Q.all := Q.all + J;
    I := P.all - 1;
    Q := J'Access;
end Punteros_1;
```

- Normalmente se usa esta representación simplificada del estado de las variables, sin indicar las direcciones exactas de la memoria.

Punteros para gestionar memoria dinámica (1)

```
procedure Punteros_2 is
    …
    P: access Integer;
    Q: access Integer;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```



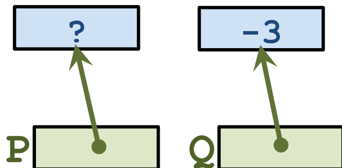
- Un puntero para acceder a memoria dinámica se declara sólo con `access`, sin incluir la palabra `all`.
- Los punteros para acceder a memoria dinámica también se inicializan automáticamente a `null`.

Punteros para gestionar memoria dinámica (2)

```
procedure Punteros_2 is
   …
   P: access Integer;
   Q: access Integer;
   …
begin
   P := new Integer;
   Q := new Integer'(-3);
   P.all := Q.all;
   Q := P;
end Punteros_2;
```

- Para reservar un espacio en memoria dinámica y hacer que un puntero apunte a él se utiliza `new` poniendo a su lado el tipo apuntado por el puntero.
- Si se quiere dar un valor inicial se utiliza en la sentencia `new` la sintaxis `'(valor_inicial)` detrás del tipo.

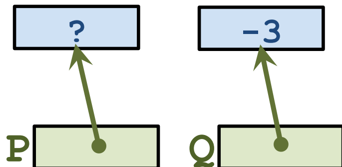
Punteros para gestionar memoria dinámica (3)




```
procedure Punteros_2 is
   …
   P: access Integer;
   Q: access Integer;
   …
begin
   P := new Integer;
   Q := new Integer'(-3);
   P.all := Q.all;
   Q := P;
end Punteros_2;
```

- Si no se da un valor inicial en la sentencia `new`, el valor apuntado queda indeterminado.

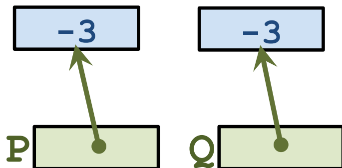
Punteros para gestionar memoria dinámica (4)




```
procedure Punteros_2 is
    …
    P: access Integer;
    Q: access Integer;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- Igual que antes, la sintaxis `.all` permite acceder a los contenidos de lo apuntado por el puntero

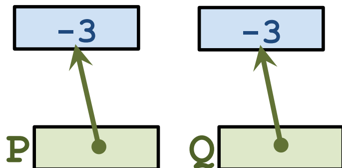
Punteros para gestionar memoria dinámica (5)




```
procedure Punteros_2 is
    …
    P: access Integer;
    Q: access Integer;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- Así, ahora `P` y `Q` almacenan el mismo valor en distintas posiciones de memoria.

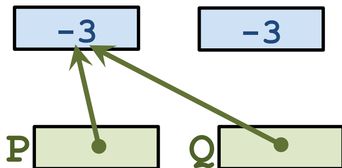
Punteros para gestionar memoria dinámica (6)




```
procedure Punteros_2 is
    …
    P: access Integer;
    Q: access Integer;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- La asignación entre variables puntero hace que apunten a la misma posición de memoria.

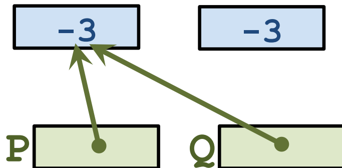
Punteros para gestionar memoria dinámica (7)




```
procedure Punteros_2 is
    …
    P: access Integer;
    Q: access Integer;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- Así, ahora `Q` apunta a la misma posición de memoria a la que apunta `P`.

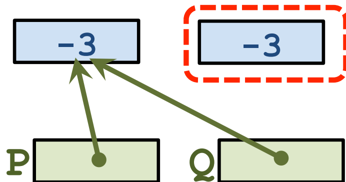
Punteros para gestionar memoria dinámica (8)




```
procedure Punteros_2 is
    …
    type Tipo_Ac_Int is
        access Integer;
    P: Tipo_Ac_Int;
    Q: Tipo_Ac_Int;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- Las variables punteros pueden declararse de tipos anónimos (como en todos las transparencias anteriores) o de un tipo con nombre (como en esta transparencia).

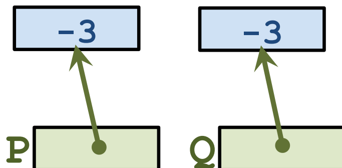
Punteros para gestionar memoria dinámica (9)




```
procedure Punteros_2 is
    …
    type Tipo_Ac_Int is
        access Integer;
    P: Tipo_Ac_Int;
    Q: Tipo_Ac_Int;
    …
begin
    P := new Integer;
    Q := new Integer'(-3);
    P.all := Q.all;
    Q := P;
end Punteros_2;
```

- Si un espacio de memoria dinámica ya no es apuntado por ningún puntero resulta innecesario, pero la memoria sigue ocupada por el programa.
- En lenguajes como Java esa zona de memoria se liberará automáticamente sin que el programador haga nada (a esto se le llama «recolección automática de basura»).
- En lenguajes como C o C++ el programador **debe** liberar esa memoria mediante una sentencia (`free`), pero utilizar mal esa sentencia es **muy peligroso**.
- En Ada está previsto que se utilice recolección automática de basura, pero gnat no lo tiene implementado.

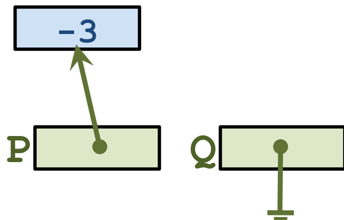
Punteros para gestionar memoria dinámica (10)




```
procedure Punteros_2 is
   …
   type Tipo_Ac_Int is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation
         (Integer, Tipo_Ac_Int);
     P: Tipo_Ac_Int;
     Q: Tipo_Ac_Int;
     …
begin
     P := new Integer;
     Q := new Integer'(-3);
     P.all := Q.all;
     Free(Q);
     Q := P;
end Punteros_2;
```

- En Ada para forzar la liberación de memoria dinámica puede usarse `Unchecked_Deallocation` para definir un procedimiento que puede llamarse `Free` (o como se quiera).
- Hay que tener mucho cuidado para poner la llamada a `Free` justo antes de mover el último puntero que apunta a la zona de memoria que se quiere liberar.

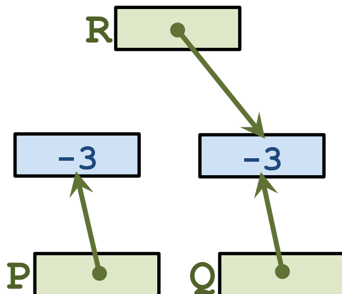
Punteros para gestionar memoria dinámica (11)




```
procedure Punteros_2 is
   …
   type Tipo_Ac_Int is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation
         (Integer, Tipo_Ac_Int);
     P: Tipo_Ac_Int;
     Q: Tipo_Ac_Int;
     …
begin
     P := new Integer;
     Q := new Integer'(-3);
     P.all := Q.all;
     Free(Q);
     Q := P;
end Punteros_2;
```

- Después del `Free`, el puntero tiene valor `null` y la zona de memoria queda marcada como libre, y puede ser asignada para otros usos.

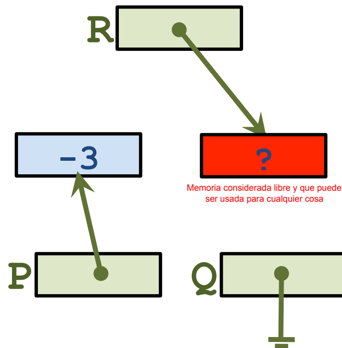
Punteros para gestionar memoria dinámica (12)




```
procedure Punteros_2 is
   …
   type Tipo_Ac_Int is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation
         (Integer, Tipo_Ac_Int);
     P: Tipo_Ac_Int;
     Q: Tipo_Ac_Int;
     …
begin
     P := new Integer;
     Q := new Integer'(-3);
     P.all := Q.all;
     Free(Q);
     Q := P;
end Punteros_2;
```

- EL PELIGRO: Que la zona a liberar aún esté apuntada por otros punteros (en este caso R).

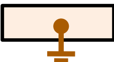
Punteros para gestionar memoria dinámica (13)



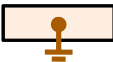

```
procedure Punteros_2 is
   …
   type Tipo_Ac_Int is access Integer;
   procedure Free is new
      Ada.Unchecked_Deallocation
         (Integer, Tipo_Ac_Int);
     P: Tipo_Ac_Int;
     Q: Tipo_Ac_Int;
     …
begin
     P := new Integer;
     Q := new Integer'(-3);
     P.all := Q.all;
     Free(Q);
     Q := P;
end Punteros_2;
```

- Ahora `R` apunta a una zona de memoria que se considera libre, por lo que puede usarse para otras cosas y contener cualquier valor impredecible.
- Cuando se use `R.all` más adelante, el comportamiento del programa es totalmente impredecible.

Punteros para gestionar memoria dinámica (14)

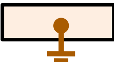


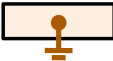

```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```



- Las variables puntero pueden apuntar a cualquier tipo de datos. En las siguientes transparencias vemos un ejemplo de cómo crear una lista enlazada.

Punteros para gestionar memoria dinámica (15)

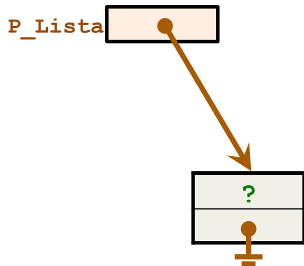





```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- P_Lista siempre apuntará a la cabecera de la lista.

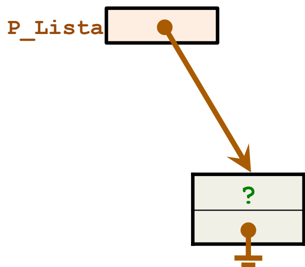
Punteros para gestionar memoria dinámica (16)

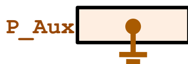


```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- Con `new` se crea una nueva celda en memoria dinámica.

Punteros para gestionar memoria dinámica (17)

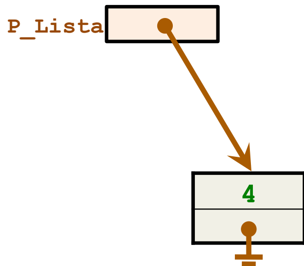


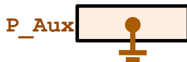



```
procedure Accesos is

     type Celda; -- declaración adelantada incompleta

     type Acceso_Celda is access Celda;

     type Celda is record
          Valor: Integer;
          Siguiente: Acceso_Celda;
     end record;

     P_Lista: Acceso_Celda;
     P_Aux: Acceso_Celda;

begin

     P_Lista := new Celda;
     P_Lista.Valor := 4;
     P_Aux := new Celda'(35, null);
     P_Aux.Siguiente := P_Lista;
     P_Lista := P_Aux;

end Accesos;
```

- Se inicializa el campo `Valor`.

Punteros para gestionar memoria dinámica (18)

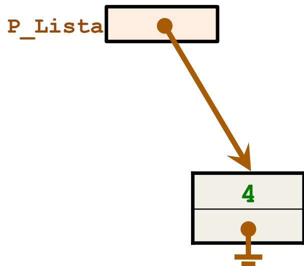


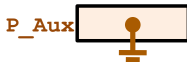



```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- `Valor` ya vale 4, los punteros están a `null` por defecto.

Punteros para gestionar memoria dinámica (19)

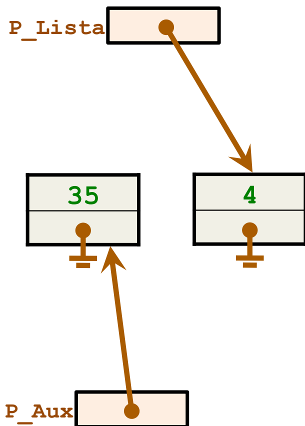





```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- P_Aux es un puntero auxiliar para ir creando nuevos elementos.

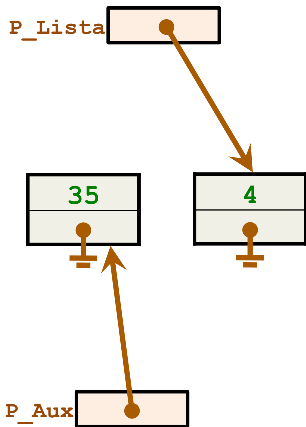
Punteros para gestionar memoria dinámica (20)




```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- new permite dar un valor a todo el registro con un agregado.

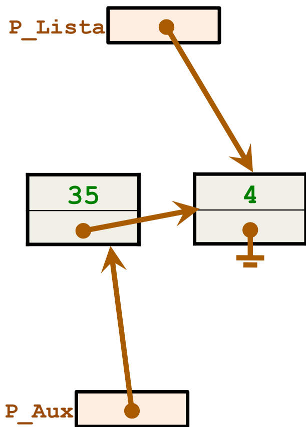
Punteros para gestionar memoria dinámica (21)




```
procedure Accesos is

     type Celda; -- declaración adelantada incompleta

     type Acceso_Celda is access Celda;

     type Celda is record
          Valor: Integer;
          Siguiente: Acceso_Celda;
     end record;

     P_Lista: Acceso_Celda;
     P_Aux: Acceso_Celda;

begin

     P_Lista := new Celda;
     P_Lista.Valor := 4;
     P_Aux := new Celda'(35, null);
     P_Aux.Siguiente := P_Lista;
     P_Lista := P_Aux;

end Accesos;
```

- El nuevo elemento se inserta en primera posición de la lista

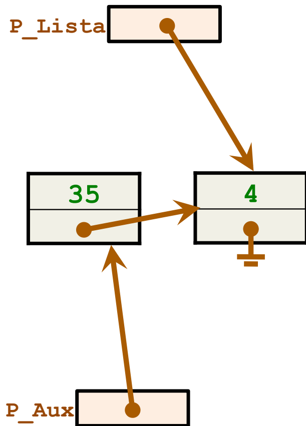
Punteros para gestionar memoria dinámica (22)




```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- Se hace que el `Siguiente` del nuevo elemento sea el actual primero.

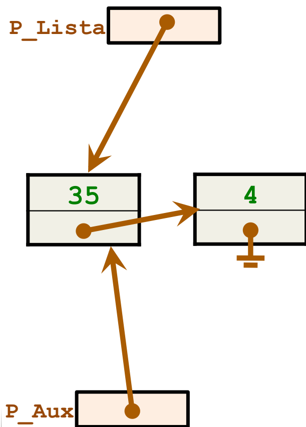
Punteros para gestionar memoria dinámica (23)




```
procedure Accesos is

     type Celda; -- declaración adelantada incompleta

     type Acceso_Celda is access Celda;

     type Celda is record
         Valor: Integer;
         Siguiente: Acceso_Celda;
     end record;

     P_Lista: Acceso_Celda;
     P_Aux: Acceso_Celda;

begin

     P_Lista := new Celda;
     P_Lista.Valor := 4;
     P_Aux := new Celda'(35, null);
     P_Aux.Siguiente := P_Lista;
     P_Lista := P_Aux;

end Accesos;
```

- Ahora `P_Lista` apuntará al nuevo primer elemento.

Punteros para gestionar memoria dinámica (24)




```
procedure Accesos is

    type Celda; -- declaración adelantada incompleta

    type Acceso_Celda is access Celda;

    type Celda is record
        Valor: Integer;
        Siguiente: Acceso_Celda;
    end record;

    P_Lista: Acceso_Celda;
    P_Aux: Acceso_Celda;

begin

    P_Lista := new Celda;
    P_Lista.Valor := 4;
    P_Aux := new Celda'(35, null);
    P_Aux.Siguiente := P_Lista;
    P_Lista := P_Aux;

end Accesos;
```

- Ahora `P_Aux` puede usarse para crear un nuevo elemento, y así sucesivamente.

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Sentencia if (I)

```
with Ada.Text_IO;

procedure Condicional is

   A: Integer := 5;

begin

   if A = 0 then
      Ada.Text_IO.Put_Line ("A vale 0");
   elsif A > 0 then
      Ada.Text_IO.Put_Line("A es positivo");
   else
      Ada.Text_IO.Put_Line("A es negativo");
   end if;

end Condicional;
```

- Las condiciones son expresiones lógicas: operaciones cuyo resultado es `True` o `False`.
- Las partes `elsif ...` y `else` son opcionales.
- Aunque en una de las alternativas haya varias sentencias NO hay que poner `begin` y `end` entre ellas.

Sentencia if (II)

```
with Ada.Text_IO;

procedure Condicional_2 is

   A,  B: Integer;

begin

   A := 5;
   B := -3;

   if A > 0 and B > 0 then
      Ada.Text_IO.Put_Line("A y B son positivos");
   end if;

end Condicional_2;
```

- Las expresiones lógicas pueden usar los operadores `and`, `or`, `xor`, `not`, y ser tan complejas como se quiera.
- Los operadores lógicos son los de menor precedencia de todos, por eso en la expresión del ejemplo no son necesarios paréntesis (aunque pueden aumentar la legibilidad).

Sentencia if (III)

```
with Ada.Text_IO;

procedure Condicional_3 is

   A,  B: Float;

begin

   A := 0.0;
   B := -3.2;

   if A > 0.0 and then B/A > 0 then
      Ada.Text_IO.Put_Line("correcto");
   end if;

end Condicional_3;
```

- El operador `and then` funciona de forma que sólo evalúa su parte derecha si es necesario, es decir, si la parte izquierda es `True` (a veces se le llama operador and "perezoso" o "en cortocircuito").
- El operador `and`, en cambio, evalúa siempre ambas partes de la expresión. En este ejemplo, usar `and then` evita que se produzca una excepción.
- El operador `or else` evalúa su parte derecha sólo si la parte izquierda es `False`.

Sentencia case

```
with Ada.Text_IO;

procedure Condicional_4 is
   I: Integer range 1..20 := 3;
begin
   case I is
      when 1 =>
         Ada.Text_IO.Put_Line("I es 1");
      when 2..10 =>
         Ada.Text_IO.Put_Line("I es pequeña");
      when 13 | 15 | 17 | 19 =>
         Ada.Tetxt_IO.Put_Line("I es grande e impar");
      when others =>
         Ada.Text_IO.Put_Line("I es grande y par");
   end case;
end Condicional_4;
```

- La expresión que acompaña al `case` (`I` en el ejemplo) necesariamente ha de ser de un tipo *contable*: `Integer` y subtipos, `Boolean` o tipos enumerados.
- En cada alternativa `when` puede ponerse un solo valor, un rango (`2..10`) o varios valores alternativos (`13 | 15`).
- Entre todas las alternativas deben estar contemplados todos los valores posibles para la expresión del `case`.
- Puede usarse `others` como última alternativa, con lo que ya se cumple que se contemplen todos los casos.

Bucle loop infinito

```
with Ada.Text_IO;

procedure Bucle_Infinito is

begin

   loop
      Ada.Text_IO.Put ("Hola... ");
      Ada.Text_IO.Put_Line ("a todos");
   end loop;

   -- las líneas siguientes no se ejecutarán nunca
   Ada.Text_IO.Put_Line ("Terminé");

end Bucle_Infinito;
```

- Todos los bucles en Ada se escriben con la sentencia `loop`.
- El bucle más sencillo que puede escribirse en Ada es un bucle `loop` infinito.
- Las sentencias que aparezcan detrás del bucle no se ejecutarán nunca.

Bucle while ... loop

```
procedure Bucle_2 is

   I: Integer := 4;

begin

   while I < 20 loop
      I := I + 1;
   end loop;

end Bucle_2;
```

- Se añade una cláusula `while` *condición* delante del `loop` que abre el bucle.
- Las condición se evalúa antes de empezar el bucle, y después de cada pasada. Si cuando se evalúe la condición su valor es `True`, las sentencias del bucle se ejecutarán una pasada más.
- El final del bucle es `end loop`, como en todos los bucles.

Bucle loop ... exit when

```
procedure Bucle_3 is

   I: Integer := 4;

begin

   loop
      I := I + 1;
      exit when I = 15;
   end loop;

end Bucle_3;
```

- Es un bucle `loop` en el que la última sentencia es `exit when` *`condición`*.
- El bucle se ejecuta siempre al menos una vez. La condición se evalúa después de cada pasada. Si la condición es `False`, las sentencias del bucle se ejecutan una pasada más.
- NOTA: La sentencia `exit` puede usarse en otros sitios y de otras maneras, pero en esta asignatura sólo puede utilizarse en el caso de un `exit when` que sea la última línea de un bucle `loop`.

Bucle for ... loop

```
with Ada.Text_IO;

procedure Bucle_4 is

begin

   for K in 1..5 loop
      Ada.Text_IO.Put_Line ("Hola");
   end loop;

   for K in reverse 1..5 loop
      Ada.Text_IO.Put_Line (Integer'Image(K));
   end loop;

end Bucle_4;
```

- Se añade una cláusula `for` *`etiqueta`* `in` *`rango`* delante del `loop` que abre el bucle.
- El bucle se ejecuta tantas veces como tamaño tiene el rango del `for`.
- La etiqueta del `for` toma en cada pasada valores consecutivos del rango.
- La etiqueta del `for` NO se declara, no es una variable.
- La etiqueta se comporta como una variable de sólo lectura que sólo existe dentro del bucle.
- No se puede avanzar en el rango de 2 en 2, ni hacerlo avanzar de golpe para salir del bucle.
- El rango `5..1` es un rango nulo. Un `for` con ese rango haría que el bucle no se ejecutara ninguna vez.
- Para recorrer un rango al revés hay que escribir `in reverse` y el rango de forma ascendente.

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Estructura general de un programa en Ada

```
-- AQUÍ VAN LAS CLÁUSULAS with DE LOS PAQUETES QUE SE UTILIZAN

procedure Ejemplo is

   -- AQUÍ VAN LAS DECLARACIONES DE TIPOS, CONSTANTES Y VARIABLES GLOBALES

   procedure Proc_1 (...) is
      -- AQUÍ VAN LAS DECLARACIONES DE TIPOS, CONSTANTES Y VARIABLES LOCALES DE Proc_1
   begin
      ...
   end Proc_1;


   procedure Proc_2 (...) is
      -- AQUÍ VAN LAS DECLARACIONES DE TIPOS, CONSTANTES Y VARIABLES LOCALES DE Proc_2
   begin
      ...
   end Proc_2;


   -- AQUÍ VAN LAS DECLARACIONES DE TIPOS, CONSTANTES Y VARIABLES LOCALES DEL PROGRAMA PRINCIPAL

begin
   ...
end Ejemplo;
```

- Dentro del procedimiento principal pueden aparecer anidados otros subprogramas (procedimientos y funciones) a los que llama el código del programa principal.
- Las variables globales deben ser las menos posibles: en vez de usarlas es mejor declararlas locales al programa principal y pasarlas como parámetros a los subprogramas.
- Dentro de cualquier subprograma pueden anidarse más subprogramas que sean llamados por el que los contiene.

¿Por qué usar subprogramas?

- NO es una cuestión estética: no es porque "queda más bonito".
- Por eso NO tiene sentido hacer primero el programa en un solo procedimiento y una vez terminado, crear subprogramas.
- Razones para usar subprogramas:
  1. No repetir código: El "corta-pega" es tentador, pero...¿qué ocurre si se descubre un error en un trozo de código repetido 10 veces por el programa? ¿Se acordará el programador de corregir en el error en las 10 repeticiones? ¿Se le olvidará hacerlo en alguna de ellas?
  2. Modularidad: Hacer el código más legible, un subprograma para cada cosa. Que los subprogramas sean pequeños: un subprograma debería caber entero en la pantalla.
- Cualquiera de las dos razones es suficiente para crear un subprograma:
  - si un trozo de código se va a usar más de una vez, se pone en un subprograma.
  - si un trozo de código se hace demasiado largo, se parte en 2 o más subprogramas.

Procedimientos (I)

Parámetros in y parámetros out

```
with Ada.Text_IO;

procedure Sub_1 is

   procedure Suma (A: in Integer; B: in Integer; C: out Integer) is

   begin
      C := A + B;
   end Suma;

   P: Integer;
   Q: Integer;

begin
   P := 7;
   Suma (P, P, Q);
   Ada.Text_IO.Put_Line ("El doble de " & Integer'Image(P) & " es " &
                         Integer'Image(Q));

end Sub_1;
```

- Los parámetros pasados en modo in sólo pueden ser leídos dentro del procedimiento, pero no modificados.
- Los parámetros pasados en modo out pueden ser modificados dentro del procedimiento, pero no pueden ser leídos antes de ser modificados.
- Los parámetros out son una forma que tiene un procedimiento para devolver valores a quien lo llama.

Procedimientos (II)

Parámetros in out

```
with Ada.Text_IO;

procedure Sub_2 is

   procedure Doblar (A: in out Integer) is
   begin
      A := 2 * A;
   end Doblar;

   P: Integer;

begin

   P := 7;
   Ada.Text_IO.Put ("El doble de " & Integer'Image(P));
   Doblar (P);
   Ada.Text_IO.Put_Line (" es " & Integer'Image(P));

end Sub_2;
```

- Los parámetros pasados en modo `in out` pueden ser leídos y modificados dentro del procedimiento cuando se quiera.
- Los parámetros `in out` son otra forma que tiene un procedimiento para devolver valores a quien lo llama.
- Para cada parámetro debe usarse el modo adecuado a lo que quiera hacerse con él.
- Si no se especifica modo para un parámetro, el modo es `in`.

Funciones

```
with Ada.Text_IO;

procedure Sub_3 is

   function Factorial (N: Positive) return Positive is
      Resultado: Integer;
   begin
      if N = 1 then
         Resultado := 1;
      else
         Resultado := N * Factorial(N-1);
      end if;
      return Resultado;
   end Factorial;

   P: Integer;

begin

   P := 7;
   Ada.Text_IO.Put_Line ("El factorial de "  & Positive'Image(P) &
                         " es " & Positive'Image(Factorial(P)));

end Sub_3;
```

- Las funciones son subprogramas que explícitamente devuelven un valor.
- El valor se devuelve a través de la sentencia `return`, que es conveniente que sea la última sentencia de la función.
- Las funciones no pueden recibir parámetros `out` ni parámetros `in out`.

Parámetros access

```
with Ada.Text_IO;

procedure Sub_4 is

   procedure Incrementar (Ptr: access Integer) is
   begin
      Ptr.all := Ptr.all + 1;
   end Incrementar;

   R: aliased Integer;

begin

   R := 7;
   Ada.Text_IO.Put_Line ("Un incremento de "  & Integer'Image(R));
   Incrementar (R'Access);
   Ada.Text_IO.Put_Line (" es " & Integer'Image(R));

end Sub_4;
```

- En la llamada, un parámetro `access` puede ser una variable puntero o el atributo `'Access` de una variable `aliased`.
- Dentro del subprograma el parámetro `access` se gestiona como una variable puntero.
- Los contenidos apuntados por el parámetro `access` pueden ser leídos y modificados dentro del procedimiento cuando se quiera.
- Los parámetros `access` son otra forma de devolver valores para los subprogramas.
- Tanto las funciones como los procedimientos pueden recibir parámetros `access`.

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Organización general del código Ada en ficheros

- Un programador siempre podría hacer su programa con todo el código metido en un solo fichero: el del procedimiento principal, con muchos subprogramas anidados dentro de él.
- Sin embargo, es más conveniente que el programador reparta los subprogramas auxiliares al principal en paquetes separados.
- De esta forma, el código escrito por un programador correspondiente a un programa en Ada está repartido entre:
  - un fichero (`.adb`) con el procedimiento principal
  - dos ficheros (`.ads`, `.adb`) por cada paquete escrito por el programador.

¿Por qué usar paquetes?

- NO es una cuestión estética: no es porque “queda más bonito”.
- Por eso NO tiene sentido hacer primero el programa en un solo fichero y una vez terminado, crear paquetes.
- Razones para usar paquetes:
  1. No repetir código **entre programas**: Cortar y pegar un subprograma que está en un programa para usarlo en otro es una práctica peligrosa.
  2. Modularidad: Hacer el código más legible, un paquete con procedimientos relacionados para la misma cosa. Los ficheros no deberían ser muy grandes, un programa principal no debería contener demasiados subprogramas.
- Cualquiera de las dos razones es suficiente para crear un paquete:
  - si un subprograma se va a usar en más de un programa debe ponerse en un paquete.
  - si un programa se hace demasiado largo por contener demasiados subprogramas, los subprogramas deben repartirse en 1 o más paquetes.

Especificación (operaciones.ads)

```
package Operaciones is

   procedure Doblar (A: in out Integer);
   function Factorial(N: Positive) return Positive;
   procedure Incrementar (Ptr: access Integer);

end Operaciones;
```

En la especificación de un paquete se colocan:

- el perfil o cabecera de los subprogramas (nombre y parámetros) que exporta el paquete
- declaraciones de los tipos y constantes que exporta el paquete
- declaraciones de las variables que exporta el paquete

Cuerpo (operaciones.adb)

```
package body Operaciones is
   procedure Doblar (A: in out Integer) is
   begin
      A := 2 * A;
   end Doblar;

   function Factorial (N: Positive) return Positive is
   begin
      if N = 1 then
         return 1;
      else
         return N * Factorial(N-1);
      end if;
   end Factorial;

   procedure Incrementar (Ptr: access Integer) is
   begin
      Ptr.all := Ptr.all + 1;
   end Incrementar;
end Operaciones;
```

En el cuerpo de un paquete se colocan:

- el código de todos los subprogramas que se exportan en la especificación
- el código de todos los subprogramas auxiliares que se necesiten para implementar los subprogramas que se exportan
- los tipos, constantes y variables necesarios para implementar los subprogramas que se exportan.

Utilización en otro programa (usa_operaciones.adb)

```
with Ada.Text_IO;
with Operaciones;

procedure Usa_Operaciones is

   I: aliased Integer;
   P: Positive;

begin

   I := 7;
   Operaciones.Doblar (I);
   Operaciones.Incrementar (I'Access);
   P := Operaciones.Factorial (I);
   Ada.Text_IO.Put_Line ("Resultado: " & Positive'Image (P));

end Usa_Operaciones;
```

- La claúsula `with` permite utilizar todo lo exportado en la especificación del paquete.
- Es necesario cualificar con el nombre del paquete lo que quiere utilizarse (aunque sea un tipo, una variable. . .).

Contenidos

1. Introducción
2. El primer programa
3. Tipos
4. Punteros
5. Estructuras de control
6. Subprogramas
7. Paquetes
8. Ficheros de texto
9. Excepciones
10. Ejemplo Final

Ada.Text_IO para gestionar ficheros de texto

- El paquete Ada.Text_IO, además de permitir trabajar con la entrada/salida estándar (por defecto, teclado y ventana de terminal), sirve también para trabajar con ficheros de textos.
- Ada.Text_IO permite leer y escribir en ficheros de texto utilizando los mismos subprogramas que para la entrada/salida estándar, pero utilizando un primer parámetro adicional que representa el fichero que se lee o escribe.

Abrir un fichero de texto

```
with Ada.Text_IO;

procedure Fich_1 is

   Fich: Ada.Text_IO.File_Type;
begin
   ...
   Ada.Text_IO.Open(Fich, Ada.Text_IO.In_File, "prueba.tmp");
   ...
end Fich_1;
```

- `Ada.Text_IO.Open` abre un fichero dado un nombre, y devuelve (como parámetro out) un descriptor del fichero abierto. El descriptor es de tipo `File_Type`.
- El descriptor del fichero se utilizará en las sentencias de lectura, escritura, y cierre del fichero abierto (en vez de utilizar el nombre).
- No hay que confundir el nombre de un fichero con el descriptor de un fichero.
- El segundo parámetro de `Ada.Text_IO.Open` es el modo de apertura del fichero:
  - `Ada.Text_IO.In_File`: Modo lectura, para leer del fichero.
  - `Ada.Text_IO.Out_File`: Modo escritura, para escribir en el fichero.
  - `Ada.Text_IO.Append_File`: Modo añadir, para escribir al final del contenido del fichero.
- Intentar abrir un fichero que no existe provoca un `Ada.Text_IO.NAME_ERROR`.
- Intentar abrir un fichero ya abierto provoca un `Ada.Text_IO.STATUS_ERROR`.

Crear un fichero de texto

```
with Ada.Text_IO;

procedure Fich_1 is

   Fich: Ada.Text_IO.File_Type;
begin
   ...
   Ada.Text_IO.Create(Fich, Ada.Text_IO.Out_File, "prueba.tmp");
   ...
end Fich_1;
```

- `Ada.Text_IO.Create` crea un fichero nuevo dado un nombre, y devuelve (como parámetro out) un descriptor del fichero creado.
- El fichero se crea vacío, para escribir en él.
- Si el fichero existiera previamente, `Ada.Text_IO.Create` borra su contenido.
- `Ada.Text_IO.Create` siempre se usa con modo `Ada.Text_IO.Out_File`.
- Intentar crear un fichero ya abierto o creado provoca un `Ada.Text_IO.STATUS_ERROR`.

Leer de un fichero de texto

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

procedure Fich_1 is
   package ASU renames Ada.Strings.Unbounded;

   Fich: Ada.Text_IO.File_Type;
   S: ASU.Unbounded_String;
begin
   ...
   Ada.Text_IO.Open(Fich, Ada.Text_IO.In_File, "prueba.tmp");
   S := ASU.To_Unbounded_String(Ada.Text_IO.Get_Line(Fich));
   ...
end Fich_1;
```

- `Ada.Text_IO.Get_Line(Fich)` lee una línea de un fichero de texto, y la devuelve como `String`. Para poder almacenarla en una variable, se necesita hacer una conversión a `Unbounded_String`.
- En el ejemplo se lee la primera línea del fichero. Una nueva llamada a `Ada.Text_IO.Get_Line(Fich)` leería la segunda línea, y así sucesivamente.
- Intentar leer un fichero que no está abierto provoca un `Ada.Text_IO.STATUS_ERROR`.

Escribir en un fichero de texto

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

procedure Fich_1 is
   package ASU renames Ada.Strings.Unbounded;

   Fich: Ada.Text_IO.File_Type;
   S: ASU.Unbounded_String;
begin
   ...
   Ada.Text_IO.Create(Fich, Ada.Text_IO.Out_File, "prueba.tmp");
   Ada.Text_IO.Put_Line(Fich, "esta es una línea");
   ...
end Fich_1;
```

- `Ada.Text_IO.Put_Line(Fich, ...)` escribe un `String` en un fichero de texto, añadiendo al final un carácter de fin de línea.
- Si no se quiere añadir el fin de línea puede utilizarse `Ada.Text_IO.Put(Fich, ...)`.
- En el ejemplo se escribe en la primera línea del fichero. Una nueva llamada a `Ada.Text_IO.Put_Line(Fich, ...)` escribiría la segunda línea, y así sucesivamente.
- Intentar escribir en un fichero que no está abierto o creado provoca un `Ada.Text_IO.STATUS_ERROR`.

Cerrar un fichero de texto

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

procedure Fich_1 is
   package ASU renames Ada.Strings.Unbounded;

   Fich: Ada.Text_IO.File_Type;
   S: ASU.Unbounded_String;
begin
   ...
   Ada.Text_IO.Create(Fich, Ada.Text_IO.Out_File, "prueba.tmp");
   ...
   Ada.Text_IO.Close(Fich);
   ...
end Fich_1;
```

- `Ada.Text_IO.Close` cierra un fichero previamente abierto con `Ada.Text_IO.Open` o `Ada.Text_IO.Create`, independientemente del modo de apertura del fichero.
- El descriptor del fichero deja de ser válido después de la sentencia `Ada.Text_IO.Close`, y ya no puede utilizarse.
- Si el fichero se ha usado para escribir en él, hasta que no se cierra no se realizan de verdad en el disco las últimas escrituras, por eso es muy importante no olvidarse de cerrar los ficheros abiertos.
- Intentar cerrar un fichero que no está abierto, o que ya se ha cerrado, provoca un `Ada.Text_IO.STATUS_ERROR`.

Contenidos

1 Introducción
2 El primer programa
3 Tipos
4 Punteros
5 Estructuras de control
6 Subprogramas
7 Paquetes
8 Ficheros de texto
9 Excepciones
10 Ejemplo Final

Comportamiento predefinido

```
with Ada.Text_IO;

procedure Excepciones_1 is
   I: Integer;
begin
   I := 0;
   I := 4 / I;  -- elevará una excepción Constraint_Error
   Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (I));
end Excepciones_1;
```

Salida del programa:

```
raised CONSTRAINT_ERROR : excepciones.adb:7 divide by zero
```

- Un error en tiempo de ejecución provoca que el programa se interrumpa y aparezca un mensaje predefinido en pantalla
- En el mensaje aparece en qué línea de qué fichero fuente se produjo la excepción.
- A veces la excepción se produce en un paquete de la biblioteca estándar de Ada, al ejecutarse un subprograma llamado por el código del programador.

Gestión de excepciones por parte del programador

```
with Ada.Text_IO;

procedure Excepciones_2 is
   I: Integer;
begin
   I := 0;
   I := 4 / I;  -- elevará una excepción Constraint_Error
   Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (I));
exception
   when Constraint_Error =>
      Ada.Text_IO.Put_Line ("Intento de dividir por 0");
end Excepciones_2;
```

Salida del programa:

```
Intento de dividir por 0
```

- El programador puede escribir código para ejecutar después de que se produzca la excepción.
- Ahora si se produce una excepción en un bloque de código, se deja de ejecutar ese código y se salta al **manejador** de la excepción que se ha producido.
- La sintaxis de la parte `exception` es similar a las cláusulas de un case, pero no tienen que aparecer todas las posibles excepciones
- Cuando se produce una excepción para la que no aparece una cláusula `when` se usa el comportamiento predefinido.

Manejador de excepciones others

```
with Ada.Text_IO;
with Ada.Exceptions;

procedure Excepciones_3 is
   I: Integer;
begin
   I := 0;
   I := 4 / I;  -- elevará una excepción Constraint_Error
   Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (I));
exception
   when Constraint_Error =>
      Ada.Text_IO.Put_Line ("Intento de dividir por 0");
   when Except:others =>
      Ada.Text_IO.Put_Line ("Excepción imprevista: " &
                            Ada.Exceptions.Exception_Name (Except) & " en: " &
                            Ada.Exceptions.Exception_Message (Except));
end Excepciones_3;
```

- Ahora ante cualquier excepción que no sea `Constraint_Error` se ejecutan las líneas de `when others`.
- A `others` se le puede poner por delante una "etiqueta" que permite llamar a algunos subprogramas del paquete `Ada.Exceptions` (por ejemplo, para recuperar el nombre de la excepción o el mensaje asociado donde aparece el número de línea).

Estructura general de un bloque de código en Ada

```
procedure Principal is
   ...
   sentencia_a;
   sentencia_b;
   sentencia_c;
   ...
end Principal;
```

- En cualquier lugar en que puede ir una sentencia en Ada puede ir un bloque, cuya sintaxis general es:

```
procedure Principal is
   ...
   sentencia_a;
   declare
      declaraciones;
   begin
      sentencia_b1;
      sentencia_b2;
      ...
   exception
      when excepción_x =>
         sentencia_x1;
         sentencia_x2;
      when excepción_y =>
         sentencia_y1;
         ...
      ...
   end;
   sentencia_c;
   ...
end Principal;
```

- Las partes `declare` y `exception` son opcionales.

Comportamiento general de las excepciones

```
procedure Principal is
   ...
   sentencia_a;
   declare
      declaraciones;
   begin
      sentencia_b1;
      sentencia_b2;
      ...
   exception
      when excepción_x =>
         sentencia_x1;
         sentencia_x2;
      when excepción_y =>
         sentencia_y1;
         ...
      ...
   end;
   sentencia_c;
   ...
end Principal;
```

- Cuando se produce una excepción en una sentencia:
  1. se interrumpe el bloque de código que la contiene
  2. se salta al manejador de excepciones de ese bloque, si lo hubiera, y se ejecutan sus sentencias
  3. se continua por la siguiente sentencia al bloque en el que se ha producido la excepción
- En el ejemplo, si al ejecutarse la `sentencia_b1` se produce la `excepción_x`, se salta a ejecutar la `sentencia_x1`, la `sentencia_x2`, y luego se ejecutará la `sentencia_c`.

Una excepción no tiene por qué terminar un programa

```
with Ada.Text_IO;

procedure Excepciones_4 is
   I, J: Integer;
begin
   I := 0;
   while I < 50 loop
      begin
         J := 40 / I;  -- elevará una excepción Constraint_Error
      exception
         when Constraint_Error =>
            Ada.Text_IO.Put_Line ("Intento de dividir por 0");
      end;

      Ada.Text_IO.Put_Line ("Resultado: " & Integer'Image (J));
      I := I + 10;
   end loop;
end Excepciones_4;
```

Salida:

```
Intento de dividir por 0
Resultado:  1074347488
Resultado:  4
Resultado:  2
Resultado:  1
Resultado:  1
```

El primer resultado se debe a que J todavía no ha sido inicializada.

Contenidos

1 Introducción

2 El primer programa

3 Tipos

4 Punteros

5 Estructuras de control

6 Subprogramas

7 Paquetes

8 Ficheros de texto

9 Excepciones

10 Ejemplo Final

El ejemplo que se muestra a continuación, además de las características vistas anteriormente, muestra:

- Cómo puede un programa recibir argumentos por la línea de comandos, mediante el paquete `Ada.Command_Line`.
- Cómo puede el programador definir sus propias excepciones.
- Cómo pasar un valor numérico contenido en un `String` a una variable `Integer` con `Integer'Value` (atributo inverso de `Integer'Image`).
- Cómo buscar en un `Unbounded_String` con la función `Index` y extraer partes con `Head` y `Tail`.

```
with Ada.Text_IO;
with Ada.Strings.Unbounded;
with Ada.Command_Line;
with Ada.Exceptions;

procedure IP_Puerto is

   package ASU renames Ada.Strings.Unbounded;

   Usage_Error: exception;
   Format_Error: exception;

   procedure Next_Token (Src: in out ASU.Unbounded_String;
                         Token: out ASU.Unbounded_String;
                         Delimiter: in String) is
      Position: Integer;
```

```
   begin
      Position := ASU.Index(Src, Delimiter);
      Token := ASU.Head (Src, Position-1);
      ASU.Tail (Src, ASU.Length(Src)-Position);
   exception
      when others =>
         raise Format_Error;
   end;

   IP: ASU.Unbounded_String;
   Parte_IP: ASU.Unbounded_String;
   Puerto:  Integer;
begin

   if Ada.Command_Line.Argument_Count /= 1 then
      raise Usage_Error;
   end if;

   Puerto := Integer'Value (Ada.Command_Line.Argument(1));
   Ada.Text_IO.Put ("Introduce una dirección IP: ");
   IP :=  ASU.To_Unbounded_String (Ada.Text_IO.Get_Line);
```

```
   for I in 1..3 loop
      Next_Token (IP, Parte_IP, ".");
      Ada.Text_IO.Put_Line ("Byte" & Integer'Image(I) &
                            " de la dirección IP: " &
                            ASU.To_String(Parte_IP));
   end loop;
   Ada.Text_IO.Put_Line ("Byte 4 de la dirección IP: " &
                         ASU.To_String(IP));
   Ada.Text_IO.Put_Line ("Puerto: " & Integer'Image (Puerto));

exception
   when Usage_Error =>
      Ada.Text_IO.Put_Line ("Uso: ip_puerto <número-de-puerto>");
   when Format_Error =>
      Ada.Text_IO.Put_Line ("Formato incorrecto en la dirección IP");
   when Except:others =>
      Ada.Text_IO.Put_Line ("Excepción imprevista: " &
                            Ada.Exceptions.Exception_Name (Except) & " en: " &
                            Ada.Exceptions.Exception_Message (Except));
end IP_Puerto;
```