Fundamentos de la Programación y la Informática
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos
Noviembre de 2024

© 2024 Miguel Angel Ortuño Pérez.
Algunos derechos reservados. Este documento se distribuye bajo la licencia *Atribución-CompartirIgual 4.0 Internacional* de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Índice

1. Introducción a la informática y la programación **5**
1.1. Elementos básicos en informática 5
1.1.1. CPU 6
1.1.2. Memoria Principal 6
1.1.3. Memoria Secundaria 7
1.1.4. Dispositivos E/S 7
1.2. Programación 9
1.2.1. El lenguaje Pascal 9
1.2.2. Compilación 11
1.2.3. Errores software 13
1.2.4. Estructura de un programa Pascal 17

2. Expresiones **23**
2.1. Directivas para el compilador 23
2.2. Palabras reservadas 23
2.3. Comentarios 24
2.4. Identificadores 25
2.5. Tipos de Datos 27
2.6. Constantes 28
2.7. Escritura 29
2.8. Operadores 36
2.9. Codificación de caracteres 37
2.10. Casting 38
2.11. Representación de los números reales 43

3. Funciones **56**
3.1. Definición de problemas 56
3.2. Diseño de programas 67
3.3. Ejemplos de uso de funciones 69

4. Selección **73**
4.1. Problemas de selección 73
4.2. if then else 73
4.3. case 87
4.4. Estructura de un programa 91
4.5. Precondiciones y postcondiciones 93

5. Procedimientos **97**
5.1. Introducción a los procedimientos 97
5.2. Variables 98
5.3. Procedimientos 106
5.4. Paso de parámetro por referencia 113
5.5. Parámetro de salida con código de error 124
5.6. Procedimiento val 126
5.7. Concatenación de cadenas 129
5.8. Ámbito de las variables 131

6. Definición de tipos y registros **136**
6.1. Nuevos tipos de datos 136
6.2. Registros 139

7. Bucles **150**
7.1. Introducción a los bucles 150
7.2. Bucles while 151
7.3. Números aleatorios 155
7.4. Bucles repeat 165
7.5. Bucles for 170

8. Arrays **179**
8.1. Introducción a los arrays 179
8.1.1. Clasificación de problemas 185
8.1.2. Problemas de acumulación 186
8.2. Problemas de búsqueda 187
8.3. Matrices 196
8.3.1. Introducción a las matrices 196
8.3.2. Matriz de números aleatorios 200

8.3.3. Suma de los valores de una matriz 201
8.3.4. Suma por filas 202
8.3.5. Suma por columnas 203
8.3.6. Generación de arrays 205
8.3.7. Suma de matrices 207
8.3.8. Máximos de una matriz 211
8.3.9. Búsqueda en matrices 212
8.4. Array de registros 215
8.5. Procesamiento de cadenas 219

9. Ficheros **226**
9.1. Introducción a los ficheros 226
9.2. Apertura 228
9.3. Lectura y escritura 228
9.4. Cierre 229
9.5. Paso de ficheros a subprogramas 230
9.6. Ejemplos 232
9.7. Lectura de campos de ancho fijo 238
9.8. Escritura de campos de ancho fijo 241

10. Memoria dinámica **242**
10.1. Introducción 242
10.2. Punteros 244
10.3. Listas encadenadas 244

11. Anexo: Acceso remoto al laboratorio **248**
11.1. Introducción al acceso remoto 248
11.2. Selección del host 248
11.3. Sesión desde Windows 250
11.4. Sesión desde macOS 252

12. Anexo: Uso básico de la shell de Unix/Linux **255**
12.1. Introducción a la shell 255
12.2. Conceptos básicos sobre la shell 255
12.3. Órdenes básicas de la shell de Unix 259

13. Anexo: El editor de texto Nano **263**
13.1. Introducción al editor de texto Nano 263
13.2. Funcionalidad principal 264
13.3. Color basado en la sintaxis 266
13.4. Indentado en Nano 267

14. Anexo: Compilación de un programa **269**

15. Anexo: Depuración **272**
15.1. Formato de cadenas 272
15.2. Depuración de un programa 275

1. Introducción a la informática y la programación

1.1. Elementos básicos en informática

Programación de Ordenadores

- Programar es darle instrucciones a un ordenador.
    - Estas instrucciones han de ser extremadamente precisas. Un cambio en una letra suele invalidar por completo un programa. Un cambio de mayúscula a minúscula, o de un espacio, puede invalidar por completo el programa. Esto es algo que al principiante le cuesta asimilar.
- Un ordenador recibe datos y genera nuevos datos según las instrucciones del programa.
- El programa también son datos, se puede cambiar tan fácilmente como el resto de datos, lo que permite que el ordenador sea de uso general.

Hardware y Software

- Hardware: parte tangible.
- Software: intangible.
    - Programas de usuario.
    - Sistema Operativo (*Operating System*) : programa intermedio entre el hardware y los programas de usuario. Acrónimo: SO, OS. Ejemplos: Microsoft Windows, Mac OS, Linux, Android, iOS, ...

Ejecutar un programa: pedirle al ordenador (al sistema operativo) que siga sus instrucciones.

Componentes de un ordenador

- CPU. *Central Processing Unit*
- Memoria principal.
- Memoria secundaria.
- Dispositivos de entrada/salida.

1.1.1. CPU

CPU

Acrónimo de *Central Processing Unit*.

- En español normalmente decimos CPU, pero también se usa *unidad central de procesamiento*, *unidad de procesamiento central*.
- Wikipedia lo define como *Hardware dentro de un ordenador u otros dispositivos programables, que interpreta las instrucciones de un programa informático mediante la realización de las operaciones básicas aritméticas, lógicas y de entrada/salida del sistema.*

Atención: a veces se le da un significado distinto: se llama CPU a la *caja* principal del ordenador (la *torre*), que contiene memoria, disco y lo que aquí llamamos CPU.

- En los ordenadores actuales, las CPU más frecuentes son las basadas en la arquitectura X86_64: los Intel i3, i5, i7, y sus equivalentes AMD.
  - Los hay más sencillos como Intel Pentium o Intel Celeron, y más potentes como Intel i9.
- Los teléfonos móviles modernos y las tablets son verdaderos ordenadores. Suelen usar una arquitectura diferente, ARM. Son más sencillos, baratos, consumen menos y se calientan menos.
  - Distintos fabricantes: Qualcomm, Nvidia, Samsung, Huawei,...
  - Los ordenadores Apple desde 2020 también emplean ARM.

1.1.2. Memoria Principal

Memoria Principal

- Normalmente se le llama *memoria*, a secas. También memoria primaria o memoria interna.
- Programa y datos están en memoria principal.
- La memoria principal es volátil (se borra al apagar), por tanto es necesario almacenar en memoria secundaria.
- Puede ser de varios tipos: RAM, ROM...

1.1.3. Memoria Secundaria

Memoria Secundaria

- Normalmente se le llama simplemente *disco*.
- Puede ser un disco duro mecánico tradicional, (HDD, *hard disk drive*), un disco de estado sólido (SSD, *Solid State Drive*), un pendrive, etc. En épocas anteriores lo habitual fue el cdrom, los floppy disk, diversos tipos de cintas magnéticas, o, en los orígenes, tarjetas de cartulina perforadas .
- La memoria secundaria es mucho más lenta que la memoria principal. Pero tiene mucha mayor capacidad, es más barata y sobre todo, no es volátil (si manejamos los dispositivos debidamente).
- La CPU no trabaja directamente sobre memoria secundaria [1], programas y datos de entrada se leen desde disco hasta memoria. Los datos de salida se escriben desde memoria hasta disco.
- La información en el disco se organiza en ficheros y directorios.
  - *fichero* y *directorio* (file / directory) son los términos informáticos tradicionales. En Microsoft Windows y otros SO se usan las palabras *documento* y *carpeta*.
- Cada vez se usan más los discos de red y los sistemas *en la nube*: son discos como los demás, pero que
  - No están físicamente cerca de la CPU, sino que se accede a ellos a través de una red informática, típicamente Internet.
  - Con frecuencia los administra una entidad distinta.

1.1.4. Dispositivos E/S

Dispositivos de Entrada Salida

Aquel tipo de dispositivo periférico de un computador capaz de interactuar con los elementos externos a ese sistema de forma bidireccional, es decir,

[1] salvo en alguna excepción

que permite tanto que sean ingresada información desde un sistema externo, como ser emitir información a partir de ese sistema. [2]. (Wikipedia)

- Con frecuencia se usan las siglas E/S (entrada salida) o I/O (input output).
- Ejemplos: Consola (pantalla + teclado), ratón, impresora, pantalla táctil, escáner, lector braille, tarjeta de red, módem...

Unidades de almacenamiento

En un ordenador, todos los datos (incluidos los programas) se almacenan como números en sistema binario.

- Los números binarios no usan los dígitos del 0 al 9, sino el 0 y el 1.
- Esto es muy conveniente: por ejemplo el 1 se puede representar por 5 voltios y el 0 por 0 voltios.
    - O mejor aún: un voltaje por debajo de 2.5 V es un 0, un voltaje superior a 2.5 V es un 1.
- Otro ejemplo: un agujero microscópico en un cdrom (*pit*) representa un 1. La ausencia de agujero (*land*) representa un 0.

Unidades de información

Unidades de información *tradicionales*

```
1 bit
8 bits = 1 byte
1024 bytes = 1 kilobyte (Kb)
1024 kilobytes = 1 megabyte (MB)
1024 megabytes = 1 gigabyte (GB)
1024 gigabytes = 1 terabyte (TB)
1024 terabytes = 1 petabyte (PB)

... Exabyte, Zettabyte, Yottabyte
```

Este sistema tiene un problema:

Usa potencias de 2 ($2^{10} = 1024$)

A pesar de que esos prefijos están reservados para las potencias de 10 ($10^3 = 1000$).

[2] La memoria secundaria realmente también es un dispositivo E/S, pero como es tan importante, en nuestra clasificación le dedicamos una categoría completa

Anexo: Unidades ISO/IEC 80000

Desde el año 1998 diversos organismos internacionales establecen que

- Los nombres anteriores (Kb, Mb, etc) deben basarse en potencias de 10.
- Los nombres correctos basados en potencias de 2 son kibibyte (KiB), mebibyte (MiB), gibibyte (GiB), tebibyte (TiB), pebibyte (PiB), exbibyte (EiB), zebibyte (Zib), yobibyte (YiB).

Sin embargo, esta norma no se emplea mucho. Sigue siendo más frecuente la notación tradicional.

- Cuando veamos que por ejemplo un disco tiene 140 Mb, no podemos estar seguros de si son 140 * 1024 o 140 * 1000.

1.2. Programación

1.2.1. El lenguaje Pascal

Lenguaje Pascal

- Creado por Niklaus Wirth a finales de los años 1960.
- Muy adecuado para la formación de programadores.
- En los años 1980 y 1990 fue muy popular en la industria, hoy es raro usarlo fuera de la enseñanza, aunque sigue siendo perfectamente válido para desarrollar (en Windows, macOS, Linux, iOS, Android...).
- Con los años han aparecido diferentes *dialectos* con pequeñas variantes, aquí usaremos *Object Pascal* con el compilador *Free Pascal*. Esta es la combinación más habitual en la actualidad, con diferencia.

- Pascal tiene muy pocas *peculiaridades*, podemos considerarlo un *máximo común divisor* de los lenguajes más habituales: prácticamente todo lo que aprendas en Pascal lo encontrarás en cualquier otro lenguaje, con cambios mínimos.
- Por ser un lenguaje sencillo, pero completo, permite centrarse en los aspectos esenciales de la programación y en la adquisición de buenas prácticas.

¿Por qué Pascal?

Inconveniente de Pascal.

- Quien aprenda a programar en Pascal, es seguro que luego tendrá que aprender otro lenguaje.
- Este es un inconveniente menor, cualquier programador (incluyendo a la mayoría de los ingenieros) tendrá que trabajar en diversos lenguajes a lo largo de su carrera. Incluso programar simultáneamente en varios lenguajes.

¿Por qué no otros lenguajes?

No es fácil decidir cuál es el mejor lenguaje para aprender a programar. Es subjetivo, diferentes especialistas tienen diferentes opiniones, todas generalmente razonables. Ningun lenguaje es óptimo para esto.

Ejemplos:

- java es posiblemente el lenguaje de programación más empleado en el mundo. Pregunta típica de principiante: *¿Por qué aprendemos Pascal, que no usa nadie, y no java?*.

  Respuesta: java tiene una sintaxis muy complicada. Y es orientado a objetos, un nivel de abstracción que no tiene sentido para un principiante. Entre otros muchos problemas. https://qr.ae/TSkd0d

- C tiene una sintaxis sencilla, pero el uso de punteros hace que su uso resulte complicado, los errores son especialmente difíciles de detectar. Se puede aprender a usar bien, pero son habilidades no especialmente necesarias para el resto de lenguajes más habituales.
- Python tiene un nivel muy alto. Su sistema de tipos dinámico oculta aspectos fundamentales que cualquie programador de cualquier lenguaje necesita conocer.
- Matlab tiene los problemas de Python, pero además es un lenguaje propietario (y caro). Y muy orientado al cálculo matemático, no es de propósito general.
- Scratch es demasiado sencillo. Está diseñado para enseñar programación a niños, no a estudiantes de ingeniería.
- Etc etc.

Pasos en la programación:

1. Definir el problema. También llamado especificar. Se pueden usar
   - Métodos formales. Estuvieron de moda un tiempo pero raramente se usan
   - Descripciones informales, detalladas, en lenguaje natural (español, inglés...).
2. Diseñar un algoritmo.

   Es un plan detallado de cómo será el programa. Normalmente se usa un lenguaje denominado *pseudocódigo*, a mitad de camino entre lenguaje natural y un lenguaje de programación.
3. Implementar.

   Es la programación en sentido estricto. Se codifican las instrucciones en un *lenguaje de programación*, en nuestro caso, Pascal.
4. Probar.
5. Corregir.

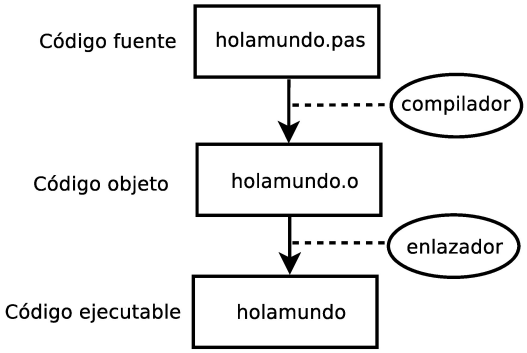
- En las metodologías de programación tradicionales (años 1970, 1980), estos 5 pasos (definir, diseñar, implementar, probar, corregir) se consideraban bien definidos y aislados. Se suponía que se debía ejecutar cada paso una vez, en cascada uno detrás de otro.
- Desde los años 1990, lo habitual es usar diversas metodologías *ágiles*, donde estos 5 pasos se repiten una y otra vez, generando prototipos cada vez más completos.

1.2.2. Compilación

Compilación

- Para escribir un programa (en Pascal o en cualquier otro lenguaje de programación) usamos un editor de texto (no un procesador de texto).

  Ejemplo: nano, gedit, atom, geany, ...



- Estas instrucciones en Pascal no se pueden ejecutar directamente, hace falta traducirlas a un lenguaje diferente: el *código máquina* de nuestra CPU, adaptado al entorno de nuestro sistema operativo. Este código sí funciona en nuestro ordenador y se denomina así, *ejecutable*.
- De esto se encarga un programa informático denominado *compilador*.

Como ejemplo, veamos cómo se compila un programa *holamundo*. Cuando se enseña un lenguaje de programación, tradicionalmente se empieza por un ejemplo mínimo llamado *hello world* que se limita a escribir este texto en pantalla.

Ejemplo en Java:

```
class HelloWorld {
  static public void main( String args[] ) {
    System.out.println( "Hello World!" );
  }
}
```

Ejemplo en Pascal:

```
program holamundo;
begin
    writeln('Hola, mundo');
end.
```

Colección de *holamundos* en diferentes lenguajes: http://helloworldcollection.de

1. El programador escribe el código fuente.
2. A partir del código fuente, el compilador genera el código objeto.
3. A partir el código objeto, el enlazador genera el código ejecutable.

Así por ejemplo

- Un estudiante de esta asignatura trabajando en el laboratorio, usará un compilador de Pascal para las CPU intel64 (celeron, i3, i5, i7, etc) sobre Linux.
- El mismo estudiante programando en su casa sobre Windows, usará un compilador de Pascal para las CPU intel64 sobre Windows.
- Un programador de aplicaciones para teléfonos Android usará (normalmente) un compilador de Java, que generará (normalmente) código para las CPU arm64 sobre Android[3].
- Un programador de iPhone trabajará (normalmente) en Objetive C, su compilador generará código para las CPU arm64, armv7 o armv7s, sobre el sistema operativo iOS.

Aunque compilación y enlazado son dos cosas distintas, normalmente con una sola orden se ejecutan automáticamente las dos cosas.

Ejemplo: `holamundo.pas`

```
Free Pascal Compiler version 3.3.1 [2018/09/14] for x86_64
Copyright (c) 1993-2018 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling holamundo.pas
Linking holamundo
7 lines compiled, 0.3 sec
```

Ejecución:

```
koji@mazinger:~/pascal$ ./holamundo
Hola, mundo
```

1.2.3. Errores software

Tipos de error en un programa

- Error de compilación.

  Son los más sencillos de detectar, nos los indicará el compilador antes de ejecutar nada. P.e. errores de sintaxis, errores con los tipos de datos...

[3]En el caso de Java hay algún paso intermedio, que en esta asignatura podemos obviar

- Errores en tiempo de ejecución.

  Un poco más complicados de detectar, solo se producen si el programa ejecuta cierta instrucción con ciertas condiciones.

  P.e. una división entre cero, un fichero que no existe...

- Error lógicos.

  También llamados *bugs* (bichos). Son los más difíciles de detectar y corregir. El programa hace algo que no genera errores, pero que no es lo que deseamos que haga.

- Defectos en la claridad del código.

  El programa puede que no produzca errores actualmente, pero su falta de calidad conlleva errores potenciales.

Si hay un error de compilación, el compilador nos indica la fila y la columna.

```
program holamundo_erroneo;
begin
    writeln['Hola, mundo'];     //  ¡Mal!
        // Esto es un error, el argumento está entre corchetes
        // cuando debería estar entre paréntesis
end.
```

Compilación.

```
koji@mazinger:~/pascal$ fpc -gl holamundo_mal.p
Free Pascal Compiler version 3.3.1 [2018/09/14] for x86_64
Copyright (c) 1993-2018 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling holamundo_mal.p
holamundo_mal.p(3,13) Error: Illegal qualifier
holamundo_mal.p(8) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

En la línea 3, columna 13, tenemos un error de tipo *Illegal qualifier*.

Errores lógicos

Cuando hablamos de *errores*, sin especificar el tipo, normalmente nos estamos refiriendo a los errores en tiempo de ejecución o a los errores lógicos.

- Los errores de compilación son fáciles de detectar por cualquier programador (a menos que sea principiante).

Probar *bien* un programa es muy complicado.

- Es imposible estar seguro al 100 % de que un programa está libre de errores. Se puede conseguir una probabilidad de error relativamente baja, pero siempre puede haber fallos, en ocasiones dramáticos.

  https://raygun.com/blog/costly-software-errors-history

Probar programas es un disciplina en sí misma, distinta de la programación.

- En entornos críticos, es normal emplear más esfuerzo (dinero) en probar un programa que en desarrollarlo.
- En entornos menos exigentes, es más frecuente (más barato) reparar los problemas causados por los errores que evitarlos.

Un programa (mínimamente realista = mínimamente complejo) siempre corre el riesgo de tener errores. La responsabilidad del programador es que el ratio sea lo más bajo posible.

Defectos en la claridad del código

Es imprescindible que el código sea claro, con un diseño adecuado y que cumpla los convenios establecidos.

- Con frecuencia el principiante, que acaba de hacer un programa de unas docenas de líneas, piensa *lo fundamental es que funcione. Que esté bonito es secundario y subjetivo.*
- Esto es completamente erróneo.
  - En la vida profesional no hay programas de unas docenas de líneas. Los programas tienen entre miles y millones de líneas.
  - Los programas de tamaño real, mal escritos, que funcionan correctamente, no existen.

Siempre hay muchas formas de hacer un programa, con calidad variable, con diversas ventajas e inconvenientes. Podrán priorizar.

- El tiempo que tarda en programarse.
- El tiempo que tarda en ejecutarse.

- La claridad.
- El tamaño del código fuente.
- La memoria consumida.
- El disco consumido.
- El uso de la red.
- Etc

Casi siempre deberíamos priorizar la claridad. Esto facilita el mantenimiento y disminuye la probabilidad del error.

- Solamente cuando la solución más clara resulte inadecuada, debemos optar por otra.
- Y solo cuando lo hayamos medido de forma feaciente, *cronómetro en mano* o *calculadora en mano*. Los humanos somos muy malos estimando todo esto *a ojo*.

Lo más importante en esta asignatura es que adquieras buenos hábitos de programación, uno de los fundamentales es hacer programas claros. Problema típico:

1. El principiante presenta un programa a su profesor, que produce un resultado correcto.
2. El profesor le indica que el programa no está bien porque es confuso y le indica una solución alternativa.
3. Para el alumno esta solución resulta más complicada, o como mucho, equivalente.

Es normal que el principiante no vea las ventajas de la solución *correcta*.

- No es raro preferir la solución que hemos elaborado nosotros, a la que hemos dedicado mucho tiempo y por tanto conocemos bien.

- Típicamente los defectos que señala el programador experimentado se manifiestan en programas *reales*, de muchos miles de líneas. Por tanto es cierto que para el ejemplo *de juguete* no hay ventajas, o son mínimas. Pero lo importante es formar al futuro programador para adquirir estas buenas prácticas.

Naturalmente el profesor procurará justificar bien su afirmación, pero aún así, este problema no tiene fácil remedio.

Diseño de algoritmos

Para diseñar un algoritmo deberíamos emplear solo tres tipos de construcción.

- Secuencia de acciones.
- Selección de acciones.
- Iteración de acciones.

Construcciones adicionales no son ni necesarias ni recomendables.

1.2.4. Estructura de un programa Pascal

Estructura de un programa en Pascal

Como referencia, indicamos aquí la estructura de un programa en Pascal: Un programa se compone de

- Cabecera (`program nombre`).
- Un bloque:
  - Parte declarativa: declaración de constantes, variables, funciones y procedimientos.
  - Parte de las sentencias: `begin` lista_de_sentencias `end`.
- Un punto.

lista_de_sentencias

- Secuencia de sentencias, separadas por *punto y coma*.

Las sentencias pueden ser:

- Simples.
  - Asignación.
  - Llamada a procedimiento.
  - Raise.
- Estructuradas.
  - Condicional (case, if).
  - Bucle: for, repeat, while.
  - with.
  - try.

Como ejemplo y como anticipo de lo que trataremos en los próximos 4 temas, mostramos el siguiente programa.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program nota_fpi;

function hace_media(compensable, entrega, teoria, prac: real): boolean;
begin
    result :=  (entrega >= compensable) and (teoria >= compensable)
            and (prac >= compensable);
end;

function media(peso_entrega, peso_teoria, peso_prac, entrega, teoria,
               prac: real): real;
begin
    result := entrega * peso_entrega + teoria * peso_teoria +
                  prac * peso_prac;
end;

const  // Constantes locales del cuerpo del programa principal
    Compensable = 4.0;

    // Peso relativo de cada apartado, en tanto por uno
    Peso_entrega = 0.25;  // Entrega prácticas
    Peso_teoria = 0.35;   // Examen teórico
    Peso_prac = 0.40;     // Examen práctico
```

```
    Entrega_jperez = 3.5;
    Teoria_jperez = 5.5;
    Prac_jperez = 5.0;

begin
    write('Nota final: ');
    if (hace_media( Compensable, Entrega_jperez, Teoria_jperez,
        Prac_jperez))
    then writeln( media(Peso_entrega, Peso_teoria, Peso_prac,
        Entrega_jperez, Teoria_jperez, Prac_jperez):0:2)
    else writeln( 'No apto' );
end.
```

Tabulación

El texto de los programas no está alineado a la izquierda, como el lenguaje natural, sino que se añade cierto espacio. Esto se denomina sangrado o tabulación.

- Una tabulación correcta es imprescindible para la claridad de un programa.
- No hay una forma única de tabular, hay distintos criterios dependiendo del gusto del programador.
    - Si el proyecto lo empezamos nosotros, podemos tabular según nuestras preferencias. Pero siempre de forma consistente.
    - Para un proyecto preexistente, debemos respetar el criterio establecido.

Hay dos formas de insertar espacios:

- Mediante el carácter *tab*.

  Es un carácter especial que significa *añadir cierto espacio horizontal*. No especifica cuánto. Depende del editor, puede estar configurado por omisión para ocupar el equivalente a 4 espacios, a 8 espacios o cualquier otro valor, es configurable.
- Mediante espacios.

Ejemplo:

```
const
    a=3;
```

A la izquierda de la *a* puedo haber insertado o bien 1 tabulador o bien 4 espacios. En una versión impresa como esta transparencia, es imposible distinguirlo.

Supongamos que teníamos el editor configurado con el tabulador a 4 espacios y que hayamos insertado un tabulador. Si en otro momento editamos con un editor configurado a 8 espacios, veremos algo así:

```
const
        a=3;
```

Esto no es un problema.

Según nuestras preferencias, podemos tabular:

- Con tabuladores.
- Con 4 espacios.
- Con 8 espacios.
- Con otro número de espacios (aunque es poco frecuente).

Pero es imprescindible mantener el mismo criterio dentro del mismo programa o proyecto.

Si en algún momento necesito cambiar el criterio en todo el proyecto, es fácil hacerlo automáticamente.

- En esta asignatura, consideramos que cada fichero es un proyecto independiente, por tanto puedes cambiar de criterio entre programa y programa (por si quieres experimentar o cambia tu gusto).
- En entornos *reales*, esto no sería aceptable.

El problema de no respetar el mismo criterio es el siguiente. Supongamos que tengo el editor con un tabulador a 4 espacios.

```
const
    a = 3;
    b = 0;
```

Supongamos que para la primera declaración usé tabuladores y para la segunda,espacios.

Si en otro momento otra persona o yo mismo trabajo con un editor con el tabulador a 8 espacios, veré lo siguiente:

```
const
        a = 3;
    b = 0;
```

¡Esto arruina por completo la claridad del programa!

Por tanto:

- Es muy conveniente configurar el editor para que muestre los caracteres invisibles. Así, representará tabuladores y espacios con algún símbolo especial (normalmente flechas o puntos).
  - En Nano, `M-P`.
- Es imprescindible prestar atención para mantener el criterio de tabulación establecido.

En esta asignatura,

- Un programa con tabulación inconsistente tendrá mala nota o suspenderá, dependiendo de la gravedad del defecto.
- Un programa que mezcle continuamente tabulaciones con espacios, estará suspenso (porque potencialmente, toda la tabulación será incorrecta).

A partir de un fichero indentado con espacios, podemos convertirlos automáticamente en tabuladores o viceversa, hay muchas herramientas.

- Posiblemente nuestro editor incluya esta funcionalidad.
- Desde la shell de Linux:
  - Convertir los tabuladores en 4 espacios.

    ```
    expand -t4 fichero_original.pas > nuevo_fichero.pas
    ```

Esto crea el fichero *nuevo_fichero.pas*, igual a *fichero_original* pero donde los tabuladores ahora son 4 espacios. Cambiando `-t4` por `-t8`, escribiría 8 espacios por cada tabulador. Observa que seguramente querrás renombrar el nuevo fichero, para que pase a tener el nombre original.

- Convertir 4 espacios en 1 tabulador

```
unexpand -t4 fichero_original.pas > nuevo_fichero.pas
```

2. Expresiones

2.1. Directivas para el compilador

Directivas para el compilador

En este curso, siempre le pediremos al compilador que sea especialmente cuidadoso con los errores. Nos advertirá o prohibirá ciertas construcciones que en principio son legales, aunque peligrosas. Para ello añadimos la siguiente línea antes de la cabecera del programa:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
```

- No te preocupes por su significado concreto, cópiala en todos tus programas

Ejemplo completo:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program holamundo;
begin
    writeln('Hola, mundo');
end.
```

2.2. Palabras reservadas

Palabras reservadas

En prácticamente cualquier lenguaje de programación hay una serie de palabras que forman parte del propio lenguaje, no se pueden usar como identificadores.

Como referencia, incluimos aquí las de nuestro dialecto de Pascal (Object Pascal):

```
and array asm begin break case const constructor continue
destructor div do downto else end false file for function
goto if implementation in inline interface label mod nil
not object of on operator or packed procedure program
record repeat set shl shr string then to true type unit until
uses var while with xor
```

2.3. Comentarios

Comentarios

En cualquier lenguaje de programación hay *comentarios*. Se trata de texto que resulta útil para las personas, pero que el compilador ignorará. En nuestro dialecto de Pascal hay varias formas de insertar comentarios:

- Todo lo que se escriba entre llaves, es un comentario. Puede ser una línea o varias.

```
{Esto es un comentario
de varias líneas}
```

- Todo lo que se escriba después de dos barras, hasta el final de esa línea, es un comentario. Por tanto es un comentario de una sola línea.

```
//  Esto es un comentario de una línea.
//  Para añadir otro comentarios, escribo de nuevo dos barras
```

Los comentarios son importantes para la calidad de un programa.

- Los comentarios tienen que aportar información *relevante*.

```
writeln( ejemplo );  // Escribe en pantalla el valor de 'ejemplo'
```

  Este comentario es una obviedad, sobra.

- Un error típico de principiante (y de no tan principiante) es pensar que poner muchos comentarios ya hace que un programa esté bien documentado.

- La información que pueda extraerse directamente del código no debería repetirse en un comentario: se corre el riesgo de modificar el ćodigo y no actualizar el comentario.

Escribir comentarios informativos, relevantes y claros es muy importante. Aunque no es fácil para el principiante.

2.4. Identificadores

Identificadores

Identificador: nombre para un elemento del programa (programa, función, procedimiento, constante, variable, etc).

- Normalmente definido por el programador (o por el programador de un *módulo*[4]).
- En Pascal solo podemos usar letras inglesas para los identificadores.
  - Esto no suele ser un problema, cualquier programa medianamente serio estará escrito en inglés (identificadores y comentarios). Otros idiomas como el español se usan solo en el interface de usuario, si procede.
- El lenguaje Pascal no distingue mayúsculas de minúsculas Apellidos, APELLIDOS y APElliDOS resulta equivalente.
  - La mayoría de los lenguajes de programación sí distinguen mayúsculas de minúsculas.
- Es importante elegir buenos identificadores, que indiquen claramente qué nombran. Un identificador ambiguo o abiertamente incorrecto es un error de claridad en el programa. No provoca errores lógicos por si mismo, pero induce al programador a crearlos.
  - El sulfato de sodio hay que etiquetarlo como *sulfato de sodio*, no como *sulfato*. Y mucho menos como *sulfato de cloro*. De lo contario ... https://youtu.be/QNTZbJSQVis
- Debemos esforzarnos en elegir buenos identificadores, aunque esto no es fácil para el principiante. Leer atentamente ejemplos de código de calidad ayuda a adquirir esta habilidad.

Ámbito de un identificador

- Algunos identificadores no se pueden repetir, tienen que ser únicos en todo el programa. Son de *ámbito global*.

[4]Módulo: fichero que contiene código para ser usado desde un programa en otro fichero distinto. En otros lenguajes a los módulos se les llama librerías

- Otros identificadores sí se pueden repetir, no importa si aparece el mismo en dos lugares distintos del programa, estarán nombrando cosas distintas. Son de *ámbito local*.

En otras palabras:

- Un identificador tiene que ser único en su ámbito. Si su ámbito es global, tiene que ser único en todo el programa. Si su ámbito es local, tiene que ser único en cierta parte del programa, pero puede repetirse en otro lugar del programa.
- *Ámbito* es el lugar de un programa donde se puede usar un identificador.

Cada lenguaje de programación tiene sus propias reglas sobre el ámbito de los identificadores, pero suelen ser muy parecidas: las mismas que Pascal.

Por ejemplo, en Pascal, el nombre de programa es de ámbito de global. El identificador que nombra a un programa no puede repetirse para nombrar otra cosa distinta en otro lugar del programa.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo;
const
    Ejemplo = 3.0;  // ¡MAL! identificador repetido
begin
    writeln( Ejemplo );
end.
```

Si intentamos compilar el ejemplo anterior, obtendremos un error.

```
koji@mazinger:~/fpi/tema02$ fpc -gl ejemplo.pas

Free Pascal Compiler version 3.0.4+dfsg-23 [2019/11/25] for x86_64
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling ejemplo.pas
ejemplo.pas(5,18) Error: Duplicate identifier "Ejemplo"
ejemplo.pas(10) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

2.5. Tipos de Datos

Tipos de datos

En Pascal manipulamos datos. Son de 5 tipos:

- Integer. Números enteros.
- Real. Números reales.
- Char. Carácteres.

  'a' es un tipo char. También ' ' y '0', que no debemos confundir con el número 0.
- String. Cadenas de texto.
- Boolean. Valores booleanos.

  Solo puede tomar dos valores: TRUE o FALSE.

Para un ordenador es muy distinto el número real 3, el numero entero 3, el carácter '3', o la cadena '3'.

- El problema es que para las personas no familirizadas con la programación, no existe esta diferencia.
- Además, una vez escrito en pantalla o en papel, es imposible distinguirlo (en el código fuente sí).

A veces representamos (en pantalla, papel o el código fuente) un número real sin decimales con un cero tras la coma para distinguirlo.

Ejemplo: el real 3, a veces se escribe como 3.0 (pero no siempre). Así queda claro que no es el entero 3.

Caracteres y cadenas

En Pascal los valores de los caracteres y de las cadenas se escriben con una comilla recta al principio y otra al final.

```
'esto es una cadena'
```

- Hay lenguajes que usan la comilla doble.
- Hay lenguajes que permiten usar tanto la comilla doble como la recta.

Observa que en el teclado español, la comilla recta es la tecla a la derecha de la tecla 0.

- No la confundas con la comilla invertida, la tecla a la derecha de la tecla p.
- En ciertas tipografías, lamentablemente la comilla recta no se representa recta, lo que induce a confusión.

2.6. Constantes

Declaración de constantes

Una constante es una entidad (una *caja*) que contiene un dato, que no cambiará durante la ejecución del programa.

- Nos referiremos a ella con un identificador. Un convenio habitual, que seguiremos en este curso, es escribirlas empezando por letra mayúscula.

Para usar una constante:

1. Podemos declarla, indicando su tipo.

   Nombre de la constante, dos puntos, tipo de dato, punto y coma.

2. La definimos, indicamos su valor.

   Nombre de la constante, igual, valor, punto y coma.

El primer punto es opcional. Casi siempre se puede elegir declarar o no declarar, va en gustos. Aquí recomendamos que no declares las constantes.

Ejemplo de declaración y definición:

```
const
    E: real ;
    E = 2.71828182845904;
```

Si declaramos y definimos, es recomendable declarar y definir en la misma línea:

```
const
    E: real = 2.71828182845904;
```

Aquí recomendamos definir sin declarar. Esto es, indicar el valor pero no el tipo[5].

```
const
    E = 2.71828182845904;
```

La definición, con o sin declaración, ha de hacerse:

- Dentro de un bloque, en la parte declarativa , después de la palabra reservada `const` y antes de la lista de sentencias (begin end).

Observaciones:

- Después de `const` no va un punto y coma.

Las constantes pueden declarse

- Al principio del programa.

  Serán constantes globales, visibles en todo el programa. Deben usarse lo mínimo posible, solo para valores *universales*, que no cambien fácilmente: p.e. el número Pi, el radio de la tierra, el tamaño de un campo en un fichero estandarizado, etc[6].

- Al principio de un subprograma.

  Típicamente, al principio del cuerpo del programa principal. Solo serán visibles en este subprograma. Aquí podemos definir p.e. datos concretos de nuestro programa.

Si tenías nociones de programación, echarás de menos las variables. Por motivos didácticos, en este curso las veremos en el tema 5. No las uses hasta entonces.

2.7. Escritura

Escritura en pantalla

- El procedimiento `write` escribe en la consola (la pantalla) los argumentos que recibe.

[5] En el tema 8 veremos que el tamaño de los arrays es necesario (y no optativo) definirlos así, sin declarar el tipo

[6] Suponiendo que todo esto sea constante en el ámbito de nuestro problema, en ciertos escenarios todos estos valores podrían ser cambiantes.

- El procedimiento `writeln` escribe en la consola los argumentos que recibe, y a continuación, un salto de línea.
- Este procedimiento escribe los valores reales en notación científica. Para usar notación decimal, añade a continuación `:0:n`, donde
  - El 0 significa que el número puede ocupar todo el espacio que necesite.
  - `n` representa el número de decimales.

  Ejemplo: `write(Tiempo_segundos:0:1)`
  - Esta es una de las pocas rarezas de Pascal, no es habitual encontrarlo en otros lenguajes.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program talla01;

const
    Talla = 'xl';
    Color = 'azul';
begin
    writeln('Talla:');
    writeln(Talla);
    writeln('Color:');
    writeln(Color);
end.
```

Con el procedimiento *writeln*, cada argumento se escribe en una línea distinta, este es el resultado:

```
Talla:
xl
Color:
azul
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program talla02;
```

```
const
    Talla = 'xl';
    Color = 'azul';
begin
    write('Talla:');
    write(Talla);
    write('Color:');
    write(Color);
end.
```

Con el procedimiento *write*, todos los argumentos se escriben la misma línea, sin espacios por medio, este es el resultado:

```
Talla:xlColor:azul
```

Normalmente usaremos una combinación de *write* y *writeln*, como en el siguiente ejemplo:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program talla03;

const
    talla = 'xl';
    color = 'azul';
begin
    write('Talla:');
    writeln(talla);
    write('Color:');
    writeln(color);
end.
```

Resultado:

```
Talla:xl
Color:azul
```

Reales en notación científica

Por omisión, nuestro dialecto de Pascal escribe todos los números reales en notación científica

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program descuento01;

const
    Precio = 12.50;
    Porcentaje_descuento = 10.0;
begin
    write('Precio: ');
    write(Precio);
    write(' Descuento: ');
    writeln(Precio * 0.01 * Porcentaje_descuento  );
    write('Precio final: ');
    writeln(Precio * (1 - 0.01 * Porcentaje_descuento));
end.
```

Resultado:

```
Precio:  1.2500000000000000E+001 Descuento:  1.2500000000000000000E+0000
Precio final:  1.1250000000000000000E+0001
```

Reales en notación convencional

Para escribir los números reales en notación convencional, añadimos `:0:n`, donde n es el número de decimales deseados.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program descuento02;

const
    Precio = 12.50;
    Porcentaje_descuento = 10.0;
begin
    write('Precio: ');
    write(Precio:0:2);
    write(' Descuento: ');
    writeln(Precio * 0.01 * Porcentaje_descuento:0:2 );
    write('Precio final: ');
    writeln(Precio * (1 - 0.01 * Porcentaje_descuento):0:2 );
end.
```

Resultado:

```
Precio: 12.50 Descuento: 1.25
Precio final: 11.25
```

Definición de reales en notación científica

Naturalmente, también podemos emplear la notación científica para definir número reales.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program cientifica;
const
    // Podemos escribir 'e' o 'E'
    Billon_europeo = 1e+12;
    Billon_americano = 1E+9;
    Cte_Planck = 6.62607015E-34;

begin
    writeln( 'Notación científica');
    writeln( 'Billón europeo:', Billon_europeo);
    writeln( 'Billón americano:', Billon_americano);
    writeln( 'Constante de Planck', Cte_Planck);
    writeln();
    writeln( 'Notación tradicional');
    writeln( 'Billón europeo:', Billon_europeo:0:0);
    writeln( 'Billón americano:', Billon_americano:0:0);
    writeln( 'Constante de Planck:', Cte_Planck:0:36);
end.
```

Resultado del programa anterior:

```
Notación científica
Billón europeo: 1.000000000000000000E+0012
Billón americano: 1.000000000E+09
Constante de Planck 6.6260701500000000011E-0034

Notación tradicional
Billón europeo:1000000000000
Billón americano:1000000000
Constante de Planck:0.000000000000000000000000000000000663
```

Atención: para usar `:0:n` es necesario que el argumento del `write` sea real, no puede ser entero.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program error_write;
```

```
const
    X = 3.0;
    Y = 7;

begin
    write('X:');
    writeln(X:0:2);
    write('Y:');
    writeln(Y:0:2);  //¡¡Mal!! La constante Y es entera
end.
```

```
Compiling error_write.pas
error_write.pas(12,18) Error: Illegal use of ':'
error_write.pas(14) Fatal: There were 1 errors compiling module, stopping
```

Expresiones: otro ejemplo

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program area_triangulo;

const
    Base = 12.43;
    Altura = 5.91;

begin
    write('Base:');
    write(Base:0:3);
    write(' altura: ');
    writeln( Altura:0:3);
    write('Área del triángulo: ');
    writeln(Base * Altura * 0.5:0:3);
end.
```

Resultado de la ejecución:

```
Base:12.430 altura: 5.910
Área del triángulo: 36.731
```

En un mismo write o writeln se pueden escribir varios argumentos, separados por comas.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program area_triangulo;

const
    Base = 12.43;
    Altura = 5.91;

begin
    write('Base:',Base:0:3);
    writeln(' altura: ', Altura:0:3);
    writeln('Área del triángulo: ', Base * Altura * 0.5:0:3);
end.
```

Resultado:

```
Base:12.430 altura: 5.910
Área del triángulo: 36.731
```

Sugerencia:

- Mientras seas principiante, escribe un solo argumento en cada write/writeln.
- Cuando tengas más soltura, puedes escribir dos argumento en cada write/writeln, pero evita escribir más de dos.

¿Por qué?

- Para evitar errores (comillas mal colocadas, paréntesis, etc).
- Para que los mensajes de error del compilador sean más claros.

2.8. Operadores

Operadores

Un operador es un símbolo o una palabra reservada que indica que se debe realizar una operación matemática o lógica sobre unos *operandos* para devolver un resultado.

- Los operandos son expresiones (valores) de entrada, en Pascal la mayoría de los operadores tienen 2 operandos, algunos tienen 1.

Ejemplo:

- $5 + 3$

El operador + tiene dos operandos: el 5 y el 3[7] y devuelve como resultado su suma.

Los operadores están muy vinculados a los tipos de datos, cada operando solo puede recibir ciertos tipos concretos de datos, para devolver cierto tipo de datos.

Ejemplo

El operador `div` es la división entera. Sus operandos han de ser números enteros. En otro caso, el compilador produce un error.

Uso correcto:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ej_div;
begin
    writeln( 5 div 2);   // Escribe 2
end.
```

Uso incorrecto:

```
    writeln( 5 div 2.0);
```

```
koji@mazinger:~/pascal$ fpc ej_div.p
Free Pascal Compiler version 3.0.0+dfsg-2 [2016/01/28] for x86_64
Copyright (c) 1993-2015 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling ej06.p
ej06.p(4,16) Error: Operator is not overloaded: "ShortInt" div "Single"
ej06.p(8) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

[7] a simple vista no podemos saber si son enteros o reales, pero en este caso no importa, el compilador hará la conversión si hace falta

Otro ejemplo incorrecto

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program tipos_mal;

const
    X = 3;
    Y = '4';
begin
    write( 'X vale ');
    writeln( X );

    write( 'Y vale ');
    writeln( Y );

    write( 'La suma vale ');
    write( X + Y );  // ¡¡MAL!!  La constante Y no es numérica
end.
```

```
Compiling tipos_mal.pas
tipos_mal.pas(15,14) Error: Operator is not overloaded: "LongInt" + "Char"
tipos_mal.pas(17) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

Potencias

Para poder usar el operador de exponenciación, es necesario añadir la cláusula `uses math;` en la cabecera, esto añade el módulo matemático.

Ejemplo: $2.1^{3.1}$

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ej_potencias;
uses math;
begin
    writeln( 2.1 ** 3.1);  // Escribe 9.97423999265870760145E+0000
end.
```

2.9. Codificación de caracteres

Codificación de caracteres

Los ordenadores almacenan toda la información, incluyendo los textos, como números (binarios).

- Cada carácter se asocia con un código numérico. A esto se le denomina *codificación de caracteres*.

  `https://es.wikipedia.org/wiki/Codificaci%C3%B3n_de_caracteres`

- En la decada de 1960 se populariza el código ASCII, que asocia cada letra (inglesa) con un número.

- Ha habido diferentes formas de representar las letras no inglesas.

- Desde la década de 2010 (aprox.) lo más habitual es emplear la norma *Unicode* codificada en *UTF-8*.

  - Es un superconjunto de ASCII: esto es, las letras inglesas se codifican exactamente igual que en ASCII.

2.10. Casting

Casting

- Pascal es *fuertemente tipado*, esto significa que en general no se pueden mezclar tipos de datos, hay que convertirlos antes para que sean homogéneos.

  Convertir un dato de un tipo a otro se llama *ahormado*. En español solemos emplear la palabra inglesa: *casting*.

- Algunas conversiones de tipos las hace el compilador automáticamente.

Operadores para enteros y reales

Hay operadores numéricos que se pueden usar con enteros o con reales, indistintamente, incluso mezclando los tipos.

```
**   Exponenciación
+ - * /
-   (operador unario de cambio de signo)
```

Realmente lo que sucede es que si un operando es entero y el otro real, el compilador convierta automáticamente el entero en real.

Ejemplo: `2 + 2.0` internamente se convierte en `2.0 + 2.0`

Operadores solo para enteros

Otros operadores solo admiten operandos enteros.

- `div`

  El operador *div* devuelve la división entera: la división de dos operandos enteros.

  Ejemplo: `5 div 2 = 2`

- `mod`

  El operador *mod* devuelve el módulo de la division entera: el resto [8].

  Ejemplo: `5 mod 2 = 1`

Si el programador se equivoca e intenta usarlos con operandos reales, el compilador dará un error. Ejemplo:

```
6.12 div 0.02     // ¡MAL!
4.0  mod 2.0      // ¡MAL!
4    mod 2.0      // ¡MAL!
```

El programador puede hacer ciertas conversiones de tipos explícitamente.

- Un entero se puede convertir en real.

  ```
  real(3)
  ```

- Un carácter se puede convertir en entero (obteniendo el código ASCII correspondiente).

  ```
  integer('a')
  ```

- Un entero se puede convertir en carácter (obtenemos el carácter correspondiente a ese código ASCII).

  ```
  char(97)
  ```

Código ASCII:
https://es.wikipedia.org/wiki/ASCII

[8] Aquí 'módulo' es sinónimo de 'resto'. Nada que ver con 'módulo' con el significado de 'librería'.

Un número real no se puede ahormar directamente a entero,
Pero disponemos de las funciones predefinidas
`trunc()` y `round()`
que reciben un número real y devuelven un entero, truncado sus decimales o redondeando.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program round_trunc;
const
    X= 4.7;
    Y= 4.5;
    Z= 4.5001;
begin
    writeln( round(X) );  // 5
    writeln( trunc(X) );  // 4
    writeln( round(Y) );  // 4
    writeln( round(Z) );  // 5
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program casting;

const
    X = 'a';
    Y = 98;
    Z = 65.7;
begin
    writeln( integer(X));  // Escribe 97

    writeln( char(Y));     // Escribe 'b'
    writeln( real(Y));     // Escribe 9.8000000000000000E+001
    writeln( real(Y):0:2); // Escribe 98.00

    { writeln( integer(Z));   // ¡Esto es ilegal!  }

    writeln( round(Z));   // Escribe  66
    writeln( trunc(Z));   // Escribe  65

    writeln( char( trunc(Z) ) ) ;   // Escribe  'A'
end.
```

Operadores de comparación

Sus argumentos pueden ser enteros o reales. Devuelven un booleano.

```
=    Igual
<>   Distinto
<    Menor
<=   Menor o igual
>    Mayor
>=   Mayor o igual
```

Un error frecuente es confundir el operador de comparación de igualdad = con el operador de asignación := [9].

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program compara;

const
    A = 50;
    B = 100;
begin
    writeln(A = B);  // FALSE
    writeln(A = A);  // TRUE
    writeln(A <> B);  // TRUE
    writeln(A > B);  // FALSE
    writeln(A <= B);  // TRUE
    writeln((A+A) >= B); // TRUE
end.
```

Es materia de los temas 3 y 4, pero adelantamos aquí estos ejemplos.

- programa `ej_comparacion_v1`;

  La función *posible_ matricula* devuelve TRUE si la nota es 10, devuelve FALSE en otro caso.

- programa `ej_comparacion_v2`;

  La función *posible_ matricula* devuelve la expresión `nota=10`.

  Valdrá TRUE si la nota es 10, FALSE en otro caso.

[9] O con el operador de comparación de igualdad en C y derivados ==, o con el operador de asignación en C y derivados, =.

Ambos programas devuelven lo mismo.

- El principiante tiende a utilizar la primera forma, que es redundante pero no es incorrecta, al principiante le resulta más claro y la claridad es conveniente.
- El programador con más experiencia preferirá la segunda.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ej_comparacion_v1;

function posible_matricula(nota: real):boolean;
begin
    if nota = 10
    then
        result := TRUE
    else
        result := FALSE
end;

const
    nota_ejemplo: real = 9.5;
begin
    writeln( posible_matricula(nota_ejemplo))
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ej_comparacion_v2;

function posible_matricula(nota: real):boolean;
begin
    result := (nota = 10); // Devuelve un booleano
end;

const
    nota_ejemplo: real = 9.5;
begin
    writeln( posible_matricula(nota_ejemplo))
end.
```

2.11. Representación de los números reales

Números reales: separador decimal

Para separar la parte decimal de la parte fraccionaria de un número real...

- En los países anglosajones se usa tradicionalmente el punto.
- En países latinoamericanos del norte de América (México, caribe) se usa tradicionalmente la coma.
- La tradición española coincide con la franco belga: la coma.
- En el año 2010, la Real Academia Española cambia de criterio y recomienda usar el punto y no la coma, aunque ambas opciones siguen siendo correctas.
- Los lenguajes de programación siempre usan el punto.
- Las hojas de cálculo usan el punto o la coma según como esté configurado el idioma en esta aplicación.

En resumen: es un asunto al que hay que prestar atención.

Hay una costumbre tradicional que actualmente desaconsejan todas las normativas y recomendaciones internacionales.

- Usar la coma para separar grupos de tres dígitos cuando el punto separe decimales.

  3,000.150 *tres mil punto ciento cincuenta* ¡Mal!

- Usar el punto para separar grupos de tres dígitos cuando la coma separe decimales.

  3.000,150 *tres mil coma ciento cincuenta* ¡Mal!

Lo que se recomienda en este caso es usar espacios, salvo que puedan inducir a confusión.

3 000 000 *tres millones*

Representación de los números reales

La representación convencional es poco adecuada para ciertos números reales. En matemáticas en general (no solo en programación) se usa entonces la *notación científica*.

- Los números reales tienen un numero de decimales potencialmente infinito.

- Pueden ser muy grandes, con un número de dígitos muy elevado.
- Pueden ser muy pequeños, con muchos ceros tras la coma.

Un número real r en notación científica se representa:
$r = c \times b^e$
c es el *coeficiente*, b es la base y e es el *exponente*.
Ejemplos: 6.022×10^{23} 1×10^{-8}
Al coeficiente comunmente se le llama *mantisa*, aunque en rigor esta denominación es incorrecta [10].

Notación de ingeniería

La notación de ingeniería es un caso particular de notación científica, donde el exponente ha de ser múltiplo de 3.
Ejemplo: 10.3×10^3
Cada uno de estos exponentes tiene un nombre, todo ello facilita la lectura.

Factor	Prefijo	Símbolo
10^{-15}	femto	f
10^{-12}	pico	p
10^{-9}	nano	n
10^{-6}	micro	µ
10^{-3}	mili	m
10^{3}	kilo	K
10^{6}	mega	M
10^{9}	giga	G
10^{12}	tera	T
10^{15}	peta	P

Coma flotante

Los ordenadores usan un caso particular de notación científica, la *representación de coma flotante* (o *coma flotante, floating point*) siguiendo el estándar IEEE 754 (año 1985).
Incluye 5 formatos básicos: tres de base dos y dos de base diez. Los más usados son:

- *binary32* (binario, simple precisión)

  Se corresponde con el tipo *real* de Free Pascal. Es el único que usaremos en esta asignatura, lo habitual en entornos no especialmente exigentes.

[10] mantisa es la parte decimal de un logaritmo.

- binary64 (binario, doble precisión).

  Se corresponde con el tipo *double* de Free Pascal.

Cada formato ocupa diferente espacio en memoria, con diferente tamaño del coeficiente y del exponente. Por tanto, cada formato tendrá distinto.

- Rango: valores mínimo y máximo representables.
- Resolución numérica: valor mínimo, no negativo, no nulo.
- Cifras signiticativas: dígitos que aporten información.

  https://es.wikipedia.org/wiki/Cifras_significativas

  Puedes consultar las características de cada formato en https://es.wikipedia.org/wiki/IEEE_754

Errores de conversión

Tomemos por ejemplo el formato real más habitual de muchos lenguajes: IEEE 754 *binary32*. Admite un exponente máximo de 8 bits, lo que equivale a 38.23 dígitos binarios.

- Podría pensarse que esto permite un rango muy grande y una precisión muy alta, que sería raro que nuestro problema requiera de más precisión.
- Pero hay otro aspecto importante a considerar: los errores de conversión binario / decimal.
  - Los humanos casi siempre usamos notación decimal, los ordenadores siempre usan internamente notación binaria, esto obliga a continuas conversiones que provocan errores.
  - Usando *binary32*, es fácil encontrar errores a partir del quinto decimal, como puedes ver en este conversor online: https://www.h-schmidt.net/FloatConverter/IEEE754.html

Por todo ello, en aquellas situaciones donde necesitemos una precisión superior a unos pocos decimales, tendremos que prestar atención a los tipos de datos reales de nuestros programas.

- Esto no solo depende del lenguaje, también del dialecto, del compilador e incluso de la arquitetura (CPU) concreta de cada caso.
- Por ejemplo en Free Pascal los tipos de real soportados son *real, single, double, extended, comp, currency.*

  https://www.freepascal.org/docs-html/ref/refsu5.html

Operadores booleanos

- `op1 and op2`

  Devuelve TRUE cuando ambos operandos son ciertos.

  Devuelve FALSE en otro caso.

- `op1 or op2`

  Devuelve TRUE cuando un operando es cierto o cuando dos operandos son ciertos.

  Devuelve FALSE en otro caso.

- `not op`

  Devuelve TRUE cuando el operando es FALSE.

  Devuelve FALSE cuando el operando es TRUE.

- `op1 xor op2`

  Devuelve TRUE cuand un operando es TRUE y otro es FALSE.

  Devuelve FALSE en otro caso.

  equivale a

  `(op1 and (not op2)) or ((not op1) and op2)`

Expresiones booleanas

- Ya estás familiarizado con las expresiones numéricas, que combinan operandos numéricos con operadores numéricos. P.e

```
1.23 + (2.4 * 12)
```

- En programación se usan mucho, además, las expresiones booleanas. Los operandos son booleanos y, por supuesto, también los operadores.

```
FALSE or not (TRUE and FALSE)
Diabetico and not Menor_edad
```

- Los operadores booleanos solo admiten operandos booleanos.

```
Diabetico and 20   //  ¡¡Mal!!
```

- Los operadores de comparación aceptan enteros o reales como operandos, y devuelven un booleano, con el que ya podemos construir expresiones booleana.

```
Diabetico and (Edad >= 18)
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program franja;

const
    Cota_inferior = -100;
    Cota_superior = 100;
    Valor = 20;
begin
    // writeln( Cota_inferior <= Valor <= Cota_superior);
    // ¡MAL! En matemáticas escribimos a<b<c. Aquí no podemos

    // writeln( Cota_inferior <= Valor and Valor <= Cota_superior);
    // ¡MAL!  and es un operador booleano, pero Valor y Valor son
    // enteros. Es necesario usar paréntesis

    writeln( (Cota_inferior <= Valor) and (Valor <= Cota_superior));
    // Correcto
end.
```

Lógica proposicional

Doble negación:

- $\neg\neg p \Leftrightarrow p$
- No es cierto que no llueva $\Leftrightarrow$ Llueve

Ambas expresiones son lógicamente equivalentes, aunque la segunda es más clara.

En lenguaje natural, es habitual usar expresiones del tipo *si... entonces*.

- *Si me avisas, entonces llevo más dinero.*
- *Si suspendes las prácticas, entonces suspendes la asignatura.*

En la especificación de un algoritmo hay que tener mucho cuidado, porque en rigor, con esta estructura estamos diciendo qué sucede si se cumple la condición, pero no estamos diciendo qué pasa si no se cumple.

- En algunos casos, posiblemente los humanos supongan que si no se cumple la condición, no se cumple la consecuencia.

  Si no me avisas, entonces no llevo más dinero.

- En otros, posiblemente no haremos esa suposición.

  Si apruebas las prácticas, ya veremos, dependes de los exámenes.

Estas imprecisiones propias del lenguaje natural no son admisibles en la especificación de un algoritmo, es importante dejar claro qué pasa si la condición es falsa: si estoy diciendo algo para ese caso o si no estoy diciendo nada.

Si (condición) entonces (consecuencia)

¿Qué pasa si no se cumple la condición?

- Caso 1. Equivalencia.

  Entonces tampoco se cumple la consecuencia.

- Caso 2. Implicación.

  Entonces no digo nada sobre consecuencia, puede que se cumpla, puede que no.

En lenguaje natural tenemos estas dos posibilidades, en lenguaje formal, solo la segunda.

Caso 1. Equivalencia

- *Si me avisas, entonces llevo más dinero. Y si no, no.*

  Esto se convierte en una equivalencia lógica. El aviso equivale a llevar más dinero.

  $aviso \implies mas_dinero \wedge \neg aviso \implies \neg mas_dinero$

  $aviso \Leftrightarrow mas_dinero$

- $p \implies q \wedge \neg p \implies \neg q$

  $p \Leftrightarrow q$

Caso 2. Implicación.

- *Si suspendes las prácticas, entonces suspendes la asignatura. Y si apruebas las prácticas, ya veremos.*

  $suspender_practicas \implies suspender_asignatura$

  (Y ya está, en un entorno formal está claro que no estoy haciendo ninguna afirmación si no se da la condición).

- $p \implies q$

Ojo con sacar conclusiones erróneas

- $p \implies q$

  ¿Equivale a ?

  $\neg p \implies \neg q$

  ¡No!

Con otras palabras, es lo mismo que acabamos de ver. De la afirmación *si me avisas, entonces llevo más dinero*, no se puede deducir que si no me avisas, no lo llevo. Es necesario indicarlo explícitamente (si procede).

Transposición de la implicación

La conclusión que sí podemos extraer es

- $p \implies q \Leftrightarrow \neg q \implies \neg p$

- *ser asturiano implica ser español*

  equivale a

  no ser español implica no ser asturiano.

Otro ejemplo

- Si he venido es porque no lo sabía.

- Si lo se no vengo.

Un poco más claro, en presente:

- Si voy es porque no lo se.

- Si lo se, no voy.

Leyes de De Morgan

Las leyes de De Morgan[11] también permiten generar expresiones booleanas equivalentes desde el punto de vista lógico.

- `not (a and b)`

  equivale a

  `(not a) or (not b)`

- `not (a or b)`

  equivale a

  `(not a) and (not b)`

  Ejemplos:
  `not (joven and rico)` ⇔ `not joven or not rico`
  `not (anciano or niño)` ⇔ `not anciano and not niño`

Aplicando la doble negación y las leyes de De Morgan, podemos escribir las expresiones booleanas de formas distintas, que

- Desde el punto de vista lógico y mátemático, serán equivalentes.
- Considerando la claridad para el humano, no serán equivalentes. Las personas entedemos mejor la *lógica positiva* (afirmaciones sin negaciones) que la *lógica negativa* (afirmaciones con negaciones).

  En programación, normalmente lo más importante es el programador, presente o futuro. En general deberemos usar la expresión equivalente con menos negaciones.

Aplicando la transposición de la implicación, podemos mejorar la claridad de las implicaciones en lógica booleana.

Ejemplos

- `Niega la siguiente expresión y simplifica el resultado`

  `p = not q and (not r or s )`

  `Solución:`

  `not p = not ( not q and (not r or s ) )`

[11] No es una errata, el nombre de su descubridor es Augustus De Morgan

```
not p =  q or not (not r or s)

not p =  q or (r and not s)
```

- ```
  Niega la siguiente expresión y simplifica el resultado

  p = (a >= 65) or (a <= 16) or q

  Solución:

  not p = (a < 65) and (a > 16) and not q
  ```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program diabetes;

const
    Diabetico = TRUE;
    Menor_edad = FALSE;
    Edad = 20;
begin
    writeln(FALSE or not (TRUE and FALSE));  // Escribe  TRUE
    writeln(Diabetico and not Menor_edad);  // Escribe TRUE

    { writeln(Diabetico and not Edad);    ¡¡ Esto es un error !!  }
    { writeln(Diabetico and not Edad < 18); ¡¡ Esto es un error !! }

    writeln(Diabetico and not (Edad < 18)); // Escribe TRUE
    writeln(Diabetico and (Edad >= 18)); // Escribe TRUE
end.
```

Ejemplo: años bisiestos

- Descripción en lenguaje natural:

  ```
  Los años múltiplos de 4 son bisiestos.
  Excepción: los múltiplos de 100, que no lo son.
  Excepción a la excepción: los múltiplos de 400 sí lo son.
  ```

  En otras palabras
  ```

```
Los años múltiplos de 4 son bisiestos.
Excepción: múltiplo de 100 y no múltiplo de 400.
```

En otras palabras

```
Un año es bisiesto si es múltiplo de 4 y no se cumple que:
es múltiplo de 100 y no es múltiplo de 400.
```

- Aplicando De Morgan, llegamos a la descripción algorítmica

```
Un año es bisiesto si es múltiplo de 4 y
(no es múltiplo de 100 o es múltiplo de 400).
```

Implementación en Pascal:

```
(A mod 4 = 0 ) and   (not (A mod 100 = 0)   or  (A mod 400 = 0) )
```

Implementación en Pascal, un poco más clara:

```
(A mod 4 = 0 ) and   ( (A mod 100 <> 0)   or  (A mod 400 = 0) )
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program bisiesto ;
const
    Anyo = 1940;

begin
    write(Anyo, ' es bisiesto:');
    writeln(
        (Anyo mod 4 = 0 )
        and
        ( (Anyo mod 100 <> 0)  or  (Anyo mod 400 = 0 ))
    );
end.
```

Ejercicio

1. Escribe en una hoja de papel un par de expresiones booleanas parecidas a las de la pg 63.

   - Una equivalencia, con la constante p a la izquierda de la igualdad y las constantes q, r y s a la derecha. Con los operadores and, or, algún not y algún paréntesis.
   - Otra equivalencia con la constante p a la izquierda, valores numéricos y constantes a la derecha. Con operadores de comparación, and y or.
   - Usa la notación de Pascal (and, or, not), no uses notación lógica ($\wedge, \vee, \neg$).

2. Escribe en otra hoja las expresiones booleanas equivalentes, negando ambos lados de la igualdad.

3. Entrega la primera hoja a un compañero para haga lo mismo. Resuelve tú su ejercicio. Comparad las soluciones para comprobar que sean iguales, corregid el problema si hay errores.

Precedencia de operadores

En matemáticas normalmente podemos distribuir los operandos entre varias líneas, haciendo cosas como $\frac{4+2}{1+1} = 3$.

- En casi todos los lenguajes de programación, nos vemos obligados a usar una sola línea.
- Si intentamos escribir la expresión anterior como $4+2/1+1$ estaremos cometiendo un error, porque el compilador lo interpreta como $4 + \frac{2}{1} + 1 = 7$.

Los operadores tienen una *reglas de precedencia*:

- Los operadores se evalúan de mayor precedecia a menor precedencia.
- A igualdad de precedencia, se evalúa de izquierda a derecha.

Precedencia en Pascal, de mayor a menor:

```
**  not  - (cambio signo)
* / div mod and
or xor + - (resta)
```

En un programa la claridad es fundamental, así que debemos usar paréntesis. Incluso es recomendable hacerlo en los casos en los que, por la precedencia de los operadores, no sería necesario.

Ejemplo

- $\frac{4+2}{1+1}$

  lo escribimos

  $(4 + 2)/(1 + 1)$.

- Si quisiéramos escribir

  $4 + \frac{2}{1} + 1$

  bastaría $4 + 2/1 + 1$.

  Pero es preferible ser muy claro:

  $4 + (2/1) + 1$.

Elementos predefinidos

Como en prácticamente cualquier lenguaje, en Pascal hay *elementos predefinidos*: funciones, operaciones y constantes definidos inicialmente en el lenguaje. No podemos declarar nuevos identificadores que usen estos nombres.

```
abs(n)      valor absoluto
trunc(n)    truncar a entero
round(n)    redondear a entero
sqr(n)      elevar al cuadrado
sqrt(n)     raíz cuadrada
chr(i)      carácter en posición i
ord(c)      posición del carácter o valor c
pred(c)     carácter o valor predecesor de c
succ(c)     carácter o valor sucesor de c
arctan(n)   arcotangente
cos(n)      coseno
exp(n)      exponencial
ln(n)       logaritmo neperiano
sin(n)      seno
low(x)      menor valor o índice en x
high(x)     mayor valor o índice en x
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program predefinidos;
begin
```

```
    writeln( abs(-3) );       // Escribe 3
    writeln( trunc(3.64) );   // Escribe 3
    writeln( round(3.64) );   // Escribe 4
    writeln( sqr(3) );        // Escribe 9
    writeln( sqrt(9) );       // Escribe 3e0
    writeln( succ('a') );     // Escribe b
    writeln( pred('B') );     // Escribe A
end.
```

3. Funciones

3.1. Definición de problemas

Definición de problemas

La definición de un problema consiste en tres cosas:

- Saber qué resolvemos.
- Saber qué nos hace falta para resolverlo.
- Saber qué tendremos cuando lo hayamos resuelto.

Esto coincide con la declaración de una función.

- Cómo se llama la función.
- Qué necesita.
- Qué devuelve.

Funciones en pascal

Una función básica en Pascal tendrá la siguiente estructura:

```
function NOMBRE_FUNCION( LISTA_DE_PARAMETROS): TIPO_DEVUELTO;
begin
    result := EXPRESION_CON_LA_SOLUCION;
end;
```

- La lista de parámetros normalmente estará formado por 1 o más parámetros. Pero también puede ser una lista vacía, sin parámetros.
- Observa que para asignar valor a *result* empleamos el operador de asignación `:=` [12], mientras que para dar valor a las constante usamos `=` [13].

Ejemplo:

```
function long_circunf(r: real): real;
begin
    result := 2.0 * Pi * r;
end;
```

[12] El mismo que usaremos para las variables, como veremos en el tema 5.

[13] Esta es una rareza de Pascal, en otros lenguajes no hay esta diferencia.

`result` es una palabra reservada que en nuestro dialecto de Pascal (Object Pascal) emplemos para darle valor a la función.

- En Pascal estándar usaríamos el mismo nombre de la función, pero esto tiene algunos inconvenientes.

```
function long_circunf(r: real): real;
begin
    long_circunf := 2.0 * Pi * r;
end;
```

Luego llamaremos a la función con un valor concreto p.e.

```
long_circunf(4.2);
```

En este caso

- 4.2 es el argumento, el valor concreto que suministramos a la función.
- r es el parámetro, el identificador que usamos para nombrar al argumento.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program longitud_circunferencia;

const
    Pi = 3.14159265;

function long_circunf(r: real): real;
begin
    result := 2.0 * Pi * r;
end;

begin
    writeln( long_circunf(4.2):0:3 );  // Escribe 26.389
end.
```

Observa que hemos declarado Pi al principio del programa, por ser una constante *universal*

Una forma alternativa de escribir el programa anterior es pasar la constante como un parámetro más de la función.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program longitud_circunferencia_v2;

const
    Pi = 3.14159265;

function long_circunf(r: real; pi: real): real;
begin
    result := 2.0 * pi * r;
end;

begin
    writeln( long_circunf(4.2, Pi):0:3 );   // Escribe 26.389
end.
```

- En este ejemplo, como Pi es una constante universalmente conocida, ambas soluciones serian aceptables. Un purista de la *programación funcional* posiblemente preferirá esta segunda solución, pero es discutible.
- Si la constante no es universalmente conocida y obvia como esta, es preferible este segundo enfoque: pasarla como parámetro.
- En caso de duda, también es preferible pasar la constante como parámetro de la función.

Importante: observa que la función *long_ circunf* no utiliza la constante `Pi` (con mayúscula) sino el parámetro `pi` (con minúscula).

Una función en Pascal bien escrita es muy similar a una función matemática .

- Acepta uno o más parámetros.
- Devuelve un valor, calculado a partir de los parámetros (y de ninguna otra cosa).

Y nada más.

- NO lee ningún otro valor del exterior, solo sus parámetros.
- NO depende del estado del programa.
- NO depende del orden de evaluación de los parámetros.
- NO provoca ningún cambio en ninguna parte, solo devuelve su valor.
    - Esto incluye NO escribir nada en pantalla.

A esto se llama *transparencia referencial*.

Así, no hay ninguna diferencia entre

- Llamar a una función, con sus argumentos.
- Escribir el valor devuelto por la función.

Ejemplo. Tenemos un programa con la función

```
function incrementa(x: integer): integer;
begin
    result := x + 1;
end;
```

Luego llamamos a esta función

```
incrementa(3);
```

Si cumplimos la propiedad de transparencia referencial (que en este curso es obligatorio), nuestro programa se comportará exactamente igual en todo si, en vez de escribir esta llamada, escribimos

```
4;
```

Se denomina *efecto lateral* a cualquier acción producida por una función. Por tanto,

Estas dos cosas son equivalentes:

- Una función tiene transparencia referencial.
- Una función no tiene efectos laterales.

Podemos decir que una función con efectos laterales *hace lo que no debe*.

- El compilador no lo va a impedir, no se va a generar ningún error. Pero eso es código de muy mala calidad, propenso a errores (aunque lo podemos ver incluso en algunos libros de programación).
- En este curso no permitiremos funciones con efectos laterales.

Ejemplo de función con varios parámetros:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program media_v1;

function media(x: real; y:real): real;
begin
    result :=  (x + y) / 2;
end;

begin  // Aquí empieza el cuerpo principal
    writeln( media(4, 5) );
end.
```

Observa que writeln está en el cuerpo principal, nunca debe estar dentro de la función (porque eso sería un efecto lateral).

Otras consideraciones sobre las funciones

- En Pascal es necesario que la definición de una función aparezca en el código fuente antes que la llamada a esa función.
- Si una función tiene varios parámetros del mismo tipo, también podemos declararlos así:

  ```
  function media(x, y: real): real;
  begin
      result :=  (x + y) / 2;
  end;
  ```

- En nuestro dialecto de Pascal, una función no puede devolver objetos *demasiado grandes* [14].

[14] La solución la veremos en el tema 5: será necesario usar un procedimiento y un parámetro de entrada-salida (parámetro por referencia).

Una función devuelve una expresión. En cualquier lugar donde se espera una expresión, se puede escribir una función.

```
writeln( media(4, 5) + 1 );    // Escribe 5.5
```

En la llamada a una función, el argumento puede ser cualquier expresión: un literal, una constante, otra función...

```
writeln( media(2 + 2, 3 + 2) + 1 );
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program media_v3;

function media(x, y: real): real;
begin
    result :=  (x + y) / 2.0;
end;

begin
    writeln( media(4, 5)+1) ;      // Escribe  5.5000000000000000E+000

    writeln( (media(4, 5)+1):0:3 );   // Escribe  5.500

    writeln( media(2+2, 3+2) + 1); // Escribe  5.5000000000000000E+000
end.
```

De nuevo, todos los writeln están fuera de la función. Lo contrario sería un defecto muy severo.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program llamadas_funcion;

function suma(x, y: integer): integer;
begin
```

```
    result :=  x + y;
end;

function incrementa(x: integer): integer;
begin
    result := x + 1;
end;

const
    A = 4;    // Estas constantes no son universales,
    B = 1;    // son ejemplos particulares. Por eso
              // van aquí y no al principio.

begin
    writeln( suma(9, 2) );              // Escribe 11
    writeln( suma(A, B) );              // Escribe 5
    writeln( suma(A, incrementa(B))) ; // Escribe 6
end.
```

Independencia argumento - parámetro

El nombre de la constante que usamos como argumento no tiene relación que el nombre del parámetro. Pueden coincidir, pero en general no tienen por qué.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program nombres_parametros;

function media(x, y: real): real;
begin
    result := (x+y) / 2;
end;

const
    A = 2.0;   // Constantes locales al cuerpo del
    B = 1.0;   // programa principal

begin
    writeln( media(A,B) );
end.
```

En el ejemplo anterior, *nombres_parametros*

- En el cuerpo del programa principal, para poder llamar a *writeln*, el compilador evalúa la expresión *media(A,B)*. Para ello, llama a la función *media* pasando como argumentos 2.0 y 1.0.
- La función *media* recibe los argumentos 2.0 y 1.0. Ignora por completo si estos valores eran constantes numéricas literales (números *tal cual*), o las constantes A y B, o son valores que se han calculado desde otra expresión, o son el resultado de una llamada a una función...

  La función solo *sabe* que el parámetro x vale 2.0 y que el parámetro y vale 1.0. Cualquier otra información no es relevante.

El siguiente ejemplo provocará un error de compilación: intenta usar las constantes A y B en la función *media*. Pero estas constantes solo se pueden usar en el cuerpo del programa principal.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program nombres_parametros_mal;

function media(x, y: real): real;
begin
        result := (A+B) / 2 ;  // ¡MAL!
end;

const
        A = 2.0;
        B = 1.0;

begin
        writeln( media(A,B) );
end.
```

```
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling nombres_parametros_mal.pas
nombres_parametros_mal.pas(7,13) Error: Identifier not found "A"
nombres_parametros_mal.pas(7,15) Error: Identifier not found "B"
nombres_parametros_mal.pas(17) Fatal: There were 2 errors compiling module,
   stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

Otro ejemplo incorrecto

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program nombres_parametros_mal_2;

const
    A = 2.0;
    B = 1.0;    // ¡MAL!

function media(x, y: real): real;
begin
    result := (x+y) / 2;
end;

begin
    writeln( media(A,B) );
end.
```

Funciona pero está mal diseñado, porque A y B son constantes globales. Deberían estar declaradas justo antes del cuerpo del programa principal para ser constantes locales.

Reutilización de nombres de parámetro

A continuación veremos un programa donde cierta función utiliza los nombres de parámetro x, y y otra función vuelve a emplear los mismos nombres.

- Esto es perfectamente correcto.
- Los parámetros solo se consideran dentro de la función donde se declaran. En programación decimos que *el ámbito de los parámetros es local.*
- Si otra función usa los mismos nombres, o no, es irrelevante.

Metáfora: una casa donde se usa la palabra *mamá* y esta identifica a una y solo una persona. En otro ámbito diferente, p.e. la casa de los vecinos, *mamá* será una persona distinta, pero esto no genera ningún problema.

El identificador debe ser único en su ámbito, pero en un programa normalmente habrá muchos ámbitos diferentes.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program reutilizacion_parametros;
function media(x, y: real): real;
begin
        result := (x+y) / 2;
end;

function suma(x, y: real): real;
begin
        result := x + y;
end;

const
        A = 2.0;
        B = 1.0;
        C = 4.0;
        D = 3.0;

begin
    writeln( media(A,B):0:1 );
    writeln( suma(C,D):0:1 );
end.
```

Importante:

- En ejemplos de funciones sencillas como estas, es tradicional usar nombres de una sola letra como x,y o tal vez a,b porque reslta obvio que se refieren al primer sumando y al segundo sumando. O al primer y segundo número de una media aritmética de dos números.
- Pero cuando las funciones no sean triviales, es necesario usar nombres de parámetro más descriptivos, como p.e *dni_ alumno*, *masa_ vehiculo*, *nota_ corte*, etc etc

Los ejemplos anteriores tenían una salida completamente *minimalista*, pero normalmente será conveniente que el programa indique qué esta calculando, con qué parámetros de entrada, así como las unidades, si procede.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program pulgada_cm;

Const
    Factor_pulg_cm = 2.54;  // Constante global, universal

function pulgada_a_cm(pulgadas: real): real;
begin
        result :=  pulgadas * Factor_pulg_cm;
end;

Const
    X = 11.0;  // Constante local

begin
        write( X, ' pulgadas son ');
        writeln( pulgada_a_cm(X):0:2, ' cm');
end.
```

El siguiente ejemplo funciona pero está mal. Copia el valor de X en la sentencia *write*, 11.0 en este caso. Si cambiásemos la constante, habría que cambiar también el texto, lo que es inaceptable.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program pulgada_cm_mal;

Const
    Factor_pulg_cm = 2.54;  // Constante global, universal

function pulgada_a_cm(pulgadas: real): real;
begin
        result :=  pulgadas * Factor_pulg_cm;
end;

Const
    X = 11.0;  // Constante local

begin
```

```
        write('11.0 pulgadas son ');  //¡¡ MAL !!
        writeln( pulgada_a_cm(X):0:2, ' cm');
end.
```

3.2. Diseño de programas

Diseño de programas

Los problemas ser resuelven dividiéndolos en subproblemas, que serán subprogramas.

Lo habitual es:

1. Primero, diseñar la solución *top-down* (arriba-abajo).

   Empezamos por el problema global, suponiendo que tenemos resueltos los subproblemas. Aplicamos esto las veces necesarias, hasta que tengamos un problema con solución directa.

2. Después, programar esa solución, *bottom-up* (abajo-arriba) .

   Programamos los subproblemas, empezando por los más sencillos. Probamos cada subprograma, con valores de ejemplo. Cuando estemos razonablemente seguros de que funciona, programamos un subprograma más complejo.

En Pascal, los subproblemas serán las funciones y los procedimientos (que veremos más adelante).

- Cualquier concepto de cierta entidad de nuestro programa deberá ser una función o procedimiento, que tendrá sentido por si mismo y que deberá ser verificado.
- El cuerpo del programa principal no debería calcular ni procesar nada, solo llamar a los subprogramas de nivel más alto.
- Los subprogramas deben ser *pequeños* ¿cuánto es pequeño? Depende.
  - Una función o procedimiento de 1 línea es perfectamente normal (si tiene entidad y sentido).
  - 5, 10, 15 líneas son valores habituales.
  - Una función o procedimiento que no cabe en una pantalla (de resolución media) debe levantarnos sospechas.

  Por supuesto, no basta con que el subprograma sea corto para poder decir que está bien escrito.

Problema: código repetido

Es fundamental que el código necesario para resolver un subproblema se escriba 1 vez y solo 1.

- Al mal programador *le da pereza* escribir la función adecuada y *copia y pega* las lineas que necesita. Este es uno de los defectos principales en los malos programas.
- Modificar unas líneas repetidas n veces obliga a hacer n modificaciones. Una función bien escrita basta con modificarla 1 vez.
  - Cualquier programa real tendrá muchas modificaciones.
- Una función bien escrita se prueba n veces. Si el código está repetido m veces, habría que probarlo $n\ x\ m$ veces.
- En un programa nunca debe haber código igual o similar repetido: se debe escribir un único subprograma, particularizado en cada caso con los parámetros necesarios.

Pruebas

Es prácticamente imprescindible probar cada uno de nuestros subprogramas por separado.

- Esta es una de las principales diferencias entre un buen programador y uno malo.
- Los principiantes suelen escribir funciones grandes y complejas, un amasijo de líneas que:
  - El autor, cuando las escribe, cree entender. Más o menos.
  - Una persona distinta, incluyendo el autor cierto tiempo después, no entenderá.

  Un código que no se entiende es imposible de probar ni corregir.

Probar un programa es una disciplina en sí misma. En entornos rigurosos:

- Se usan herramientas automáticas que comprueban cada subprograma de forma sistemática, con grandes conjuntos de parámetros de entrada.

- Las pruebas las hacen personas distintas, incluso en lenguajes distintos. Esto no exime al programador de hacer sus propias pruebas, son pruebas adicionales.
- Cada vez que se añade o modifica algo, se repiten todas las pruebas, completas, desde cero.

En entornos menos exigentes deberemos, al menos:

- Hacer varias llamadas a cada una de nuestras funciones de forma independiente, con parámetros controlado y *escritos a mano*, observando la salida.
- Probar algunos parámetros con valores *bajos*, algunos *medios* y algunos *altos*, prestando atención a los extremos que suelen ser los conflictivos.

3.3. Ejemplos de uso de funciones

Volumen de un cilindro hueco

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program cilindro_hueco;
const      // Constantes globales, universales
    Pi = 3.1415926;  // Observa que usamos =, no :=

function area_circulo(r: real): real;
begin
    result := Pi*sqr(r);
end;

function area_corona(r_interior: real; r_exterior: real): real;
begin
    result := area_circulo(r_exterior) - area_circulo(r_interior);
end;

function vol_cil_hueco(r_interior, r_exterior, alt: real): real;
begin
    result := area_corona(r_interior, r_exterior) * alt;
end;

const    // Constantes del cuerpo del programa
    Rmin = 3.2;
```

```
    Rmax = 4.3;
    Alt = 2.8;

begin
    //writeln( area_circulo(1));
    //writeln( area_circulo(0.1));
    //writeln( area_circulo(0));
    //writeln( area_circulo(100000));

    //writeln( area_corona( 10, 10));
    //writeln( area_corona( 0, 0));
    //writeln( area_corona( 0.1, 0.1));

    writeln('v = ', vol_cil_hueco(Rmin, Rmax, Alt) :0:3);
end.
```

Letra o número ASCII

Escribamos un programa que :

- Reciba un código ASCII en binario, como 7 números enteros con valor 0 o 1.
- Devuelva TRUE si se trata de un número o una letra, devuelva FALSE en otro caso.

Subproblemas que tendremos que resolver:

- Convertir un número binario en decimal.
- Saber si un código ascii se corresponde con un dígito.
- Saber si un código ascii se corresponde con una mayúscula.
- Saber si un código ascii se corresponde con una minúscula.

ASCII significa *American Standard Code for Information Interchange*, por tanto se trata siempre de letras inglesas.

¿Cómo convertir un número binario en decimal?

Supongamos 4 dígitos binarios: d1, d2, d3, d4.

- valor_decimal $= d1 * 2^3 + d2 * 2^2 + d3 * 2^1 + d4 * 2^0$

  Ejemplo: 1100 = 1*8 + 1*4 + 0*2 + 0*1 = 12

Observa que el valor de un número decimal se calcula de forma análoga:

- $6320 = 6*1000+3*100+2*10+0*1 = d1*10^3+d2*10^2+d3*10^1+d4*10^0$

```
function binario_a_decimal(d1,d2,d3,d4,d5,d6,d7 : integer): integer;
// Recibe 7 dígitos correspondientes a un número binario, lo convierte
// en decimal
begin
    result := d1 * 2 ** 6 +
              d2 * 2 ** 5 +
              d3 * 2 ** 4 +
              d4 * 2 ** 3 +
              d5 * 2 ** 2 +
              d6 * 2 ** 1 +
              d7 * 2 ** 0 ;
end;
```

Para saber si es dígito, mayúscula o minúscula:

```
function es_digito_ascii( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con un  dígito
begin
    result :=  ( x >= ord('0'))   and   (x <= ord('9'));
end;

function es_minuscula( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con letra minúscula (inglesa)
begin
    result :=  ( x >= ord('a'))   and   (x <= ord('z'));
end;

function es_mayuscula( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con letra mayúscula (inglesa)
begin
    result :=  ( x >= ord('A'))   and   (x <= ord('Z'));
end;
```

Para saber si es número o letra:

```
function es_numero_o_letra( x: integer): boolean;
// Recibe un código ascii como número entero, indica si se corresponde
// con un digito o con una letra (inglesa)
begin
    result :=
        es_digito_ascii(x) or es_minuscula(x) or es_mayuscula(x);
end;
```

Ejemplo completo: https://gsyc.urjc.es/ mortuno/fpi/letra_o_numero.pas

4. Selección

4.1. Problemas de selección

Problemas de selección

Hasta ahora hemos visto problemas de solución directa:

- En el tema 2, la solución al problema era una expresión, numérica o booleana, construida como una secuencia operandos y operadores, donde los operandos eran o constantes o funciones predefinidas.
- En el tema 3, aprendimos a usar funciones, pero la solución seguía siendo una única expresión, con la novedad de que los operadores podían ser no solo constantes y funciones predefinidas, sino funciones definidas por nosotros.
- En este tema, veremos las *sentencias de control*, que permiten como seleccionar una expresión u otra, o ejecutar una acción u otra, a partir de cierta condición booleana.

  En Pascal se hace mediante las sentencias *if-then-else* y *case*.

4.2. if then else

if then else

if *condición* then
  sentencia/s a ejecutar si la condición es cierta
else
  sentencia/s a ejecutar si la condición es falsa

- La condición puede ser cualquier expresión: una combinación de constantes, funciones, variables...
- En la rama *then* o en la rama *else* puede haber:
  - Una única sentencia.
  - Una lista de sentencias, dentro de un bloque *begin end*.
- La sentencia *if then else* , como cualquier otra, se puede escribir escribir tanto en el cuerpo del programa principal como en el cuerpo de un subprograma.

```
if mi_expresion then
    writeln('Bla Bla')  // Correcto
else
    writeln('Bla Bla Blá');
```

MUY IMPORTANTE

Es una única sentencia, en Pascal nunca se pone ';' antes del else. Este es un error de programación muy común (Pascal tiene pocas *rarezas* pero tal vez esta sea una).

```
if mi_expresion then
    writeln('Bla Bla');  // ¡¡MAL!! (sobra el punto y coma)
else
    writeln('Bla Bla Blá');
```

Si en la *rama then* o en la *rama else*, o en ambas, es necesario ejecutar más de una sentencia, se declara un bloque *begin end*.

```
if mi_expresion then begin
    write('Bla ');
    writeln('Bla Bla');
end   // Atención, antes del else, nunca se escribe punto y coma
else begin
    write('Bla Bla ');
    writeln('Blá');
end;
```

- Observa que nuestro convenio es que las sentencias tengan exactamente un nivel más de sangrado que *if, end* y *else*.

En Pascal el punto y coma no se usa para finalizar sentencias, sino para delimitarlas. Por tanto, la última sentencia no necesita punto y coma. Aunque es más sencillo ponerla siempre, como haremos en esta asignatura. En este caso el compilador entiende que al final del bloque hay una sentencia nula.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program if_then_else;

const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;
```

```
begin
    if Temperatura >= Limite_fiebre then
        writeln('Fiebre')
    else
        writeln('Temperatura normal');
end.
```

- En este curso, nuestro convenio será escribir la palabra reservada *if*, la condición y la palabra reservada *then* en una línea
- De esta forma, las sentencias a ejecutar tendrán exactamente 1 nivel de tabulación más que el *if* y el *else*.

Una vez más: recuerda que el carácter ';' separa sentencias. if-then-else es una única sentencia, no puede haber un ';' en medio[15].

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then_else_MAL;

const
    Limite_fiebre = 37.5;

var
    temperatura: real = 37.8;

begin
    if temperatura >= Limite_fiebre then
        writeln('Fiebre') ; // ¡¡MAL!! Sobra el ;
    else
        writeln('Temperatura normal');
end.
```

Es necesario que la tabulación sea consistente, pero también hay otros criterios posibles y habituales, p.e escribir *if*, *then* y *else* en la misma columna.

[15] Excepto dentro de un bloque begin-end, naturalmente.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then_else_v02;

// Otro convenio posible, aunque no lo seguiremos aquí

const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;

begin
    if Temperatura >= Limite_fiebre
    then
        writeln('Fiebre')
    else
        writeln('Temperatura normal');
end.
```

Otra posibilidad (discrepante con el convenio de este curso) escribir bloques begin-end siempre, aunque no haga falta.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then_else_v03;

// Otra posibilidad más: escribir siempre begin-end

const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;

begin
    if Temperatura >= Limite_fiebre then begin
        writeln('Fiebre');
    end
    else begin
        writeln('Temperatura normal');
    end; // El punto y coma tras el end puede omitirse
end.
```

Si la rama del *then* o la rama del *else* tienen más de una sentencia, entonces sí será necesario el uso de *begin end*.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then_else_v04;

const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;

begin
    if Temperatura >= Limite_fiebre then begin
        writeln('Fiebre');
        write('La temperatura es ');
        write(Temperatura-Limite_fiebre:0:1);
        writeln(' grados superior a lo normal');
    end
    else
        writeln('Temperatura normal');
end.
```

Ejecución:

```
Fiebre
La temperatura es 0.3 grados superior a lo normal
```

- Nuestro convenio es que si escribimos begin, irá en la misma línea de *if condición then*.
- Observa que las sentencias de la rama *then* aparecen con una tabulación adicional, igual que las de la rama *else*.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then_else_v05;
    // Tabulación discutible
const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;
```

```
begin
    if Temperatura >= Limite_fiebre
    then
        begin
            writeln('Fiebre');
            write('La temperatura es ');
            write(Temperatura-Limite_fiebre:0:1);
            writeln(' grados superior a lo normal');
        end
    else
        writeln('Temperatura normal');
end.
```

Este enfoque tiene el inconveniente de añadir un nivel de tabulación innecesario, es preferible tabular como en `if_then_else_v04`.

Números mágicos

- En un programa, una constante numérica literal (un número *tal cual*) solo puede estar en la definición de una constante, en ningún otro sitio.
- Poner una constante numérica literal en mitad del código es una mala práctica denominada *número mágico*. El compilador no lo impedirá, pero es un código defectuosos, potencialmente problemático.

Ejemplo incorrecto:

```
 if Temperatura >= 37.5 then
     writeln('Fiebre');
```

Ejemplo correcto:

```
const
    Limite_fiebre = 37.5;
begin
    if Temperatura >= Limite_fiebre then
        writeln('Fiebre');
end
```

Pero hay excepciones. Si a juicio del desarrollador hay algún número cuyo valor es muy obvio, puede ser conveniente escribir un número *tal cual*, sin definir una constante.

```
if n >= 0  then
   result := 'positivo'
else
   result := 'negativo';
```

Problema: omisión del begin-end

Es muy importante usar begin-end cuando una de las ramas tiene más de una sentencia.

- Si me olvido del begin-end en la rama *then*, no es un problema demasiado serio porque resulta un error de sintaxis, el *else* queda descolocado y el compilador me avisará con un error.

  ```
  Fatal: Syntax error, ";" expected but "ELSE" found
  ```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program error_olvido_begin_end_1;
function comprueba_positivo(x:integer):boolean;
begin
    if x >= 0 then   //  ¡¡Mal!!
        writeln('Entra en rama then');
        result := True
    else
        result := False;
end;
const
    Test = 3;
begin
    writeln(comprueba_positivo(Test));
end.
```

Recuerda que las funciones no deben tener efectos laterales. Pero en este caso el writeln puede ser aceptable, para una prueba, que borraremos.

Pero si olvido el begin-end en la rama else, el error es más severo, porque la sintaxis es correcta.

- Resultará un error lógico.

- El compilador considerará que la primera sentencia pertenece a la rama else, pero que ahí acaba la rama.
- El resto de sentencias aparentan ser correctas, se ejecutarán siempre, sin importar la condición del *if*.
- La tabulación, que resultará incorrecta, hace que el error pueda ser más difícil de localizar.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program error_olvido_begin_end_2;
function comprueba_positivo(x:integer):boolean;
begin
    if x >= 0 then
        result := True
    else
        writeln('Entra en rama else');
        result := False;    // ¡¡Mal!!
end;

const
    Test = 3;
begin
    writeln(comprueba_positivo(Test));
end.
```

El comportamiento (indeseado) del ejemplo anterior queda más claro si lo tabulamos de forma coherente con lo que hemos escrito (no lo que queríamos escribir).

```
function comprueba_positivo(x:integer):boolean;
begin
    if x >= 0 then
        result := True
    else
        writeln('Entra en rama else');

    result := False;  //  Se ejecuta siempre, está fuera
                      //  de la rama else
end;
```

En resumen: si la rama *then* o la rama *else* tienen más de una sentencia, es imprescindible escribir un bloque *begin end*.

- Si me olvido en la rama *then*, es malo, porque el compilador detectará un error.
- Si me olvido en la rama *else*, es peor, porque se producirá un error que el compilador no puede detectar.

Omisión del else

El else no es obligatorio, si nuestro algoritmo no ejecuta nada en caso de incumplimiento de la condición, lo omitimos.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_then;

const
    Limite_fiebre = 37.5;
    Temperatura = 37.8;

begin
    if Temperatura >= Limite_fiebre then
        writeln('Fiebre');
end.
```

If anidado

En la rama *then* puede haber otra sentencia *if-then-else*

Este ejemplo es correcto, aunque peligroso.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program anidacion_correcta;

const
    Mayoria_edad = 18;
    Adolescencia = 13;
    Edad = 14;

begin
    if Edad < Mayoria_edad then
        if Edad < adolescencia then
```

```
            writeln(Edad, ': Niño')
        else
            writeln(Edad, ': Adolescente');
end.
```

El *else* se corresponde con el *then* más próximo.
Ejecución:

```
14: Adolescente
```

El problema es que no queda del todo claro, si no conocemos bien el criterio de Pascal, o si nos descuidadmos, podemos pensar, erróneamente que el *else* se corresponde al primer *if*.

El error anterior es aún más severo (más probable) si el programa está mal tabulado:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program MALA_TABULACION;

const
    Mayoria_edad = 18;
    Adolescencia = 13;
    Edad = 14;

begin
    if Edad < Mayoria_edad then
        if Edad < Adolescencia then
            writeln(Edad, ': Niño')
    else    //  ¡¡Mal tabulado!!
        writeln(Edad, ': Adolescente');
end.
```

- En el ejemplo erróneo anterior, la tabulación nos haría pensar que se escribirá *adolescente* cuando $edad >= Mayoria_edad$.
- Como hemos dicho, lo que realmente hace el programa es escribir *adolescente* cuando $edad < Mayoria_edad$ y $edad >= adolescencia$.

- Para evitar este problema, una práctica recomendable que aquí estableceremos como convenio es que cuando en la rama *then* haya otra sentencia *if-then-else*, la *protegeremos* con un bloque *begin-end*.

```
if condicion1 then begin
    if condicion2 then
        sentencia1
    else
        sentencia2
end
else
    sentencia3
```

Observa que escribimos el *if*, el *end* y el *else* en la misma columna

.

Cuando el segundo *if-then-else* esté en la rama *else*:

- Tendremos una estructura muy habitual, llamada *if encadenados*.
- Entonces no añadiremos el bloque *begin-end* (a menos que sea necesario porque haya más de una sentencia).
- Ya no se da el problema potencial que hemos descrito del *else* ambiguo.

if encadenados

Los *if* encadenados, esto es, una sucesión de *else if*, *else if*, *else if*, son una estructura muy habitual.

- *Si se da cierta condición haz esto, y si no, haz lo otro, y si tampoco, esto otro, y si tampoco ...*

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program if_encadenado;

const
    Mayoria_edad = 18;
    Adolescencia = 13;
    Edad = 10;
```

```
begin
    if Edad >= Mayoria_edad then
        writeln(Edad, ': Adulto')
    else if Edad > Adolescencia then
        writeln(Edad, ': Adolescente')
    else
        writeln(Edad, ': Niño');
end.
```

Resultado:

```
10: Niño
```

if anidado

En vez de encadenar

```
if then
else if
else if
```

tambien podríamos anidar, esto es, añadir un *begin-end* después del else y meter dentro el if-then-else

```
if then
else begin
   if ... then .. else
end
```

Pero siempre es preferible encadenar, la estructura *else if* es muy habitual, muy clara y añadiendo otro *begin end*, solo lo complicamos.

Varios if-then-else: resumen

Como resumen de todo lo anterior, recuerda:
Si una sentencia *if-then-else* tiene otro *if-then-else*...

- En la rama *then*, es potencialmente peligroso, así que siempre la anidaremos en un bloque *begin-end*.
- En la rama *else*, es un if encadenado *else if, else if, else if*. No es peligroso, es muy común. No añadimos bloque *begin-end* (a menos, naturalmente, que sea imprescindible porque hay otras sentencias).

Anidamiento múltiple

¿Podemos escribir un if-then-else dentro de otro if-then-else que está dentro de un if-then-else que está un if-then-else que...?

- El lenguaje lo permite.
- Pero si anidamos más de 2 o 3 niveles, muy probablemente estaremos escribiendo un programa de muy mala calidad, difícil de entender y propenso a errores.
    - Esto es uno los defectos peores y más típicos de quienes no saben programar.
- El anidamiento excesivo casi siempre indica que el programador no ha sabido descomponer su problema en subproblemas (escribiendo nuevas funciones).

Recuerda que el encadenamiento múltiple (else if, else if, else if) es un caso diferente, que, bien hecho, no tiene por qué dar problemas.

Sentencias if sucesivas

Posiblemente la mejor solución es evitar completamente el anidamiento (evitar un *if then else* dentro de otro): usando una secuencia de sentencias *if then* independientes y sucesivas.

```
if condicion1 and condicion2 then
    sentencia1

if condicion1 and not condicion2
    sentencia2

if not condicion1 then
    sentencia3
```

- Incluyendo las condiciones en una única expresión, evitamos niveles de anidamiento y resulta más legible.
- Este enfoque no siempre es posible, en cada caso tendremos que evaluar qué resulta más claro.
- Eso sí: es muy importante que las condiciones sean correctas. Un error típico es asignar dos ramas distintas a un valor frontera [16]. O no asignar ninguna rama a un valor frontera.

[16] En ese caso se ejecutan ambas.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ifs_sucesivos;

const
    Mayoria_edad: integer = 18;
    Adolescencia: integer = 13;

var
    Edad: integer = 10;

begin
    if Edad >= Mayoria_edad then
        writeln(Edad, ':Adulto');

    if (Edad > Adolescencia) and (Edad < Mayoria_edad) then
        writeln(Edad, ':Adolescente');

    if Edad <= Adolescencia then
        writeln(Edad, ':Niño');
end.
```

Resultado:

```
10:Niño
```

El ejemplo anterior es sencillo pero no muy realista: normalmente usaremos una función. Debemos recordar que una función bien escrita nunca escribe en pantalla.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ifs_sucesivos_02;

const
    Mayoria_edad: integer = 18;
    Adolescencia: integer = 13;

function clasifica_edad(edad: integer): string;
begin
    if edad >= Mayoria_edad then
```

```
        result := 'Adulto';

    if (edad > Adolescencia) and (Edad < Mayoria_edad) then
        result := 'Adolescente';

    if edad <= Adolescencia then
        result := 'Niño'
end;

const
    Edad = 10;
begin
    write(Edad, ':');
    writeln( clasifica_edad(Edad));
end.
```

Resultado:

```
10:Niño
```

4.3. case

Case

Otra sentencia de control disponible en Pascal es `case`. Permite ejecutar diferentes acciones a partir de un valor discreto.

- Es similar a `if`, la diferencia es que `if` considera un booleano y `case` puede considerar, además, tipos más complejos como integer o char.
    - Pero nunca real: ha de ser un valor discreto
- Se parece mucho a los if-then-else encadenados. De hecho, cualquier cosa que se haga con case, también se puede hacer con else-if, else-if.
    - Hay lenguajes que no tienen nada parecido a `case`, sus diseñadores consideraron que `else-if` es suficiente.
    - La ventaja es que `case` puede resultar más claro que `else-if`.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_case;

const
    Edad = 14;
begin
    case Edad of
    0 :
        writeln('Bebé');
    1..12 :
        writeln('Niño');
    13..17 :
        writeln('Adolescente');
    otherwise     // sin dos puntos
        writeln('Adulto');
    end;    // 'end' de case, único sin 'begin'
end.
```

- El compilador evalúa la expresión que escribamos a continuación de la palabra reservada *case*. En este ejemplo, la constante *edad*.
- Separamos todos los posibles valores, pudiendo usar rangos. Indicamos las acciones a realizar para cada conjunto de casos.
- Si el valor no está dentro de ninguno de los rangos descritos, se ejecutan las sentencias a continuación de *otherwise*.
- La rama *otherwise* se puede omitir, pero es preferible ponerla siempre.

Ejemplo de case en una función

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program case_en_funcion;
function clasificacion_edad(edad: integer): string;
begin
    case edad of
    0 :
        result := 'Bebé';
```

```
    1..12 :
        result := 'Niño';
    13..17 :
        result := 'Adolescente';
    otherwise     // sin dos puntos
        result := 'Adulto';
    end;    // 'end' de case, único sin 'begin'
end;
const
    Edad_cliente = 14;
begin
    writeln( clasificacion_edad(Edad_cliente));
end.
```

```
Adolescente
```

Ejemplo erróneo

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program case_erroneo;

function clasificacion_edad(edad: integer): string;
begin
    case edad of
    0 :
        result := 'Bebé';
    1..13 :
        result := 'Niño';
    13..17 :    // ¡Mal! El caso de 13 años está duplicado
               // El compilador dará un error
        result := 'Adolescente';
    otherwise
        result := 'Adulto';
    end;
end;
const
    Edad_cliente = 14;
begin
    writeln( clasificacion_edad(Edad_cliente));
end.
```

Otro ejemplo

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program case_hexa;
function valor_digito_hex(digito: char): integer;
begin
    case digito of
    '0':
        result := 0;
    '1'..'9':
        result := ord(digito)-ord('0');
    'A'..'F':
        result := 10+ord(digito)-ord('A');
    'a'..'f':
        result := 10+ord(digito)-ord('a');
    otherwise
        result := 0;
    end;
end;
const
    Digito = 'a' ;
begin
    writeln(digito, ':', valor_digito_hex(Digito)); // Escribe a:10
end.
```

Case con múltiples sentencias

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program case_varias_sentencias;
const
    Edad = 14;
begin
    case Edad of
    0..1 : begin
        write('Menor de edad, ');
        writeln('bebé');
    end;
    2..12 : begin
        write('Menor de edad, ');
        writeln('niño');
    end;
```

```
    13..17 : begin
        write('Menor de edad, ');
        writeln('adolescente');
    end;
    otherwise     // sin dos puntos
        writeln('Adulto');
    end;    // 'end' de case, único sin 'begin'
end.
```

Ejemplo: fecha válida

- Versión inicial. https://gsyc.urjc.es/ mortuno/fpi/fecha_valida_v01.pas
- Versión mejorada. https://gsyc.urjc.es/ mortuno/fpi/fecha_valida_v02.pas
- Versión con if then else. https://gsyc.urjc.es/ mortuno/fpi/fecha_valida_v03.pas
- Versión *estropeada*, como suele hacerlo un principiante. https://gsyc.urjc.es/ mortuno/fpi/fecha_valida_v04.pas

4.4. Estructura de un programa

Estructura de un programa

Como hemos visto,

- Los programas aceptan datos de entrada, los procesan mediante subprogramas (funciones o procedimientos) y producen datos de salida.

Estos subprogramas se pueden clasificar de muchas formas, pero una estructura típica es preproceso, proceso principal y postproceso.

- Subprogramas de preproceso.

  Toman los datos brutos de la entrada, comprueban que sean correctos y los preparan para el proceso principal, típicamente adaptando unidades o formatos.

- Subprogramas de proceso principal.

  Realizan las tareas fundamentales del programa, a partir de datos *limpios*.

- Subprogramas de postproceso, también llamados de formateado de salida.

  Los datos que salen del proceso principal, se adaptan al formato concreto necesario para su uso en la siguiente etapa.

Estas ideas son universales en programación, aunque el vocabulario exacto puede cambiar.

Ejemplo 1

Videojuego, módulo de movimiento de un personaje.

- Preproceso:

  Las órdenes del usuario pueden venir de un joystick, del teclado o del ratón. El preproceso convertirá todo esto a un único formato que especifique el movimiento.

- Proceso principal:

  Recibe la órdenes de movimiento, la situación del personaje y el universo relevante, genera la nueva situación.

- Postproceso:

  Esa situación se presenta en pantalla.

Este es un ejemplo para ilustrar las ideas, un juego real es más complicado.

Ejemplo 2

Módulo de un procesador de texto que inserta una imagen en un documento.

- Preproceso:

  La imagen que facilita el usuario puede estar en muchos formatos: jpg, png, gif, bmp, etc. Internamente, el programa maneja el formato bmp, así que todos los ficheros que no estén en este formato, se convierten a bmp.

- Proceso principal:

  La imagen se posiciona sobra la estructura interna que representa el documento.

- Postproceso:

  A partir de la estrucura interna del documento, se genera el pdf para imprimir o se compone el fragmento de página visible en ese momento.

Este es un ejemplo para ilustrar las ideas, un procesador de texto real es más complicado.

Ejemplo 3

Asistente virtual (Siri, Amazon Alexa, Google Assistant, Cortana).

- Preproceso:

  A partir del sonido de la voz, se extrae el texto y la identidad de quien habla.

- Proceso principal.

  Mediante técnicas de inteligencia artificial, se hace el procesamiento del lenguaje natural.

- Postproceso:

  Las órdenes se ejecutan: abrir una página web, leerla, reproducir una canción, encender una luz, ...

De nuevo, este ejemplo espera ser ilustrativo pero está muy simplificado.

4.5. Precondiciones y postcondiciones

Precondiciones y postcondiciones

- Cada uno de estos conjuntos de subprogramas (preproceso, proceso y post-proceso) estará formado en general por varias funciones. O tal vez solo una, o incluso ninguna (nada que hacer).
- Para la correcta ejecución de una función de cualquier tipo es necesario que se den:
  - Precondiciones.

    Condiciones que han de cumplir los parámetros para que se pueda ejecutar la función.

    Si no se cumplen, los parámetros recibidos serán erróneos y la función, si está bien programada, no se ejecutará.
  - Postcondiciones.

    Condiciones que ha de cumplir el resultado de la función. Si no se cumplen, es que la función está mal programada.

Ejemplos de precondición:

- En una función que divida dos números, el divisor ha de ser no nulo.
- En una función que calcule el factorial de un número, este ha de ser entero y positivo.

Ejemplos de postcondición:

- El factorial es un número ha de ser entero y positivo.
- El seno de un ángulo ha de ser un número real entre -1 y 1.

Cuando programemos una función, es muy importante comprobar las precondiciones. No hacerlo es:

- Muy peligroso y difícil de detectar: podemos hacer 1000 ejecuciones (con parámetros correctos) que provocarán resultados correctos. En la ejecución 1001 la función puede recibir parámetros errónes. Si no los trata correctamente, normalmente se producirá un error severo.
- Uno de los fallos de programación más habituales.

Una función puede tener postcondiciones que sea conveniente comprobar, pero en la práctica esto no es tan habitual.

En general, nuestras funciones deberán comprobar...

- Las precondiciones, casi siempre.
- Las postcondiciones, algunas veces.

¿Dónde comprobar precondiciones y postcondiciones?.

- Si las precondiciones o postcondiciones son sencillas (tal vez una línea o dos), pueden hacerse en la misma función.
- Si tienen cierta complejidad, requerirán una nueva función.

Ejemplo: inverso multiplicativo

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program inverso_multiplicativo;

function inverso(x: real): real;
begin
    result := 1/x;
end;

const
    A = 4.3;

begin
    if A <> 0 then begin  // Precondición
        write('El inverso de ', A:0:3 );
        writeln(' es ', inverso(A):0:3);
    end
    else
        writeln( 'Error: el 0 no tiene inverso');
end.
```

Ejemplo: cambio de divisas

Supongamos una aplicación de comercio electrónico. Internamente trabaja siempre con dólares, pero los clientes pueden pagar en libras o euros, que el sistema se encarga de convetir en dólares.

- Preproceso:

  Convertir la divisa de entrada en dólares, si es necesario.

- Proceso principal:

  (No haremos).

- Postproceso:

  (No haremos).

Algunos conceptos sobre el dominio del problema.

- Cada divisa tiene un código ISO 4271, formado por 3 letras maýusculas:

  USD: Dólar norteamericano.

  EUR: Euro.

  GBP: Libra esterlina.

  etc.

- El precio de una divisa frente a otra se indica concatenando los códigos ISO 4217.

Ejemplo: EURUSD = 1.13 significa que necesitamos 1.13 USD para comprar 1 EUR.

- A la primera divisa (EUR en este caso) se la denomina *divisa base*, es la divisa que compramos.

- La segunda divisa (USD) es la *divisa cotizada*, la divisa que usamos para pagar.

Si necesitamos el par inverso, podemos dividir entre 1.
USDEUR = 1 / 1.13 = 0.88
(aunque en la vida real puede haber pequeñas diferencias por las comisiones).

Nuestro programa recibirá:

- Un importe, un número real.

- El código ISO 4271 de la divisa, una cadena.

La función principal, *a_ dolar*, convierte la divisa de entrada en dólares.

- La precondición es que la divisa sea EUR, GBP o USD. Si no se cumple, no se invocará a la función.

- La función *a_ dolar* nunca recibirá una divisa distinta a estas tres, pero *por si acaso*, hacemos una comprobación redundante. Si algo falla y no se detectó en la comprobación de precondiciones, el programa se detiene abruptamente.

https://gsyc.urjc.es/ mortuno/fpi/divisas.pas

5. Procedimientos

5.1. Introducción a los procedimientos

Como hemos visto en el tema 3, una función bien escrita no puede tener efectos laterales, esto es, tiene que tener *transparencia referencial*.

- Tiene que devolver su resultado a partir de los parámetros de entrada, y nada más.
- No puede tener ningún efecto en ningún sitio, no puede modificar nada, no puede escribir nada en pantalla.

Función mal escrita:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program sin_transparencia_referencial;

function media(a,b: integer): real;
begin
    result := (a + b ) / 2;
    writeln('La media de ',a , ' y ' ,b, ' es ', result); // ¡MAL!
end;

const
    A = 2;
    B = 3;

begin
    writeln( media(A,B));
end.
```

Función bien escrita:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program transparencia_referencial;

function media(a,b: integer): real;
begin
```

```
    result := (a + b ) / 2
end;

const
    A = 2;
    B = 3;

begin
    write('La media de ',A );
    write(' y ', B);
    writeln(' es ',media(A,B):0:3);
end.
```

En este curso, hasta ahora solo hemos usado constantes y funciones sin efectos laterales. Por tanto, todas las ejecuciones del programa producían exactamente el mismo resultado. A esto se le denomina *programación funcional*.

Esto es lo más adecuado en muchas ocasiones, pero si los programas solo tuvieran este comportamiento, estarían muy limitados. Cuando resulte conveniente, usaremos dos elementos más:

- Variables.
- Procedimientos.

5.2. Variables

Variables

Una constante, como su nombre indica, es un valor que permanece inalterado durante toda la ejecución de un programa. Una variable es un nombre para un valor que podrá cambiar durante la ejecución.

- Al igual que las constantes, las variables se declaran (indicamos su nombre y tipo) y se definen (les damos valor).
- La declaración y la definición de una variable es prácticamente igual que la declaración y definición de una constante, pero con la palabra reservada `var` en vez de `const`.

  ```
  var
      marca, modelo: string;
      cilindrada : integer;
  ```

- Las constantes se pueden declarar y definir o solamente definir, sin declarar. Pero las variables se declaran y definen siempre.

Las variables se declaran en la parte declarativa de los bloques, junto a las constantes, inmediatamente antes de la parte de las sentencias (el begin-end).

En otras palabras, podemos declarar variables:

- Antes del begin de una función o un procedimiento.
- Antes del begin del cuerpo del programa principal.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variable;
var
    aceleracion: integer;

begin
    aceleracion := 12;
    writeln(aceleracion);
end.
```

Al igual que en la declaración de constantes:

- Después de `var` no hay *punto y coma*.
- Hay un *punto y coma* al final de cada declaración.

En estos casos usamos identificadores como *a, b, c, etc* porque son ejemplos *que no hace nada*. En un ejercicio no tan trivial, usaríamos nombres descriptivos: P.e. *importe*, *velocidad*, *hora*, *dni*, etc.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variables01;

function suma(a,b: integer):integer;
var
    c: integer;    // Variable local de la función suma
```

```
begin
    c := a + b;
    result := c;
end;

const   // Constantes locales al programa principal
    Sumando1 = 12;
    Sumando2 = 3;

var   // Variables locales al programa principal
    resultado : integer;

begin
    resultado := suma(Sumando1, Sumando2);
    writeln(resultado);  // 15
end.



{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variables02;

function suma(a,b: integer):integer;
var
    c: integer;    // Variable local de la función suma
begin
    c := a + b;
    result := c
end;

var   // Variables locales al programa principal
    sumando1,sumando2,resultado : integer;

begin
    sumando1 := 4;
    sumando2 := 9;
    resultado := suma(sumando1,sumando2);
    writeln(resultado);  // 13

    sumando1 := 2;  // Podemos cambiar el valor de las variables.
    sumando2 := 6;  // No es necesario definir nuevas constantes.
```

```
    z := suma(sumando1,sumando2);
    writeln(resultado);  // 8
end.
```

El programador es responsable de inicializar todas las variables (darles un valor inicial).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variable_no_inicializada;

var
    a: real;

begin
    writeln(a);   // ¡¡MUY MAL!!   No hemos inicializado la variable
end.
```

- Si lo olvidamos, el compilador no dará un error. Como mucho un aviso.

```
Warning: Variable "a" does not seem to be initialized
```

Una variable se puede inicializar (puede recibir su primer valor) o bien en la declaración, o bien en el cuerpo del programa / funcion / procedimiento.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program inicializacion_variable;

var
    a: real = 3;
    b : real ;

begin
    b := 4.2 ;
    writeln(a);
    writeln(b);
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program uso_de_variables;

var
    x, y : integer;

begin
    x := 3;
    y := x + 1 ;
    x := y div 2 ;    // 4 div 2
    writeln(x);       // 2
    writeln(y);       // 4
end
```

Independencia argumento - parámetro

Si pasamos una variable como argumento en una llamada a un subprograma, el nombre de la variable es completamente independiente del nombre del parámetro. Pueden coincidir, pero en general no tienen por qué.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program nombres_parametros;
function media(x, y: real): real;
begin
    result := (x+y) / 2;
end;

var
    a,b : real;
begin
    a := 5;
    b := 3;
    writeln( media(a,b) );
end.
```

Solo es relevante que el primer argumento vale 5, y el segundo, 3.

Asignación vs Comparación

Es importante no confundir el operador de asignación := con el operador de comparación =

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program asignacion_vs_comparacion;

var
    x : integer;

begin
    x := 0 ;
    x := x + 1 ;   // Ahora x vale 1
    writeln(x = x + 1);  // Escribe FALSE
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program asignacion_vs_comparacion_02;

var
    x : integer;
    a : boolean ;

begin
    x := 3 ;
    a := x = x + 1;
    a := (x = x + 1);    // Lo mismo, más claro
    writeln(a) ;         //  Escribe FALSE

    a := x = 4 ;         //  x vale 3
    writeln(a) ;         //  Escribe FALSE

    x := x + 1 ;         //  ahora x vale 4
    a := x = 4 ;
    writeln(a);          //  Escribe TRUE
end.
```

Pero recuerda que para dar valor a las constantes, en Pascal se usa =, no :=.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
```

```
program asignacion_vs_comparacion_03;

const
    X := 3;   // ¡¡MAL!!

begin
    writeln(X) ;
end.
```

Al intentar compilar este ejemplo obtendremos un error.

```
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling asignacion_vs_comparacion_03.pas
asignacion_vs_comparacion_03.pas(6,7) Fatal: Syntax error, "=" expected
    but ":=" found
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

Ahormados

El ahormado (*casting*) de variables funciona igual que el ahormado de constantes.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program casting_variables;

var
    a: integer;
    x: real;
    c: char;
begin
    a := 3;
    x := a;  // La conversión de integer en real es automática

    c := 'Z';
    x := integer(c);
    writeln(x:0:0);  // escribe 90
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program casting_variables_02;

var
    x: char;
    y: integer;
    z: real;
begin
    x := 'a';
    writeln( integer(x));  // Escribe 97

    y:= 98;
    writeln( char(y));   // Escribe 'b'
    writeln( real(y));   // Escribe 9.8000000000000000E+001

    z:= 65.7;
    { writeln( integer(z));   // ¡Esto es ilegal!  }

    writeln( trunc(z));       // Escribe  65

    writeln( char(  trunc(z)  ) );   // Escribe  'A'

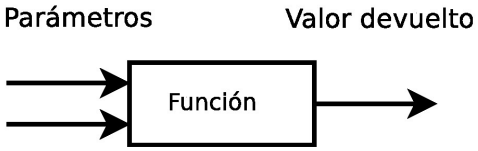
    // Esto es redundante, trunc(z) ya devuelve un integer
    writeln( char( integer( trunc(z) ) ) );   // Escribe  'A'
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program casting_variables_03

var
    c: char;
begin
    c := 'A';

    writeln ( ord(c) );         // Escribe 65

    c := chr( ord(c) + 1 );
    writeln(c);                // Escribe 'B'
```




```
    writeln( succ(c) );         // Escribe 'C'
end.
```

5.3. Procedimientos

Procedimientos

Recordemos que una función es un subprograma que:

- No deberían tener efectos laterales.
- Devuelve un valor.
- No puede modificar sus argumentos.

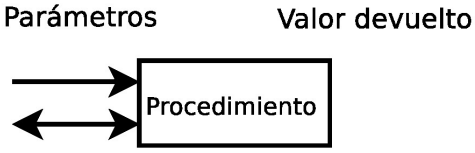
Un procedimiento:

- Se espera que tengan efectos laterales. Por ejemplo, escribir en pantalla.
- Puede modificar sus argumentos (aunque no es obligatorio).
- No devuelve ningún valor (aunque pueden modificar sus argumentos).

Un procedimiento es una acción con nombre.

Una función:

- Recibe 1 o más parámetros de entrada[17].
- Devuelve exactamente 1 valor.
- Si está bien escrita, no tiene efectos laterales.



Un procedimiento:

- Puede no recibir ningún parámetro, puede recibir uno, puede recibir más de uno.
- Alguno de los parámetros puede ser de salida (o no).
- No devuelve ningún valor.

Supongamos que solo conocemos las funciones y tenemos este código:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program sin_procedimiento;
var
    c: char;
    i: integer;
begin
    c := 'a';
    i := ord(c);
    writeln('El código ASCII de ',c,' es ',i);

    c := 'b';
    i := ord(c);
    writeln('El código ASCII de ',c,' es ',i);
end.
```

```
El código ASCII de a es 97
El código ASCII de b es 98
```

Claramente es deseable factorizar esto (evitar el *copia y pega*), pero no es aceptable que una función tenga efectos laterales.

[17] En ocasiones excepcionales puede no tener ninguno

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program procedimiento;

procedure escribe_ascii(x:char);
var
    i: integer;

begin
    i := ord(x);
    writeln('El código ASCII de ',x,' es ',i);
end;

var
    c: char;
begin
    c := 'a';
    escribe_ascii(c);

    c := 'b';
    escribe_ascii(c);
end.
```

```
El código ASCII de a es 97
El código ASCII de b es 98
```

- Un procedimiento tendrá parámetros, de entrada y/o de salida.
- También podrá tener variables locales, para almecenar valores temporales que resulten convenientes.
- Jamás podremos poner el mismo nombre a una variable y a un parámetro, en nuestro caso el compilador generará un error *duplicate identifier*.

```
procedure escribe_ascii(x:char);
var x: integer;   // ¡¡MAL!!

begin
    x := ord(x);
    writeln('El código ASCII de ',x,' es ',x);
end;
```

Como las funciones, los procedimientos pueden no tener argumentos.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program procedure_holamundo;

procedure escribe_holamundo;
begin
    writeln('Hola, mundo');
end;

begin
    escribe_holamundo;
end.
```

- También podríamos declararlo y usarlo con paréntesis, pero sin argumentos.

  ```
  ...
  procedure escribe_holamundo();
  ...
  escribe_holamundo();
  ```

Un principiante puede pensar:

- *Las funciones calculan expresiones, pueden seleccionar y hacer bucles. Pero no pueden escribir en pantalla.*
- *Los procedimientos hacen lo mismo que las funciones, pero además sí pueden escribir en pantalla.*

... entonces los procedimientos son mejores. Usemos procedimientos para todo.

¡No! Los funciones *calculan cosas* y los procedimientos *hacen cosas con las cosas calculadas.*

- En este curso, prácticamente lo único que *hacemos* es escribir en pantalla.
- En este curso puede parecer caprichosa esa limitación de las funciones. En nuestros ejemplos, las funciones podrían escribir en pantalla y aparentemente *no pasaría nada.*

Pero el objetivo de este curso es adquirir hábitos de programación para entornos *reales*. En un entorno *real*, la necesidad de la separación función / procedimiento resulta muy evidente, porque podríamos tener.

- Una función que *calcula algo*.
- Un procedimiento que escribe ese algo en pantalla.
- Otro procedimiento para insertarlo en una base de datos.
- Otro procedimiento para escribir un informe para impresora.
- Otro procedimiento para enviar el dato a Hacienda.
- Otro procedimiento para preparar el dato para ver en un móvil.
- Otro procedimiento para ver el dato en un tablet.
- Etc etc etc.

En muchos lenguajes de programación no hay procedimientos, todo son funciones. Pero este principio es aplicable igualmente: si nuestro diseño es bueno tendremos:

- *funciones función*, con transparencia referencial, que *calculan cosas*. Importante: devuelven valores.
- *funciones procedimiento*, con efectos laterales, que hacen cosas con las cosas calculadas por las funciones. Importante: no devuelven valores.

La idea fundamental, en cualquier lenguaje, es:

1. Un subprograma calcula las cosas.
2. Otro subprograma distinto, hace cosas con lo calculado por el subprograma anterior.

En Pascal, el 1) suele ser una función, pero puede ser un procedimiento si necesitamos devolver más de una cosa (y no queremos usar registros).
En Pascal, el 2) es siempre un procedimiento.

Readln

Con el procedimiento *readln* podemos leer un valor introducido por teclado y almacenarlo en una variable.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ejemplo_readln;

var
    c: char;
begin
    writeln('Escribe un carácter');
    readln(c) ;
    write('Has escrito ');
    writeln(c);
end.
```

Recuerda que un nombre de variable como *c* es típico en esta clase de ejemplos sencillos. En un programa real, tendríamos que elegir un nombre descriptivo.

Observa que readln no devuelve nada. Ningún procedimiento devuelve nada. Si intentamos hacer esto.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ejemplo_readln_mal;

var
    c: char;
begin
    writeln('Escribe un carácter');
    c := readln();   // ¡¡MAL!!
    write('Has escrito ');
    writeln(c);
end.
```

El compilador da un error:

```
Compiling ejemplo_readln_mal.pas
ejemplo_readln_mal.pas(8,17) Error: Incompatible types: got "untyped" expected "Char"
ejemplo_readln_mal.pas(11) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
```

Ejemplo mínimo:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program ejemplo_readln_02;

var
    i: integer;
begin
    readln(i) ;
    writeln(i);   // Escribe el entero que hayamos introducido
                    // Da error de ejecución si no es un entero
end.
```

Ejemplo un poco más completo:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_readln_03;

var
    i: integer;
begin
    writeln('Escribe un número entero: ');
    readln(i) ;
    write('La raiz cuadrada de ',i,' es ');
    writeln( sqrt(i) );   // Escribe la raiz cuadrada del número
                          // Da error de ejecución si no es un entero
end.
```

Este cálculo lo hacemos dentro del procedimiento porque es trivial, una raiz cuadrada. Si fuera un poco más complejo, el cálculo debería ir en una función aparte.

Halt

Con el procedimiento `halt` podemos concluir la ejecución del programa.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_halt;
```

```
var
    i: string;
begin
    writeln('Escribe "q" si quieres concluir');
    readln(i) ;
    if i='q' then
        halt      // Recuerda que aquí no se escribe ';'
    else
        writeln('El programa sigue...');
end.
```

5.4. Paso de parámetro por referencia

Paso de parámetro por referencia

- Paso por valor:

  La forma más habitual de pasar parámetros a funciones o procedimientos es el *paso por valor*[18]. Es la que hemos visto hasta ahora. El subprograma recibe una copia del parámetro, si el subprograma modifica el parámetro, los cambios se pierden al finalizar el subprograma.

- Paso por referencia.

  Anteponiendo la palabra reservada var a un parámetro, pasa por referencia, no por valor. Esto significa que su valor puede modificarse dentro del procedimiento, y el cambio se verá después de la llamada al procedimiento.

  - Observa que el procedimiento predefinido *readln()* está definido de forma que el parámetro se pasa por referencia.

Ejemplo de paso por valor

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program paso_por_valor;

procedure p(x:integer);
begin
    x := 0;
    writeln(x);   // Escribe 0
```

[18] En ocasiones denominado paso por copia

```
end;

var
    i: integer;
begin
    i := 3;
    p(i);
    writeln(i);     // Escribe 3, la variable no ha cambiado
end.
```

Ejemplo de paso por referencia

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program paso_por_referencia;

procedure p(var x:integer);
begin
    x := 0;
    writeln(x);   // Escribe 0
end;

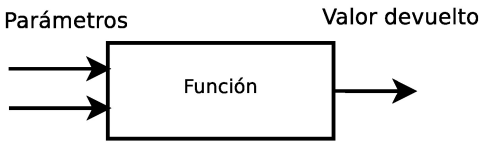
var
    i: integer;
begin
    i := 3;
    p(i);
    writeln(i);    // Escribe 0, el procedimiento ha cambiado
                   // la variable
end.
```

La única diferencia en el fuente es añadir `var`, pero el comportamiento cambia por completo.

Así, cuando pasemos parámetros a un procedimiento, normalmente.

- Si queremos que cambien, pasamos por referencia, anteponiendo `var`.
- Si no queremos que cambien, pasamos por valor, sin anteponer `var`.

Con una excepción: objetos que ocupen mucha memoria.



Excepción: objetos grandes

Como hemos visto, al hacer un paso por valor (sin *val*), el procedimiento recibirá una copia del parámetro. Esta copia:

- Duplica la memoria consumida.
- Tarda cierto tiempo.

El paso por referencia ahorra memoria y tiempo. Con tipos de datos que ocupan poco espacio, como los que hemos visto, esto no es importante. Pero sí lo será para objetos que ocupen mucha memoria.

- Este ahorro también será necesario si una función intenta devolver un objeto muy grande: no podrá hacerlo. Será necesario escribir un procedimiento, donde el valor a devolver será un parámetro pasado por referencia.

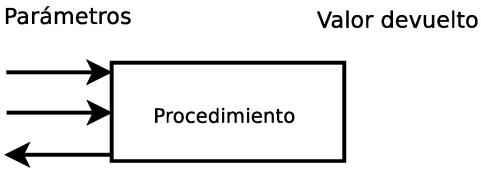
En otras palabras

- A partir de una función, siempre es posible escribir un procedimiento equivalente. Basta eliminar el valor devuelto y añadir un parámetro de salida, con ese valor.
- Siempre es *posible* pero ¿cuándo es *necesario*?.
    - Cuando el valor a devolver es *grande*. En otros casos no tiene sentido.

Ejemplo 1 (gráfico)

Si tenemos una función con dos parámetros (de entrada):

Podemos convertirla en un procedimiento con dos parámetros de entrada y uno de salida.



Ejemplo 2 (código) : Dado este programa que usa una función con un parámetro de entrada.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program pts_funcion;

const
    EURPTS = 166.386; // Constante global, universal

function pts_a_eur(pts : real) : real;
begin
    result := pts / EURPTS;
end;

procedure escribe_resultado(pts, eur: real);
begin
    write(pts:0:0 , ' PTS son ');
    writeln(eur:0:2 , ' EUR');
end;

var
    importe_pts, importe_eur : real;
begin  // Cuerpo del programa principal
    importe_pts := 500;
    importe_eur := pts_a_eur(importe_pts);
    escribe_resultado(importe_pts, importe_eur);

    importe_pts := 1000000;
    importe_eur := pts_a_eur(importe_pts);
    escribe_resultado(importe_pts, importe_eur);
end.
```

Podemos escribir un programa equivalente, basado en un procedimiento equivalente con un parámetro de entrada y otro de salida.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program pts_proc;

const
    EURPTS = 166.386; // Constante global, universal

procedure pts_a_eur(pts : real; var eur: real);
begin
    eur := pts / EURPTS;
end;

procedure escribe_resultado(pts, eur: real);
begin
    write(pts:0:0 , ' PTS son ');
    writeln(eur:0:2 , ' EUR');
end;

var
    importe_pts, importe_eur : real;
begin
    importe_pts := 500;
    pts_a_eur(importe_pts, importe_eur);
    escribe_resultado(importe_pts, importe_eur);

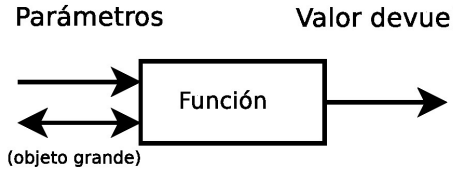
    importe_pts := 1000000;
    pts_a_eur(importe_pts, importe_eur);
    escribe_resultado(importe_pts, importe_eur);
end.
```

Este ejemplo ha servido para ilustrar el concepto, no tiene ninguna otra utilidad ni tampoco sentido, porque *importe_ eur* es un único número, ocupa muy poca memoria.

- Pero es necesario que sepamos usar el paso por valor. Si necesitamos devolver un objeto grande como por ejemplo un *array* con 1000 precios (lo veremos en el tema 8), entonces es imprescindible. También para devolver más de un objeto.

Paso por valor en funciones

También se pueden pasar parámetros por referencia a una función (no solo a un procedimiento).



- Exactamente de la misma forma, anteponiendo `var`.
- Normalmente no haremos esto. Pero como ahorra tiempo y memoria, para objetos grandes puede ser imprescindible. Un paso por valor de un objeto *grande* provocará un *desbordamiento de pila*.

    1. Observa que no hablamos de *devolver* algo grande, como en el caso anterior, sino de *recibir* algo grande.

- ¿Cuándo es *grande*? No es fácil saberlo a priori. Depende de la versión del compilador y del sistema operativo [19].
- Si es necesario pasar un objeto grande por referencia a una función, el programador será responsable de no modificarlo. El compilador lo permitirá, pero será un efecto lateral, que es un mal diseño (y en este curso, un ejercicio suspenso).

Si alguno de los parámetros es un objeto *grande*, tendremos que pasarlo por referencia.

- Desde el punto de vista del compilador, es un parámetro de entrada salida.
- Aunque el compilador nos permita modificarlo, nunca debemos hacerlo. Nuestro único objetivo es ahorrar memoria.
- En caso de que el objeto muy grande sea el valor a devolver, no podemos usar una función, sino un procedimiento (donde el valor a devolver será un parámetro de entrada/salida).

[19] En las versiones que manejamos ahora, el compilador de Windows acepta objetos mayores que el de Linux, pero esto podría cambiar en cualquier momento

Paso por referencia. Resumen

- Para pasar un parámetro por referencia, se antepone la palabra reservada *var* en la declaración .
- Hace dos cosas:
  1. Que el parámetro sea de entrada-salida, esto es, que la modificación del parámetro dentro del subprograma afecte también a la variable pasada como argumento. Lo usaremos en procedimentos.
  2. Ahorra memoria. Lo usaremos cuando el objeto sea grande. Esto es imprescindible tanto en funciones como en procedimientos.
- El compilador nos permitirá declarar un parámetro con paso por referencia en una función (aunque no sea un parámetro grande), por tanto nos permitirá modificarlo. Pero esto es un mal diseño.
- En nuestro dialecto de Pascal (Object Pascal), las funciones no pueden devolver objetos grandes. Por tanto es necesario usar un procedimiento, con un parámetro de salida.

En otras palabras:

- Cuando usamos paso por referencia (*anteponer var*) suceden dos cosas:
  1. El compilador pemite que modifiquemos el parámetro.
  2. Se ahorra tiempo y memoria.
- Pero cuando solo necesitemos una (ahorrar), no debemos usar la otra (modificar). Sería un mal diseño.

Otro ejemplo de paso por referencia

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ej_incrementa;

procedure incrementa(var a:integer);
begin
    a := a + 1;
end;
```

```
var
    i: integer;
begin
    i := 3;
    incrementa(i);
    writeln(i);      // Escribe 4
end.
```

Si olvidamos añadir `var`, estamos pasando por valor y los cambios en el parámetro desaparecerán cuando la función termine su ejecución (En este caso, no era lo deseado).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}

program incrementa_mal;

procedure incrementa(a:integer);  // MAL: olvido var
begin
    a := a + 1;
end;

var
    i: integer;
begin
    i := 3;
    incrementa(i);
    writeln(i);      // Escribe 3
end.
```

Este ejercicio tan sencillo (incrementar un número, usando un subprograma), también podemos hacerlo usando una función, no solo un procedimiento.

Basta usar la variable tanto a la izquierda como a la derecha de la asignación.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program incrementa_con_funcion;

function incrementa(a:integer):integer;
begin
```

```
    result := a + 1;
end;

var
    i: integer;
begin
    i := 3;
    i := incrementa(i);
    writeln(i);      // Escribe 4
end.
```

Recuerda:

- Las funciones devuelven valores, las usamos para formar expresiones. Por ejemplo, si `f` y `g` son funciones, escribimos cosas como

```
writeln(f(x));
y := f(x);
z := f(g(x));
y2 := f( alfa, beta, gamma);
y3 := f( g(x1), g(x2), g(x3) );
```

f, *g*, *h*, son nombres típicos de función en ejemplos como este, donde la función en sí misma no existe o es lo de menos. En un ejercicio completo los nombres serían descriptivos. P.e. *maximo*, *factorial*, *calcula_rozamiento*, *tipo_aplicable*, etc.

- Los procedimientos no devuelven nada. Tal vez modifiquen un parámetro pero no podemos usarlos a la derecha del operador de asignación o como parámetro de una función o procedimiento.

Por ejemplo, si `p` y `q` son procedimientos, escribiremos cosas como

```
p;
p(x);
q(x, y, f(z));
```

Pero jamás

```
writeln(p(x));    // MAL
x := p(x);        // MAL
p( q(x) );        // MAL
```

Si lo intentamos, el compilador dará un error.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program incrementa_mal_02;

procedure incrementa(var a:integer);
begin
    a := a + 1;
end;

var
    i: integer;
begin
    i := 3;
    i := incrementa(i);  // ¡¡MAL!! No es una función
    writeln( i );
end.
```

```
incrementa_mal_02.pas(13,7) Error: Invalid assignment, procedures
return no value
```

Recuerda que una función devuelve un valor, y solo uno.

Si necesitamos un subprograma que devuelva no solo un valor, sino dos o más, hay dos formas de hacerlo:

- Usar un procedimiento y pasar argumentos por referencia.
- Usar registros (como veremos en el siguiente tema).

```
program devolver_pareja;

procedure division_entera(dividendo, divisor: integer;
    var cociente, resto: integer );
begin
    cociente := dividendo div divisor;
    resto := dividendo mod divisor;
end;

procedure escribir_valores(dividendo, divisor,
    cociente, resto:integer);
begin
```

```
    write('La división entera entre ', dividendo);
    write(' y ', divisor);
    write(' es ', cociente);
    writeln(' con un resto de ', resto);
end;

const
    Dividendo_ejemplo = 9;
    Divisor_ejemplo = 2;
var
    dividendo, divisor, cociente, resto : integer;
begin
    dividendo := Dividendo_ejemplo;
    divisor := Divisor_ejemplo;

    division_entera(dividendo, divisor, cociente, resto);
    escribir_valores(dividendo, divisor, cociente, resto);
end.
```

Resultado:

```
La división entera entre 9 y 2 es 4 con un resto de 1
```

Observa que en el ejemplo anterior, en la declaración, hemos tenido que separar los argumentos pasados por valor de los argumentos pasados por referencia.

Esto es, hemos escrito:

```
procedure division_entera(dividendo, divisor: integer;
        var cociente, resto: integer );
```

y no

```
procedure division_entera(dividendo, divisor,
        var cociente, var resto : integer); // ¡¡MAL!!
```

5.5. Parámetro de salida con código de error

Parámetro de salida con código de error

Veamos ahora otro caso, muy habitual, en que un subprograma devuelve dos (o más) *cosas*.

Es muy normal que un subprograma no pueda realizar su tarea principal porque algo se lo ha impedido:

- Tal vez no se cumple alguna precondición.
- Tal vez se ha producido algún error.

El subprograma devolverá dos valores: un código de éxito/error y el valor calculado, si procede.

En el ejemplo *inverso_ multiplicativo* del tema 4, cuando una precondición se incumplía, el programa principal lo detectaba y mostraba un mensaje de error.

- En un programa real no haremos algo así, en el cuerpo del programa principal normalmente no tomaremos decisiones, y menos aún de tan bajo nivel. En el programa principal nos limitaremos a llamar a subprogramas.
- Ahora que sabemos usar procedimientos y parámetros de salida, veremos una forma mucho más adecuada.
    - El cuerpo del programa principal llama a un procedimiento.
    - El procedimiento comprueba si puede llamar a la función.

Un subprograma intentará hacer un cálculo y devolverá dos parámetros de salida:

Si el cálculo es posible:

- El valor calculado.
- Un código que significa *todo bien, sí, he podido calcular.*

Si el cálculo no es posible:

- Un código que signfica *algo ha fallado.*
- En esta caso, el primer parámetro no contiene nada, un valor indeterminado[20], no debe consultarse.

[20] suele ser 0 pero puede cambiar en cualquier ejecución

Tendremos dos subprogramas:

- Uno que hace el cálculo (o lo intenta), y devuelve el código de éxito/error y el valor.

  Este subprograma en principio sería una función, pero las funciones en Pascal no pueden devolver dos valores. Por tanto ha de ser un procedimiento, pero un *procedimiento-función*, esto es, un procedimiento que *calcula cosas*.

- Otro que llama al subprograma anterior y *hace cosas*: escribir en pantalla el resultado. Este subprograma será un procedimiento.

Recuerda que hay lenguajes donde solo hay funciones, no procedimientos. Lo importante en cualquie lenguaje es separar muy claramente.

- Los subprogramas que *calculan cosas*.
- Los subprogramas que *hacen cosas*. Por ejemplo, mostrar al usuario el resultado del cálculo.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program inverso02;
procedure calcula_inverso(x: real; var inverso: real;
        var codigo: integer);
// Si es posible calcular el inverso multiplicativo de x, en el
// parámetro de salida 'codigo' devuelve un 0 y en el parámetro
// de salida 'inverso', el inverso.
// Si no es posible calcular el inverso, devuelve un código
// distinto de 0.
begin
    if x <> 0 then begin
        inverso := 1/x;
        codigo := 0;
    end else begin
        codigo := 1;
    end
end;
```

```
procedure escribe_inverso(x: real);
var
    codigo : integer;
    inverso : real;
begin
    calcula_inverso(x, inverso, codigo);
    if codigo = 0 then begin
        write('El inverso de ', x:0:3 );
        writeln(' es ', inverso:0:3);
    end
    else begin
        write( 'El número ', x:0:3);
        writeln( ' no tiene inverso');
    end;
end;
```

Observa que *calcula_ inverso* es un procedimiento, no una función. No podemos hacer esto:

```
    inverso := calcula_inverso(x);  // ¡¡MAL!!
```

```
var
    ejemplo : real;
begin
    ejemplo := 23;
    escribe_inverso(ejemplo);

    ejemplo := 0;
    escribe_inverso(ejemplo);
end.
```

Resultado:

```
El inverso de 23.000 es 0.043
El número 0.000 no tiene inverso
```

5.6. Procedimiento val

Procedimiento val

`val` es un procedimiento que hace uso de esta característica (modificar dos parámetros para devolver dos valores)

Sirve para convertir una cadena de texto en un número, si es posible.

Ejemplos:

- La cadena '12' se puede convertir en el entero 12.
- La cadena '4r2' no se puede convetir en un entero.
- La cadena '4x2', tampoco (`val` convierte números, no expresiones).
- La cadena ' 8.2' se puede convertir en real (los espacios a la izquierda no importan).
- La cadena '24.8 ' no se puede convetir en real (los espacios a la derecha sí dan error).

Sus parámetros son:

- Primer parámetro. Parámetro de entrada. Una cadena.
- Segundo parámetro. Parámetro de salida. Contendrá un número del tipo necesario: entero o real [21], según la conversión que necesitamos hacer. Contendrá el valor numérico de la cadena de entrada, si es posible.
- Tercer parámetro. Parámetro de salida, un código. Si vale 0 indica que la conversión ha sido posible. En otro caso, indica la posición de la cadena donde se produjo el error.

Observa que la cadena se *convierte* en número, si es posible. Pero esto no significa que el primer parámetro se altere, sino que en el tercer parámetro tendremos el valor numérico correspondiente.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_val;  // En un programa real solo escribiríamos
var                   // i,r cuando el código es correcto
    i,codigo : integer;
    r: real;
begin
    val('3', i, codigo);          // ok
    writeln(i, ', ',  codigo);    // 3, 0

    val('x', i, codigo);          // error
    writeln(i, ', ',  codigo);    // 0, 2
```

[21] o algún tipo enumerado

```
    val('3.7', r, codigo);         // ok (r es real)
    writeln(r, ', ', codigo);     // 3.7000000000000002E+000,0

    val('2.m', r, codigo);         // error
    writeln(r, ', ', codigo);     // 0.0000000000000000E+000,3

    val(' 3', r, codigo);           // ok
    writeln(r, ', ', codigo);     // 3.0000000000000000E+00,0

    val('3.7', i, codigo);         // error. (i es integer, no real)
    writeln(i, ', ', codigo);     // 0, 2
end.
```

El ejemplo anterior ilustra el funcionamiento de val, pero en un programa real nunca deberíamos consultar *sin más* el valor de i o de r.

- Solamente usaremos el valor devuelto cuando el código sea 0.
- Cuando el código es distinto de 0, las variable i o r tienen un valor no definido: apuntan a una zona de memoria indeterminada.
    - Lo más frecuente es que esta zona no haya sido usada previamente y su valor sea 0.
    - Pero también puede tener el valor de cualquier otra variable previa, un número aleatorio.
    - Podemos ejecutar el programa 10 000 veces y obtener un 0, y obtener cualquier otra cosa la vez 10 001. Problemas de este tipo son los que hacen que un programa tenga errores que solo son detectados al cabo de muchos años de funcionamiento aparentemente correcto.

A continuación veremos ejemplos correctos de uso `val`.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_val_02;

var
```

```
    s: string;
    i, codigo: integer;

begin
    write('Introduce un número entero ');
    readln(s) ;  // Lo que leemos es una cadena, 'por si acaso'
    val(s, i, codigo);
    if (codigo) = 0 then
        writeln(i)
    else
        writeln('Error, ',s,' no es un entero');
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_val_03;

var
    s: string;
    codigo: integer;
    r: real;

begin
    write('Introduce un número real ');
    readln(s) ;
    val(s, r, codigo);
    if (codigo) = 0 then
        writeln(r)
    else
        writeln('Error, ',s,' no es un real');
end.
```

5.7. Concatenación de cadenas

Concatenación de cadenas

Se pueden concatenar (unir) cadenas con el operador +.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program concatenacion;
```

```
var
    s1, s2, s3: string;
begin
    s1 := 'hola, ';
    s2 := 'mundo';
    s3 := s1 + s2;
    writeln(s3);    // hola, mundo

    s3 := 'concatenando ' + 'cadenas';
    writeln(s3);    // concatenado cadenas
end.
```

Procedimiento str

`str` hace el paso inverso de `val`: toma un número o un tipo enumerado y lo convierte en cadena.

Recibe dos parámetros:

- El número o enumerado a convertir.
- Una cadena, donde escribirá el resultado.

```
program ejemplo_str;

var
    i: integer;
    r: real;

    s1, s2: string;
begin
    i := 4;
    r := 2.2311;

    str(i, s1);  // Convierte i a cadena y lo guarda en s1
    str(r:0:2, s2);  // Convierte r:0:2 a cadena y lo guarda en s2
    write(s1  );
    writeln(', ' + s2 );
        // Escribe 4, 2.23
end.
```

5.8. Ámbito de las variables

Ámbito de las variables

El ámbito de una variable es la zona del programa donde esa variable existe, donde se puede leer y escribir.

Según su ámbito, las variables pueden ser:

- Locales.
  - Locales del cuerpo principal, declaradas después de todas las funciones y procedimientos, justo antes del begin del cuerpo principal.
  - Locales de una función o procedimiento, declaradas justo antes del begin de esa función o procedimiento.
- Globales

  Son extremadamente peligrosas.
  - Algunas metodologías permiten que se usen en contadísimas excepciones y con mucho cuidado.
  - Para otras metodologías, no deben usarse nunca, en ningún caso. En este curso:

    Variable global $\Longrightarrow$ Suspenso seguro.

Variable global

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variable_global;
    //  Pésimo ejemplo. Suspenso seguro
var
    x: integer;

    // Esta variable es global. Es visible en todo el código que viene
    // a continuación, esto es, a todo el programa. Defecto muy severo.

procedure incrementa();
begin
    x := x + 1;
end;

begin
    x := 3;
```

```
    incrementa();
    writeln(x);       // Escribe 4
end.
```

Variable local del cuerpo principal

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variable_local;

function incrementa(x:integer):integer;
begin
    x := x + 1;
    result := x;
end;

var     // Este es el lugar correcto para declarar las variables
        // que usa el cuerpo principal del programa. Así, serán
        // variables locales del cuerpo principal
    x: integer;
begin
    x := 3;
    x := incrementa(x);
    writeln(x);     // Escribe 4
end.
```

Variable local de un subprograma

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program variable_local_02;

function incrementa(x:integer):integer;
var
    y:string;   // Este es un lugar correcto para declarar una
                // variable. Es una variable local de la función

begin
    y := 'No uso esta variable pero es un ejemplo';
    x := x + 1;
    result := x;
end;
```

```
var
    x: integer;
begin
    x := 3;
    x := incrementa(x);
    writeln(x);     // Escribe 4
end.
```

Problemas de las variables globales

Entre los problemas de las variables globales, destacamos dos:

- Nunca podemos estar seguros de que una variable global sea verdaderamente global, porque una variable local puede taparla.
- Supongamos que.

    1. Un subprograma trabaja con una variable global y almacena cierto valor.
    2. El programa sigue su curso, llamando a otros subprogramas.
    3. El subprograma vuelve a trabajar con la variable global. Pero en el intervalo 2, cualquier otros subprograma puede haber hecho cualquier otra cosa con ese. valor

El primer problema es serio, el segundo, mucho peor. Los veremos en los siguientes ejemplos.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ambitos_01;
var
    // Nunca se deben declarar variables aqui. Ahora lo hacemos para
    // mostrar los problemas
    nombre: string ;   // Variable global, suspenso seguro.

procedure p( nombre: string);
    // El argumento n oculta la variable global
begin
    writeln( nombre );  // Escribe María, el valor local, no el global
end;
```

```
begin
    nombre := 'Juan' ;        // Definimos la variable global
    p( 'María' );
end.



{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ambitos_02;
var
    // Nunca se deben declarar variables aqui. Ahora lo hacemos para
    // ver los problemas

    nombre: string ;   // Variable global, suspenso seguro

function averigua_aeropuerto(codigo_iata: string):string;
begin
    if codigo_iata = 'MAD' then
        nombre := 'Madrid-Barajas Adolfo Suárez'
    else
        nombre := 'Aeropuerto Desconocido';
    result := nombre;
end;
   // Esta función está machacando la variable global

var
    codigo_iata:string;        // Variables correctas, locales
    nombre_aeropuerto:string;

begin
    nombre := 'Juan García' ;      // Definimos la variable global
    codigo_iata := 'MAD' ;
    nombre_aeropuerto := averigua_aeropuerto(codigo_iata);
    writeln('Pasajero:', nombre);
    writeln('Aeropuerto destino:', nombre_aeropuerto);
end.
```

Resultado:

```
Pasajero:Madrid-Barajas Adolfo Suárez
Aeropuerto destino:Madrid-Barajas Adolfo Suárez
```

Resolución de ecuaciones de 2º grado

Versión básica https://gsyc.urjc.es/ mortuno/fpi/ecuacion_grado2.pas

Versión mejorada https://gsyc.urjc.es/ mortuno/fpi/ecuacion_grado2_v02.pas

6. Definición de tipos y registros

6.1. Nuevos tipos de datos

Nuevos tipos datos

Ya conocemos los tipos de datos primitivos básicos en Pascal: integer, real, boolean, char, string.

- Una variable integer (entera) contendrá un dato de tipo integer, serán *literales* como p.e. 2, 17 o -1250.
- Una variable real contendrá un dato de tipo real, con literales como 4.3909999999999997E+001 o -5.7294910000000003E+004.
- Una variable boolean contendrá un dato boolean, con los literales TRUE o FALSE.
- Una variable de tipo char (carácter) caracter contendrá datos de tipo char, con literales como 'V', '@' o '3'.
- Una variable de tipo string (cadena) contendrá un dato de tipo string, con literales como 'hola, mundo' o 'x'.

Tipos enumerados

Pero podemos definir nuevos tipos de datos para manejar en el programa entidades abstractas propias de nuestro problema.

- Tipos enumerados:

  Contendrán literales con sentido en el universo de nuestro problema, como *sota*, *caballo*, *rey*

  o

  despegue, *ascenso*, *descenso*.

Tipos Enumerados

Definimos un nuevo tipo a base de enumerar sus elementos, que serán literales especificados por nosotros, literales propios del dominio de nuestro problema.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program enumerados_01;
type
    TipoDiaSem = ( lun, mar, mie, jue, vie, sab, dom );
```

```
    TipoCarta = ( uno, dos, tres, cuatro, cinco, seis, siete,
        sota, caballo, rey );

    TipoPalo = ( oros, bastos, copas, espadas );
var
    carta: TipoCarta;
const
    No_laborable : TipoDiaSem = dom;
begin
    carta := uno;
    writeln(carta);   // Escribe  uno
    writeln(No_laborable);   // Escribe dom
end.
```

Como sabes, la declaración de una variable es p.e.

```
var
    x : real;
```

La definición de un nuevo tipo es p.e.

```
type
    TipoFigura = (sota, caballo, rey);
```

- La definición empieza tras la palabra reservada *type*. Sin ';'.
- Es conveniente poder distinguir fácilmente los identificadores que correspondan a nuevos tipos de datos, en este curso estableceremos el convenio de que el nombre del tipo empiecen por la palabra *Tipo* y con *NotacionCamello*.
    - Juntar palabras sin guiones ni barras bajas, poniendo en mayúscula la primera letra de cada palabra.
- Después del nombre de tipo se escribe '=', ni ':' ni ':='.
- Definimos el tipo enumerando todos sus elementos, entre paréntesis y finalizando cada uno en ';'.

- Hemos visto que el número 3 y el caracter '3' son distintos, aunque se vean iguales en pantalla.
- Por el mismo motivo, la cadena *oros* y el TipoPalo *oros* son distintos, aunque se vean iguales en pantalla.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program enumerados_02;
type
    TipoPalo = ( oros, bastos, copas, espadas );
var
    palo: TipoPalo;
    s: string;
begin
    palo := oros;
    s := 'oros';
    writeln(palo)  ;  // oros
    writeln(s);     // oros

  {  palo := 'oros'; ¡ERROR!  palo es de TipoPalo, no string   }
  {  s := oros;      ¡ERROR!  s es de tipo string, no TipoPalo }
end.
```

Hay lenguajes que no tienen tipos enumerados, se puede programar sin ellos, empleando por ejemplo cadenas.

Pero los enumerados tienen muchas ventajas, nombraremos 4 (hay más).

- Ventaja 1.

  No hace falta filtrar valores erróneos, con código como

  ```
  if (valor <> 'oros') and (valor <> 'bastos')
      and (valor<>'copas') and (valor <> 'espadas') then error();
  ```

  (Porque `valor` siempre estará dentro de los enumerados, el compilador impediría lo contrario).

- Ventaja 2.

  Los elementos tienen orden, podemos escribir expresiones como

  `(carta > sota) and (carta < rey)`

- Ventaja 3

  Disponemos de las funciones `succ()` y `pred()`, que devuelven el valor posterior y el valor anterior, respectivamente.

- Ventaja 4.

  Disponemos de las funciones `high(TIPO_ENUMERADO)` y `low(TIPO_ENUMERADO)` que nos devuelven el mayor y el menor valor posible dentro del tipo enumerado.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program enumerados_03;
type
    TipoTemperatura = ( muy_frio, frio, caliente, muy_caliente);
var
    t: TipoTemperatura;
begin
    t := frio;

    if t < caliente then
        writeln('Encender calefaccion');  // Escribe el mensaje

    writeln(high(TipoTemperatura)) ;    // muy_caliente
    writeln(low(TipoTemperatura)) ;     // muy_frio
    writeln(succ(t)) ;                  // caliente
    writeln(pred(t));                   // muy_frio
end.
```

6.2. Registros

Registros

Ya conocemos los siguientes tipos:

- Tipos primitivos (boolean, integer, real, char, string).

- Tipos enumerados.

Todos ellos son *tipos elementales*, definen un único elemento, un único valor. Además de estos, en Pascal como en casi cualquier lenguaje de programación, tenemos tipos compuestos, formados por la agregación de varios tipos simples.

Veremos ahora los *registros*, un tipo de datos compuesto.

Es muy común que tengamos datos variados pero con mucha relación entre ellos, que queramos tratar como una única cosa.

- Porque sean propiedades del mismo ente.
- Porque se usan juntos habitualmente.
- Porque son la salida de una función.
- ...

Podemos definir un nuevo tipo de datos *record* (registro), como una serie de datos elementales agregados.

A cada elemento de un registro se le llama *campo*.

Definición de un registro

Para definir un tipo de datos registro:

- Usamos la palabra reservada *type*, como para los demás tipos.
- El nombre del tipo lo escribimos en NotacionCamello, anteponiendo la palabra *Tipo*.
- Después del nombre, el carácter '=', ni ':' ni ':='.
- Palabra reservada *record*.
- Declaramos el nombre y tipo de cada elemento, finalizado en ';'.
- Concluimos con *end*.
    - Observa que es un *end* similar al de *case*: es imprescindible y no llega ningún *begin* asociado.

```
type
    TipoNotas = record
        entrega : real;
        teorico : real;
        practico : real;
    end;
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program registros_01;
type
    TipoNotas = record
        entrega : real;
        teorico : real;
        practico : real;
    end;
var
    nota_jperez : TipoNotas;
    nota : TipoNotas;
begin
    nota_jperez.entrega := 7.5;
    nota_jperez.teorico := 4.3;
    nota_jperez.practico := 4.2;

    // Ahora nota_jperez es una única variable que contiene
    // los tres valores

    nota := nota_jperez;
    // Podemos copiarlos todos a la vez en otra variable

    // writeln(calcula_media(nota_jperez));
    // Podemos pasarlos a una función como un único parámetro
end.
```

Para acceder a cada campo, escribimos el nombre del registro, un punto y el nombre del campo.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program registros_02;
```

```
type
    TipoNotas = record
        entrega : real;
        teorico : real;
        practico : real;
    end;

var
    nota_jperez : TipoNotas;
    nota_media : real;
begin
    nota_jperez.entrega := 7.5;
    nota_jperez.teorico := 4.3;
    nota_jperez.practico := 4.2;

    nota_media :=  ( nota_jperez.entrega + nota_jperez.teorico +
                     nota_jperez.practico) / 3 ;
end.
```

No podemos escribir todo el registro con una única llamada a `write()` o `writeln()`, estos procedimientos no saben manejar los registros. Es necesario escribir campo a campo.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program registros_03;
type
    TipoNotas = record
        entrega : real;
        teorico : real;
        practico : real;
    end;
var
    nota_jperez : TipoNotas;
begin
    nota_jperez.entrega := 7.5;
    nota_jperez.teorico := 4.3;
    nota_jperez.practico := 4.2;

   {writeln( nota_jperez ); ERROR, no se pueden escribir directamente}
```

```
    writeln(nota_jperez.entrega:0:2);
    writeln(nota_jperez.teorico:0:2);
    writeln(nota_jperez.practico:0:2);
end.
```

Lo más razonable es usar un procedimiento, escrito por nosotros, que sí sepa manejar nuestro registro.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program registros_04;
type
    TipoNotas = record
        entrega : real;
        teorico : real;
        practico : real;
    end;
procedure escribe_nota(nota:TipoNotas);
begin
    writeln('Entrega de prácticas:',nota.entrega:0:2);
    writeln('Examen teórico:',nota.teorico:0:2);
    writeln('Examen práctico: ',nota.practico:0:2);
end;
var
    nota_jperez : TipoNotas;
begin
    nota_jperez.entrega := 7.5;
    nota_jperez.teorico := 4.3;
    nota_jperez.practico := 4.2;
    escribe_nota(nota_jperez);
end.
```

Pascal tampoco sabe comparar registros, tendremos que programar nuestra función.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program registros_05;
type
    TipoNotas = Record
```

```
        entrega : real;
        teorico : real;
        practico : real;
    end;

function reales_iguales(r1, r2, epsilon: real):boolean;
// No es correcto comparar dos reales con la expresión r1=r2
// porque podría haber una mínima diferencia provocada
// por los errores de redondeo. (Si fueran enteros sí
// sería correcto)
begin
    result := abs(r1-r2) < epsilon;
end;


function notas_iguales(n1,n2:TipoNotas; epsilon:real): Boolean;
begin
    result := reales_iguales(n1.entrega, n2.entrega, epsilon) and
        reales_iguales(n1.teorico, n2.teorico, epsilon) and
        reales_iguales(n1.practico, n2.practico, epsilon);
end;

const
    Epsilon = 0.01;
var
    nota_jperez, nota_mgarcia : TipoNotas;
begin
    nota_jperez.entrega := 7.5;
    nota_jperez.teorico := 4.3;
    nota_jperez.practico := 4.2;
    nota_mgarcia.entrega := 7.5;
    nota_mgarcia.teorico := 4.3;
    nota_mgarcia.practico := 4.1999987622;

  { writeln (nota_jperez = nota_mgarcia);  ¡Mal!   }
    writeln( notas_iguales(nota_jperez,nota_mgarcia, Epsilon))
       // TRUE
end.
```

https://gsyc.urjc.es/ mortuno/fpi/registros_05.pas

Distancia entre dos puntos

Otro caso donde claramente es conveniente usar registros: coordenadas de un punto.

Por ejemplo, coordenadas cartesianas de un punto en el plano.

- Un punto será un registro.
- Los campos x e y contendrán sus coordenadas.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program coordenadas;
uses math;

type
    TipoPunto= record
        x: real;
        y: real;
    end;

function distancia(a, b: TipoPunto): real;
begin
    result := sqrt((b.x-a.x)**2 + (b.y-a.y)**2);
end;
```

Distancia entre dos puntos en un plano:
https://tinyurl.com/yxxvec3v

```
procedure escribe_punto(a: TipoPunto);
begin
    write('(', a.x:0:3);
    write(',');
    write(a.y:0:3,')');
end;
```

```
var
    p1,p2 : TipoPunto;

begin
    p1.x := 4;
    p1.y := 0;
    p2.x := -1;
    p2.y := 0;

    write('La distancia entre ');
    escribe_punto(p1);
    write(' y ');
    escribe_punto(p2);
    writeln(' es ', distancia(p1,p2):0:3);
end.
```

```
La distancia entre (1.000,3.000) y (0.000,-1.000) es 4.123
```

https://gsyc.urjc.es/ mortuno/fpi/distancia.pas

Devolver varios valores en una función

Sabemos que una función devuelve exactamente 1 valor. Si necesitamos más, tenemos dos soluciones:

1. Usar un procedimiento con parámetros de salida, como vimos en el tema 5.

2. Devolver un registro.

En el siguiente ejemplo, la función *division_ entera* devuelve un registro con el cociente y el resto de una división entera.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program devolver_registro;
type
    TipoResulDiv = record
        cociente : integer;
```

```
        resto : integer;
    end;

function division_entera(dividendo, divisor: integer):TipoResulDiv;
begin
    result.cociente := dividendo div divisor;
    result.resto := dividendo mod divisor;
end;

procedure escribe_resultado(dividendo, divisor: integer;
    resultado: TipoResulDiv);
begin
    write('La división entera entre ', dividendo);
    write(' y ', divisor);
    write(' es ', resultado.cociente);
    writeln(' con un resto de ', resultado.resto);
end;

var
    dividendo, divisor : integer;
    resultado : TipoResulDiv;
begin
    dividendo := 9;
    divisor := 2;

    resultado := division_entera(dividendo, divisor);
    escribe_resultado(dividendo, divisor, resultado);

end.
```

```
var
    dividendo, divisor : integer;
    resultado : TipoResulDiv;
begin
    dividendo := 9;
    divisor := 2;

    resultado := division_entera(dividendo, divisor);
```

```
    write('La división entera entre ', dividendo);
    write(' y ', divisor);
    write(' es ', resultado.cociente);
    writeln(' con un resto de ', resultado.resto);
end.
```

```
La división entera entre 9 y 2 es 4 con un resto de 1
```

Devolver varios valores en una función

Una situación habitual donde resulta muy conveniente que una función devuelva un registro es el siguiente:

- Si todo ha ido bien, queremos devolver el valor calculado.
- Si algo ha fallado (precondición, postcondición o cualquier otro problema), queremos indicarlo.

Para ello devolvemos un registro con dos campos:

- Un código que indique si hubo problemas o no.
- El valor, si todo fue bien. Si el código indica error, esta valor es irrelevante.
    - Distintos lenguajes y liberías usan distintos convenios. P.e. el procedimiento `val`, y muchos otros, devuelve un entero con valor 0 en este caso.

En el siguiente ejemplo, devolveremos una cadena.

- *ok*, para indicar la ausencia de errores.
- En otro caso, la descripción del problema.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program inverso_02;
type
    TipoResultado= record
        codigo:string;
        valor: real;
    end;
```

```
function inverso(x: real): TipoResultado;
begin
    if x = 0 then
        result.codigo := 'Error, el divisor no puede ser nulo'
    else begin
        result.codigo := 'ok';
        result.valor := 1/x;
    end;
end;

procedure escribe_inverso(x:real; calculado: TipoResultado);
begin
    if calculado.codigo = 'ok' then begin
        write('El inverso de ', x:0:3 );
        writeln(' es ', calculado.valor:0:3);
    end
    else
        writeln( calculado.codigo );
end;

var
    x : real;
    calculado : TipoResultado;

begin
    writeln('Escribe un número');
    readln(x);

    calculado := inverso(x);
    escribe_inverso(x, calculado);
end.
```

https://gsyc.urjc.es/ mortuno/fpi/inverso_02.pas

7. Bucles

7.1. Introducción a los bucles

Bucles

Un bucle es una estructura que permite ejecutar una o más sentencias todas las veces necesarias. En Pascal, como en casi cualquier lenguaje de programación tenemos instrucciones para hacer bucles...

- Mientras se cumpla cierta condición:

```
while CONDICION_DE_PERMANENCIA do
    SENTENCIAS
```

- Hasta que se cumpla cierta condición:

```
repeat
    SENTENCIAS
until CONDICION_DE_SALIDA
```

- Un número predeterminado de veces:

```
for VARIABLE := INICIO to FIN do
    SENTENCIAS
```

Antes de escribir un bucle, necesitamos tener claro lo siguiente:

- ¿Sabemos el número de veces que se ejecutará el bucle? (Es un valor en una variable, una constante, una expresión o una función).

  Sí $\implies$ Lo más adecuado es `for`.

  No:

  - ¿Se va a ejecutar la primera vez?.
    Seguro que sí $\implies$ Lo más adecuado es `repeat until`.
    Tal vez no $\implies$ La única opción es `while do`.

- ¿Qué hay que hacer en cada iteración?.
- ¿Cuál es la condición de salida?.

Con `while do` podemos hacer cualquier tipo de bucle, aunque hay ocasiones donde `for` o `repeat until` es más adecuado (más sencillo).

7.2. Bucles while

Bucles while

Ejemplo de bucle while mínimo. Ejecutar algo 3 veces.

(Más adelante veremos que `for` sería más adecuado, pero empezaremos por while, que sirve para cualquier tipo de bucle).

- Tendremos una variable que pasa por los estados 1, 2 y 3.
- Como norma general las variables tienen que tener nombres completos y descriptivos, pero tradicionalmente en este caso se hace una excepción y se le suele llamar `i` (por *indice*).
- Repite una sentencia mientras la condición de permanencia `(i <= 3)` sea cierta.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_00;
var
    i: integer;
begin
    i := 1;
    while i <= 3 do
        i := i + 1;
    writeln(i);   // Escribe 4
end.
```

- Este ejemplo es poco práctico, raramente repetiremos solamente una sola sentencia.

Ejemplo más realista: repetir un bloque begin-end mientras la condición de permanencia sea cierta.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_01;
var
    i: integer;
begin
    i := 1;
    while i <= 3 do begin
        writeln( 'Probando bucles');
```

```
        i := i + 1;
    end;
end.
```

Resultado:

```
Probando bucles
Probando bucles
Probando bucles
```

En Pascal, la forma habitual de hacer algo `n` veces es contar desde 1 hasta n, como hemos visto.

Pero también podemos contar desde 0 hasta n-1, que es equivalente (y lo habitual en otros lenguajes).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_02;
var
    i: integer;
begin
    i := 0;
    while i < 3 do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

Resultado:

```
Probando bucles
Probando bucles
Probando bucles
```

- El error más habitual en este tipo de programas es confundir el 0 con el 1, el `n` con el `n-1`, el < con el <= ,etc.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_03;
var
```

```
    i: integer;
begin
    i := 1;
    while i < 3 do begin   // Cuidado, este son 2 ejecuciones
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

1. Otro error frecuente es olvidar actualizar la condición de salida `i := i + 1`. En este caso el programa entraría en un *bucle infinito*. Tendríamos que abortarlo desde el terminal con `ctrl c`.

- Observa que, con enteros, $i < 4 \Leftrightarrow i <= 3$.

  Podemos usar cualquiera de las dos expresiones.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_04;
var
    i: integer;
begin
    i := 1;
    while i < 4 do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

En los ejemplos anteriores hemos escrito `3` o `4` como *constantes literales*, esto es, como números *tal cual*.

- Así, estos ejemplos han quedado más claros. Los humanos entedemos mejor `3` que `n`.
- Pero en un programa normal no deberíamos hacerlo, esto sería un *número mágico*.

Los números mágicos tienen (al menos) dos problemas:

1. No queda claro *de dónde sale*. Ponerle un nombre (usar una constante) resulta más claro.
2. Si necesitamos cambiarlo, basta modificar la definición de la constante, no es necesario buscarlo y cambiarlo por todo el código.

El mismo bucle, sin números mágicos:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_05;
const
    N = 3;
var
    i: integer;
begin
    i := 1;
    while i <= N do begin
        writeln( 'Probando bucles');
        i := i + 1;
    end;
end.
```

Estos ejemplos donde la condición de permanencia en el bucle es la comparación de un contador con una constante, son buenos para ilustrar el uso de `while`.

- Aunque veremos enseguida que `for` sería más adecuado.

Para usar `while` de una manera más realista, necesitamos algo que el programador no conozca en el momento de escribir el código fuente.

Podría ser:

- Una entrada del usuario.
- La lectura de un fichero o cualquier otro dato.
    - Realmente, para el programa leer la entrada del usuario es leer un fichero.
- Una consulta al reloj.
- Un número aleatorio.
- ...

7.3. Números aleatorios

Generación de números aleatorios

En ocasiones necesitamos que un programa genere números aleatorios.

- Un ordenador no es capaz de generar números verdaderamente aleatorios.
- A menos que disponga de hardware específico para ello. Ejemplos: [1], [2], [3]

Normalmente nos basta usar números *pseudoaletorios*:

- A partir de un número inicial denominado *semilla*, de forma matemática se genera una serie de números casi aleatorios.
- Es necesario cambiar la semilla en cada ejecución del programa, si no, la secuencia pseudoaleatoria siempre es la misma.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_00;
begin
    writeln(random);
    writeln(random);
    writeln(random);
end.
```

Este programa, como no cambia la semilla, siempre devuelve la misma serie (los mismos tres números reales).

Para observar bien estos programas con números aleatorios, ejecutalos en tu ordenador. Puedes descargar todos los ejemplos en

https://gsyc.urjc.es/ mortuno/fpi/tema07.zip

- La función `randomize` (sin argumentos) inicializa la semilla, usando la hora del sistema, en segundos.
- La función `random` (sin argumentos) devuelve un número real pseudoaleatorio, $0 \leq numero < 1$.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_01;
begin
    randomize;  // Cambia la semilla del generador de números
                // pseudoaleatorios. Usa la hora, en segundos
    writeln(random);
    writeln(random);
    writeln(random);
end.
```

Ahora el programa *baraja* la semilla inicial, con lo que en cada ejecución devuelve una serie distinta.

- Si necesitamos un número real entre 0 y n, basta multiplicar el valor que devuelve `random` por n.
- Si necesitamos un entero, truncamos.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_02;
const
    N = 10;
begin
    randomize;

    writeln(random * N) ;
    // Nº real mayor o igual que 0, menor que N

    writeln( trunc( random * N) + 1 ) ;
    // Nº entero entre 1 y N
end.
```

Es muy común necesitar un número entero entre 0 y n-1,.

- También se puede conseguir directamente pasando un argumento (**entero**) a `random()`.
- `random(n)`, devuelve un número entero pseudoaleatorio, $0 \leq numero < n$.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_03;
const
    N = 6;
begin
    randomize;
    writeln( random(N) + 1);  // Número de 1 a 6
end.
```

Recuerda, esto es solo aplicable a los enteros. Para reales, `random * N`).

Para reutilizar este código, escribimos una función.

```
1  {$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
2  program dado_01;
3
4  function tira_dado(caras_dado:integer):integer;
5  begin
6      result := random(caras_dado) + 1 ;
7  end;
8
9  const
10     Caras_dado = 6;
11 begin
12     randomize;
13     writeln( tira_dado(Caras_dado) );
14 end.
```

- En la línea 10 definimos la constante local `caras_dado`.
- En la línea 13 llamamos a la función `dado` pasando 6 como argumento.
- En el cuerpo de la función, línea 6, el parámetro `caras_dado` tiene el mismo nombre y el mismo valor (en este ejemplo) que la constante local `Caras_dado`, pero son dos cosas distintas.

Un problema del uso de `randomize` se basa en la hora en segundos, por lo que todos los números generados el mismo segundo, tendrán la misma semilla.

- Para evitarlo, usamos el procedimiento `delay()` que detiene la ejecución del programa el tiempo que indiquemos. Así nos aseguramos de que la semilla sea siempre distinta.

  Si estamos seguros de que no vamos a pedir dos números aleatorios seguidos en el mismo segundo, no hace falta usar `delay()`.

- Para poder usar `delay` es necesario incluir el módulo `crt`, añadiendo después de *program*

  `uses crt;`.

- Para usar más de un módulo, los separamos por comas.

  `uses math, crt;`

- Observa que tanto `delay` como `randomize` los ejecutamos una sola vez, en el cuerpo principal del programa. No en cada llamada a la función `tira_dado()`.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program dado_02;
uses crt;    // Necesario para delay

function tira_dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

const
    Caras_dado = 6;

begin
    delay(1000);   // 1000 milisegundos = 1 segundo
    randomize;
    writeln( tira_dado(Caras_dado) );
end.
```

- Cada vez que llamamos a la función *tira_dado()* se genera un nuevo valor aleatorio, diferente (salvo casualidad).

- Si queremos hacer varias cosas diferentes con el valor del dado (por ejemplo escribirlo y sumarlo), habrá que guardar el valor del dado en una variable y usar esa variable.
- Un error frecuente entre principiantes es el mostrado a continuación, donde sumamos el valor de dos dados.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program dado_03_mal;
uses crt;    // Necesario para delay
const
    Caras_dado : integer = 6;

function dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

begin
    delay(800);
    randomize;
    write( 'primer dado:');
    writeln( dado(Caras_dado) );

    write( 'segundo dado:');
    writeln( dado(Caras_dado) );

    write( 'suma:');
    writeln( dado(Caras_dado) + dado(Caras_Dado));  // ¡¡MAL!!
end.
```

Resultado:

```
primer dado:3
segundo dado:3
suma:4
```

Este resultado no tiene sentido, porque las tiradas del dado que estamos usando para escribir son distintas de las empleadas para el cálculo. El ejemplo a continuación corrige el error.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program dado_04;
uses crt;
const
    Caras_dado : integer = 6;

function dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

var
    dado1,dado2: integer;

begin
    delay(800);
    randomize;
    dado1 := dado(Caras_dado);
    write( 'primer dado:');
    writeln( dado1);

    dado2 := dado(Caras_dado);
    write( 'segundo dado:');
    writeln( dado2);

    write( 'suma:');
    writeln( dado1 + dado2);
end.
```

```
primer dado:2
segundo dado:5
suma:7
```

Tiremos 3 dados

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_06;
uses crt;
```

```
function tira_dado(caras_dado:integer):integer;
begin
    result := random( caras_dado ) + 1 ;
end;

var    // Recuerda que si las variables se definen antes de
       // las funciones --> son globales --> suspenso seguro
    i: integer;
const
    Caras_dado = 6;
    N = 3;
begin
    delay(1000);
    randomize;
    i := 1;
    while i <= N  do begin
        writeln( tira_dado(Caras_dado));
        i := i + 1;
    end;
end.
```

Observa este problema:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program while_07;
uses crt;

function tira_dado(caras_dado:integer):integer;
begin
    randomize;   //  ¡Mal!
    result :=  random( caras_dado ) + 1
end;

const
    Caras_dado = 6;
    N = 3;
var
    i: integer;
begin
    delay(1000);
```

```
    i := 1;
    while i <= N do begin
        writeln( tira_dado(Caras_dado));
        i := i + 1;
    end;
end.
```

Resultado:

```
4
4
4
```

Siempre salen tres números iguales: tal vez 4 4 4, tal vez 2 2 2, otras veces 6 6 6, etc.

- El problema es que invocamos a `randomize` (*barajamos*) cada vez que tiramos un dado. El segundero no habrá cambiado, y por tanto el resultado de barajar siempre será el mismo.
- La solución es invocar a `randomize` una vez y solo una vez. El lugar más adecuado es al comienzo del programa principal.

Otro ejemplo: parchís. Tiramos dados hasta que salga 5. Este ya es un ejemplo donde es más adecuado: `while` que `for`

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program salida_parchis_01; // Tira dados hasta que salga 5
uses crt;
function tira_dado(caras_dado:integer):integer;
begin
    result := random(  caras_dado ) + 1 ;
end;
const
    Caras_dado : integer = 6;
    Dado_salida : integer = 5;
var
    puntos: integer;
begin
    delay(1000);
```

```
    randomize;
    puntos := 0;  // Tenemos que iniciar puntos, con un valor
             // que fuerce la primera ejecución del bucle
    while puntos <> Dado_salida do begin
        puntos := tira_dado(Caras_dado);
        writeln( puntos );
    end;
end.
```

Cuando usamos números aleatorios, prácticamente siempre habrá que:

- Generar el número aleatorio según lo especificado
- Almacenarlo en una variable
- Usar esa variable para mostrarlo y usarlo en cualquier sentencia de control

Ejemplo:

- Supongamos que queremos tirar un dado hasta que salga un valor superior o igual a un valor objetivo
- Empecemos con una versión incorrecta

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_mal;
[...]  // Declaramos dado() y Caras_dado como siempre
var
    objetivo: integer;

begin
    randomize;
    objetivo := 4;
    repeat
        writeln(dado(Caras_dado));
    until dado(Caras_dado) >= objetivo;  // ¡¡MAL!!
end.
```

Ejemplo de ejecución:

```
2
5
3
```

El problema de la versión anterior es que tirábamos un dado, escribíamos su valor en pantalla, pero para la condición de salida, volvíamos a tirar un nuevo dado. Versión correcta:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program random_ok;
[...]  // Declaramos dado() y Caras_dado como siempre
var
    puntos: integer;
    objetivo: integer;

begin
    randomize;
    objetivo := 4;
    repeat
        puntos := dado(Caras_dado);
        writeln(puntos);
    until puntos >= objetivo;
end.
```

Ejemplo de ejecución:

```
2
5
```

Las chapas es un juego de azar donde los jugadores apuestan sobre el resultado del lanzamiento de dos monedas.

https://es.wikipedia.org/wiki/Chapas_(juego_de_apuestas)

- Cada jugador apuesta por *cara* o *cruz* y lanza dos monedas.
- Se considera *cara* si salen dos *caras*, se considera *cruz* si salen dos *cruces*.
- En otro caso, se lanzan las monedas las veces que sean necesarias.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program chapas_01;
    // Tira dos monedas hasta que sean iguales
```

```
uses crt;

   // Definimos la función tira_dado() como en ejemplos anteriores

function tira_moneda(): string;
var
    valor : integer;
begin
    valor := tira_dado(2);
        // Permitimos este número mágico porque todas
        // las monedas tienen 2 caras
    if valor = 1 then
        result := 'cara'
    else
        result := 'cruz';
end;

var
    moneda1, moneda2 : string;
begin
    delay(1000);
    randomize;
    moneda1 := 'cara';  // Iniciamos la monedas, con un valor
    moneda2 := 'cruz';  // que fuerce la primera ejecución
    while moneda1 <> moneda2 do begin
        moneda1 := tira_moneda();
        moneda2 := tira_moneda();
        writeln( moneda1, ' ', moneda2);
    end;
end.
```

7.4. Bucles repeat

Bucles repeat

En los ejemplos del parchís y de las chapas, estábamos seguros de que era necesario ejecutar el bucle al menos una vez.

Antes de la sentencia while...

- Aún no teníamos ningun valor para los dados o las monedas.
- Pero forzamos la condición de permanencia en el bucle para que la primera vez, siempre fuera cierto.

Los bucles `repeat` son más cómodos para estos casos donde sabemos que el bucle se tiene que ejecutar al menos una vez, y evaluar luego la condición de salida.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program repeat_01;
var
    i: integer;

begin
    i := 1;
    repeat
        writeln( 'Bucle repeat');
        i := i + 1;
    until i = 4;
end.
```

Ejecución:

```
Bucle repeat
Bucle repeat
Bucle repeat
```

- Observa que `repeat` es la única sentencia que espera una lista de sentencias, no hace falta usar begin end.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program salida_parchis_repeat;
    // Tira dados hasta que salga 5
uses crt;


function tira_dado(caras_dado:integer):integer;
begin
    result := random(  caras_dado ) + 1 ;
end;

const
```

```
    Caras_dado: integer = 6;
var
    puntos: integer;
begin
    delay(1000);
    randomize;
    repeat
        puntos := tira_dado(Caras_dado);
        writeln( puntos );
    until puntos = 5;
end.
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program chapas_repeat;
    // Tira dos monedas hasta que sean iguales
uses crt;

// Definimos la función tira_moneda() como en los casos anteriores

var
    moneda1, moneda2 : string;

begin
    delay(1000);
    randomize;
    repeat
        moneda1 := tira_moneda();
        moneda2 := tira_moneda();
        writeln( moneda1, ' ', moneda2);
    until moneda1 = moneda2;
end.
```

Observa la versión de estos dos programas usando `repeat` y no `while do`
Ya no hace falta iniciar las variables antes del bucle, se les da valor dentro del cuerpo del bucle, antes de evaluar la expresión de salida.

- La ejecución es equivalente.

- Pero el código es más sencillo, más *elegante* $\Longrightarrow$ los errores del programador son menos probables.

Observa también que

```
repeat
    sentencias
until condicion_salida
```

equivale a

```
{forzar condicion_salida = FALSE}
while not condicion_salida do begin
    sentencias
end
```

La condición de permanencia es un booleano. La condición de salida es otro booleano, la negación del anterior.

- Ya sabemos aplicar doble negación, De Morgan, etc.

En lenguaje natural sucede lo mismo.
diviértete hasta que no puedas más
diviértete mientras puedas

Bucles repeat y lenguaje natural

No es raro que nos encontremos una especificación incorrecta en lenguaje natural parecida a la siguiente:

```
Leer un valor
Si es erróneo, descartarlo y volver a leer
```

Una implementación ingenua, pero que cumple la letra de lo pedido sería:

```
valor := lee_valor
if valor_erroneo(valor) then
    valor := lee_valor;
```

Difícilmente esto será lo deseado, puesto que si el segundo valor vuelve a ser erróneo, no se corrige.

Probablemente el autor de esa especificación esperaba que se sobreentendiera lo siguiente:

```
Repite la lectura de un valor
    Hasta que no sea erróno
```

- Los *sobreentendidos* en ingeniería no son aceptables. Todo debe ser explícito.
- Si (por algún extraño motivo) realmente se deseara corregir una vez y solo una, habría que dejarlo muy claro.

```
Leer un valor
Si es erróneo, descartarlo y volver a leer

Si a la segunda es erróneo, nos lo quedamos. Ya no leemos de nuevo
```

Lectura desde teclado de número dentro de rango

Vamos a pedir al usuario que escriba un número entre 0 y 10.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program lectura_numero;

procedure lee_numero(limite_inf, limite_sup: integer; var n:integer);
var
    sal : boolean = FALSE;
    s : string;
    codigo : integer;
begin
    repeat
        write('Introduce un número entero entre ');
        write(limite_inf, ' y ' , limite_sup);
        writeln;
        readln(s) ;
        val(s, n, codigo);
        if (codigo = 0 ) and (n >= limite_inf) and (n <= limite_sup) then
            sal := True
        else
            writeln('Error, ',s,' no es un entero en el rango pedido');
    until sal;
end;


const
    LimiteInf = 0;
    LimiteSup = 10;
var
    numero: integer;

begin
    lee_numero(LimiteInf, LimiteSup, numero);
    writeln('Número indicado por el usuario: ',numero);
end.
```

Resultado:

```
Introduce un número entero entre 0 y 10
jj
Error, jj no es un entero en el rango pedido
Introduce un número entero entre 0 y 10
```

```
12
Error, 12 no es un entero en el rango pedido
Introduce un número entero entre 0 y 10
0
Número indicado por el usuario: 0
```

Observaciones:

- El límite inferior y el superior se define en constantes locales al cuerpo del programa principal, que luego se pasan como parámetro al procedimiento.
- El parámetro `n` se pasa por referencia, para que el valor generado dentro del procedimiento no se pierda al salir del procedimiento.

7.5. Bucles for

Bucles for

Cuando se conoce de antemano el número de veces que se debe ejecutar el bucle, lo más adecuado es usar `for`.

- Este *conocimiento* puede ser en el momento de escribir el programa, lo que se llama *en tiempo de compilación*. En ese caso, el número de iteraciones será una constante.
- En otras ocasiones se conocerá al ejecutar el programa. Se llama *en tiempo de ejecución*. El valor será una variable, una expresión o el resultado de una función.

```
for variable_de_control := valor_inicial to valor_final do
    sentencia;
```

o bien

```
for variable_de_control := valor_inicial to valor_final do begin
    sentencia_1
    sentencia_2
    ..
    sentencia_2
end
```

La variable de control puede ser de cualquier tipo *ordinal*, esto es, discreto, que tenga un número finito de elementos: enteros, caracteres, booleanos o cualquier enumerado.

- La variable de control nunca puede sear real, intentarlo generaría un error de compilación.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_01;
const
    N = 3;
var
    i : integer;
begin
    for i := 1 to N do
        writeln( 'Bucle for');
end.
```

Resultado:

```
Bucle for
Bucle for
Bucle for
```

En un programa no debe haber *números mágicos*, pero considerar el número 1 como mágico seguramente es excesivo. Especialmente en Pascal, donde lo habitual es empezar a contar en 1 (y no en 0).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_02;
const
    N = 3;
var
    i : integer;  // i, j, k son nombres tradicionales para índices:
                  // variables que iteran sobre enteros
begin
    for i := 1 to N do begin
```

```
        write( i, ' ' );
    end;
    writeln;  // Para añadir nueva línea al final
end.
```

Resultado:

```
1 2 3
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_mal_01;
var
    i: real;  //  ¡Mal!  La variable ha de ser un tipo ordinal

begin
    for i := 1 to 3 do        // Error de compilación
                              // ordinal expression expected
        write( i, ' ' );
    writeln ;
end.
```

Para que el incremento de la variable de control sea negativo, en vez de `to` escribiremos `downto`.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_03;
const
    N = 3;
var
    i : integer;

begin
    for i := N downto 1 do
        write( i, ' ' );
    writeln;
end.
```

Resultado:

```
3 2 1
```

Si por error intentamos hacer un bucle decreciente sin usar *downto*, se ejecutará 0 veces.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_mal_02;
const
    N = 3;
var
    i: integer;

begin
    for i := N to 1 do
            // Error lógico. Se ejecuta 0 veces
        writeln( 'Holamundo de bucles for');
end.
```

Ejemplo con char:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program for_04;
var
    c: char;

begin
    for c := 'a' to 'f' do
        write( c, ' ' );
    writeln;
end.
```

Resultado:

```
a b c d e f
```

Es muy habitual anidar bucles `for`:

```
program matriz;
const
    Filas: integer = 4;
```

```
    Columnas: integer = 5;
var
    i, j: integer;
begin
    for i := 1 to Filas do begin
        for j := 1 to Columnas do
            write(' [',i,',',j,']');
        writeln();
    end;
end.
```

Resultado:

```
[1,1] [1,2] [1,3] [1,4] [1,5]
[2,1] [2,2] [2,3] [2,4] [2,5]
[3,1] [3,2] [3,3] [3,4] [3,5]
[4,1] [4,2] [4,3] [4,4] [4,5]
```

- Observa que el primer bucle ahora tiene más de una sentencia, lo que obliga a usar un bloque begin-end.

Máximo y mínimo

Busquemos el máximo y el mínimo de una serie de dados:

- En una variable guardaremos el mínimo provisional. En otra, el máximo provisional.
- En la primera iteración, el mínimo provisional será el valor obtenido, el único disponible. También será el máximo provisional.
- En las siguientes iteraciones:
  - Si el valor obtenido es menor que el mínimo provisional, ese valor pasará a ser el nuevo mínimo.
  - Si el valor obtenido es mayor que el máximo provisional, ese valor pasará a ser el nuevo máximo.
- Al final de todas las iteraciones, los valores provisionales pasan a ser definitivos.

En una primera solución, usaremos dos sentencias condicionales anidadas.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program min_max_01;
uses crt;

function tira_dado(caras_dado:integer):integer;
begin
    result := random(caras_dado) + 1;
end;

procedure escribe_minmax(minimo,maximo:integer);
begin
    write('Valor mínimo: ',minimo);
    writeln(' Valor máximo: ',maximo);
end;

procedure busca_minmax(n, caras_dado: integer;
    var minimo, maximo: integer);
var
    i : integer;
    resultado : integer;
begin
    for i := 1 to n do begin
        resultado := tira_dado(caras_dado);
        write( resultado, ' ' );
        if i = 1 then begin
            minimo := resultado;
            maximo := resultado;
        end
        else begin
            if resultado > maximo then
                maximo := resultado;
            if resultado < minimo then
                minimo := resultado;
        end
    end;
    writeln;
end;

var
    minimo, maximo : integer;
const
```

```
    N = 6;
    CarasDado = 48;
begin
    delay(1000);
    randomize();

    busca_minmax(N, CarasDado, minimo, maximo);
    escribe_minmax(minimo,maximo);
end.
```

Podemos hacer una solución más elegante con un único condicional:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program min_max_02;
user crt;
// El resto del programa es idéntico al anterior, solo cambia
// este procedimiento
procedure busca_minmax(n, caras_dado: integer;
    var minimo, maximo: integer);
var
    i : integer;
    resultado : integer;
begin
    for i := 1 to N do begin
        resultado := tira_dado(caras_dado);
        write( resultado, ' ' );

        if (i = 1) or (resultado > maximo) then
            maximo := resultado;

        if (i = 1) or (resultado < minimo) then
            minimo := resultado;
    end;
    writeln;
end;
```

Ejemplo: triángulo

Programa que escriba

```
*
* *
* * *
* * * *
* * * * *
```

- Bucle que se ejecuta tantas veces como la altura del triángulo.
- Cada fila tiene tantos *puntos de tinta* como la variable de control.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program triangulo;
procedure escribe_fila(ancho: integer; tinta: string);
var
    i: integer;
begin
    for i := 1 to ancho do begin
        write(tinta);
    end;
    writeln();
end;
procedure escribe_triangulo(altura: integer; tinta: string);
var
    fila: integer;
begin
    for fila := 1 to altura do begin
        escribe_fila(fila, tinta);
    end
end;
const
    Alto = 5;
    Tinta: string = '* ';
begin
    escribe_triangulo(Alto, Tinta);
end.
```

Ejemplo: triángulo invertido

Ahora queremos un programa que escriba

```
*
* *
```

```
* * *
* * * *
* * * * *

* * * * *
* * * *
* * *
* *
*
```

Para obtener la expresión del número de puntos de tinta del triángulo invertido, usamos la técnica habitual:

1. Poner los valores de unos cuantos casos particulares.
2. Generalizar para `n`.

```
Triangulo invertido, ancho 5

fila 1     5 puntos   (5 = 5-1+1)
fila 2     4 puntos   (4 = 5-2+1)
fila 3     3 puntos   (3 = 5-3+1)
fila 4     2 puntos   (2 = 5-4+1)
fila 5     1 puntos   (1 = 5-5+1)

                  ancho - fila + 1
```

https://gsyc.urjc.es/ mortuno/fpi/triangulos.pas

8. Arrays

8.1. Introducción a los arrays

Arrays

Un array es un conjunto de elementos del mismo tipo, con un cierto orden.

- El número máximo de elementos lo debe fijar el programador en el código fuente, no puede cambiar *sobre la marcha*.
- El array ocupa siempre el mismo espacio en memoria, esté lleno o vacío siempre ocupa el máximo tamaño disponible.

La clase más sencilla de array se llama *vector*. Un nombre equivalente, alternativo es *tabla*. Un vector es:

- Un conjunto de elementos del mismo tipo, ordenados mediante un índice (solo uno).

Posteriormente veremos un tipo array un poco más avanzado, llamado *matriz*, que se caracteriza por tener dos índices.

En cualquier tipo de array, los índices suelen ser números naturales.

Empecemos por un ejemplo muy básico. Funciona aunque necesita mejoras.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_01;

var
    vector : array[1..3] of string;  // Número mágico (de momento)

begin
    vector[1] := 'Primero';
    vector[2] := 'Segundo';
    vector[3] := 'Tercero';

    writeln(vector[1]);
    writeln(vector[2]);
    writeln(vector[3]);
end.
```

```
var
    vector : array[1..3] of string;  // Número mágico (de momento)
```

- Declaramos `vector` como `array`.
- Entre corchetes indicamos el ordinal de su primer elemento y el ordinal del último, separados por dos puntos consecutivos (`..`).
- Como es costumbre en Pascal, en este ejemplo el primer elemento es el 1. Podríamos usar el 0, pero resulta menos *idiomático*. También podríamos usar cualquier otro número positivo o negativo, con tal de que sea entero.
- Despues de la palabra reservada `of`, indicamos el tipo de datos que tendrá cada elemento del array.
- Para referirnos a un elemento del array, escribimos el nombre del array y añadimos entre corchetes el índice.

En este ejemplo usamos un índice que no va de 1 a n.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_02;

var
    vector : array[-1..1] of string;

begin
    vector[-1] := 'Sotano';
    vector[0] := 'Planta baja';
    vector[1] := 'Primero';

    writeln(vector[-1]);
    writeln(vector[0]);
    writeln(vector[1]);
end.
```

Normalmente usaremos un bucle para recorrer el array.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_03;

var
    vector : array[1..3] of string;
    i : integer;

begin
    vector[1] := 'Primero';
    vector[2] := 'Segundo';
    vector[3] := 'Tercero';

    for i := 1 to 3 do
        writeln(vector[i]);
end.
```

También podemos inicializar un array (variable o constante) enumerando todos sus elementos, entre paréntesis.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_03bis;

var
    vector : Array[1..3] of string = ('Primero', 'Segundo', 'Tercero');
    i : integer;

begin
    for i:= 1 to 3 do
        writeln(vector[i]);
end.
```

Si intentamos acceder a una posición fuera de rango, obtendremos un error de compilación (gracias a la directiva de compilación `{$R+}` que ponemos en cada programa).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_mal_01;

var
    vector : array[1..3] of string;

begin
    vector[4] := 'Cuarto'; //  ¡Mal!
end.
```

Resultado:

```
Compiling array_mal_01.pas
array_mal_01.pas(8,11) Error: range check error while evaluating constants
(4 must be between 1 and 3)
array_mal_01.pas(10) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
```

Pero si no ponemos directivas al compilador para que detecte los desbordamientos de rango, el programa compilará (con un *warning*) e incluso se ejecutará.

Será un programa vulnerable a todo tipo de problemas de seguridad, una bomba de relojería en potencia.

```
// Sin directivas de compilación
program array_mal_02;

var
    vector : array[1..3] of string;

begin
    vector[4] := 'Cuarto';
    writeln(vector[4]);
end.
```

Resultado:

```
Cuarto
```

En los ejemplos anteriores, declarábamos la variable con un número mágico. Esto es incorrecto, deberíamos usar una constante.

- En otros casos podríamos declarar y definir esta constante así:

```
const
    TamanioVector : integer = 3;  // ¡Mal en este caso!
```

- Pero como esta constante la usaremos para declarar una variable (será el tamaño del array) tenemos que definirla (darle su valor) pero no declararla (indicar su tipo):

```
const
    TamanioVector = 3;   // Correcto
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_04;
const
    TamanioVector = 3;

var
    vector : array[1..TamanioVector] of string;
    i : integer;

begin
    vector[1] := 'Primero';
    vector[2] := 'Segundo';
    vector[3] := 'Tercero';

    for i := 1 to TamanioVector do
        writeln(vector[i]);
end.
```

Hasta ahora declarábamos las variables directamente como arrays del tamaño correspondiente.

```
var
    vector : array[1..TamanioVector] of string;
```

Pero esto es un problema en potencia. Si necesitamos cambiar ese tipo de datos, tendremos que tocar muchas variables y muchos parámetros. Lo adecuado es declarar un nuevo tipo de datos. En este caso, `TipoVector`.

- Por convenio, estos nombres de tipo empezarán por el prefijo *Tipo* y estarán escritos en *NotacionCamello* [22].

```
const
    TamanioVector = 3;
type
    TipoVector = array[1..TamanioVector] of string;
var
    vector : TipoVector;
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_05;
    // Declaramos un tipo para el array

const
    TamanioVector = 3;

type
    TipoVector = array[1..TamanioVector] of string;

var
    vector : TipoVector;
    i : integer;

begin
    vector[1] := 'Primero';
    vector[2] := 'Segundo';
    vector[3] := 'Tercero';

    for i := 1 to TamanioVector do
        writeln(vector[i]);
end.
```

[22] Juntar dos palabras, de forma que cada una empiece por mayúscula

Usemos un vector de enteros para almacenar las tiradas de un dado.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program array_06;  // Guardamos la salida de un dado
uses crt;
const
    TamanioVector = 5;
type
    TipoVector = array[1..TamanioVector] of integer;
// Definimos tira_dado() como en el tema 7
var
    i: integer;
    vector : TipoVector;
const
    CarasDado = 6;

begin
    delay(800);
    randomize;

    for i := 1 to TamanioVector do begin
        vector[i] := tira_dado(CarasDado);
    end;

    for i := 1 to TamanioVector do begin
        write(vector[i], ' ');
    end;
    writeln;
end.
```

Resultado:

```
4 5 2 5 3
```

8.1.1. Clasificación de problemas

Clasificación de problemas

Recapitulemos:

- En los temas 2 y 3, vimos cómo resolver problemas cuya solución era directamente una expresión (una *fórmula*).

- En el tema 4 trabajamos en la resolución de problemas donde era necesario considerar distintos casos por separado.
- En el tema 7 tratamos los bucles, estructura básica para poder trabajar con arrays. Los problemas con arrays a su vez podemos subdividirlos en:
    1. Problemas de acumulación de valores.
    2. Problemas de búsqueda.
    3. Problemas de maximización

Esta taxonomía es completa. Aplicando esta familia de técnicas (y sus combinaciones) podremos resolver cualquier problema de programación que se nos plantee.

8.1.2. Problemas de acumulación

Problemas de acumulación

Calculemos la media aritmética de un vector de reales:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program media_vector;
const
    TamanioVector = 3;
type
    TipoVector = array[1..TamanioVector] of real;

function suma_vector(vector: TipoVector): real;
var
    suma: real;
    i: integer;
begin
    suma := 0.0;
    for i := 1 to TamanioVector do begin
        suma := suma + vector[i];
    end;
    result := suma;
end;

function media(vector: TipoVector): real;
begin
    result := suma_vector(vector) / TamanioVector;
```

```
end;
const
    VectorPrueba: TipoVector = (115.4, 121.9, 111.9);
begin
    writeln(media(VectorPrueba) :0:2);
end.
```

Resultado:

```
116.40
```

8.2. Problemas de búsqueda

Problemas de búsqueda

Vamos a buscar un valor concreto en nuestro array:

- Una función, `busca_valor` recorrerá todo el array comparando cada valor con el deseado.
- Si lo encuentra, devolverá su posición.
- Si no lo encuentra, devolverá un valor especial. Un valor que sabemos que nunca puede ser una posición, por lo que podemos darle el significado *no encontrado*.
    - Como estamos empezando a contar nuestros arrays desde 1, el 0 es una posición *imposible*. Por tanto podemos darle el significado *valor no encontrado*.
    - Muy a menudo se usa -1 para indicar *no encontrado*. También se usan constantes especiales como `nil`.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program busqueda_real_mal;
    // Búsqueda ERRONEA de un real en un array

const
    TamanioVector = 3;
type
    TipoVector = array[1..TamanioVector] of real;
```

```
function busca_valor(vector: TipoVector; valor: real): integer;
    // Esta función busca un valor en un vector y devuelve su
    // posición. Si no existe, devuelve 0
var
    i: integer;
begin
    result := 0;
    for i := 1 to TamanioVector do begin
        if vector[i] = valor then
            result := i;
    end
end;

const
    VectorPrueba: TipoVector = (115.4, 121.9, 111.9);
    ValorBuscado: real = 121.900001;
begin
    writeln(busca_valor(VectorPrueba, ValorBuscado));
end.
```

Resultado:

```
0
```

El programa no ha encontrado un valor exactamente igual al buscado, aunque hay otro apenas una millonésima menor.

Observa que en la función de búsqueda NO hemos hecho

```
if LO_ENCUENTRO then
    result := LA_POSICION
else
    result := 0;
```

Sino que hemos seguido este otro esquema, que es muy normal en problemas de búsqueda:

```
De momento no lo he encontrado
Busco por todo el array
    Si aparece, lo he encontrado
```

En caso de que no se encuentre nunca, la sentencia *de momento no lo he encontrado* se convierte en definitiva, porque ninguna otra altera el valor.

El ejemplo anterior, *busqueda_real_mal* sería correcto para buscar enteros, cadenas, enumerados, etc.

- Pero no podemos buscar así un número real.
- Es muy frecuente que errores en el redondeo de las expresiones matemáticas provoquen errores muy pequeños, del orden de 1^{-12} o inferiores.
- Estos errores son suficientes para que dos números reales se consideren distintos cuando en la práctica son iguales.

Para buscar un número real, siempre tenemos que comprobar que la diferencia entre el número a considerar y el número objetivo sea menor que cierto epsilon (valor muy pequeño, despreciable).

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program busqueda_real;
    // Buscamos un valor real en un vector
const
    TamanioVector = 4;
type
    TipoVector = array[1..TamanioVector] of real;

function busca_valor(vector: TipoVector; valor, epsilon: real): integer;
    // Esta función busca un valor real en un vector, permitiendo
    // cierto margen (epsilon). Devuelve su posición, o 0 si no existe
var
    i: integer;
begin
    result := 0;
    for i := 1 to TamanioVector do begin
        if abs(vector[i] - valor) <= epsilon then
            result := i;
    end
end;

const
    VectorPrueba: TipoVector = (93.0, 86.9, 0, 86.2);
    ValorBuscado: real = 86.873;
```

```
    Epsilon: real = 0.03;
begin
    writeln(busca_valor(VectorPrueba, ValorBuscado, Epsilon))
end.
```

Resultado:

```
2
```

El ejemplo anterior ya puede ser aceptable, pero se puede mejorar:

- En todos los casos recorríamos todos los valores.
- Pero si encontramos un valor que se corresponda con lo solicitado, ya no es necesario seguir buscando.

Este tipo de mejoras solo merecen la pena si estamos seguros de que el ahorro de tiempo es importante (hay muchos datos). En otro caso, el tiempo de programador suele ser más valioso.

- A continuación veremos una versión que comete un error al intentar aplicar esta mejora.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program busqueda_real_mal_02;
    // Buscamos en un vector un valor real lo baste próximo a un
    // valor pedido. Si lo encontramos, dejamos de buscar.
    // Versión errónea, olvidamos el caso de que no exista

const
    TamanioVector = 4;
type
    TipoVector = array[1..TamanioVector] of real;

function busca_valor(vector: TipoVector; valor, epsilon: real): integer;
    // Busca un valor en el vector, permitiendo un margen epsilon.
    // Devuelve su posición, o 0 si no existe
var
    i: integer;
```

```
    sal: boolean;  // Cuando sea cierto, salgo del bucle
begin
    result := 0;
    i := 1;
    sal := FALSE;
    repeat
        if abs(vector[i] - valor) <= epsilon then begin
            result := i;
            sal := TRUE;
        end;
        i := i+1;
    until sal;   // ¡Mal! Si no existe el valor, i se desborda
end;

const
    VectorPrueba: TipoVector = (93.0, 87.4, 0, 86.2);
    ValorBuscado = 95;
    Epsilon = 0.6;
begin
    writeln(busca_valor(VectorPrueba, ValorBuscado, Epsilon))
end.
```

Resultado:

```
Runtime error 201 at $00000000004010D8
  $00000000004010D8 line 23 of busqueda_real_mal_02.pas
  $000000000040118F line 36 of busqueda_real_mal_02.pas
  $000000000040104C
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program busqueda_real_02;
    // Búsqueda de un real, versión mejorada.
    // Si lo encontramos, dejamos de buscar.

const
    TamanioVector = 4;
type
    TipoVector = array[1..TamanioVector] of real;

function busca_valor(vector: TipoVector; valor, epsilon: real): integer;
    // Esta función busca un valor en el array, permitiendo
```

```
    // cierto margen epsilon. Devuelve su posición en el vector. Si no
    // existe, devuelve 0
var
    i: integer;
    sal: boolean;  // Cuando sea cierto, salgo del bucle
begin
    result := 0;
    i := 1;
    sal := FALSE;
    repeat
        if abs(vector[i] - valor) <= epsilon then begin
            result := i;
            sal := TRUE;
        end;
        if i= TamanioVector then
            sal := TRUE
        else
            i := i+1;
    until sal
end;

const
    TiemposPrueba: TipoVector = (93.0, 87.4, 0, 86.2);
    ValorBuscado: real = 95;
    Epsilon: real = 0.6;
begin
    writeln(busca_valor(TiemposPrueba, ValorBuscado, Epsilon))
end.
```

Resultado:

```
0
```

Presentaremos ahora un ejemplo donde queremos comprobar si **todos** los valores cumplen cierta condición. Este tipo de problemas se resuelve así:

- En una variable booleana indicamos si se cumple la condición o no.
- Inicialmente almacenamos en esta variable que sí se cumple.
- Ahora nuestro objetivo es encontrar un caso donde no se cumpla. Basta con uno.

- Recorremos todo el array en un bucle.
    - Si aparece un caso que incumple, ya podemos marcar que la condición no se cumple y forzar la salida del bucle.
    - Si llegamos al final del array, salimos del bucle.

Tras la busqueda, si se mantiene la suposición provisional inicial de que la condicion se cumple, podemos afirmar que definitivamente se cumple. Y si no, no.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program todos_en_margen;
// Queremos saber si todos los valores de un vector cumplen
// la condición de estar en el intervalo definido por un
// valor de referencia +- cierto margen

const
    TamanioVector = 4;
type
    TipoVector = array[1..TamanioVector] of real;

function comprueba_margen(
    vector: TipoVector; referencia, margen: real): boolean;
    // Esta función comprueba si todos los vector están dentro
    // de un valor promedio, más menos cierto margen
var
    i: integer;
    sal: boolean;  // Cuando sea cierto, salgo del bucle
begin
    result := TRUE; // De momento supongo que todos cumplen
    i := 1;
    sal := FALSE;   // De momento indico que no acabe el bucle
    repeat
        if abs(vector[i] - referencia) > margen then begin
            // Valor fuera del intervalo
            result := False;
            sal := TRUE;
        end;
        if i= TamanioVector then
            sal := TRUE
        else
```

```
            i := i+1;
    until sal;
end;

const
    Margen = 5 ;  // No le llamamos epsilon, no es muy pequeño
    VectorPrueba1 : TipoVector = (115.4, 110.9, 111.9, 114.1);
    VectorPrueba2 : TipoVector = (115.4, 124.2, 111.9, 120.7);
var
    valor_referencia : real = 113.0;
begin
    writeln(comprueba_margen(
        VectorPrueba1, valor_referencia, Margen));  // TRUE

    writeln(comprueba_margen(
        VectorPrueba2, valor_referencia, Margen))  // FALSE
end.
```

Comprobemos que todos los tiempos están dentro de cierto margen, expresado porcentualmente:

https://gsyc.urjc.es/ mortuno/fpi/en_margen_relativo.pas

Máximo y Mínimo

Queremos buscar los valores mínimo y máximo de un vector.

- Provisionalmente, tomamos el primer valor como máximo y como mínimo.
- Recorremos todo el array desde la segunda posición, cuando algún valor sea mayor que el máximo provisional o menor que el mínimo provisional, actualizamos el máximo / mínimo provisional.

Otra enfoque posible podría ser:

- Tomar provisionalmente como mínimo, un valor que sabemos que será mejorado por cualquiera. Un valor más alto que cualquier valor posible.
- Tomar provisionalmente como máximo, un valor que sabemos que será mejorado por cualquiera. Un valor menor que cualquier valor posible.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program minimo_maximo_01;
    // Muestra los valores mínimo y máximo de un array
const
    TamanioVector = 4;
    TamanioMinVector = 2;
type
    TipoVector = array[1..TamanioVector] of real;

procedure calcula_min_max(
    vector: TipoVector; var minimo, maximo : real);
var
    i : integer;
begin
    minimo := vector[1];
    maximo := vector[1];
    for i := 2 to TamanioVector do begin
        if vector[i] < minimo then
            minimo := vector[i]
        else if vector[i] > maximo then
            maximo := vector[i];
    end;
end;

procedure escribe_min_max(vector: TipoVector);
var
    maximo, minimo : real;
begin
    if TamanioVector < TamanioMinVector then
        writeln('El vector es demasiado pequeño')
    else begin
        calcula_min_max(vector, minimo, maximo);
        write('Minimo: ', minimo:0:1);
        writeln(' Máximo: ', maximo:0:1);
    end;
end;

const
    vector_prueba: TipoVector = (93.0, 87.4, 91.3 , 86.2);
```

```
begin
    escribe_min_max(vector_prueba);
end.
```

Resultado:

```
Minimo: 86.2 Máximo: 93.0
```

Busquemos ahora no el máximo y el mínimo, sino la posición en el vector del máximo y el mínimo.

```
procedure minimo_maximo(
    vector: TipoVector; var pos_min, pos_max : integer);
var
    i : integer;
begin
    pos_min := 1;
    pos_max := 1;
    for i := 2 to TamanioVector do begin
        if vector[i] < vector[pos_min] then begin
            pos_min := i;
        end
        else if vector[i] > vector[pos_max] then begin
            pos_max := i;
        end;
    end;
end;
```

https://gsyc.urjc.es/ mortuno/fpi/minimo_maximo_02.pas

8.3. Matrices

8.3.1. Introducción a las matrices

Matrices

Podemos usar arrays de dos (o más) índices. Se denominan *matrices*.

- Declaramos los índices separados por comas.

```
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;
```

- Para acceder a la matriz, escribimos los índices dentro de los corchetes, separados por comas.

```
matriz[i,j] := 0;
write(matriz[i,j]);
```

Recuerda que en Pascal normalmente se empieza a contar en 1, mientras que en muchos otros lenguajes, por influencia de C, se empieza por el 0. El criterio de Pascal facilita el procesamiento de vectors y arrays, para N elementos los bucles pueden ir desde 1 hasta N. En otros lenguajes los bucles se recorrerían desde 0 hasta $N-1$, lo que resulta ligeramente más complicado.

El primer índice indica la fila, el segundo, la columna

```
aij    i: fila   j: columna
                                        Recorrer por    Recorrer por
fila 1    a11   a12   a13               filas           columnas
fila 2    a21   a22   a23
fila 3    a31   a32   a33               -------->       |   |   |
fila 4    a41   a42   a43               -------->       |   |   |
                                        -------->       v   v   v
          col1  col2  col3
```

- Manteniendo fija la i y cambiando la j, recorremos la matriz por filas

  Para i fijado a 1: *a11, a12, a13*

  para i fijado a 2: *a21, a22, a23*

  ...

- Manteniendo fija la j y cambiando la i, recorremos la matriz por columnas

  para j fijado a 1: *a11, a21, a31, a41*

  para j fijado a 2: *a12, a22, a32, a42*

  ...

Este primer ejemplo necesita varias mejoras

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program matriz_01;
const
    Filas = 2;
    Columnas = 3;
```

```
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;
var
    matriz : TipoMatriz;
    i,j: integer; // Estos nombres de variables son muy cortos,
                  // pero son tradicionales para los índices

begin   // ¡Mal Diseño!
    matriz[1,1] := 11;
    matriz[1,2] := 12;
    matriz[1,3] := 13;
    matriz[2,1] := 21;
    matriz[2,2] := 22;
    matriz[2,3] := 23;
    for i := 1 to Filas do
        for j := 1 to Columnas do
            write(matriz[i,j], ' ');
end.
```

Resultado:

```
11 12 13 21 22 23
```

Observa que *TipoMatriz* es un tipo global, visible en todo el programa por definirse al principio.

- Las variables nunca deben ser globales. Sin embargo es muy habitual que los tipos sean globales (aunque puedan ser a veces locales).

La siguiente versión corrige los problemas de la anterior:

- El código se divide en subprogramas.
- Añade *writeln* tras la impresión de cada fila, para que se imprima en una nueva línea.

```
program matriz_02;
const
    Filas = 2;
    Columnas = 3;
type
```

```
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

procedure inicia_matriz(var matriz:TipoMatriz);
var
    i,j : integer;
begin
    for i := 1 to Filas do
        for j:= 1 to Columnas do
            matriz[i,j] := i * 10 + j;
end;

procedure escribe_matriz(matriz:TipoMatriz);
var i,j: integer;
begin
    for i := 1 to Filas do begin
        for j:= 1 to Columnas do
            write(matriz[i,j], ' ');
        writeln;
    end;
end;

var
    matriz : TipoMatriz;
begin
    inicia_matriz(matriz);
    escribe_matriz(matriz);
end.
```

Resultado:

```
11 12 13
21 22 23
```

Observaciones

- El parámetro *matriz* del procedimiento *inicia_ matriz* es necesariamente de salida, se pasa por referencia (anteponiendo la palabra reservada *var*).
- El bucle de escritura de las filas ahora tiene dos líneas, por lo que se añade un bloque *begin end*.

8.3.2. Matriz de números aleatorios

El siguiente programa genera una matriz donde cada posición es un valor aleatorio.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program matriz_aleatoria;
uses crt;  // Necesario para delay
const
    Filas = 3;
    Columnas = 4;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
begin
    result := random(caras_dado) + 1;
end;

procedure inicia_matriz(var matriz:TipoMatriz);
var
    i,j : integer;
const
    CarasDado = 6;
begin
    for i := 1 to Filas do
        for j:= 1 to columnas do
            matriz[i,j] := tira_dado(CarasDado);
end;

procedure escribe_matriz(matriz:TipoMatriz);
var i,j: integer;
begin
    for i := 1 to Filas do begin
        for j:= 1 to Columnas do
            write(matriz[i,j], ' ');
        writeln;
    end;
end;

var
    matriz : TipoMatriz;
```

```
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
end.
```

Resultado:

```
2 2 3 5
4 6 4 1
2 6 1 4
```

8.3.3. Suma de los valores de una matriz

El siguiente programa suma todos los valores de una matriz:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program suma_matriz;
uses crt;  // Necesario para delay
const
    Filas = 2;
    Columnas = 3;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
// Iguales a los ejemplos anteriores

function sumatorio_matriz(matriz:TipoMatriz): integer;
var i,j, sumatorio: integer;
begin
    sumatorio := 0;
    for i:= 1 to Filas do
        for j:= 1 to Columnas do
            sumatorio := sumatorio + matriz[i,j];
    result := sumatorio;
end;

var
```

```
    matriz : TipoMatriz;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
    writeln('Suma de elementos: ',sumatorio_matriz(matriz));
end.
```

Resultado:

```
1 4 6
3 2 1
Suma de elementos: 17
```

8.3.4. Suma por filas

Este programa suma los elementos de una matriz, por filas:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program sumatorio_filas;
uses crt;  // Necesario para delay
const
    Filas = 3;
    Columnas = 4;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
// Iguales a los ejemplos anteriores

procedure suma_por_filas(matriz:TipoMatriz);
var i,j, sumatorio: integer;
begin
    for i:= 1 to Filas do begin
        sumatorio := 0;
        for j:= 1 to Columnas do
            sumatorio := sumatorio + matriz[i,j];
        write('Suma fila ', i);
        writeln(': ', sumatorio);
```

```
    end;
end;

var
    matriz : TipoMatriz;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
    suma_por_filas(matriz);
end.
```

Resultado:

```
3 4 6 3
5 4 4 1
3 5 3 5
Suma fila 1: 16
Suma fila 2: 14
Suma fila 3: 16
```

Este programa es muy parecido al anterior, pero

- Inicia el sumatorio una vez por cada fila, no una vez para toda la matriz.
- El subprograma principal es *suma_por_filas*. Es un procedimiento, no una función. Recuerda que las funciones no deben tener efectos laterales (como escribir en pantalla).

8.3.5. Suma por columnas

Sumemos ahora las columnas de una matriz:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program sumatorio_columnas;
uses crt;  // Necesario para delay
const
    Filas = 3;
    Columnas = 4;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
```

```
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
// Iguales a los ejemplos anteriores

procedure suma_por_columnas(matriz:TipoMatriz);
var i,j, sumatorio: integer;
begin
    for j:= 1 to Columnas do begin
        sumatorio := 0;
        for i:= 1 to Filas do
            sumatorio := sumatorio + matriz[i,j];
        write('Suma columna ', j);
        writeln(': ', sumatorio);
    end;
end;

var
    matriz : TipoMatriz;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
    suma_por_columnas(matriz);
end.
```

Resultado:

```
4 1 5 1
4 2 6 4
2 6 4 3
Suma columna 1: 10
Suma columna 2: 9
Suma columna 3: 15
Suma columna 4: 8
```

Observaciones:

- Normalmente procesamos *por filas*, esto es, en el orden *de lectura* (occidental): de izquierda a derecha y de arriba a bajo. Fijamos la fila, procesamos cada columna y bajamos a la fila siguiente.
- Pero ahora necesitamos procesar por columnas. Así que el primer bucle es el de las columnas, y dentro, el bucle de las filas.

- Es recomendable que siempre reserves la variable i para las filas y la j para las columnas. En este ejemplo algún programador podría tener la tentación de intercambiarlas, pero seguramente induciría confusión.
- Es especialmente fácil *despistarse* con estos algoritmos. En cualquier programa es importante probar bien cada funcion, pero en estos casos, mucho más.

8.3.6. Generación de arrays

Generación de arrays

- Para empezar por algo sencillo, en los ejemplos previos hemos escrito en pantalla el resultado de las operaciones deseadas.
  - Con un procedimiento, porque las funciones bien escritas ni escriben en pantalla ni tienen ningún otro efecto lateral.
- Normalmente no haremos eso, sino que generaremos un nuevo vector o una nueva matriz. Que posteriormente escribiremos en pantalla, si procede.
  - Una función nos devolverá el vector o la matriz con el resultado. A menos que sea *demasiado grande*, entonces tendremos que usar un parámetro de salida de un procedimiento.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program sumatorio_filas_02;
uses crt;  // Necesario para delay
const
    Filas = 4;
    Columnas = 3;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;
    TipoVector = array[1..Filas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
// Iguales a los ejemplos anteriores
```

```
procedure escribe_vector(var mi_vector:TipoVector);
var i: integer;
begin
    for i := 1 to Filas do
        write(mi_vector[i], ' ');
    writeln;
end;

function suma_por_filas(matriz:TipoMatriz):TipoVector;
var i,j, sumatorio: integer;
begin
    for i:= 1 to Filas do begin
        sumatorio := 0;
        for j:= 1 to Columnas do
            sumatorio := sumatorio + matriz[i,j];
        result[i] := sumatorio;
    end;
end;

var
    matriz : TipoMatriz;
    vector_suma : TipoVector;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
    vector_suma := suma_por_filas(matriz);
    writeln('Sumatorio por filas:');
    escribe_vector(vector_suma);
end.
```

Resultado:

```
5 5 1
1 2 1
6 1 1
2 5 5
Sumatorio por filas:
11 4 8 12
```

Generar el vector es muy similar a escribir los resultados, pero

- En vez de escribir con un procedimiento, usamos una función que devuelve un `TipoVector`, asignando a `result[i]` el valor obtenido.
    - Si el vector a devolver fuera *grande* habría que usar un procedimiento con un parámetro de salida.
- En este caso necesitamos un vector con tantos elementos como filas. Si hiciéramos algún proceso por columnas, el vector tendría tantos elementos como columnas.

8.3.7. Suma de matrices

Suma de matrices

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program adicion_matrices;
uses crt;  // Necesario para delay
const
    Filas = 4;
    Columnas = 3;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
// Igual a los ejemplos anteriores

function suma_matrices(a,b:TipoMatriz):TipoMatriz;
var
    i, j: integer;
begin
    for i:= 1 to Filas do
        for j:=1 to Columnas do
            result[i,j] := a[i,j] + b[i,j];
end;

var
    matriz_1, matriz_2, matriz_3 : TipoMatriz;
begin
    randomize();
    delay(800);
```

```
    inicia_matriz(matriz_1);
    inicia_matriz(matriz_2);
    writeln('Matriz 1');
    escribe_matriz(matriz_1);
    writeln('Matriz 2');
    escribe_matriz(matriz_2);
    matriz_3 := suma_matrices(matriz_1, matriz_2);
    writeln('Matriz suma:');
    escribe_matriz(matriz_3);
end.
```

Resultado:

```
Matriz 1
2 4 6
3 5 1
5 3 6
4 3 6
Matriz 2
4 4 3
3 6 5
3 6 5
4 2 3
Matriz suma:
6 8 9
6 11 6
8 9 11
8 5 9
```

Observaciones:

- Al escribir la matriz suma, algunos elementos ocupan un espacio y otros dos (según sean menores que 10 o no). Esto queda un poco *feo*.
- Podemos solucionarlo indicando que todos los valores ocupen al menos dos espacios, añadiendo `:2` en el *write*.

  ```
  write(matriz[i,j]:2, ' ');
  ```

De esta forma el resultado sería:

```
 6  8  9
 6 11  6
 8  9 11
 8  5  9
```

Las tres matrices del programa anterior aparecen una debajo de la otra. Sería conveniente mostrarlas una al lado de otra. Para ello:

- Necesitamos un procedimiento que escriba la fila *i-ésima* de una matriz.
- Para cada fila, invocaremos a este procedimiento, para cada una de las tres matrices.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program matrices_horizontales;
uses crt;  // Necesario para delay
const
    Filas = 4;
    Columnas = 3;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
// Igual a los ejemplos anteriores

procedure escribe_matriz(matriz:TipoMatriz);
// Como en los ejemplos anteriores, pero con
//     write(matriz[i,j]:2, ' ');

procedure escribe_fila(matriz:TipoMatriz; i:integer);
var
    j: integer;
begin
    for j := 1 to Columnas do
        write(matriz[i,j]:2);
        write(' ');
    write('  ');
end;

procedure escribe_3_matrices(
    matriz_1, matriz_2, matriz_3: TipoMatriz);
var
```

```
    i : integer;
begin
    for i := 1 to Filas do begin
        escribe_fila(matriz_1, i);
        escribe_fila(matriz_2, i);
        escribe_fila(matriz_3, i);
        writeln;
    end;
end;

var
    matriz_1, matriz_2, matriz_3 : TipoMatriz;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz_1);
    inicia_matriz(matriz_2);
    inicia_matriz(matriz_3);
    writeln('Matriz 1');
    escribe_matriz(matriz_1);
    writeln('Matriz 2');
    escribe_matriz(matriz_2);
    writeln('Matriz 3:');
    escribe_matriz(matriz_3);
    writeln;
    escribe_3_matrices(matriz_1, matriz_2, matriz_3);
end.
```

Resultado:

```
Matriz 1
 6  5  4
 4  1  3
 4  1  5
 2  1  5
Matriz 2
 3  5  1
 6  5  4
 6  6  2
 3  2  1
Matriz 3:
 1  5  6
 3  2  5
 5  5  1
 2  4  3
```

```
6  5  4    3  5  1    1  5  6
4  1  3    6  5  4    3  2  5
4  1  5    6  6  2    5  5  1
2  1  5    3  2  1    2  4  3
```

8.3.8. Máximos de una matriz

Máximos de una matriz

El siguiente programa calcula un vector que contiene el valor máximo de cada fila de una matriz:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program maximos_matriz;
uses crt;  // Necesario para delay
const
    Filas = 4;
    Columnas = 3;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of integer;
    TipoVector = array[1..Filas] of integer;

function tira_dado(caras_dado:integer):integer;
procedure inicia_matriz(var matriz:TipoMatriz);
procedure escribe_matriz(matriz:TipoMatriz);
procedure escribe_vector(var mi_vector:TipoVector);
// Iguales a los ejemplos anteriores

function maximo_fila(matriz:TipoMatriz; i:integer):integer;
   // Devuelve el valor máximo de la fila i de la matriz
var j,  max: integer;
begin
    if Columnas < 2 then begin
        writeln('La matriz tiene que tener al menos 2 columnas');
        halt;
    end;

    max := matriz[i,1];
    for j:= 2 to Columnas do
        if matriz[i,j] > max then
            max := matriz[i,j];
    result := max;
end;
```

```
function maximo_por_filas(matriz:TipoMatriz):TipoVector;
// Devuelve un vector con los máximos de cada fila
var
    i : integer;
begin
    for i := 1 to filas do begin
        result[i] := maximo_fila(matriz,i);
    end;
end;

var
    matriz : TipoMatriz;
    vector_maximos : TipoVector;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz);
    escribe_matriz(matriz);
    vector_maximos := maximo_por_filas(matriz);
    writeln('Maximos de cada fila:');
    escribe_vector(vector_maximos);
end.
```

Resultado:

```
2 1 2
6 6 3
2 5 4
1 2 1
Maximos de cada fila:
2 6 5 2
```

8.3.9. Búsqueda en matrices

Búsqueda en matrices

Todo lo visto sobre búsqueda en vectores es aplicable a búsqueda en matrices.

- Como los vectores tienen un índice, hay un bucle. Las matrices tienen dos índices, por tanto basta añadir un segundo bucle, anidado.

Exactamente igual que en las búsquedas sobre vectores.

- Para buscar cierto valor, suponemos inicialmente que no lo hemos encontrado con un booleano a *False*. Si aparece, lo ponemos a *True*.
- Para comprobar si todos los valores cumplen cierta condición, suponemos inicialmente que sí la cumplen (booleano a *True*). Si alguno, basta con uno, la incumple, ponemos el booleano a *False*.

¿Hay algún múltiplo de K en la matriz?

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program algun_multiplo; // Busca en una matriz un múltiplo de k

[...]

function es_multiplo(numero, k: integer): boolean;
begin
    result := (numero mod k ) = 0
    // Si la división entera de un número entre k tiene
    // como resto 0, es que el número es múltiplo de k
end;

function algun_multiplo(matriz: TipoMatriz;k: integer): boolean;
var
    i,j: integer;
begin
    result := False;  // De momento no hay ninguno
    for i:= 1 to Filas do
        for j:= 1 to Columnas do
            if es_multiplo( matriz[i,j] , k) then
                result := True
end;

var
    matriz : TipoMatriz;
const
    CarasDado  =  24 ;
    K = 7 ;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz, CarasDado);
```

```
    escribe_matriz(matriz);
    if algun_multiplo(matriz, K) then
        writeln('Hay algún múltiplo de ',K)
    else
        writeln('No hay ningún múltiplo de ',K)
end.
```

https://gsyc.urjc.es/ mortuno/fpi/algun_multiplo.pas

¿Son todos los reales mayores que K?

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program todos_mayores;
// Indica si en una matriz de reales todos los números son
//mayores que K
uses crt;  // Necesario para delay
const
    Filas = 3;
    Columnas = 5;
type
    TipoMatriz = array[1..Filas, 1..Columnas] of real;


function genera_real(cota_superior:real):real;
begin
    result := random() * cota_superior;
end;

procedure inicia_matriz(var matriz:TipoMatriz; cota_superior: real);
var
    i,j : integer;
begin
    for i := 1 to Filas do
        for j:= 1 to Columnas do
            matriz[i,j] := genera_real(cota_superior);
end;


procedure escribe_matriz(matriz:TipoMatriz);
var i,j: integer;
begin
```

```
    for i := 1 to Filas do begin
        for j:= 1 to Columnas do
            write(matriz[i,j]:6:2);
        writeln;
    end;
end;

function todos_mayores(matriz: TipoMatriz;k: real): boolean;
var
    i,j: integer;
begin
    result := True;  // De momento todos son mayores
    for i:= 1 to Filas do
        for j:= 1 to Columnas do
            if  matriz[i,j] <= k then
                result := False;
end;

var
    matriz : TipoMatriz;
const
    CotaSuperior  =  10 ;
    K = 1.0;
begin
    randomize();
    delay(800);
    inicia_matriz(matriz, CotaSuperior);
    escribe_matriz(matriz);
    if todos_mayores(matriz, K) then
        writeln('Todos los valores son mayores que ',K:4:2)
    else
        writeln('No todos los valores son mayores que ',K:4:2)
end.
```

https://gsyc.urjc.es/ mortuno/fpi/todos_mayores.pas

8.4. Array de registros

Array de registros

Hasta ahora hemos visto arrays de tipos elementales (enteros, reales, etc) pero es mucho más habitual usar arrays de registros.

En el siguiente ejemplo

- Definiremos un registro de TipoAeropuerto, para almacenar código IATA y nombre de un aeropuerto.
- Definiremos un array de TipoAeropuerto.
- Escribiremos procedimientos para dar valor al array y para imprimirlo.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program aeropuertos_01;
const
    NumAeropuertos = 5;
type
    TipoAeropuerto = record // TipoAeropuerto en singular
        codigo : string;
        nombre : string;
    end;
    TipoAeropuertos = array[1..NumAeropuertos] of TipoAeropuerto;
        // TipoAeropuertos en plural, el array

procedure escribe_aeropuertos(aeropuertos:TipoAeropuertos);
var
    i : integer;
begin
    for i := 1 to NumAeropuertos do begin
        write(aeropuertos[i].codigo, ' ' );
        writeln(aeropuertos[i].nombre);
    end;
end;

procedure inicia_aeropuertos(var aeropuertos:TipoAeropuertos);
begin
    aeropuertos[1].codigo := 'MAD';
    aeropuertos[1].nombre := 'Adolfo Suárez Madrid Barajas Airport';
    aeropuertos[2].codigo := 'LHR';
    aeropuertos[2].nombre := 'Heathrow Airport';
    aeropuertos[3].codigo := 'CDG';
    aeropuertos[3].nombre := 'Charles de Gaulle Airport';
    aeropuertos[4].codigo := 'CGN';
    aeropuertos[4].nombre := 'Cologne Bonn Airport';
    aeropuertos[5].codigo := 'OVD';
```

```
    aeropuertos[5].nombre := 'Asturias Airport';
end;

var
    aeropuertos : TipoAeropuertos;
begin
    inicia_aeropuertos(aeropuertos);
    escribe_aeropuertos(aeropuertos);
end.
```

Resultado:

```
MAD Adolfo Suárez Madrid Barajas Airport
LHR Heathrow Airport
CDG Charles de Gaulle Airport
CGN Cologne Bonn Airport
OVD Asturias Airport
```

Ahora escribiremos un programa para buscar el nombre de aeropuerto correspondiente a un código IATA.

- Inicialmente, el resultado provisional será la cadena *Aeropuerto desconocido*.
- Recorreremos todo el array, si algún código IATA es igual al buscado, el resultado será el aeropuerto correspondiente.
- En el programa *aeropuerto_ 02* buscamos en todo el array, incluso aunque lo hayamos encontrado.
- El programa *aeropuerto_ 03* está mejorado, porque si encuentra el aeropuerto, deja de buscar.

```
programa aeropuertos_02;
[...]
function busca_aeropuerto(aeropuertos:TipoAeropuertos; codigo:string): string;
var
    i:integer;
begin
```

```
    result := 'Aeropuerto desconocido';
    for i := 1 to NumAeropuertos do
        if aeropuertos[i].codigo = codigo then
            result := aeropuertos[i].nombre;
end;

var
    aeropuertos : TipoAeropuertos;
begin
    inicia_aeropuertos(aeropuertos);
    writeln(busca_aeropuerto(aeropuertos, 'LHR'));
    writeln(busca_aeropuerto(aeropuertos,'LPA'));
end.
```

Resultado:

```
Heathrow Airport
Aeropuerto Desconocido
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program aeropuertos_03;
[...]
function busca_aeropuerto(aeropuertos:TipoAeropuertos; codigo:string): string;
var
    i : integer = 1;
    sal : boolean = FALSE;
begin
    result := 'Aeropuerto desconocido';
    repeat
        if aeropuertos[i].codigo = codigo then begin
            result := aeropuertos[i].nombre;
            sal := TRUE;
        end;
        if i = NumAeropuertos then
            sal := TRUE
        else
            i := i + 1;
    until sal;
end;
```

Ejemplo: normalización de valores

En este ejemplo manejamos una lista de registros, que contiene una fecha de vuelo y una duración de vuelo.

- Calculamos el tiempo normalizado de cada vuelo *i-ésimo*, que será tiempo[i] / media(tiempos).
- Añadimos el tiempo normalizado a otro campo del registro.

https://gsyc.urjc.es/ mortuno/fpi/normaliza_01.pas
Resultado:

```
Fecha       Tiempo  Tiempo Normalizado
2018.11.18   92.4   1.002
2018.11.19   96.6   1.048
2018.11.20   88.7   0.962
2018.11.21   91.1   0.988

Tiepo medio: 92.200
```

8.5. Procesamiento de cadenas

Procesamiento de cadenas

En muchos programas tendremos que tratar cadenas. Una función básica es `length()`, que devuelve el número de caracteres.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program longitud_cadena;
var
    s: string;

begin
    s := 'hola';
    writeln(length(s));      // 4
    // La función se llama length(), ni len(), ni lengh(),
    // ni lenght, ni lenth(), ni...
end.
```

Podemos acceder al caracter *i-ésimo* de una cadena tratándola como un array de caracteres.

- De hecho, el tipo básico de cadenas que estamos usando es precisamente eso, un array de char.

Vamos a *deletrear* una palabra, escribiéndola letra a letra, con un guión separando las letras.

- En el programa *deletrea_ 01*, escribimos un guión después de cada letra.
- En *deletrea_ 02*, escribimos un guión después de cada letra, excepto en la última.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program deletrea_01;
var
    s: string;
    i: integer;

begin
    s := 'hola';
    for i:= 1 to length(s) do
        write(s[i], '-');
    writeln;
end.
```

Resultado:

```
h-o-l-a-
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program deletrea_02;
var
    s: string;
    i: integer;

begin
    s := 'hola';
    for i:= 1 to length(s)-1 do
        write(s[i], '-');
    writeln(s[i+1]);
        // En otros lenguajes no podemos usar la variable del
        // bucle fuera del bucle, en Pascal sí.
end.
```

Resultado:

```
h-o-l-a
```

Escritura de una cadena al revés

En muchos programas usaremos la *cadena vacía* `''`.

- Está formado por una comilla, cero caracteres y otra comilla.
- Su longitud es cero.
- Es completamente distinta de una cadena con uno o más espacios (cuya longitud sería 1, 2, 3...).
- Es completamente distinta a una cadena no definida.

En el siguiente ejemplo, invertiremos una cadena:

1. Partimos de la cadena vacía.
2. Vamos concatenado cada carácter, empezando por el final.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program invierte_01;

function invierte(s:string):string;
var
    salida : string;
    i: integer;
begin
    salida := '';
    for i := length(s) downto 1 do
        salida := salida + s[i];
    result := salida;
end;

var
    s: string;
begin
    s := 'hola';
```

```
    writeln(invierte(s));
end.
```

Resultado:

```
aloh
```

Ahora que sabemos invertir una cadena, vamos a buscar palíndromos.

- En el programa *palindromo_ 01* intentamos buscar palíndromos comparando una cadena con su cadena invertida. Pero no funciona, porque falta eliminar los espacios.
- Para eliminar espacios:
  - Podemos usar una función o un procedimiento. Mostraremos aquí ambas cosas.
    - En el programa *quita_ espacios_ 01*, quitamos los espacios usando un procedimiento.
    - En el programa *palindromo_ 02*, quitamos los espacios usando una función.
  - Empezamos por la cadena vacía.
  - Vamos concatenando todos los caracteres de la cadena original, excepto los espacios.
  - Permutamos la cadena resultante con la original.
- En *palindromo_ 02* eliminamos los espacios, y por tanto, detectamos los palíndromos correctamente.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program palindromo_01;
    // Primer intento, erróneo
function invierte(s:string):string;
var
    salida : string;
    i: integer;
```

```
begin
    salida := '';
    for i := length(s) downto 1 do
        salida := salida + s[i];
    result := salida;
end;
function palindromo(s:string):boolean;
begin
    result := (s = invierte(s));
end;

begin
    writeln(palindromo('ana'));   // TRUE, ok
    writeln(palindromo('hola, mundo'));  // FALSE, ok
    writeln(palindromo('acaso hubo buhos aca')); // FALSE
        // En el tercer ejemplo fallan los espacios
end.



{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program quita_espacios_01;
procedure quita_espacios(var s:string);
var
    salida : string;
    i: integer;
begin
    salida := '';
    for i := 1 to length(s) do
        if s[i] <> ' ' then
            salida := salida + s[i];
    s := salida;
end;

var
    s: string;
begin
    s := 'hola mundo';
    quita_espacios(s);
    writeln(s);
end.
```

Resultado:

```
holamundo
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program palindromo_02;
function invierte(s:string):string;
var
    salida : string;
    i: integer;
begin
    salida := '';
    for i := length(s) downto 1 do
        salida := salida + s[i];
    result := salida;
end;

function quita_espacios(s:string):string;
var
    salida : string;
    i: integer;
begin
    salida := '';
    for i := 1 to length(s) do
        if s[i] <> ' ' then
            salida := salida + s[i];
    result := salida;
end;

function palindromo(s:string):boolean;
begin
    s := quita_espacios(s);
    result := (s = invierte(s));
end;

begin
    writeln(palindromo('son mulas o civicos alumnos'));
    writeln(palindromo('hola, mundo'));
```

```
    writeln(palindromo('la ruta nos aporto otro paso natural'));
end.
```

Resultado:

```
TRUE
FALSE
TRUE
```

9. Ficheros

9.1. Introducción a los ficheros

Introducción a los ficheros

Todos los datos que hemos usado hasta ahora eran volátiles. Se perdían al concluir la ejecución del programa. Por el contrario, un *fichero*:

- Es una colección de datos *persistente*. No se pierde cuando acaba el programa ni cuando se apaga el ordenador.
- Tiene un nombre, basta usar ese nombre para leer o escribir el fichero. P.e `holamundo.pas`.
    - El nombre puede incluir un *trayecto* (*path*) , esto es, una lista de directorios y/o subdirectorios. P.e. `~/fpi/practica01/holamundo.pas`. Si el nombre no incluye trayecto, se entiende que el fichero está en el *directorio actual*.

Los ficheros los gestiona el sistema operativo (Linux, Windows, macOS,etc).

Los ficheros pueden contener varios tipos de datos:

- Texto.

  Son los más frecuentes, los únicos que veremos en este curso. Trabajaremos con ficheros formados por una secuencia de cadenas de texto, que usan el carácter *salto de línea* como separador.

- Binarios.

  Contienen números reales, enteros o cualquier otro tipo de dato y/o combinación de tipo de dato.

Hay dos formas de acceso a un fichero:

- Acceso secuencial.

  Leemos o escribimos los datos, uno tras otro.

  El único que veremos este curso.

- Acceso aleatorio.

  Leemos o escribimos cualquier posición del fichero. No significa *al azar*, sino cualquier posición que establezca el programa, de forma determinística.

Para usar un fichero hay que:

- Abrirlo, especificando el *modo de apertura*.
  - Modo lectura. El uso que haremos del fichero será leer sus valores.
  - Modo escritura. Borraremos el contenido anterior del fichero para escribir nuevos valores.
  - Otros modos: lectura-escritura, adición, ...
- Leer o escribir.
- Cerrarlo.

Un mismo fichero tiene dos nombres diferentes, es importante no confundirlos.

- El nombre del fichero que maneja el sistema operativo.

  Es el nombre que se verá en el disco, en el gestor de ficheros del sistema operativo, en cualquier otro programa.

  (Llamémosle *nombre externo*, aunque en programación se le suele llamar *nombre de fichero*, a secas).

- El nombre que tendrá el fichero internamente en nuestro programa.

  (Llamémosle *nombre interno*, aunque en programación se le suele llamar *fichero*, a secas, o *descriptor de fichero*). Será una variable:

  - No de ningún tipo básico (entero, real, cadena...) ni compuesto (registro) ni una colección (array).
  - Sino de un tipo nuevo: tipo fichero. En nuestro caso de tipo *text*.

Con frecuencia, nuestro programa manejará estos dos nombre

- Un *string* conteniendo el nombre del fichero (*nombre externo*).
- Un tipo fichero (*text*) (nombre interno).

En la apertura del fichero indicamos el nombre externo del fichero y qué variable (nombre interno) usamos para ese fichero. Y lo normal es que, tras la apertura, el nombre externo ya no lo volvamos a usar más. Usaremos solo el nombre interno.

9.2. Apertura

Apertura de un fichero

En casi todos los lenguajes de programación, la apertura de un fichero se hace con una sentencia o función o método llamado *open*, a la que se pasa como argumento el nombre del fichero y el modo de apertura.

Pero nuestro dialecto de Pascal (Object Pascal) es un poco peculiar.

Para abrir un fichero:

- Declaramos el fichero como una variable de tipo *text* (más o menos como en cualquier otro lenguaje).
- Usamos el procedimiento *assign* para indicar el nombre externo de fichero que tendrá nuestra variable interna.
- Usamos el procedimiento:
  - *reset*
    para indicar que el fichero se abra en modo lectura.
  - *rewrite*
    para indicar que el fichero se abra en modo escritura.

9.3. Lectura y escritura

Lectura y escritura de un fichero de texto

La lectura y escritura de un fichero de texto es muy similar a la lectura desde el teclado y escritura en pantalla.

- Usamos los procedimientos *write*, *writeln* y *readln*.
- Lo único que cambia es que añadimos como primer parámetro el nombre interno del fichero.

De hecho, tanto el teclado como la pantalla se consideran ficheros a estos efectos.

- En las lecturas con *readln*, si no se indica el fichero, se entiende que se refiere a leer desde el teclado.
- En las escrituras con *write* y *writeln*, si no se indica el fichero, se entiende que se trata de la pantalla.

Respecto a la lectura de ficheros de texto:

- Como su nombre indica, solo contienen texto. Si vamos a interpretar ese texto como números, tendremos que usar el procedimiento *val*.

Respecto a la escritura de ficheros de texto:

- Solo podremos escribir directamente tipos básicos: reales, enteros, caracteres, cadenas y booleanos.
- Si necesitamos llevar a fichero tipos más complejos (registros, vectores, matrices, etc) tendremos que escribir cada tipo básico por separado.
- El propio procedimiento *write* / *writeln* se encarga de convertir estos tipos básicos en texto.

Ejemplo:

```
readln(fichero,linea);
    // Lee en la cadena 'linea' una linea desde el fichero de
    // nombre interno 'fichero'

writeln(fichero, 'hola, mundo');
    // escribe en el fichero de nombre interno 'fichero' la constante
    // cadena 'hola, mundo'

writeln(fichero, linea);
    // escribe en el fichero de nombre interno 'fichero' la cadena
    // contenida en el string 'linea'
```

9.4. Cierre

Cierre de un fichero

Es muy sencillo, basta con invocar al procemiento *close* y pasarle como argumento el nombre (interno) del fichero.

```
close(fichero);
```

- Una buena práctica es que la apertura y el cierre del fichero estén la misma función o procedimiento, en líneas prácticamente contiguas o muy próximas (que se vea en la misma pantalla).
- El procesamiento del fichero irá en una función o procedimiento aparte (a menos que sean unas pocas línea: 8, 10...).

```
assign(fichero, nombre_fichero);
reset(fichero);
procesa_fichero(fichero);
close(fichero);
```

Un programa siempre debe cerrar todos sus ficheros (aunque olvidarlo no suele ser catastrófico, normalmente lo hará el sistema operativo por nosotros cuando el programa concluya).

9.5. Paso de ficheros a subprogramas

Paso de ficheros a subprogramas

En los diseños más sencillos, un subprograma recibe el nombre del fichero:

- Recibirá el nombre externo del fichero como parámetro.
- Declarará el nombre interno como variable local.
- Abrirá, procesará y cerrará el fichero.

En casos no tan básicos, un subprograma recibe el fichero

- Un subprograma abrirá el fichero.
- Pasará el nombre interno a un segundo subprograma. Solo el interno, el externo ya no lo necesitará.
- El segundo subprograma recibirá (por referencia) el nombre interno del fichero y lo procesará.
- El subprograma inicial cerrará el fichero.

En nuestro dialecto de Pascal (Object Pascal), cuando se pasa un fichero a un subprograma, es necesario pasarlo explícitamente por referencia[23].

Esto es necesario para que el subprograma tenga acceso al archivo real y pueda modificarlo (por ejemplo, escribir datos en él o mover el puntero interno).

[23]En otros dialectos también se pasa por referencia, pero es implícito

Subprograma recibe el nombre del fichero:

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program escritura_03;

procedure escribe_fichero(nombre_fichero: string);
   // El procedimiento recibe el 'nombre externo'
   // del fichero. Lo recibe por copia.
var
    fichero: text;
begin
    assign(fichero, nombre_fichero);
    rewrite(fichero);
    writeln(fichero, 'Hola, mundo.');
    close(fichero);
end;

var
    nombre_fichero:string;
begin
    nombre_fichero := 'ejemplo.txt';
    escribe_fichero(nombre_fichero);
end.
```

Subprograma recibe el fichero

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program escritura_04;

procedure escribe_fichero(var fich: Text);
   // El procedimiento recibe el 'nombre interno'
   // del fichero. Ha de hacerse por referencia.
   // El fichero ya está abierto.
begin
    writeln(fich, 'Hola, mundo.');
end;

var
    fichero: text;
```

```
    nombre_fichero: string;
begin
    nombre_fichero := 'ejemplo.txt';
    assign(fichero, nombre_fichero);
    rewrite(fichero);
    escribe_fichero(fichero);
    close(fichero);
end.
```

9.6. Ejemplos

Escritura de una línea de texto

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program escritura_01;
var
    fichero: text;
begin
    assign(fichero, 'texto.txt');   // Ponemos nombre al fichero
    rewrite(fichero);               // Apertura en modo escritura
    writeln(fichero, 'hola,mundo'); // Escribimos en el fichero
    close(fichero);
end.
```

Uso de write y writeln en ficheros

Recuerda: *write* y *writeln* se usan en ficheros de texto igual que en pantalla. Basta añadir como primer argumento el nombre (interno) del fichero

Ejemplo en pantalla

```
write('Área del triángulo: ');
writeln(area:0:3);
```

Ejemplo en fichero:

```
write(fichero, 'Área del triángulo: ');
writeln(fichero, area:0:3);
```

(además de, por supuesto, abrir el fichero antes y cerrarlo después)

El siguiente programa escribe en pantalla:

```
Base:12.430 altura: 5.910
Área del triángulo: 73.461
```

Y exactamente lo mismo en el fichero *area_ triangulo.txt*:

```
Base:12.430 altura: 5.910
Área del triángulo: 73.461
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program area_triangulo_fichero;

function area_triangulo(base,altura:real):real;
begin
    result := base * altura;
end;

procedure escribe_area_pantalla(base, altura, area: real);
begin
    write('Base:');
    write(Base:0:3);
    write(' altura: ');
    writeln( Altura:0:3);
    write('Área del triángulo: ');
    writeln(area:0:3);
end;

procedure escribe_area_fichero(base, altura, area: real; nombre_fichero:string);
var
    fichero : text;
begin
    assign(fichero, nombre_fichero);   // Ponemos nombre al fichero
    rewrite(fichero);                  // Apertura en modo escritura
    write(fichero, 'Base:');
    write(fichero, Base:0:3);
    write(fichero, ' altura: ');
    writeln(fichero, Altura:0:3);
    write(fichero, 'Área del triángulo: ');
    writeln(fichero, area:0:3);
    close(fichero)
end;

const
    Base = 12.43;
    Altura = 5.91;
```

```
    NombreFichero = 'area_triangulo.txt';
var
    area : real;

begin
    area := area_triangulo(base, altura);
    escribe_area_pantalla(base, altura, area);
    escribe_area_fichero(base, altura, area, NombreFichero);
end.
```

https://gsyc.urjc.es/ mortuno/fpi/area_triangulo_fichero.pas

Lectura de una lína de texto

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program lectura_01;

var
    fichero: text;
  // Nombre interno de nuestro fichero. Una variable de tipo
  // text, esto es, de tipo 'fichero de texto'
    linea: string;
begin
    assign(fichero, 'texto.txt');   // Ponemos nombre al fichero
    // 'texto.txt' es un string con el nombre externo del fichero

    // En lo sucesivo siempre usamos el 'nombre interno'

    reset(fichero);                 // Lo abrimos en modo lectura
    readln(fichero, linea);         // Leemos una línea del fichero
    close(fichero);                 // Cerramos el fichero
    writeln(linea);                 // Escribimos en pantalla la línea
end.
```

Escritura de varias líneas en un fichero

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program escritura_02;
var
    fichero : text;
    i : integer;
```

```
    iteraciones : integer = 3;
begin
    assign(fichero, 'texto.txt');   // Ponemos nombre al fichero
    rewrite(fichero);               // Apertura en modo escritura
    for i := 1 to iteraciones do begin
        writeln(fichero, i, ' hola,mundo');
    end;
    close(fichero);
end.
```

Contenido del fichero:

```
1 hola,mundo
2 hola,mundo
3 hola,mundo
```

Lectura de todas las líneas de un fichero

Es muy frecuente que queramos leer un fichero completo, de forma secuencial.

- El tamaño de un fichero no es constante, no podemos usar un bucle `for` como en los arrays.
- Disponemos de una función `eof()` (end of file), que recibe como argumento el nombre (interno) del fichero y devuelve TRUE si hemos llegado hasta el final y por tanto no se puede seguir leyendo.

Lo habitual es usar una sentencia `while`:

```
while not eof(fichero) do
begin
    readln(fichero, linea);
    procesa_linea(linea);
end
```

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program lectura_02;

var
    fichero: text;   // Tipo de datos 'fichero de texto'
    linea: string;
begin
    assign(fichero, 'texto.txt');   // Ponemos nombre al fichero
```

```
    reset(fichero);                  // Lo abrimos en modo lectura

    while not eof(fichero) do
    begin
        readln(fichero, linea);
        writeln(linea);
    end;
    close(fichero);                 // Cerramos el fichero
end.
```

El siguiente ejemplo (`lectura_03`) es un caso típico de procesamiento de un fichero línea a línea.

- Un procedimiento (o tal vez una función) recorre un fichero completo en modo lectura.
- Un procedimiento (o tal vez una función) procesa cada línea.
- En este ejemplo el procesamiento de la línea consiste en escribirla sin más, pero podría ser cualquier otra cosa.

Este esquema nos servirá para resolver la mayoría de los problemas comunes relacionados con el procesamiento de ficheros.

- En particular, todos los de este curso.
- Lo único necesario sera reescribir el subprograma *procesa_linea*.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program lectura_03;
procedure procesa_linea(linea:string);
begin
    writeln(linea);
end;

procedure procesa_fichero(var fichero:text);
var
    linea: string;
begin
    while not eof(fichero) do
    begin
```

```
        readln(fichero, linea);
        procesa_linea(linea);
    end;
end;

const
    Nombre_fichero = 'texto.txt';
var
    fichero : text;
begin
    assign(fichero, Nombre_fichero);    // Ponemos nombre al fichero
    reset(fichero);                     // Lo abrimos en modo lectura

    procesa_fichero(fichero);

    close(fichero);                     // Cerramos el fichero
end.
```

Observaciones:

- Es necesario que exista el fichero en el directorio actual. En este caso, `texto.txt`. Si no, se disparará un error de ejecución (*runtime error*).
- Un programa *real* de mínima calidad no debería generar ningún error de ejecución.
    - O bien comprobaríamos que el fichero realmente existe, antes de intentar abrirlo.
    - O bien capturaríamos el *runtime error* (capturaríamos una excepción).

Escritura, lectura y proceso

Ejemplo:

- Escribimos n números aleatorios en un fichero.
- Leemos todo el fichero y calculamos la media aritmética de sus valores.

https://gsyc.urjc.es/ mortuno/fpi/escritura_lectura_01.pas

9.7. Lectura de campos de ancho fijo

Formato de datos en ficheros de texto

Hay muchas formas de organizar los datos dentro de un fichero de texto. En cada momento de la historia de la informática ha ido cambiando el formato que solía ser *más popular*. (Aunque siempre se han usado los formatos *antiguos*).

- Campos de ancho fijo.

  El único que veremos aquí. Muy sencillo. Aún lo siguen usando algunas aplicaciones bancarias.

- CSV (y variantes).

  comma-separated values.

- XML

  eXtensible Markup Language.

- JSON

- YAML

- ... y muchos otros.

Campos de ancho fijo

Un fichero de texto con campos de ancho fijo tiene un aspecto como por ejemplo este:

```
AAA -17.353-145.510Anaa Airport
AAB -26.693 141.048Arrabury Airport
AAC  31.073  33.836El Arish International Airport
AAD   6.096  46.638Adado Airport
AAE  36.822   7.809Rabah Bitat Airport
AAF  29.728 -85.027Apalachicola Regional Airport
AAG -24.104 -49.789Arapoti Airport
AAH  50.823   6.186Aachen-Merzbrück Airport
AAI -13.025 -46.884Arraias Airport
AAJ   3.899 -55.578Cayana Airstrip
```

- Posición 1 a 3: código IATA.
- Posición 4 a 11: latitud.
- Posición 12 a 19: longitud.
- Posición 20 hasta fin de línea: nombre del aeropuerto.

Para procesar estos ficheros:

- Los leemos línea a línea.
- Descomponemos cada línea en sus campos.

  La función `copy(linea, i, n)`

  acepta como argumentos:

  - Una cadena de texto.
  - Un entero `i`.
  - Un entero `n`.

  y devuelve la subcadena que empieza en la posición `i`, con longitud de `n` caracteres.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program descomposicion_01;
    // Descomponemos una línea de texto en campos de longitud fija.
    // Los números están como texto, los convertimos en números
    // con el procedimiento val

const
    Pc01 = 1;   // principio del campo01
    Pc02 = 3;   // principio del campo02

var
    lc01, lc02: integer;  // Longitud de los campos 01 y 02
    linea,  campo01, campo02: string;
    codigo : integer; // para val
    nombre : string;
    dorsal : integer;

begin
    linea := '01Juan Garcia';
    lc01 := Pc02 - Pc01;
    lc02 := length(linea) - Pc02 + 1;
    campo01 := copy(linea, Pc01, lc01);
```

```
    campo02 := copy(linea, Pc02, lc02);
    val(campo01, dorsal, codigo);
    if codigo <> 0 then begin
        writeln('Valor de entrada incorrecto');
        halt;
    end;
    nombre := campo02;
    writeln(dorsal, ' ', nombre);   // 1 Juan García
end.
```

En los campos de tipo cadena normalmente tendremos que eliminar los espacios a la derecha.

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program descomposicion_02;
function quita_espacios_dcha(cadena: string): string;
var
    i : integer;
begin
    i := length(cadena);
    while cadena[i] = ' ' do
        i := i-1;
    result := copy(cadena,1, i);
end;

const
    Pc01 = 1;   // principio del campo01
    Pc02 = 6;   // principio del campo02
    Pc03 = 26;  // principio del campo03
    Eol = 46; // fin de linea (end of line)

var
    lc01, lc02, lc03: integer; // Longitud campos 01, 02, 03
    linea,  campo01, campo02, campo03: string;
    cp, poblacion, provincia : string;

begin
         //            1         2         3         4
         //   12345678901234567890123456789012345678901234567890123456
    linea := '28943Fuenlabrada         Madrid              ';
```

```
    lc01 := Pc02 - Pc01;
    lc02 := Pc03 - Pc02;
    lc03 := Eol - Pc03;

    campo01 := copy(linea, Pc01, lc01);
    campo02 := copy(linea, Pc02, lc02);
    campo03 := copy(linea, Pc03, lc03);

    cp := quita_espacios_dcha(campo01);
    poblacion := quita_espacios_dcha(campo02);
    provincia := quita_espacios_dcha(campo03);

    write(cp, ' ');
    write(poblacion,', ');
    writeln('(', provincia, ')');
end.
```

9.8. Escritura de campos de ancho fijo

Escritura de campos de ancho fijo

Para escribir campos de ancho fijo, creamos subcadenas de ancho fijo y luego las concatenamos.

- Las subcadenas de los campos numéricos los creamos con el procedimiento `str`.
- Para las subcadenas de los campos de texto:
  - Podemos usar nuestra propia función.
  - Podemos usar la función `padright()` de la librería `strutils`.

https://gsyc.urjc.es/ mortuno/fpi/composicion_01.pas
https://gsyc.urjc.es/ mortuno/fpi/composicion_02.pas
https://gsyc.urjc.es/ mortuno/fpi/composicion_03.pas

10. Memoria dinámica

10.1. Introducción

Formas de almacenamiento de datos en memoria

Hay tres formas de almacenar datos en la memoria del ordenador:

- Variables y constantes globales, también llamadas *estáticas*.

  Ya las conocemos.

- Variables y constantes locales, también llamadas *automáticas*.

  Ya las conocemos.

- Memoria dinámica.

  Dedicaremos este tema a presentar una introducción a la memoria dinámica.

Memoria estática

Como sabemos, se usa de dos formas:

- Variables globales. Muy peligrosas. En este curso, *supenso seguro*.
- Constantes globales. Las usamos en ocasiones: para el tamaño de un array o para magnitudes físicas o matemáticas universales.

A las variables y constantes globales también se las llama *estáticas*, porque se pueden usar en todo el programa, *viven* en memoria durante toda la vida del programa.

Memoria automática

- Variables locales.

  De un procedimiento, de una función o del cuerpo principal de un programa.

- Parámetros de funciones y procedimientos.

  Su ámbito está restringido a un subprograma (función o procedimiento), no se pueden usar fuera de su ámbito.

A estas variables y parámetros también se les llama *variables automáticas*, porque se crean en memoria automáticamente cuando el programa entra en su subprograma, y se liberan automáticamente al abandonar el subprograma.

Tanto en la memoria estática como en la memoria automática, teníamos una limitación muy severa: antes de usar una variable, constante o parámetro teníamos que fijar y limitar el tamaño que ocuparía.

- Un número real, un boolean, un entero, etc almacenan exactamente 1 valor.
- Una cadena (en nuestro dialecto de Pascal, Object Pascal sin directivas adicionales) permite un máximo de 255 caracteres.
- En un array el programador indica el número de posiciones. Ese espacio queda reservado y ocupado, no importa si se usa o no.
    - Si es necesario almacenar más datos de los previstos inicialmente, no se puede.
    - Si no se ocupa todo el array con valores útiles, se desperdicia memoria.

Memoria dinámica

La *memoria dinámica* es una técnica que permite que el programador no indique en el código fuente del programa cuánto espacio ocupará un dato o estructura de datos, si no que en *tiempo de ejecución*, esto es, *sobre la marcha*, el propio programa va ocupando y liberando la memoria que necesita.

Esto puede gestionarlo:

- El programador.
- El lenguaje de programación.
- En muchos lenguajes de programación, el manejo de la memoria dinámica es responsabilidad del programador. Esto tiene cierta dificultad y es propenso a errores, aunque se puede conseguir un código muy eficiente, muy rápido, que consume muy poca memoria.
    - Es típico de lenguajes de los años 1970 y 1980 como C y Pascal, aunque hay muchas excepciones.
- En otros lenguajes de programación, el propio compilador (o intérprete) se ocupa de gestionar la memoria dinámica. Esto los hace más fáciles de usar y con menos errores. Aunque el código suele ser menos eficiente.
    - Es típico de lenguajes de los años 1990 y posteriores como python, java y javascript. Aunque hay muchas excepciones.

10.2. Punteros

Punteros

Usando memoria dinámica, para manejar un valor ya no tratamos con una única entidad, una *variable*, sino con dos: *punteros* apuntando a *datos*.

- El puntero no contiene el dato directamente, sino la dirección de memoria donde está el dato. El puntero *apunta al dato*.
- A partir del puntero se optiene el dato, a esto se llama *aplicar una indirección*, o, coloquialmente, *atravesar el puntero*.

- En los lenguajes como C y Pascal, el programador es responsable de:
  - Reservar memoria para un dato.
  - Asegurarse de que el puntero apunta a una zona de memoria correctamente reservada.
  - Liberar la memoria cuando ya no sea necesaria.
- En lenguajes como python, java y javascript, muchas de estas tareas se hacen automáticamente.
  - Por ejemplo, la memoria la libera *de vez en cuando* el *recolector de basura*. Pero el hecho de que este recolector se ejecute sin control directo del programador puede ser un inconveniente.

10.3. Listas encadenadas

Listas encadenadas

Hay muchas formas de manejar memoria dinámica, la más sencilla es la *lista encadenada*, también llamada *lista enlazada*. Es una estructura similar a una cadena (de eslabones, no de texto). Está formada por nodos.

- Se inicia la lista como una cadena vacía.
- Cada nodo contiene información de un elemento (típicamente un registro) y apunta al nodo siguiente.
- Se añaden uno a uno los nodos necesarios.
- Cuando ya no son necesarios, se liberan. Sería equivalente a quitar eslabones de la cadena metafórica.

Declaración e inicialización

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program punteros_01;

type
    TipoLista = ^TipoNodo;  // TipoLista es un puntero a un nodo

    TipoNodo = record
        valor: integer;    // El valor que nos interesa
        sig: TipoLista;    // Puntero al siguiente nodo
    end;

procedure inicia_lista(var lista:TipoLista);
begin
    lista := nil
end;
```

Adición de nuevo elemento

```
procedure aniade_elemento(var lista:TipoLista; valor:integer);
var
    nodo : TipoLista;
begin
    new(nodo);              // Reservamos memoria para un nuevo nodo
    nodo^.valor := valor;   // Le damos su valor
    nodo^.sig := lista;     // Enlazamos el nodo a la cadena
    lista := nodo           // Ahora la lista empieza por el nuevo nodo
end;
```

Procesamiento de la lista

```
procedure recorre_lista(var lista:TipoLista);
begin
    while (lista <> nil) do begin  // Mientras no estemos en el fin
        writeln(lista^.valor);     // Procesamos el nodo
        lista := lista^.sig        // Pasamos al siguiente elemento
    end
end;
```

Este procesamiento recorre la lista en orden inverso al de creación: el primer elemento en entrar es el último en salir.

Liberación de memoria

```
procedure libera_lista(var lista:TipoLista);
var
    previo : TipoLista;
begin
    while (lista <> nil) do begin
        previo := lista;       // Guardamos el elemento anterior
        lista := lista^.sig;   // La lista apunta al siguiente
        dispose(previo)        // Borramos el anterior
    end
end;
```

Uso de los procedimientos anteriores

```
var
    lista : TipoLista;
begin
    inicia_lista(lista);
    aniade_elemento(lista, 1);
    aniade_elemento(lista, 2);
    aniade_elemento(lista, 3);

    recorre_lista(lista);
    libera_lista(lista);
end.
```

Código fuente completo: https://gsyc.urjc.es/ mortuno/fpi/punteros_01.pas

Recomendaciones

Para un estudiante de introducción a la programación, el uso de memoria dinámica tiene cierta complejidad. Para evitar errores, reutiliza este código con los mínimos cambios imprescindibles.

- Usa las mismas funciones y tipos de datos, puedes usar exactamente los mismos nombres.
- Limítate a cambiar el campo *valor* de *Tiponodo*.
    - Para estructuras sencillas, reemplaza *valor* por los campos que necesites.

    - Para estructura un poco más complejas, sustituye *valor* por un tipo registro, que contenga los campos necesarios.

- En el procedimiento *recorre_lista*, reemplaza el *writeln* por la llamada al subprograma que corresponda.

11. Anexo: Acceso remoto al laboratorio

11.1. Introducción al acceso remoto

Las prácticas de la asignatura

- Tienen que funcionar en el laboratorio linux de la EIF
- Tendrás que entregarlas en el laboratorio

Pero normalmente trabajarás en el ordenador de tu casa, que resulta más cómodo. En la asignatura aprenderás

- A entrar desde casa en tu cuenta del laboratorio linux
- A sincronizar tu ordenador de casa con tu cuenta del laboratorio

Para abrir tu cuenta linux, sigue estas instrucciones https://labs.eif.urjc.es
Recuerda que esta cuenta es distinta a la cuenta de dominio único de la URJC (que usas por ejemplo para los ordenadores Windows). Para trabajar en el laboratorio desde casa necesitas

1. Elegir una máquina (también llamada *host*) que esté activa
2. Abrir una sesión en la máquina usando algún cliente del protocolo ssh (*secure shell*)

11.2. Selección del host

Elegir una máquina activa
En esta página web encontrarás el listado de máquinas del laboratorio
https://labs.eif.urjc.es/index.php/parte-de-guerra/
Sugerencia: guarda esta página en tus marcadores, la usarás mucho

- También puedes encontrarla buscando en google

  parte de guerra eif

- Puedes usar cualquier máquina del campus de Fuenlabrada, ya sea física o virtual

- No importa si hay varias personas en la misma máquina, pero si todos usais la misma, tal vez podría tener problemas de rendimiento. Por tanto, elige una al azar
- El *parte de guerra* no es completamente fiable. Si alguna máquina no te permite entrar, prueba con otra

Ejemplos de nombre de máquina:
`f-l3202-pc05`
`f-l-vm04`

- Para acceder a un *host* desde cualquier lugar de internet (que no sea el propio laboratorio), es necesario indicar su nombre completo (*FQDN, fully qualified domain name*)
- En nuestro caso el FQDN es el nombre de máquina, añadiendo el sufijo `aulas.eif.urjc.es`

  p.e.

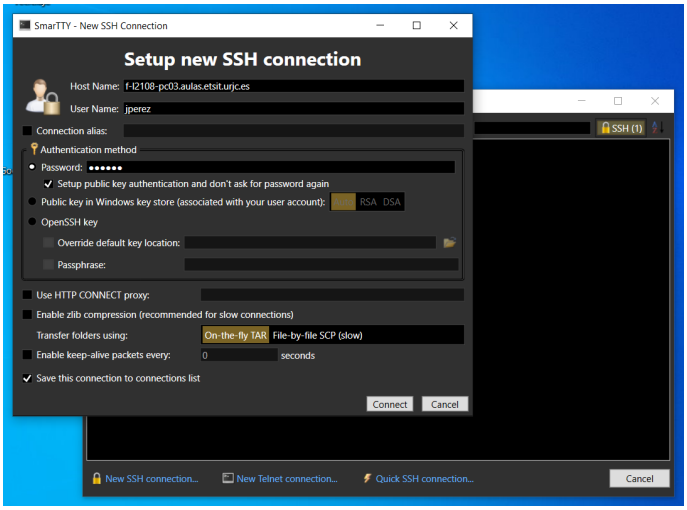
  `f-l3202-pc05.aulas.eif.urjc.es`

  `f-l-vm04.aulas.eif.urjc.es`

La primera vez que abras sesión en una máquina, el cliente ssh mostrará un mensaje parecido a este

```
The authenticity of host 'f-l3202-pc05.aulas.eif.urjc.es'
can't be established.  ECDSA key fingerprint is
SHA256:ucEkxgpIobKhgwOb979NY97fmuaTtWwewdLa//SxVtk.
Are you sure you want to continue connecting (yes/no)?
```

- Esto significa que para estar 100 % seguros de que ningún atacante suplanta la identidad de la máquina deberíamos revisar esta huella digital
- Como no estamos en un entorno especialmente peligroso, podemos contester *yes* sin comprobar nada. Esto guarda la huella digital, y no volverá a mencionarla a menos que
  - Suframos un verdadero ataque
  - El administrador reinstale la máquina sin conservar la huella



SmarTTY: nueva conexión

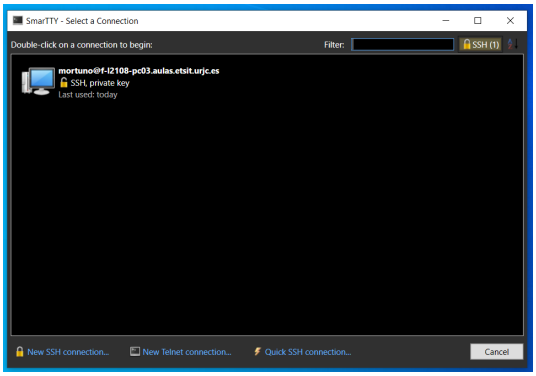
11.3. Sesión desde Windows

Sesión ssh desde Windows

Para Microsoft Windows hay muchos clientes ssh disponibles

- Posiblemente el más usado es Putty, pero es un poco antiguo. No permite el uso de pestañas, y la configuración para evitar teclear contraseñas en cada sesión es un poco complicada
- Aquí recomendamos SmarTTY. Es gratuito, sencillo y potente. Cualquier buscador te indicará que puedes descargarlo desde https://sysprogs.com/SmarTTY

1. Al iniciar SmarTTY, nos pedirá que seleccionemos una conexión. Elegimos *New SSH connection*
2. En el campo *host name* escribimos el nombre completo de la máquina. P.e. *f-l2108-pc03.aulas.eif.urjc.es*
3. En el campo *user name* escribimos nuestro nombre de usuario en el laboratorio (*jperez, mgarcia,...*)
4. En el campo *password* escribimos nuestra contraseña
5. Dejamos el resto de parámetros en su valor por omisión y pulsamos *connect*



6. La primera vez que nos conectemos a una máquina nos aparecerá una ventana titulada *Save host key* donde nos mostrará su huella digital. Como no estamos en un entorno especialmente sensible, la guardamos sin más como nos recomienda.

- Si todo ha ido bien, SmarTTY nos preguntará si preferimos un *terminal inteligente* (*Start with a smart terminal*) o un terminal normal (*Start with a regular terminal*)
- Elegimos el terminal normal y marcamos la opción *remember the choice* para que no pregunte de nuevo. Hecho esto, ya podemos trabajar en la sesión
- Pulsando el icono que representa un signo de más de color verde, podemos añadir una nueva sesión normal en otra pestaña
- Aquí puedes ver una sesión de ejemplo: https://youtu.be/pV2E5Tfr1aI

  Atención, en este vídeo se usan direcciones obsoletas (aulas.etsit.urjc.es), no las actuales: aulas.eif.urjc.es

Una vez que hayamos creado la conexión, si en otro momento queremos entrar en la misma máquina no es necesario repetir todos los pasos, los datos de la conexión se habrán guardado y basta con hacer doble click sobre su icono

11.4. Sesión desde macOS

Sesión ssh desde macOS

En macOS no es necesario instalar ningún programa

1. Ejecutamos *Terminal*
2. En el menú *Shell* elegimos la opción *Nueva conexión remota*
3. En el panel izquierdo (*Servicio*) debe estar seleccionada la opción *Shell segura (ssh)*
4. En el panel derecho (*Servidor*), pulsamos el botón con el signo más para añadir la dirección de la máquina a la que nos queremos conectar. P.e. *f-l2108-pc03.aulas.eif.urjc.es*
5. En el campo *Usuario* escribimos nuestro nombre de usuario en el laboratorio Linux. (*jperez, mgarcia,...*)
6. Pulsamos conectar

- Para abrir varios terminales, basta pulsar de nuevo *Conectar*
- Normalmente, cuando escribes una contraseña, el cliente no la muestra en pantalla, pero aparece un asterisco o similar cada vez que pulsas una letra

  ```
  contraseña: *******
  ```

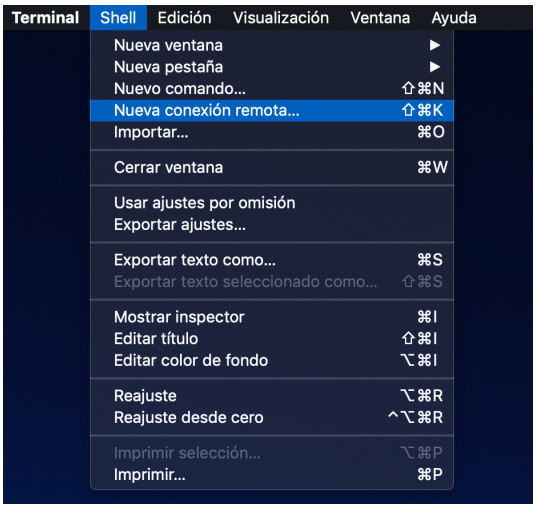
  En este caso no, la escritura de la contraseña es completamente invisible. Que esto no te confunda

- Aquí puedes ver una sesión de ejemplo: https://youtu.be/f8CJINHulgs

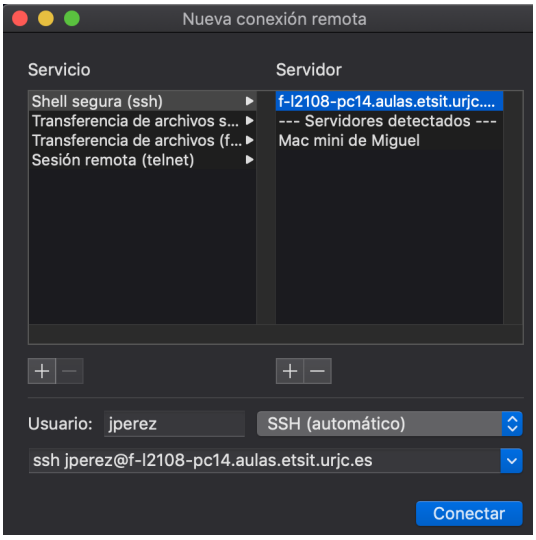
  Atención, en este vídeo se usan direcciones obsoletas (aulas.etsit.urjc.es), no las actuales: aulas.eif.urjc.es

Alternativas

El uso de un cliente ssh es posiblemente la forma más sencilla de hacer las prácticas de FPI desde casa. Con la ventaja de que usarás a diario el mismo entorno que en el examen. Pero si lo prefieres, puedes emplear cualquier otra solución



Terminal de macOS



Parámetros de la conexión

- Acceso gráfico mendiante VNCweb o cualquier otro cliente VNC

  Inconvenientes: algunas teclas no funcionan bien. Si tu acceso a internet no es bueno, no trabajarás cómodo. Si olvidas cerrar la sesión remota, puedes tener problemas en una sesión local

- Instalar el compilador de FreePascal en tu máquina local y sincronizar los ficheros con FreeFileSync

  Inconvenientes en Windows: tendrás que saber manejar la shell de Windows y la de Linux, que son ligeramente distintas

  Inconvenientes en macOS: la instalación del compilador a veces es problemática

- Instalar una imagen de máquina virtual similar a la de los laboratorios y sincronizar los ficheros

  Inconveniente: puede ser ligeramente incómodo

- Usar un compilador de Pascal online

  Inconveniente: puede ser ligeramente incómodo. Tendrás que sincronizar tus ficheros a mano. En el examen usarás un entorno (el del laboratorio) distinto al que usaste durante el curso

- Instalar Linux en una partición de tu ordenador y sincronizar los ficheros

  Inconveniente: es una cierta complicación, innecesaria en algunas asignaturas

- Usar WSL2

  Inconveniente: es una cierta complicación, innecesaria en algunas asignaturas

- ...

12. Anexo: Uso básico de la shell de Unix/Linux

12.1. Introducción a la shell

Uso básico de la shell de Unix/Linux

Hasta ahora has manejado ordenadores usando interfaces gráficos de usuario, con ratón (o similar), ventanas, menús, botones, etc.

Aquí aprenderas a manejar lo más elemental de la *shell* de Unix/Linux.

- Unix es una familia de sistemas operativos, a la que pertenece Linux.
- La *shell* es un programa que nos permite manejar nuestro sistema usando solo teclado y pantalla en modo texto, sin gráficos ni ratón.

Es una forma de trabajar más antigua y un poco más complicada que los interfaces gráficos, pero con con ventajas importantes.

12.2. Conceptos básicos sobre la shell

Terminal

A la combinación de teclado y pantalla sin gráficos se le llama *terminal* o también *consola*. Desde el terminal, manejamos la shell.

Con un terminal se puede trabajar de dos formas:

- En *local*, esto es, usar el ordenador que tenemos delante de nosotros.
- En *remoto*. Usando un cliente del protocolo ssh, como p.e. SmarTTY o Terminal, podemos trabajar en una máquina en la otra punta del mundo, exactamente igual que si la tuviéramos a un metro.

Sesión

En nuestro caso, una *sesión* es un intercambio de información entre el usuario y el ordenador.

- Empieza cuando el usuario (o su cliente, p.e. SmarTTY) introduce sus credenciales (nombre de usuario y contraseña) en el sistema.
- Concluye cuando el usuario lo decide o cuando algún error lo fuerza.
- En una sesión en modo texto, el usuario escribe órdenes en el terminal y el ordenador devuelve los resultados de las órdenes.

  A las órdenes también se les llama *comandos*.

Ficheros y directorios

Fichero y *directorio* son los nombres tradicionales en Unix para lo que normalmente conoces como *documento* y *carpeta*. Podemos considerarlos sinónimos, usa los que prefieras.

- Hay que tener cuidado con la palabra *fichero*, porque en Unix, los directorios son un caso particular de fichero. En otras palabras: cuando decimos *fichero* podemos referirnos a un fichero *ordinario* (un documento) o a un directorio.

Nombres de fichero (y directorio)

Como sabes, cada fichero tiene un nombre y tal vez una extensión. La extensión es el sufijo del nombre, a partir del último punto.

- Por ejemplo en el fichero llamado `holamundo.pas` la extensión *pas* indica que se trata de un fichero en código fuente de Pascal.

Esto es igual que en Windows. Pero en los nombres de ficheros en Unix/Linux hay dos diferencias importantes respecto a Windows:

1. Uso de espacios.
2. Uso de mayúsculas.

1. No es recomendable que un nombre incluya espacios.

   - En Windows es frecuente usar nombres con espacios, como
     `primer ejemplo.docx`
   - En la shell esto serían dos ficheros: por un lado `primer` y por otro `ejemplo.docx`
   - Hay varias soluciones para este problema, aquí recomendamos usar la barra baja (_) en vez del espacio.
     `primer_ejemplo.docx`

2. Mayúsculas y minúsculas son letras distintas. Si un enunciado te pide por ejemplo un fichero llamado `holamundo.pas`, no puedes llamarlo `Holamundo.pas`, es un nombre distinto.

Directorios

- Directorio *home*

  Cuando un usuario tiene cuenta en una máquina, puede escribir en diversos sitios, pero se reserva para él un directorio donde guardar su trabajo. En español se puede llamar *carpeta personal*, *directorio hogar*, etc. Pero posiblemente lo más habitual es llamarlo *home*, en inglés, a secas. Se representa por la virgulilla (`~`).

  Virgulilla en el teclado:

  - Windows y Linux: AltGr 4 (y un espacio)
  - macOS: opt ñ

  Cuidado: si copias y pegas una virgulilla desde un pdf, estarás pegando una virgulilla diferente, más pequeña, que no funciona.

- Directorio actual:

  En una sesión, el usuario *está* en cierto directorio: el directorio actual. Siempre que el usuario escriba una orden sobre un directorio, mientras no indique lo contrario, se supone que se refiere al directorio actual.

- Subdirectorio:

  Directorio que está dentro de otro directorio.

Argumento (de una orden)

Cuando escribimos órdenes de shell, podemos añadirles parámetros adicionales a los que se llama *argumentos*.

- Ejemplo:

  `cd ..`

  Es la orden `cd` con el argumento `..`

- Para indicar cual es el comportamiento de una orden cuando no especificamos argumentos, decimos *por omisión la orden hace ...*

Opción

Una orden puede incluir *opciones*. Las opciones modifican el comportamiento de las órdenes de la shell. Se escriben como un guión seguido de una o más letras.

- Ejemplo:

  `rm -r probando`

  Esto ejecuta la orden de shell `rm`, con la opción `r` . El argumento es *probando*.

Observa que:

- No es lo mismo la opción que el argumento.
- No es lo mismo una letra minúscula que una mayúscula.
- En la opción (u opciones), no puede haber espacios entre el guión y la(s) letra(s) `rm - r  probando  # ¡Esto está mal!`

Prompt

El *prompt* es la línea de texto que vemos en el terminal cuando la shell está preparada para que escribamos una orden. P.e.

`jperez@f-l-vm01:~$`

Es importante que sepamos interpretar el prompt porque aporta mucha información útil. En este ejemplo vemos.

- Nuestro nombre de usuario (jperez)
- El nombre de host (f-l-vm01)
- El directorio actual (virgulilla, es decir, *home*)

Observa que:

- La arroba separa el nombre de usuario del nombe de host.
- Los dos puntos separan el nombre de host del directorio actual.
- El dólar indica el fin del prompt, y que podemos escribir a continuación.

Path

Path significa *trayecto*. Es un texto que de forma compacta especifica dónde está un fichero.

Ejemplo:

- `holamundo.pas`

  Esto es un nombre sin *path*. No especifica dónde está

- `~/fpi/practica01/holamundo.pas`

  Esto es un nombre con *path completo*. Significa que en mi directorio *home*, hay un directorio llamado *fpi*, dentro, un subdirectorio llamado *practica01*, y dentro, un fichero llamado *holamundo.pas*.

Observa que los nombres de directorio están separados por el carácter barra (/), igual que en las direcciones de internet.

En Windows para este propósito se emplea la barra invertida (\)

12.3. Órdenes básicas de la shell de Unix

Órdenes básicas

Las órdenes básicas que necesitarás aquí son:

Con directorios:

- Ver su contenido (`ls`)
- Ver su estructura (`tree`)
- Entrar en un directorio (`cd`)
- Salir de un directorio (`cd ..`)
- Crear un directorio (`mkdir`)
- Borrar un directorio (`rm -r`)

Con ficheros:

- Borrar un fichero (`rm`)
- Si es un fichero de texto, editarlo (`nano`)
- Si es un programa en pascal, compilarlo (`fpc`)

ls

Abreviatura de *list*. Sirve para ver el contenido de un directorio.

- `ls`

  Muestra un listado de los ficheros y subdirectorios del directorio actual.

- `ls -l`

  Listado *largo*. No solo vemos el nombre de los ficheros, también su fecha de creación, tamaño y algunos otros *atributos*.

cd

Abreviatura de *change directory*. Sirve para cambiar el directorio actual, esto es, para *entrar* en un directorio o *salir* de él.

- `cd ejemplo`

  Si en el directorio actual hay un subdirectorio llamado *ejemplo*, entraremos en él.

- `cd`

  La orden *cd* sin indicar ningún argumento, nos lleva al *home*, esto es, equivale a `cd ~`.

- `cd ..`

  Estos dos puntos (en horizontal y sin espacios por medio) representan al directorio padre de cada directorio. Esta orden hace que el directorio actual pase a ser el directorio padre. En otras palabras, salimos del directorio en el que estamos.

mkdir

Abreviatura de *make directory*. Sirve para crear directorios.

- `mkdir fpi`

  Crea un directorio llamado *fpi* en el directorio actual. Si por ejemplo mi directorio actual es el *home*, esta orden creará `~/fpi`.

- `mkdir ~/fpi`

  En este caso indico el nombre del fichero con su path completo. Por tanto, se creará exactamente ahí, sin importar cual sea mi directorio actual.

rm

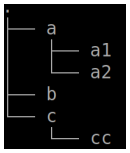
Abreviatura de *remove*. Sirve para borrar uno o más ficheros.

- `rm ejemplo`

  Borra un fichero llamado *ejemplo* del directorio actual. Si es un directorio no lo borrará.

- `rm -r ejemplo`

  La opción `-r` significa *recursive*. Con esta opción la orden `rm` borra ficheros y también directorios, recursivamente. Esto es, recorriendo y borrando sucesivamente todos los subdirectorios que haya.



tree

Sirve para ver la estructura en forma de árbol de todos los ficheros, directorios y subdirectorios contenidos dentro de mi directorio actual.

Si `tree` nos devuelve por ejemplo esta salida:

Significa que

- En el directorio actual tenemos tres elementos: a, b y c. Observarmos que a y c son subdirectorios, pero en el caso de b no podemos distinguir si se trata de un subdirectorio vacío o un fichero.
- Dentro de a, están los ficheros o directorios a1, y a2.
- Dentro de c, el fichero o directorio cc.

Visualización de salidas largas

En ocasiones, un comando puede generar una salida con muchas líneas, de forma que no quepan en tu terminal. Para ver la salida completa, tienes varias soluciones:

1. Usar la barra de desplazamiento vertical de tu terminal (si está disponible).
2. Añadir al comando la barra vertical y el comando `less`. P.e.

```
ls | less
tree | less
```

   - De esta forma, podrás usar las flechas arriba y abajo del teclado para desplazarte por la salida.
   - Para escribir la barra vertical, pulsa simultáneamente las teclas *alt gr* y *1*.

exit

`exit`

Finaliza la shell actual, por tanto cierra la sesión.

- Si no teníamos ningún programa funcionando y cerramos la ventana del terminal, el efecto es el mismo. Pero es una buena costumbre cerrarlo todo ordenadamente.
- Si había un programa funcionando, por ejemplo un editor de texto abierto, y cerramos la ventana *por las malas*, sin usar exit, podremos tener problemas en la siguiente sesión.

En este vídeo puedes ver una sesión básica https://youtu.be/70BUma0M4ao

13. Anexo: El editor de texto Nano

13.1. Introducción al editor de texto Nano

El editor de texto Nano

Nano es un editor de texto para usar usar desde el terminal, libre y gratuito. Lo usarás para editar ficheros en tu cuenta del laboratorio

- Está disponible para Linux, macOS y Windows. Pero para trabajar en tu ordenador seguramente no merece la pena instalarlo. De momento puedes usar el editor de texto preestablecido en tu equipo
- Es sencillo y suficiente para tareas básicas. Algunas cosas las denomina de forma un poco distinta a los programas modernos, pero es fácil de usar

Uso de Nano

Para empezar a editar un fichero, teclea en el terminal

```
nano nombre_del_fichero.txt
```

o

```
nano nombre_del_fichero.pas
```

- Si el fichero existía, lo abre. Si no, crea uno nuevo
- En las dos últimas líneas del terminal, Nano nos irá mostrando los atajos de teclado que necesitaremos normalmente
- El acento circunflejo (`^`) significa `Ctrl`
- `M-` significa *tecla meta*
  - En Windows y Linux, pulsar `Alt`, mantener pulsado y pulsar la letra que corresponda
  - En macOS, pulsar `Esc`, soltar y pulsa la letra que corresponda
- Los atajos aparecen escritos en mayúsculas, pero es indiferente usar mayúsculas o minúsculas

13.2. Funcionalidad principal

Funcionalidad principal

- `^O` Write out

  Guarda el fichero. Equivalente al `Save` o `Guardar` de otros programas

Seleccionar texto

1. Llevamos el cursor al principio del texto que queremos seleccionar
2. Pulsamos *shift* y lo mantenemos pulsado
3. Llevamos el cursor al final del texto a seleccionar. La posición actual del cursor quedará excluida de la selección

- `^K` Cut text

  Cortar texto. Por omisión, una línea. Si hemos seleccionado algo, copia la selección

- `^U` Paste text

  Pega el texto cortado o copiado

- Copiar

  Para no perder el texto original, esto es, copiar, podemos hacer dos cosas:

  1. Cortar, pegar lo que acabamos de cortar, ir donde queramos pegar, pegar
  2. `M-6` Esto es,
     `Alt 6` (Linux, Windows)
     `Esc 6` (macOS)

- `M-U` (`Alt u`) Undo

  Deshace el último cambio

- `M-E` (`Alt e`) Redo

  Rehace lo último que hayamos deshecho

- `^W` Where is

  Busca una cadena de texto. Equivalente al `Search` o `Buscar` de otros programas

- `^X` Exit

  Sale del programa. Si hay alguna modificación que no hemos guardado, nos preguntará si queremos guardar las modificaciones (yes/no) o si ya no queremos salir (cancel)

En este vídeo puedes ver una demostración https://youtu.be/U3WBWZHvf7Q

Observa que usamos dos formas distintas de copiar y pegar (o de cortar y pegar)

- Normalmente trabajaremos dentro del mismo fichero y podremos usar las funciones del propio Nano

```
M-6  ^U        (Copiar y pegar. Alt 6 en Windows y Linux, Esc 6 en macOS)
^K   ^U        (Cortar y pegar)
```

- Cuando queramos copiar y pegar o cortar y pegar entre dos ficheros, será necesario hacer algo diferente. Aquí recomendamos copiar y pegar usando el entorno gráfico de nuestro ordenador local (botón secundario del ratón)[24]

[24] El inconveniente es que esto será ligeramente distinto en Windows, Linux o macOS

Copiar y pegar con el ratón no siempre es posible. Supongamos que quieres copiar y pegar las directivas, sin ratón:

1. Descarga este holamundo en la carpeta donde necesites el fichero. Para ello, haz clic con el botón derecho en el siguiente enlace, en la opción *guardar enlace como* http://ortuno.es/holamundo.pas

2. Abre el fichero con nano. Quédate con las directivas, modifica lo que necesites.

3. Guarda el fichero (`^O`), y cuando nano te muestre el nombre del fichero a escribir, cámbialo por el nombre adecuado.

Ver línea y columna en Nano

El compilador te dirá en qué línea y columna están los errores, así que te será útil que Nano te muestra la línea y la columna donde está el cursor

- Edita con el propio Nano un fichero llamado *.nanorc* en tu *home*

  ```
  nano ~/.nanorc
  ```

- Dentro escribre el siguiente texto

  ```
  set constantshow
  ```

Observa que el nombre de este fichero empieza por punto. Esto hace que sea un fichero *oculto*. Podrás editarlo como cualquier otro, pero

- Al hacer un listado normal con `ls`, este fichero no se verá

- Cuando quieras ver todos los ficheros del directorio actual, incluyendo los ocultos, deberás añadir la opción `-a`

  ```
  ls -a
  ```

13.3. Color basado en la sintaxis

Color basado en la sintaxis

Para que Nano use distintos colores para los distintos elementos de un programa Pascal (esto es, de los ficheros .pas)

1. Descarga el fichero http://ortuno.es/pascal.nanorc en el directorio `~/fpi/`

   Esto puedes hacerlo de dos formas

- Editar un fichero `~/fpi/pascal.nanorc`, copiar el contenido del fichero desde el navegador, pegar en Nano
- ```
  cd ~/fpi
  wget http://ortuno.es/pascal.nanorc
  ```

2. Añade en el fichero de configuración de Nano (`~/.nanorc`) la siguiente línea

   ```
   include ~/fpi/pascal.nanorc
   ```

Como siempre, estas instrucciones debes seguirlas al pie de la letra. Una mayúscula o un espacio mal puestos provocarán que nada funcione

13.4. Indentado en Nano

Indentado en Nano

- Activar y desactivar la visualización de caracteres invisibles `M-P`

  Esto es:

  - Windows:

    alt p
  - macOS:

    esc p

- Nano representa los tabuladores con un ancho por omisión de 8 caracteres. Si preferimos, otro valor (normalmente 4), añadimos la siguiente línea en `~/.nanorc`

  ```
  set tabsize 4
  ```

- Para que Nano reemplace todos los tabuladores por espacios, añadimos la siguiente línea en `~/.nanorc`

  ```
  set tabstospaces
  ```
  ```

Correspondencia de paréntesis

Es muy normal tener expresiones con muchos paréntesis, lo que resulta propenso a errores. Los editores suelen tener una opción que, señalando un paréntesis abierto, nos indica el paréntesis cerrado correspondiente. Y viceversa. En Nano:

- Llevamos el cursor a un paréntesis
- Pulsamos `M-]`

  Esto es,

  - Windows:

    alt altgr +
  - macOS:

    esc altgr +

Para que la correspondencia de paréntesis funcione, es necesario editar

- O bien editar el fichero `~/.nanorc`
- O bien el fichero `/etc/nanorc` (si somos los administradores de la máquina)

Y añadir / descomentar las líneas

```
set brackets ""')>]}"
set matchbrackets "(<[{)>]}"
```

(En el laboratorio esto ya está hecho, si instalas Nano en casa, deberás hacerlo tú)

Personalización automática

- El siguiente script realiza automáticamente todas las personalizaciones que hemos descrito

  http://ortuno.es/custom_nano.py
- En el laboratorio puedes ejecutar en un terminal

  `~mortuno/custom_nano.py` [25]

[25] Recuerda que la virgulilla normalmente no se puede copiar y pegar desde un pdf

14. Anexo: Compilación de un programa

Section

Compilación de un programa en Pascal

- En las primeras clases de la asignatura, usarás el cliente de ssh de forma que la edición y la compilación la harás en una máquina linux del laboratorio.

  No importa que tu pc de casa sea Windows, Linux o Mac. A todos los efectos estás trabajando en Linux.

- En clases posteriores, podrás instalar en tu ordenador de casa el compilador que te corresponda, para editar, compilar y ejecutar en tu propia máquina.

  - Esto no es imprescindible, requiere un pequeño esfuerzo adicional pero puede resultar más cómodo.
  - Tendrás que sincronizar con cuidado tus ficheros de casa con tus ficheros del laboratorio. Para la asignatura, solo importa la versión que tengas en el laboratorio.

Compilar desde el terminal

Una vez situados en el directorio donde esté el código fuente del programa en Pascal que queramos compilar, escribimos

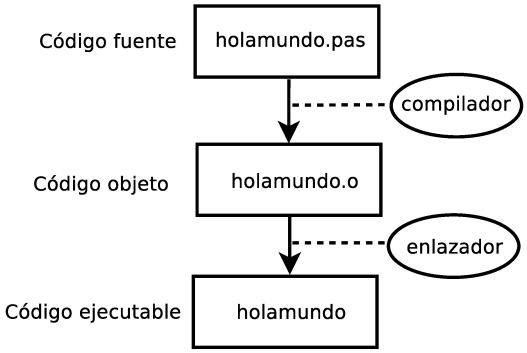
```
fpc -gl nombre_fichero.pas
```

- En este curso, los ficheros con el código fuente tendrán la extensión `.pas`, que es la más habitual para pascal.
- Las opciones `-gl` no son imprescindibles, pero sí son convenientes para que el compilador muestre los mensajes de error con más claridad.
- El compilador generará un fichero `.o` con el código objeto. Podemos ignorarlo y/o borrarlo.
- En Linux y macOS, el fichero ejecutable tendrá el mismo nombre que el fichero con el código fuente, pero sin extensión.

  ```
  nombre_fichero
  ```

- En Microsoft Windows, los ficheros ejecutables tienen extensión `.exe`.

  ```
  nombre_fichero.exe
  ```



1. El programador escribe el código fuente, con cualquier editor.
2. A partir del código fuente, el compilador genera el código objeto.
3. A partir el código objeto, el enlazador genera el código ejecutable.

Ejecución de un programa

- Linux, macOS.

  Después de compilar, si no ha habido errores basta teclear

  `./nombre_fichero`

- Windows.

  Teclea el nombre del fichero sin más

  `nombre_fichero`

En este vídeo puedes ver una demostración: https://youtu.be/NX1I9Su3TtM

Errores frecuentes (I)

Para un principiante, los errores más frecuentes son:

- Intentar compilar sin haberse situado en el directorio donde está el código fuente. En otras palabras, omitir las órdenes `cd` correspondientes.

  Solución: fíjate siembre bien en el *prompt*, la secuencia entre el *dos puntos* y el dólar, indica cuál es tu directorio actual. P.e.

  ```
  jperez@f-l3210-pc23:~/fpi/practica01$
  ```

  Significa que el directorio actual es

  ```
  ~/fpi/practica01
  ```

Errores frecuentes (II)

- Intentar ejecutar el código fuente (el fichero con extensión `.pas`), en vez de el ejecutable (el fichero sin extensión).
  - Error:
    `./holamundo.pas`
  - Correcto:
    `./holamundo`

Errores frecuentes (III)

- Equivocarse en una letra, una mayúscula, confundir una barra baja con un espacio, poner un espacio de más, etc.
  - En este curso siempre escribiremos los nombres de ficheros en minúsculas y sin espacios.
- Tener el mismo código fuente en directorios distintos, editar en uno de ellos pero luego compilar en otro.

Instalación del compilador en tu ordenador

- Windows:

  Bájate y ejecuta el instalador desde la página de free pascal.

  Elige la opción AMD64/intel 64 https://www.freepascal.org/download.var
- macOS
  1. Instala *Xcode* desde la app store.
  2. Instala *Xcode Command Line Tools*, ejecutando en un terminal

     ```
     xcode-select --install
     ```

  3. Bájate y ejecuta el instalador desde la página de free pascala.
     Elige la opción AMD64/intel 64 https://www.freepascal.org/download.var
- Ubuntu Linux.

  En la laboratorio ya está instalado. En casa ejecuta en un terminal:

  ```
  sudo apt update
  sudo apt upgrade
  sudo apt install fpc
  ```

15. Anexo: Depuración

15.1. Formato de cadenas

Formato de cadenas

Ya sabemos usar *write* y *writeln* para componer un mensaje en pantalla. Supongamos que necesitamos algo como esto

```
x: 3.47 y: 12.15 z: 0.29
```

Podemos generarlo así:

```
write('x:',x:0:2);
write(' y:',y:0:2);
writeln(' z:',z:0:2);
```

Esto funciona, pero tiene tres problemas

1. Es farragoso y propenso a errores
2. No permite un ajuste fino de la presentación de los números
3. No es aplicable en algunos casos. Por ejemplo en la librería de depuración que usaremos aquí

Casi todos los lenguajes de programación nos ofrecen una solución más conveniente: dar formato a una cadena

Esta forma de componer cadenas se hace popular por las funciones *printf* y *sprintf* del lenguaje C. Muchos otros lenguajes lo han incluido, con las mismas opciones y sintaxis

- Usan un *microlenguaje* muy rico, especialmente para indicar cómo queremos que se muestren los números
- Aquí veremos solo las opciones más elementales, pero hay muchas otras: alineamiento a derecha, izquierda, centrado, número de decimales, ceros a la izquierda, diversos tipos de notación científica, base decimal, hexadecimal, etc

Función format

En Pascal disponemos de la función *format*.

- Para usarla es necesario añadir al comienzo del programa

  ```
  uses SysUtils;
  ```

Recibe dos argumentos

- Una *format string* que especifica el texto invariable y el formato del contenido variable

  Ejemplo:

  ```
  'Nombre: %s   Nota: %f   Convocatoria:%d'
  ```

  Un signo de porcentaje seguido de una letra es un *especificador de formato*

- Un array de argumentos, que indica cual será el contenido variable. Casi siempre serán variables

  Ejemplo:

  ```
  [nombre,nota,convocatoria]
  ```

Devuelve una cadena, donde cada especificador de formato habrá sido reemplazado por un argumento del array

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program ejemplo_format_01;

uses  SysUtils;

var
    nombre : string;
    nota : real;
    convocatoria : integer;
    mensaje: string;

begin
    nombre := 'Juan García';
```

```
    nota := 6.25;
    convocatoria := 1;

    mensaje := format('Nombre: %s   Nota: %f   Convocatoria:%d',
        [nombre,nota,convocatoria]);

    writeln(mensaje);
end.
```

Resultado:

```
Nombre: Juan García   Nota: 6.25   Convocatoria:1
```

Observa que el primer especificador de formato (`%s`) se reemplaza por el primer argumento del array, el segundo especificador (`%f`) por el segundo argumento y así sucesivamente

- `%s` indica una cadena
- `%f` indica un número real (en lenguaje C el tipo es *float*)
- `%d` indica un número entero, decimal

Hay muchos otros especificadores, estos son los principales[26]

Para escribir el simbolo de porcentaje como literal y no como especificador, se usa `%%`

```
format( 'Aprobados: %f %%',[aprobados]);
```

Resultado:

```
Aprobados: 68.31 %
```

Obviamente, tendremos que prestar atención para que

- El número de especificadores coincida con el número de argumentos del array
- Los tipos coincidan. Por ejemplo no tiene sentido indicar `%s` para un valor numérico

[26] puedes encontrar el resto buscando en el web 'printf format'

Si nos equivocamos, no obtendremos un error de compilación sino uno de ejecución

```
An unhandled exception occurred at $00000000004348B2:
EConvertError: Invalid argument index in format
"Nombre: %s   Nota: %s   Convocatoria:%d"
  $00000000004348B2
  $00000000004359EA
  $0000000000435F26
  $0000000000401182 line 17 of ejemplo_format_01.pas
```

15.2. Depuración de un programa

Depuración de un programa

Depurar es el proceso de localizar y corregir los fallos de un programa

- Es una habilidad especialmente importante: es normal que un programador emplee la mayor parte de su tiempo en la depuración, no en el análisis ni en la escritura de código
- Los programas raramente funcionan a la primera. El programador tiene que detectar los problemas y corregirlos. Lo fundamental es *ver* qué está pasando
- La depuración empieza en el mismo momento de la programación. Cuando empezamos a escribir cada función deberíamos ir pensando cómo depurarla, sin esperar a que se manifiesten los problemas

Como estudiante siempre podrás pedir ayuda al profesor. Pero antes, debes intentar depurar el programa por tí mismo

- Un buen profesor con frecuencia no arreglará tu programa, sino que te dará las pautas para depurarlo
- Te estás formando. Que tu programa funcione es un objetivo subordinado al objetivo principal: que aprendas a programar y por tanto a depurar
- Es muy frecuente que un profesor de programación tenga que decirle a un estudiante algo como: *Tu programa no tiene buena solución. El código es complicado, no está bien dividido en subprogramas, tienes muchos niveles de if/while/for... Tú mismo no lo entiendes, es seguro que no hay un fallo sino muchos, porque te has puesto a escribir líneas de forma desorganizada y sin comprobar cada paso. Borra todo y vuelve a empezar. Pero ahora haz esto, esto y ...*

Supongamos que necesitamos una función que calcule una potencia (sin usar **) y escribimos algo así

```
function potencia(base: real; exponente: integer): real;
var
    i : integer;
    mensaje : string;
begin
    result := 1;
    for i := 0 to abs(exponente) do begin
        result := result * base;
    end;

    if exponente < 0 then begin
        result := 1 / result;
    end;
end;
```

Los probamos por ejemplo con

```
base := 2;
exponente := 3;
```

Y obtenemos el resultado erróneo 16

- Tal vez nos demos cuenta de que el problema es que el bucle *for* debería empezar en 1 y no en 0, pero si no es el caso, iremos *generando trazas*
- Esto consiste en ver, paso a paso, los valores que se van generando, para analizar el comportamiento del programa

- Una solución sencilla y frecuente, aunque *chapucera* es ir escribiendo en pantalla los valores de la variable, usando *write* o la sentencia equivalente en nuestro lenguaje

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program hola_log;

uses slog, SysUtils;

function potencia(base: real; exponente: integer): real;
var
    i : integer;
    mensaje : string;
begin
    result := 1;
    for i := 0 to abs(exponente) do begin
        result := result * base;
        mensaje := format('i:%d  result:%f', [i,result]);
        writeln(mensaje);
    end;

    if exponente < 0 then begin
        result := 1 / result;
        writeln('Exponente negativo, multiplico por -1');
    end;
end;

var
    base : real;
    exponente : integer;
begin
    base := 2;
    exponente := 3;
    writeln(potencia(base, exponente):0:3);

    exponente := -3;
    writeln(potencia(base, exponente):0:3);
end.
```

Resultado:

```
i:0  result:2.00
i:1  result:4.00
i:2  result:8.00
i:3  result:16.00
16.000
```

```
i:0  result:2.00
i:1  result:4.00
i:2  result:8.00
i:3  result:16.00
Exponente negativo, multiplico por -1
0.063
```

- Esto tiene el problema de que *ensucia* la pantalla, mezclando información de depuración con la verdadera salida del programa. Rompe completamente el principio de la transparencia referencial: las funciones no deberían escribir nada en pantalla, pero escribir una traza es escribir
- La solución igualmente *chapucera* es: *cuando acabe de probarlo, quito los mensajes de traza*
  - Lo cual es pesado, lleva tiempo. Además puede ser complicado localizar dónde se generan la trazas
  - Es muy habitual que, en el futuro, se vuelvan a detectar problemas en el programa. Entonces habría que reescribir las trazas, un esfuerzo repetido, una pérdida de tiempo (al borrar y al reescribir)

La forma adecuada de generar trazas es mediante una librería de *logs* [27]

- Una librería es un fichero que contiene funciones, que son llamadas desde el código escrito en otro fichero
- En cualquier lenguaje tendremos librerías de log. O podremos programar la nuestra, es sencillo
- Para esta asignatura hemos preparado la librería *slog* (*simple log*. Puedes descargarla en https://gsyc.urjc.es/ mortuno/slog.pas

Uso de slog

Para usar slog en un programa

- Añade al principio del programa al menos la librería *slog*

  ```
  uses slog;
  ```

  Probablemente querrás usar format, así que normalmente escribirás

[27] La palabra española es bitácora, pero raramente se usa en este contexto

```
uses slog, SysUtils;
```

- Incluye el fichero slog.pas en el mismo directorio que el programa

slog escribirá las trazas en el fichero de log, que por omisión se llamará *slog.log* y estará en el mismo directorio

Para escribir en el fichero de log

- Llama a la función `slog.debug()`, pasando como argumento la cadena a escribir

Para que las trazas dejen de generarse

- Invoca al procedimiento *setquietmode* con el parámetro *True*

```
slog.setquietmode(True);
```

Ejemplo

```
program hola_log;

uses slog, SysUtils;

var
    cadena : string;
    mi_entero : integer;
    mi_real : real;

begin
    cadena := 'Hola,mundo';
    mi_entero := 3;
    mi_real := 2.5;

    slog.debug('Hola mundo, esto es un mensaje de depuración');

    slog.setquietmode(True);
    slog.debug('Esto no aparecerá en el log');

    slog.setquietmode(False);
    slog.debug('Esto vuelve a salir en el log');

    slog.debug(format('la cadena vale %s, el entero %d, y el real %f',
```

```
        [cadena, mi_entero, mi_real]));

    mi_entero := mi_entero + 1;
    mi_real := mi_real * 2;
    slog.debug(format('ahora el entero vale %d y el real %f',
        [mi_entero, mi_real]));
end.
```

Al ejecutar el programa anterior

- En pantalla no aparecerá nada
- Contenido del fichero *slog.log*:

```
2020-12-18 20:33:28 DEBUG Hola mundo, esto es un mensaje de depuración
2020-12-18 20:33:28 DEBUG Esto vuelve a salir en el log
2020-12-18 20:33:28 DEBUG la cadena vale Hola,mundo, el entero 3, y el real 2.50
2020-12-18 20:33:28 DEBUG ahora el entero vale 4 y el real 5.00
```

Cada vez que usemos slog, los mensajes nuevos se irán añadiendo al fichero *slog.log*, sin borrar los mensajes anteriores. Cuando ya no necesites la traza

- Escribe al principio de tu programa

  ```
  slog.setquietmode(True);
  ```

- No es necesario que borres las llamadas a *slog.debug* (excepto tal vez las líneas muy obvias que probablemente no vuelvas a necesitar)

Otro ejemplo

```
{$mode objfpc}{$H-}{$R+}{$T+}{$Q+}{$V+}{$D+}{$X-}{$warnings on}
program potencia_03;
uses slog, SysUtils;
function potencia(base: real; exponente: integer): real;
var
    i : integer;
    traza : string;
begin
    result := 1;
    for i := 0 to abs(exponente) do begin
        result := result * base;
        traza := format('i:%d  result:%f', [i,result]);
```

```
        slog.debug(traza);
    end;

    if exponente < 0 then begin
        result := 1 / result;
        traza := format('Exp. negativo, multiplico por -1. result:%f',
            [result]);
        slog.debug(traza);
    end;
end;

var
    base : real;
    exponente : integer;

begin
    base := 2;
    exponente := 3;
    writeln(potencia(base, exponente):0:3);

    exponente := -3;
    writeln(potencia(base, exponente):0:3);
end.
```

En el fichero *slog.log* veremos la siguiente traza

```
2020-12-18 20:55:03 DEBUG i:0  result:2.00
2020-12-18 20:55:03 DEBUG i:1  result:4.00
2020-12-18 20:55:03 DEBUG i:2  result:8.00
2020-12-18 20:55:03 DEBUG i:3  result:16.00
2020-12-18 20:55:03 DEBUG i:0  result:2.00
2020-12-18 20:55:03 DEBUG i:1  result:4.00
2020-12-18 20:55:03 DEBUG i:2  result:8.00
2020-12-18 20:55:03 DEBUG i:3  result:16.00
2020-12-18 20:55:03 DEBUG Exp. negativo, multiplico por -1. result:0.06
```

Que debería servir para darnos cuenta de que el error ha sido empezar el bucle por 0 y ejecutarlo 4 veces

¿Qué trazar?

Cuando nuestro subprograma no hace lo que esperamos, el problema puede estar en

- Los parámetros que recibe, que no son los que deberían

- Las expresiones principales del cálculo
- La asignación del resultado al valor de retorno o a los parámetros de salida

En otras palabras: el error puede estar en cualquier parte, podemos necesitar trazar cualquier cosa

- Pero tampoco tiene sentido trazar todas y cada una de las variables y parámetros, esto añadiría demasiada confusión tanto al código fuente como a la traza
- En cada caso debemos decidir qué trazar