Programación en Pascal.

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Septiembre de 2022

Contenidos

Programación de Ordenadores

- Programar es darle instrucciones a un ordenador.
- 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.

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.

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...

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.

[1] salvo en alguna excepción

- 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.

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, 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...

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

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 ($10^3 = 1000$).

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.

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 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.

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.

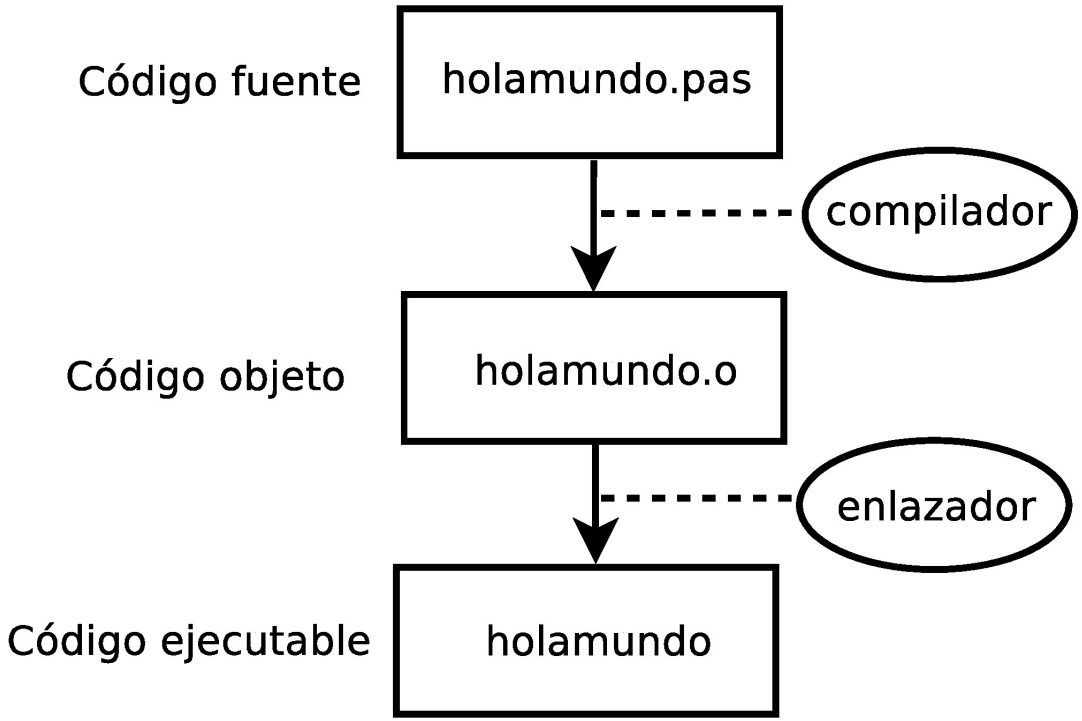
Ejemplo en Java:

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

Ejemplo en Pascal:

```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.

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

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
```

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...
- 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.

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
    ```