

## **Parte II**

# **Introducción al lenguaje C. Construcción de algoritmos y programas informáticos**



## Tema 2a

# Programas informáticos: qué son y cómo se construyen

Un programa informático no es más que un conjunto de instrucciones y datos que ejecuta un ordenador.

Por supuesto, un ordenador hace esto mediante su procesador, que obtiene de la memoria del ordenador “palabras” que significan, tanto instrucciones que tiene que realizar (esas palabras forman un diccionario de “opcodes” o códigos de operación específicos del tipo de procesador), como los datos que tiene que emplear en ellas (estos datos son siempre números, aunque después se puedan interpretar de otras formas).

El procesador del ordenador es capaz de hacer unas pocas operaciones básicas: leer y escribir “palabras” en la memoria de acceso aleatorio (RAM), operar con las “palabras” cargadas en sus registros (haciendo con ellas simples operaciones matemáticas: sumar, restar, cambiar signo, mover bits, funciones matemáticas estándar, etc.), enviar y recibir “palabras” a otros dispositivos conectados al ordenador (el disco duro, la tarjeta de red, la tarjeta de sonido, etc.).

Inicialmente, los procesadores eran suficientemente sencillos como para programarlos pasándoles los “opcodes” y los datos sobre los que tenían que efectuarse las operaciones. Todavía se puede hacer, aunque de una manera más legible por el programador, programando en (lenguaje) ensamblador. He aquí el ejemplo más sencillo con el que un ordenador se comunica con el mundo (diciéndole “Hola”):

```
.data
LC0:
    .ascii "Hola\u201c mundo!\n\0"
.text
    .global main
    .extern printf
main:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    movl $LC0, (%esp)
    call printf
    movl $0, %eax
    leave
    ret
```

Todo lo que hace este programa es escribir “Hola mundo!” en la pantalla del ordenador. Mejor dicho, lo que hace es enviarle a una “función” proporcionada por el sistema operativo (`printf`) los datos suficientes para que aparezca en pantalla (o donde el sistema operativo considere) la famosa frase. El resto del código (los “opcodes” representados por las instrucciones del procesador, como “`movl`”, “`call`”, etc.) y los datos (como la secuencia de bytes que representa “Hola mundo!”) sólo sirve para preparar esta llamada. No se incluyen en este ejemplo todas las operaciones que tiene que hacer el sistema operativo para reconocer los datos como un programa, ubicarlo en memoria, etc.

Afortunadamente, toda esta complejidad (necesaria) para indicarle al procesador qué operaciones debe hacer, con su limitado conjunto de operaciones, se puede enriquecer agrupando las operaciones y dándoles un nombre; luego, se llama al paquete de instrucciones (que llamaremos procedimiento o función). Esto es lo que se hace al llamar a “`printf`”: dicha función requiere más de 5000 instrucciones del procesador, pero que son siempre las mismas.

Esta estrategia es la que se encuentra en la base de los lenguajes de programación: consisten en un conjunto muy rico de operaciones (algo que los humanos podemos “manejar”) cada una de las cuales incluye una gran cantidad de operaciones sencillas, las pocas que puede realizar el procesador.

Un ejemplo de lenguaje de programación es el “C”, que vamos a estudiar. En este lenguaje, el anterior programa se escribiría como:

```
#include <stdio.h>
int main()
{
    printf("Hola mundo\n");
    return 0;
}
```

Este código es comparativamente más corto y, unas secciones más adelante, será considerablemente más claro.

## El lenguaje C: ¿por qué debe estudiarlo un físico?

Aunque parezca una pregunta extraña, el lenguaje C se ha incorporado hace muy poco al arsenal de herramientas informáticas de los físicos. El lenguaje tradicional para el cálculo numérico (de físicos, matemáticos, ingenieros, ...) ha sido el Fortran. El nombre de este lenguaje significa “traductor de fórmulas” (FORmula TRANslator, en Inglés), y es eso lo que hace, precisamente: traduce las fórmulas matemáticas al lenguaje de la máquina (datos, “opcodes” y demás). Esto genera programas muy rápidos haciendo cálculos, pero poco más. La traducción literal no es siempre la mejor traducción.

El lenguaje C apareció como un lenguaje para programar sistemas operativos, no para hacer cálculos. El sistema operativo UNIX fue el primero escrito en este lenguaje. La estructura del lenguaje C, por esa razón, facilita la división del código en secciones reutilizables, que se llaman “funciones” (en otros lenguajes se habla también procedimientos o subrutinas, pero no nos pararemos a ver la diferencia entre esos conceptos).

El paso del ensamblador a un lenguaje como el Fortran representa una abstracción de las operaciones concretas a representaciones más familiares (por ejemplo, de escribir “`movl 2,%eax \\\ movl 3,%ebx \\\ add %eax,%ebx`” se pasó a tener “`a=2 \\\ b=3 \\\ c=a+b`”).

El paso del Fortran al C representa la posibilidad de crear nuevas funciones y nuevas maneras de acceder libremente a los recursos de la máquina (digamos a la memoria o al

disco) con total libertad, de acceder a todas las funcionalidades del sistema operativo (sin necesidad de que nadie provea “librerías”, “unidades” o “componentes” de enlace), . . . y todo de una manera tan flexible, que casi se puede decir que el programador adapta el lenguaje a sus necesidades. De todo esto se deduce que quien escribe esta introducción es un fanático del lenguaje C.

¿Por qué, entonces, debe un físico aprender C? Básicamente por tres razones.

- La primera es que un físico no siempre quiere hacer cálculos con fórmulas matemáticas. Puede querer hacer cálculos con otras cosas, como cadenas de símbolos; es conocida la participación de muchos físicos teóricos en el desarrollo de algoritmos de búsqueda usados para la decodificación de secuencias genéticas. Puede querer hacer cálculos con máquinas o autómatas celulares, para simular procesos de formación de patrones o de evolución de sistemas biológicos, como veremos en un capítulo posterior. Puede querer, también, guardar o recuperar información en forma de imágenes, y para ello tiene que poder acceder con cierta libertad al disco y a la memoria. También puede querer adquirir datos de un dispositivo de laboratorio, y para ello tiene que poder llamar a funciones que “saben” cómo enviar y recibir datos de un dispositivo en concreto. Todas estas ventajas las proporciona el lenguaje C.
- La segunda es más filosófica, y tiene que ver con la actitud del físico. Un físico no se conforma con tener una herramienta; siempre quiere saber cómo funciona. Un lenguaje como el C acerca al programador a la forma de funcionar de la máquina que está programando (la llamada “máquina virtual”, dentro del mundo UNIX<sup>1</sup>); es más, le permite “dominarla”. Otros lenguajes como el Basic buscan acercarse al ser humano, pero eso los limita, porque los seres humanos somos muy diferentes de las máquinas. Otros, como el Fortran buscan proporcionar una herramienta “para salir del paso”, pero uno pronto querrá más . . .
- La tercera es más práctica. El lenguaje C es el más sencillo de todos los lenguajes que permiten hacer todo esto. Un físico puede desarrollar toda su carrera (y su investigación) sin necesidad de conocer otro lenguaje de programación. Luego, si uno quiere aprender más sobre teoría de programación, usará el C++ (programación orientada a objetos). En el extremo contrario, si uno quiere hablar, ya no con un ordenador, si no con un chip particular, usará el ensamblador (el que se adapte a los “opcodes” de ese chip. . . si es que no se ha desarrollado un compilador C adaptado a él).

## Manuales y referencias del lenguaje C

Para aprender lenguaje C existen muchas referencias tanto bibliográficas como en Internet. Aquí indicaremos dos, disponibles en Español:

- Kernigan & Ritchie
- Aprenda lenguaje ANSI C como si estuviera en primero

---

<sup>1</sup>No confundir esta “máquina virtual”, que es la máquina (real) vista “a través del sistema operativo”, con las máquinas virtuales como Java o .NET. Estas últimas son programas que se ejecutan en nuestra máquina pero que emulan el funcionamiento de otras máquinas “ideales”; los programas para esas máquinas virtuales consisten en datos y “opcodes” que, en lugar de ser pasados directamente al procesador de nuestra máquina, son primero interpretados y traducidos por el programa de “máquina virtual” al lenguaje de nuestro procesador.

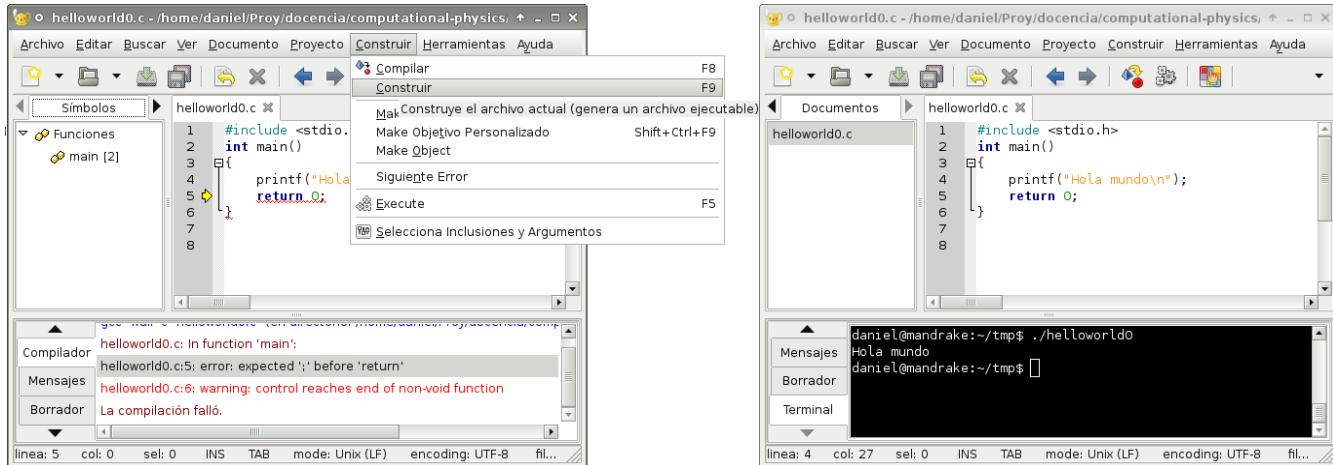


Figura 2a.1: Entorno de desarrollo Geany para GNU Linux.

Dicho esto, es tan cierto que alguien pueda aprender a programar en lenguaje C estudiando esas magníficas obras, como que alguien pueda aprender Francés estudiándose una gramática y un diccionario.

Por eso, lo que viene a continuación explica cómo instalar un compilador de C, cómo compilar programas, cómo modificarlos, etc. todo a través de ejemplos, esto es, leyéndolo, escribiéndolo y “hablándolo”, que es como realmente se aprende el lenguaje.

## 2a.1. Compiladores de C: descarga e instalación

Como se ha dicho más arriba, el sistema operativo UNIX está escrito en C. El descendiente más moderno y accesible de UNIX es el sistema operativo GNU/Linux. Además de su diseño estándar (UNIX es en la actualidad un “estándar” de sistema operativo de alto rendimiento) y de estar escrito en C (lo que, en sí, ya es una ventaja para el programador), Linux tiene otra ventaja más: la documentación sobre él es ilimitada. Si algo se puede hacer, en Internet está cómo hacerlo en Linux.

Linux se construye con el compilador de C del GCC (GNU Compiler Collection). En las primeras distribuciones de Linux, este compilador se instalaba al inicio como una parte fundamental del sistema operativo. Actualmente, es necesario instalarlo manualmente. En dos de las grandes distribuciones Linux (Debian y Fedora), esto se hace simplemente como:

<pre>apt-get install gcc</pre>	(Debian)
<pre>yum install gcc</pre>	(Fedora)

El compilador es solamente el programa que convierte el código que nosotros escribimos en el programa (el “ejecutable”) que cargará en memoria el sistema operativo y que se ejecutará en nuestra máquina (por nuestro procesador). Esto quiere decir que necesitaremos otro programa además para escribir el código (un “editor de texto”).

Un programa en Linux que nos ayuda en esta tarea es Geany. Este programa proporciona un “entorno de desarrollo” básico: un editor con sintaxis resaltada, una ventana de terminal para ejecutar comandos, una ventana con los resultados de la compilación (usualmente los “errores de la compilación”), un gestor de proyectos, varios menús para llamar al compilador, etc. El aspecto que tiene se puede ver en las figuras 2a.1.

Tomaremos el par Geany + gcc como la herramienta estándar en Linux.

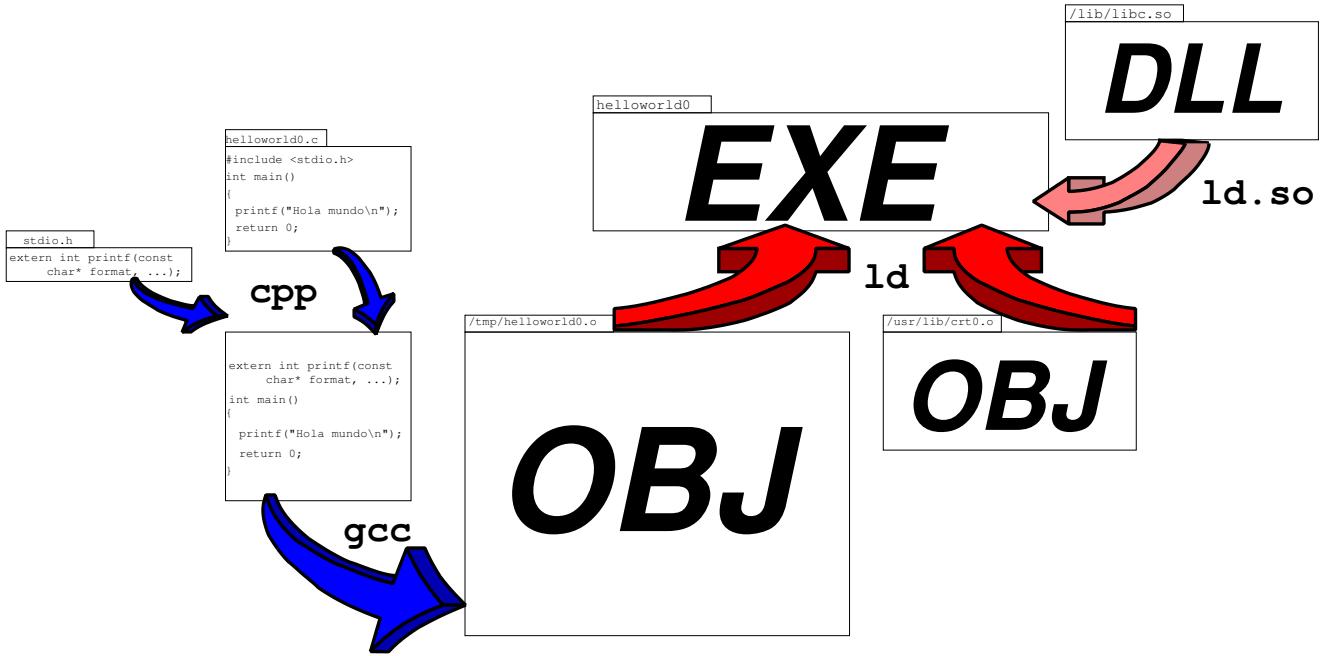


Figura 2a.2: Diagrama de flujo de la compilación de un programa.

### ¿Qué hacer si “se tiene Windows”?

La mejor opción en Windows es Code::Blocks: una aplicación gráfica con editor con sintaxis resaltada, ventana de mensajes, gestor de proyectos, menús de compilación, depurador, etc., es decir, un completo entorno de desarrollo. Y lo mejor es que puede instalar simultáneamente el compilador GCC (el mismo usado en Linux). La información sobre este entorno se encuentra en:

<http://www.codeblocks.org>

## 2a.2. Compilación, enlazado y ejecución de programas

Para saber cómo funciona el entorno de desarrollo y el compilador elegidos —o mejor dicho, para saber qué falla cuando no funcionan— es importante saber cómo se llega desde el programa “Hola mundo!” de más arriba a teclear “helloworld0” (que es como llamaremos al ejecutable) y ver cómo nuestro esfuerzo (y el del compilador) se ven recompensados con una línea de saludo en la consola, producida por nuestro programa.

En la figura 2a.2 se muestra un diagrama de bloques con el flujo de información que se produce. Partimos de un archivo de texto “helloworld0.c”, en codificación ASCII o, lo que se suele llamar “texto plano”; este archivo contiene el código con la sintaxis C (esperamos que correcta) que hemos escrito. Cuando llamamos al compilador sólo le decimos el nombre de este archivo de código y el nombre del ejecutable que queremos que produca (“helloworld0” en Linux, “helloworld0.exe” en Windows). Al decirle esto, “gcc” comienza una cascada de operaciones concatenadas unas con otras: preprocessamiento, “parsing” o interpretación, compilación y enlazado.

1. El preprocessamiento lo realiza el preprocessador, un programa llamado “cpp” (C PreProcessor). Se encarga de interpretar unas “directivas” sencillas (que no constituyen un

lenguaje en sí mismas, sino una especie de reglas de sustitución) que aparecen marcadas como líneas que empiezan por "#". Un ejemplo en nuestro primer programa es la primera línea: #include <stdio.h>. Esta línea incluye todo el código que existe en el archivo de "cabeceras" (o archivo H, de "header"), previamente interpretado por el propio preprocesador "cpp". En los archivos H se declaran las funciones disponibles para ser usadas en nuestro programa; en particular, la función "printf". Sin el archivo H, el compilador no sabría cuantos o de qué tipo son los argumentos que espera la función "printf". Veremos más adelante otras directivas del preprocesador muy útiles.

2. El "parsing" o interpretación de la sintaxis consiste en identificar las palabras clave del lenguaje (como "int" o "return"), los limitadores de ámbito (las "llaves"), las variables y funciones (como "main" o "printf"), los literales (como la cadena, entrecomillada, "Hola mundo!\n"), etc. Cualquier error, como la falta de un simple ":" al final de una línea, detendrá el proceso, ya que el código puede ser ambiguo.
3. La compilación consiste en, una vez identificada la estructura del programa, las estructuras de control en él, las funciones, las variables, las operaciones, etc. traducir éstas a instrucciones que entendería el procesador de nuestra máquina. Esto generaría (si lo guardásemos en el disco duro) un archivo "objeto" (con extensión O) que contiene todas las instrucciones correspondientes a lo que nosotros hemos escrito; nada más.
4. El enlazado sirve para unir nuestro "objeto" con otros "objetos" para construir un ejecutable. Esto lo lleva a cabo el programa "ld". Los objetos que une al nuestro son los que indican al sistema operativo, por ejemplo, cuál es la primera función que se ejecutará (main, en todos nuestros programas en C), o los objetos que proveen algunas funciones que nosotros no hemos escrito, o las "bibliotecas de funciones" (de enlace dinámico) que contienen otras de las funciones que nosotros llamamos; por ejemplo, la función "printf" se encuentra, en Linux, en la biblioteca "/lib/libc.so", y en Windows en la "c:\Windows\System32\MSCRT.DLL". Más adelante veremos que, cuando empleemos una función matemática como la raíz cuadrada, "sqrt", deberemos decirle al GCC dónde encontrarla (para que se lo indique al "linker"): añadiremos la opción "-lm", que significa que la biblioteca de enlace dinámico se encuentra en un archivo llamado "libm.so", en alguna parte (convenida) de nuestro sistema de ficheros.

El programa "gcc" realiza todas estas operaciones él solo, por lo que se le llama siempre "el compilador" (la operación que no hace otro programa como "cpp" o "ld").

Es muy importante tener en cuenta un detalle de cómo se llevan a cabo todas estas etapas: una justo a continuación de otra. Esto significa que, según el preprocesador completa la "traducción" de una línea, se la pasa al "parser"; cuando el "parser" encuentra una instrucción en C completa, se la pasa al compilador propiamente dicho; cuando el compilador genera las instrucciones binarias que equivalen a esa instrucción en C, las añade a un "objeto" (que guarda en un archivo temporal). Cuando el objeto está completo, el "linker" lo une, indexa y referencia con los demás objetos, produciendo, finalmente, el ejecutable. Este flujo continuado diferencia al C de otros lenguajes porque obliga, por ejemplo, a declarar todo lo que se va a utilizar antes de utilizarlo, como el caso de la función "printf", que está declarada en "stdio.h" como

```
extern int printf (const char *format, ...);
```

Pronto veremos que todas las funciones y variables que se vayan a emplear deberán haber sido previamente declaradas.

A continuación sigue una sección práctica para hacer todas estas operaciones de varias maneras, y ganar familiaridad con las herramientas.

## 2A.2. COMPILACIÓN, ENLAZADO Y EJECUCIÓN DE PROGRAMAS

2a-7

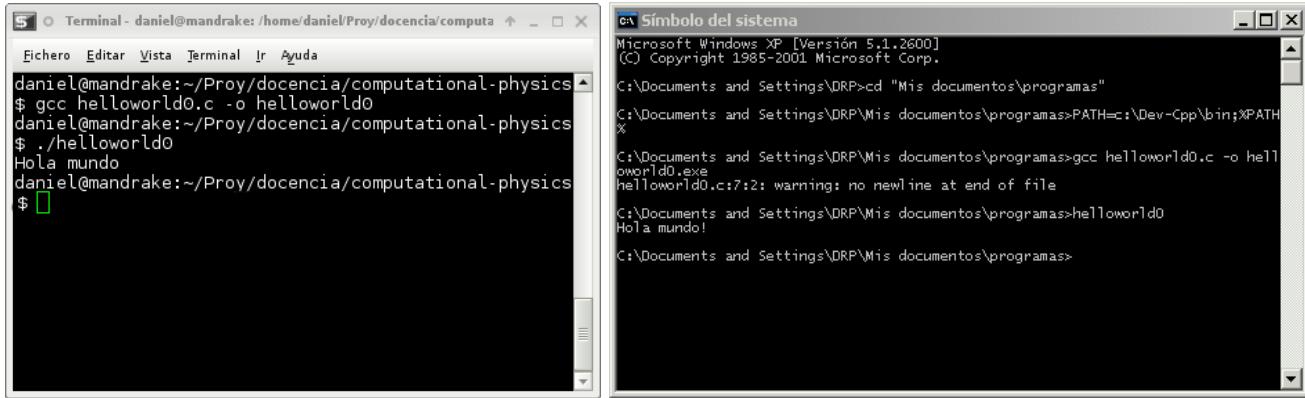


Figura 2a.3: Compilar el programa “Hola mundo!” usando “gcc” en Linux (izquierda) y Windows (derecha) desde “línea de comandos”.

### Compilación desde la línea de comandos

Para poder compilar, desde una terminal o consola, un programa C usando el compilador “gcc” es necesario, en primer lugar, que éste se encuentre en la ruta de búsqueda de ejecutables: el “PATH”. En Linux, esto es automático: el compilador se instala en “/usr/bin” como la mayoría de los ejecutables. En Windows, la instalación se lleva a cabo en el directorio que indica el usuario, y hay que indicar que se incluya dicho directorio en el “PATH”<sup>2</sup>.

En la figura 2a.3 se muestra como se llevan a cabo los pasos de compilación (entendida como lo que hace el “gcc”) y ejecución del programa “Hola mundo!”.

Lo primero es moverse al directorio en el que se encuentra el archivo que contiene el programa en C.

A continuación se llama al compilador con tres argumentos de línea de comandos: el nombre del archivo C, un indicador de opción (que comienza con “-” o “–”) relativo al nombre de la salida (“-o”, de “output”, salida en Inglés), seguido del nombre de este archivo de salida ejecutable.

```
cd ~/home/daniel/Proy/docencia/computational-physics
gcc helloworld0.c -o helloworld0
```

Si todo va bien, el compilador terminará sin decir nada. Esto es lo que sucede en el ejemplo en Linux de la figura 2a.3.

Si sucediera algo sospechoso, dará un mensaje de “Warning” (advertencia), pero acabará produciendo el ejecutable. Esto es lo que sucede en el ejemplo en Windows de la figura 2a.3: “gcc” sospecha que el archivo no esté completo, porque acaba bruscamente, sin una línea en blanco final. Añadir esa línea en blanco, hará desaparecer la advertencia.

Si hubiera un error en la sintaxis o una incoherencia lógica (evidente) en el programa, dará un mensaje de “Error”. Este mensaje se ve con mucha más frecuencia de la que uno desearía... basta borrar un simple “;” en el código del programa...

La explicación que sigue al “Error” o “Warning” es casi siempre suficiente para repararlo.

Para ejecutar el programa, basta escribir su nombre precedido de la ruta para llegar a él. Esto es así en Linux, porque sólo los programas en los directorios de la variable “PATH” se

<sup>2</sup>Si se decide emplear este método para compilar todos los programas, se debe incluir este directorio en el “PATH” del sistema. Para ello, en Windows XP, se seleccionan las “Propiedades” de “Mi PC”, y en la pestaña “Opciones Avanzadas”, el botón “Variables de entorno”. Una vez ahí, hay que añadir el nombre del directorio a la variable de usuario “PATH”.

buscan como ejecutables (como el propio “gcc”, porque “/usr/bin” está en el “PATH”); los demás, es necesario indicar dónde están, basta con decir “en el directorio actual”, representado por “./”, así

```
./helloworld0
```

En Windows esta precaución no es necesaria: en este sistema operativo el directorio de trabajo se entiende que está en el “PATH” (aunque no aparezca explícitamente)<sup>3</sup>.

Si todo funciona correctamente, veremos en el terminal la salida producida por el programa. Si algo falla, como veremos más adelante, el propio programa o el sistema operativo se encargarán de indicárnoslo (si es grave/evidente).

## Compilación usando Geany

1. Geany proporciona un editor de texto con sintaxis resaltada. Además de esta ventana, que es la que más espacio ocupa, existen otras dos: en la izquierda se muestran una serie de listas relativas al código que se está editando y debajo una serie de pestañas informan sobre el estado del editor o de los procesos de compilación (por ejemplo, errores).

La más inmediata lista a la izquierda es la que muestra los documentos abiertos. Inicialmente sólo aparece uno, “sin título”; tan pronto se guarde y se le asigne un nombre (como “helloworld0.c”) cambiará esta lista. Con las flechas de la parte superior, se puede acceder a otra lista que, más adelante, nos será útil: la lista de elementos declarados en el programa. Una vez escrito el programa “Hola mundo”, el único elemento declarado que aparece es la función “main”; las demás declaraciones están en “stdio.h”, que es incluida, pero no se está editando.

En la parte inferior se puede ver, en la pestaña “Estado”, las operaciones que se hacen con el programa: cuando y con qué nombre se crea o guarda. La pestaña de “Compilador” es la que nos informará de la llamada que hace a “gcc”, del éxito de la compilación, o de los errores o advertencias. Como el compilador informa del número de línea en la que se produce cada error, haciendo click sobre cada mensaje en la lista Geany desplazará el cursor del editor a dicha línea para que nos sea más fácil modificar el código.

2. Los mismos pasos que se han llevado a cabo desde la línea de comandos, se pueden realizar con Geany seleccionando las opciones desde los menús. Por ejemplo, igual que antes generamos el ejecutable con “gcc”, ahora se puede elegir “Construir > Construir” para generar el ejecutable. En la ventana inferior, pestaña de “Compilador”, nos mostrará que hemos llamado, precisamente a “gcc” como

```
gcc -Wall -o helloworld0 helloworld0.c
```

La novedad respecto al modo en que lo hicimos antes es la inclusión de la opción de línea de comandos `-Wall`. Esta opción hace que el informe de “Warnings” sea muy detallado.

3. Una opción más rápida que compilar y generar el ejecutable como hemos hecho es estrictamente compilar (menú “Construir > Compilar”), generando un archivo O... si toda la sintaxis es correcta. Esto se usa muchas veces para comprobar la sintaxis (aunque

<sup>3</sup>Debe tenerse en cuenta, si se llama al programa desde otro directorio, que el separador de directorios en Windows es la “barra invertida”, no la de “dividir”. Esto dará sus problemas en C, ya que la barra invertida tiene significado propio en este lenguaje (nacido, recordemos, antes que el MS-DOS y el Windows).

## 2A.2. COMPILACIÓN, ENLAZADO Y EJECUCIÓN DE PROGRAMAS

2a-9

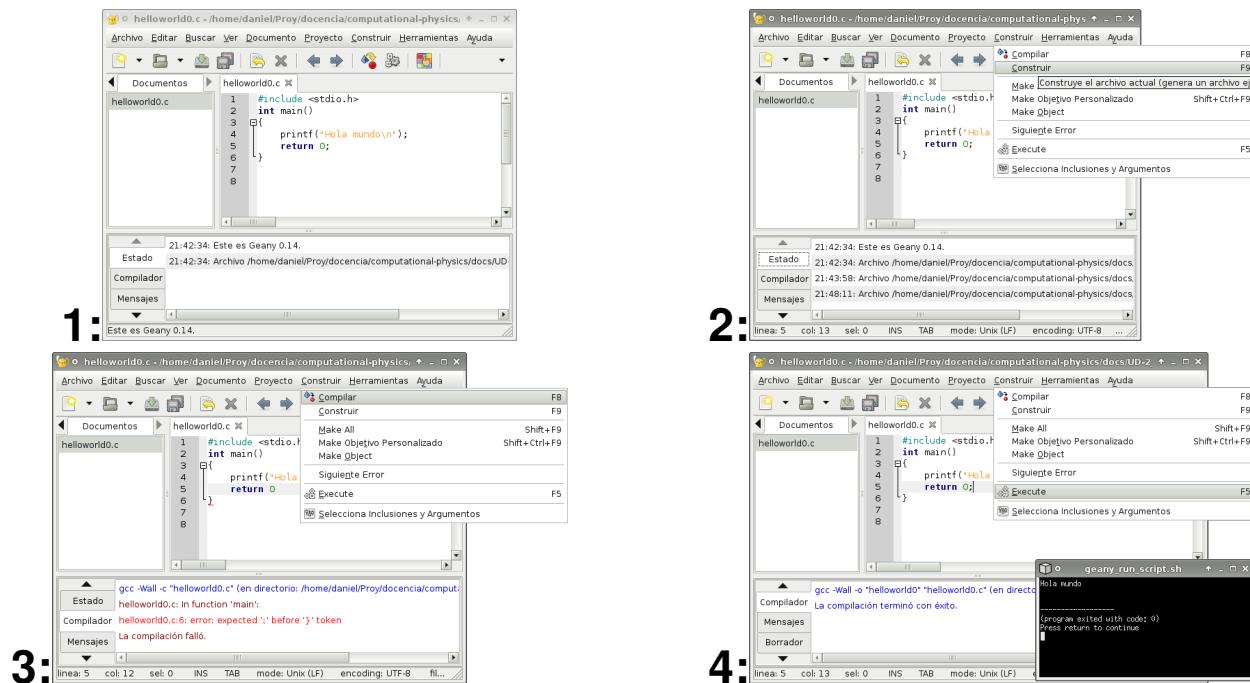


Figura 2a.4: Pasos para crear, compilar y ejecutar el programa “Hola mundo!” usando Geany.

en esto también nos ayuda Geany, resaltando dónde nos falta cerrar una llave, o unas comillas, o poner un “;”). En la pestaña de “Compilador” podemos ver cómo se hace esto (parar el proceso en la compilación, generando el archivo O, sin llegar al enlazado):

```
gcc -Wall -c helloworld0.c
```

4. Para hacer pruebas de ejecución de nuestro programa, Geany también proporciona un menú: “Construir > Ejecutar”. Cuando lo seleccionamos, se abre una ventana de terminal (con fondo negro, habitualmente) dentro de la que puede ver la salida de nuestro programa (“Hola mundo!”) y un mensaje final de “Pulse enter para terminar...”, tras lo cual se cerrará la ventana. Para poder ejecutar el programa, por supuesto, debe haberse creado antes.

## Compilación usando Code::Blocks

Todos los entornos de desarrollo se parecen. Para no repetir simplemente lo que se ha hecho con Geany en Linux, vamos a crear un proyecto en Code::Blocks.

Un proyecto consiste en un conjunto de archivos de código C que contienen distintas partes del código de un programa. Un programa muy largo, se suele dividir en varios archivos. También se hace esto cuando alguno de esos archivos se ha “heredado” de otro proyecto, o cuando creamos nuestros propios archivos H y nuestras propias bibliotecas de funciones.

1. Para crear un proyecto en Code::Blocks seleccionaremos el menú: *File > New > Project*. El nuevo proyecto queremos que sea una *Console application* (aplicación de consola) y, en el siguiente diálogo diremos que el lenguaje de programación será C. Después, en otro diálogo le daremos un nombre: “HolaMundo”. El programa nos preguntará por la carpeta en la que guardaremos el proyecto y todos los archivos referidos a él (nuestros

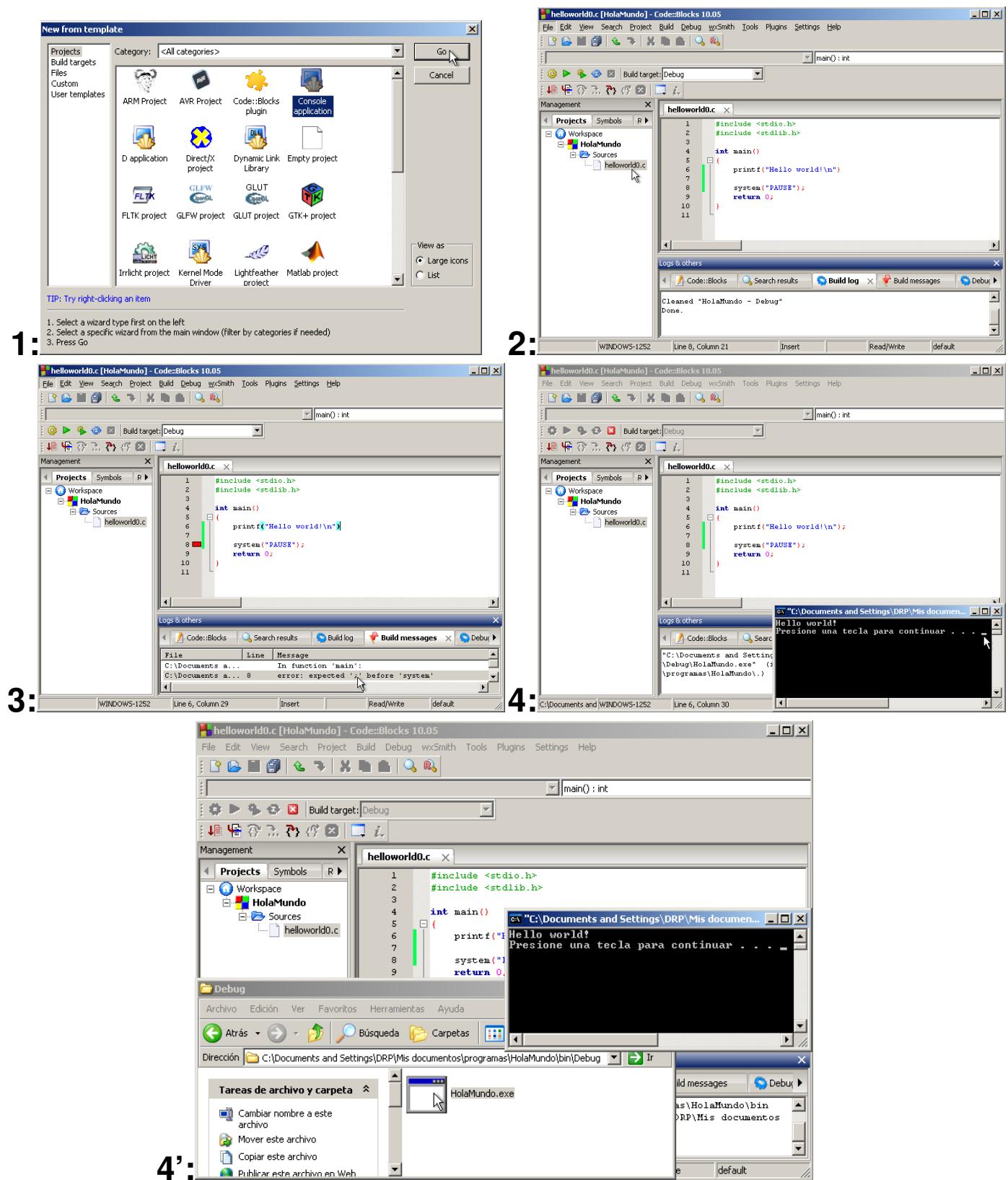


Figura 2a.5: Pasos para crear, compilar y ejecutar el proyecto “Hola mundo!” usando Code::Blocks.

### 2A.3. APPLICACIONES AUXILIARES

2a-11

y creados por el entorno de desarrollo y por el compilador); conviene que cada proyecto tenga su carpeta. En ella creará un archivo “HolaMundo.cbp” (la extensión CBP indica que es un *Code::Blocks Project*).

2. Al elegir la opción *Console application*, le indicamos a *Code::Blocks* qué prototipo de aplicación queríamos. Por eso, veremos que ya tenemos el “esqueleto” de la aplicación de consola creado: un archivo “main.c”. Toda aplicación de consola puede empezar así, con este esqueleto. Elijamos *File > Save file as ...* (guardar archivo como) y cambiémosle el nombre a “helloworld0.c”, como siempre. En el árbol de proyecto (*Workspace*) sobre el archivo “main.c” seleccionemos *Remove file from project* (quitar archivo del proyecto) y luego *Add files* (añadir archivos), para añadir el nuevo “helloworld0.c” que hemos guardado.

Podemos *Compile current file* (compilar el archivo actual), lo que en algunos contextos se llama “módulo” del programa. Esto generará, en el correspondiente subdirectorio del proyecto un archivo O. Eso quiere decir que no se ha llamado al “linker”. El proceso es muy rápido y nos permite detectar errores en nuestro archivo, sin perder tiempo compilando todo el programa. Si, por ejemplo, nos hubiéramos olvidado un “;”, ese error aparecería en la pestaña de mensajes del “Compilador”.

Podemos también *Build* (construir) el programa. Esto generará todos los archivos O de todos los módulos que no hayan sido compilados o que hayan sido modificados desde la última vez; luego lo enlazará todo en un ejecutable. Ésta es la opción preferida, porque genera el ejecutable que queremos.

3. A la construcción del programa le puede seguir su ejecución con *Build and run* (construir y ejecutar) o se puede hacer *Run* (ejecutar) el programa ya compilado. También se le puede, esta vez en el menú de *Project*, *Set program's arguments...* (fijar los argumentos), es decir, los parámetros de línea de comandos que usaremos más adelante, igual que si llamásemos al programa desde una consola y escribiésemos los parámetros a continuación del nombre de nuestro programa.

Otra alternativa es ejecutar el programa haciendo doble click sobre el ícono del ejecutable en una ventana del explorador de Windows.

## 2a.3. Aplicaciones auxiliares

Un único programa no tiene por qué hacerlo todo. Esto está en contraposición con los gigantescos programas a los que estamos habituados, pero no es el mejor diseño (como también estamos acostumbrados a sufrir por parte de dichos gigantescos programas). El mejor diseño es tener muchos programas pequeños, interoperables entre ellos, que hacen, cada uno, una tarea. De este modo, si se mejora una de esas tareas, sólo se rescribirá ese programa, no el conjunto de ellos. Esta es la filosofía tras el sistema UNIX: muchas herramientas simples y un sistema operativo que facilita la conexión entre ellas.<sup>4</sup>

<sup>4</sup>Esta filosofía se relaciona con la filosofía KISS (igual que “beso” en Inglés), por las siglas de “Keep It Small and Simple” (manténlo pequeño y sencillo).

## La documentación: man y apropos

Para escribir un programa se necesitarán muchas funciones. Unas las escribirá el programador, pero muchas otras están ya incluidas en las bibliotecas del sistema y, en Linux/UNIX, documentadas mediante la utilidad `man`. Así, por ejemplo, para acceder a la documentación de la función que interpreta cadenas como números enteros, `atoi`, se escribe en la línea de comandos:

```
man atoi
```

Cuando no se conoce el nombre de una función, o programa, pero sí lo que hace, se puede buscar con la instrucción `apropos`. Por ejemplo, si queremos imprimir algo (en Inglés “print”), buscaremos:

```
apropos print
```

y en respuesta recibiremos un largo listado, entre el cual figuran (entre muchas otras):

```
printf (1) - format and print data
printf (3) - formatted output conversion
printw (3ncurses) - print formatted output in curses windows
psignal (3) - print signal message
pwd (1) - print name of current/working directory
```

De todas ellas, la función C es `printf` (3). El número entre paréntesis la distingue de un programa, que también se llama `printf`, indicando que, para buscar la documentación, hay que llamar a `man` como:

```
man 3 printf
```

## Gnuplot

Otra herramienta que utilizaremos muy a menudo para representar los datos numéricos que produzcan nuestros programas es el programa “Gnuplot”. “Gnuplot” es una herramienta muy potente dedicada exclusivamente a representar datos gráficamente. En Linux se puede instalar con “yum” o con “apt-get”, del mismo modo que el “gcc”.

En Linux, “Gnuplot” se ejecuta desde una consola, llamándolo por su nombre en minúsculas “gnuplot” (algo habitual en Linux). En la línea de comandos se teclean las órdenes que se quiere que execute<sup>5</sup>.

Como ejemplo, en la tabla 2a.1 se muestra el contenido de un archivo de datos “sennal.dat” y los comandos de “Gnuplot” necesarios para representar esos datos<sup>6</sup>, una función que los aproxima (“Gnuplot” también podría calcular una) y una recta alrededor de la que oscilan los datos.

<sup>5</sup>Existe una aplicacioncita con interfaz gráfica que recurre a “Gnuplot” para hacer representaciones de datos contenidos en archivos del disco, llamada “PlotDrop”. Esta herramienta puede ser suficiente para empezar, pero su excesiva simplicidad no le hace justicia a “Gnuplot”

<sup>6</sup>Los comandos de “Gnuplot” pueden ser tan largos como se quiera. El final de cada uno lo indica el final de la línea de texto o un “;”. Para facilitar su lectura, sin embargo, se pueden romper en varias líneas, indicando que continúa en la siguiente con una “\” al final de cada línea del comando inconcluso. Eso es lo que hemos hecho en la tabla que sigue.

## 2A.3. APLICACIONES AUXILIARES

2a-13

Datos: sennal.dat		Código “Gnuplot”
#X	Y	
0	9	plot "sennal.dat" with points 3, \
1	9	1.5+10*sin(1.2*x) with lines,\
2	-2	1.5 notitle with lines
3	-7	
4	-2	set terminal png
5	10	set output "sennal.png"
6	11	replot
7	0	
8	-10	set terminal wxt
9	-4	

Tabla 2a.1: Ejemplo de datos en un archivo de texto (la primera línea es un comentario descriptivo) y comandos de “Gnuplot” para representarlos y guardar la gráfica resultante como una imagen PNG.

```

#parámetros de ajuste
y0=1.5
A=10
w=1.2

#función modelo
f(t)=y0+A*sin(w*t)

plot "sennal.dat" title "datos experimentales" \
      with points 3, \
      f(x) with lines,\ 
      1.5 notitle with lines

#salida a archivo
set terminal png
set output "sennal.png"
replot

set terminal wxt

```

Tabla 2a.2: Ejemplo de “Gnuplot”, contenido en un archivo de texto llamado “sennal.plt”. Para ejecutarlo, se escribirá: load "sennal.plt"

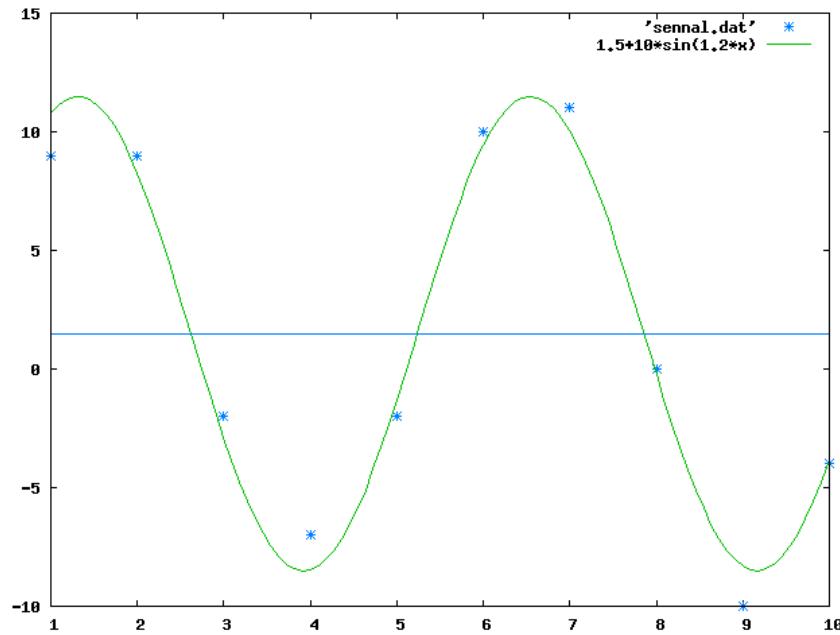


Figura 2a.6: Representación con “Gnuplot” de unos datos “experimentales”, de una función que los ajusta y de una línea de valor medio.

- El comando usado, “plot”, tiene una sintaxis sencilla: o bien se le indica el archivo de datos ( $X, Y$ ), o bien se le indica la expresión de la variable  $x$  que se quiere representar. Opcionalmente, se le puede indicar la forma de representar los datos (con puntos, líneas, etc. y también se puede indicar el tipo/color numerado 1, 2, 3, ...). Por defecto, “Gnuplot” añade una leyenda; ésta se puede modificar haciendo que no aparezca (con “notitle”) o indicando el nombre que se quiere dar con “title” seguido de una cadena entrecomillada con el nombre (por ejemplo, se puede sustituir “notitle” por “title ‘media’”). Más información sobre `plot` se puede obtener escribiendo “`help plot`”.
- Despues de representar los datos, se le dice a “Gnuplot” que, en vez de usar como salida una ventana gráfica, exporte la gráfica a una imagen en formato PNG (Portable Network Graphics), dibujándola de nuevo (`replot`); el nombre del archivo con la imagen será “`sennal.png`”. En la figura 2a.6 se muestra el resultado, que es prácticamente idéntico al mostrado en la ventana gráfica por “Gnuplot”.
- Al acabar, se vuelve a decir a “Gnuplot” que siga usando la ventana gráfica (“`wxt`” o “`x11`” en Linux, “`windows`” en Windows) para mostrar nuevas gráficas.

Las instrucciones de “Gnuplot” se pueden guardar en un archivo de texto y ejecutarlas después con “`load`”. En la tabla 2a.2 se muestra un ejemplo más completo basado en el anterior.

Finalizar diciendo que todo esto se puede hacer también en Windows, descargando “Gnuplot para Win32” desde

<http://www.gnuplot.info>

Desde aquí también se puede encontrar mucha información sobre el programa, el lenguaje, las capacidades, etc. Sin embargo, la mejor referencia en la Red sobre “Gnuplot” es la de T. Kawano, accesible desde

## 2A.3. APLICACIONES AUXILIARES

2a-15

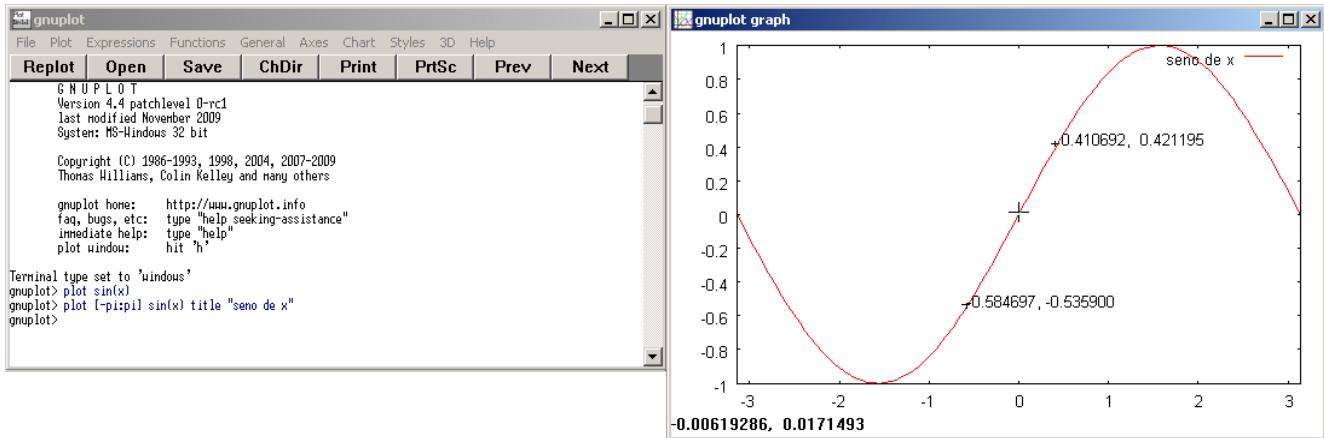


Figura 2a.7: Imagen de la interfaz gráfica de Gnuplot para Win32.

<http://lowrank.net/gnuplot/index-e.html>

(además de en Inglés, también está disponible en Japonés).



## Tema 2b

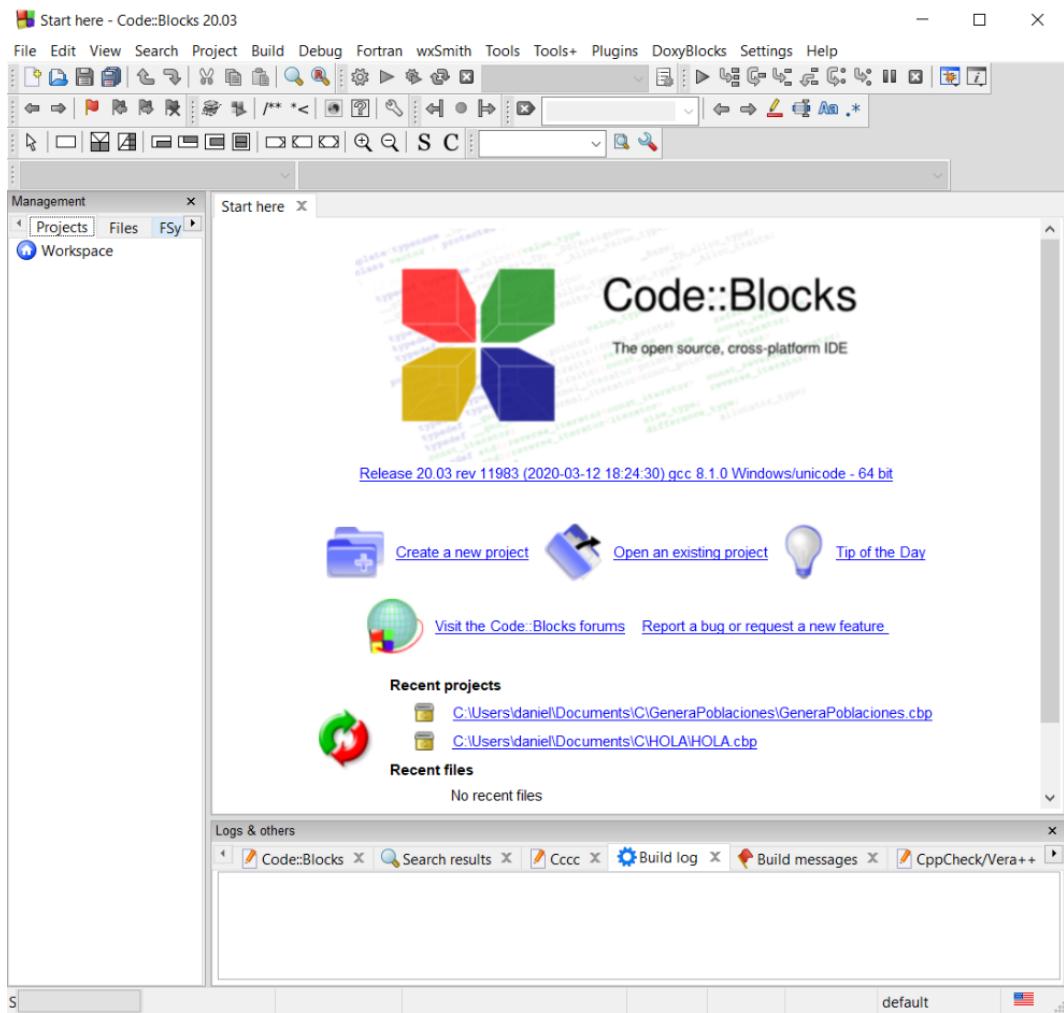
# Proyecto de C con Code::Blocks

El entorno de desarrollo que emplearemos preferentemente en esta segunda parte de la asignatura para programar en C será Code::Blocks. Ya ha sido introducido en el capítulo XXX, donde se explicó desde dónde se descarga, qué aspecto tiene, etc. Aquí describiremos paso a paso cómo se usa Code::Blocks para crear el programa ejecutable y cómo se llama a éste desde la consola de Windows.

Los pasos serán muy sencillos: crear un proyecto de programación en C, crear un archivo de código y escribir el código en él, compilarlo y construir el ejecutable, corregir el código si la compilación da error y, por fin, ejecutarlo para comprobar que funciona y obtener los resultados de un cálculo. Explicaremos todos estos pasos a través de dos ejemplos sencillos.

### 2b.1. Creación de un proyecto en C

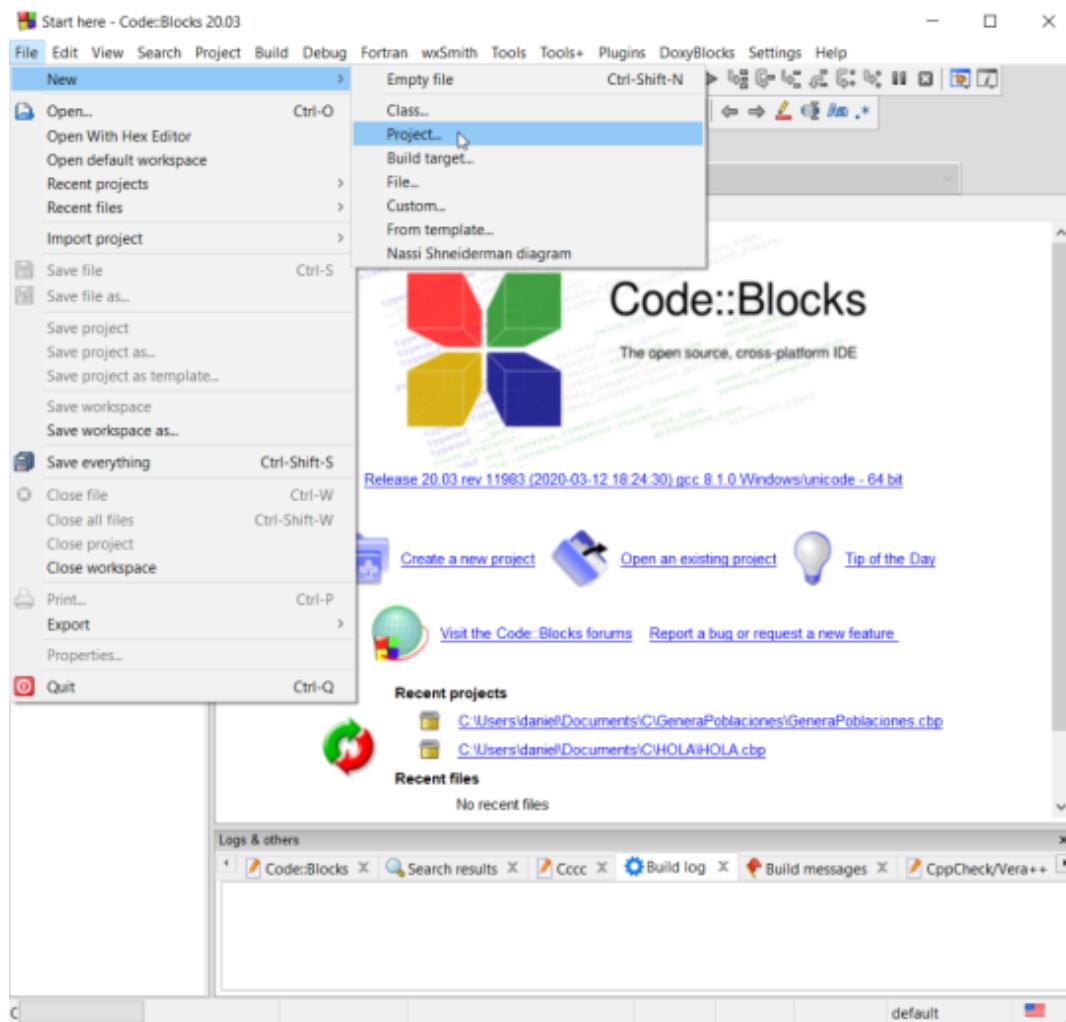
Comencemos abriendo el programa Code::Blocks. Éste es el aspecto típico de un entorno de desarrollo con menús en la parte superior, una serie de barras de botones de herramientas, un par de paneles, uno de proyecto y otro para edición (que, en este primer momento sólo muestra información sobre la versión de Code::Blocks), y un panel inferior que informa sobre los procesos que se ejecutan y los errores (“logs”).



Para ver cómo crear un proyecto en C con el que construir un programa ejecutable a partir de un código que escribamos en ese lenguaje, empezaremos por ir al menú “File > New > Project”.

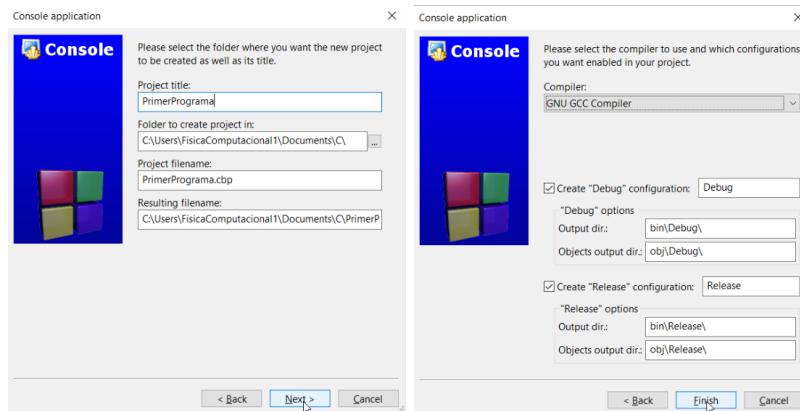
## 2B.1. CREACIÓN DE UN PROYECTO EN C

2b-3

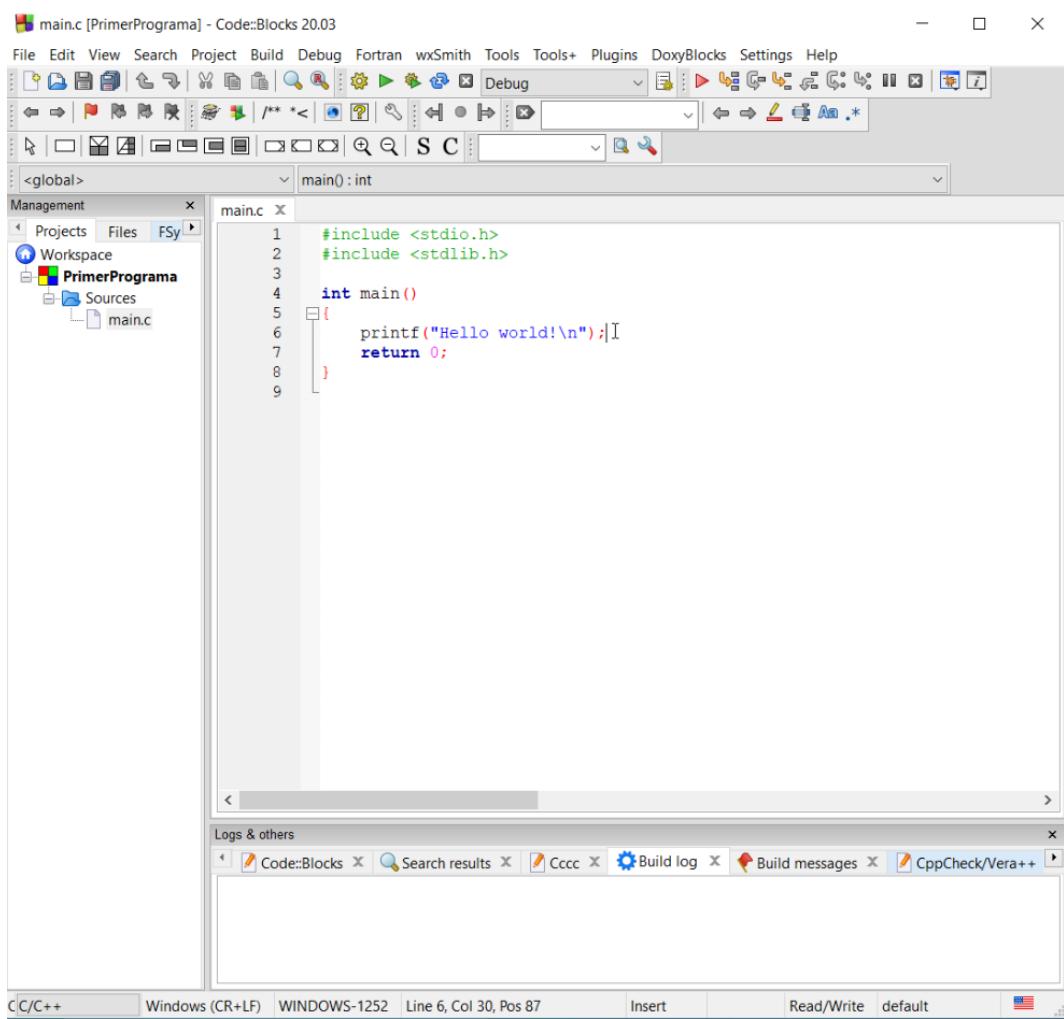


Code::Blocks es un entorno de desarrollo muy completo, que no sólo sirve para programar aplicaciones sencillas en C, como las que haremos en este curso. Por eso, nos guiará a través de una serie de diálogos para que seleccionemos entre las diferentes opciones: tipo de programa (elegiremos aplicación de consola), configuración de la aplicación (saltaremos este paso), lenguaje de programación de la aplicación (elegiremos C), nombre y ubicación de los archivos del proyecto (le daremos un nombre y lo guardaremos en una carpeta de nuestro directorio de usuario) y, por último, podremos elegir el compilador que usaremos (debería aparecer seleccionado GCC, que es el compilador que se instala junto con Code::Blocks al elegir la instalación completa).

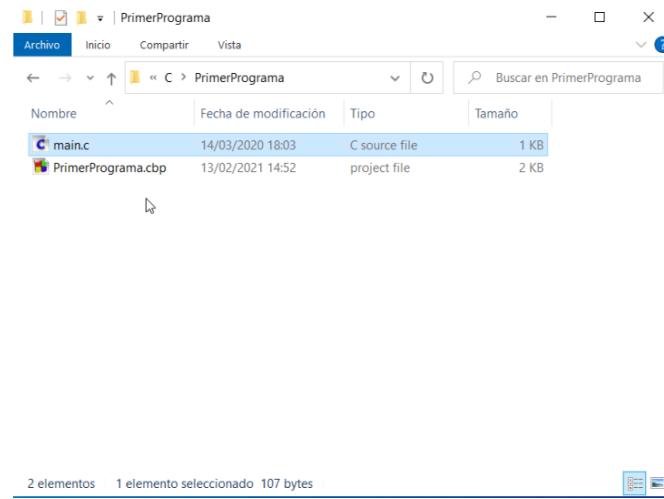




Tras completar todos estos pasos, aparecerá en el panel de proyectos nuestro “PrimerPrograma” (el nombre que hemos dado a este proyecto). Además, Code::Blocks, poblará este proyecto con un código en C de ejemplo, a partir del cuál podemos empezar a programar. El nombre del archivo de código es “main.c” y, si hacemos doble click en su nombre, se abrirá en el panel de edición: se trata del programa “Hello World” u “Hola Mundo”.



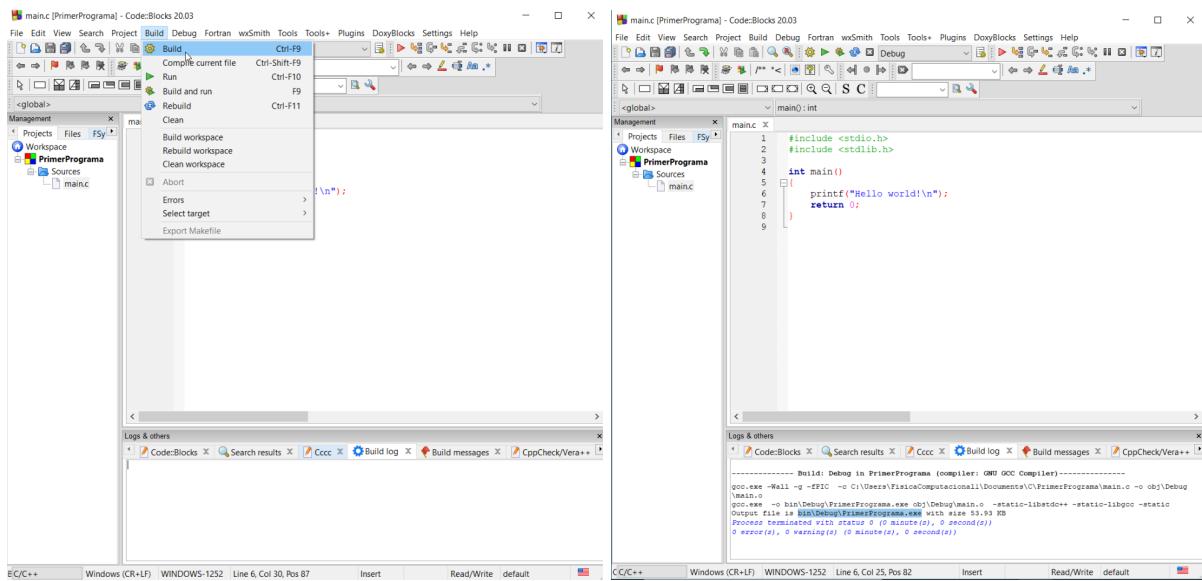
Si abrimos con el explorador de archivos la carpeta en la que pedimos a Code::Blocks que crease nuestro proyecto “PrimerPrograma”, veremos que contiene el archivo “PrimerPrograma.cbproj” (con las opciones de nuestro proyecto de Code::Blocks) y el archivo “main.c” que estamos editando.



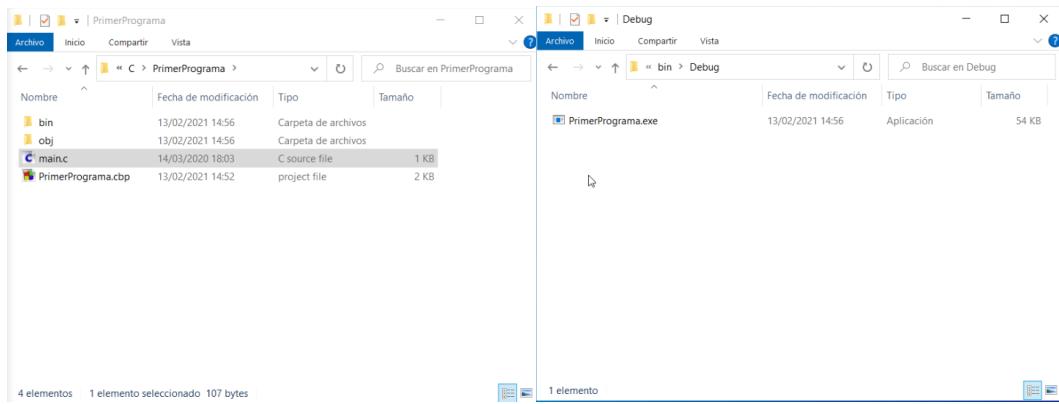
## **2b.2. Compilación y creación de un archivo ejecutable**

Este simple programa que ha añadido Code::Blocks a nuestro proyecto ya es posible compilarlo y enlazarlo para generar un programa ejecutable por nuestro ordenador. En Code::Blocks esto se hace en un único paso: desde el menú “Build > Build”.

Veremos que, en respuesta a este menú, se generan unos mensajes en el panel de información (“Build logs”): éasas son las llamadas al compilador y el enlazador (en nuestro caso, ambos son el mismo programa “gcc.exe”); entre los argumentos pasados podremos encontrar la ubicación del archivo ejecutable en el subdirectorio “bin/Debug” de la carpeta de proyecto; el ejecutable se llama igual que el proyecto, “PrimerPrograma.exe”.

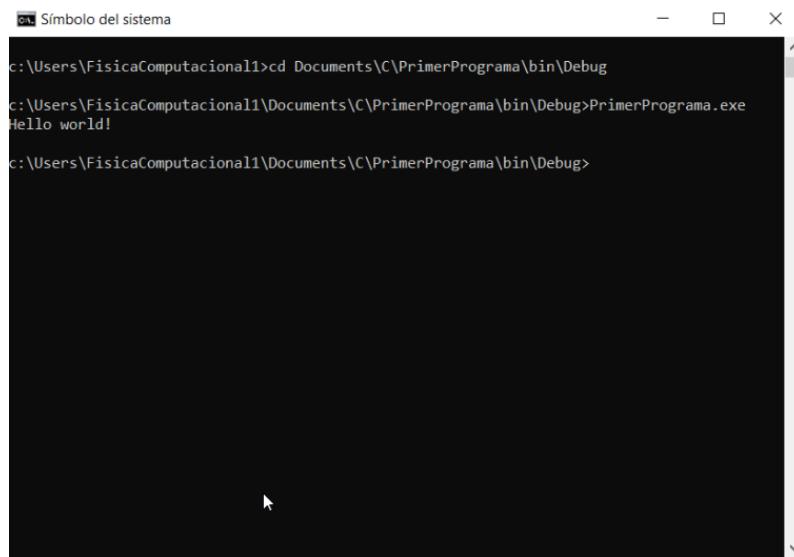


Si inspeccionamos ahora desde el explorador de archivos la carpeta de programa, veremos que contiene dos nuevas carpetas: una, llamada “obj”, contiene los resultados de la compilación (archivos “objeto”) y otra, llamada “bin”, contiene un directorio “Debug” con el ejecutable en su interior.



## Ejecución del programa: la consola y los argumentos de línea de comandos

Para ejecutar este programa vamos a utilizar la consola. La abriremos, iremos a la dirección de la carpeta de “Debug” (con el comando “cd” cambiamos el directorio de trabajo) y allí teclearemos el nombre del ejecutable (en este caso “PrimerPrograma.exe”). Veremos que el mensaje “Hello world!” aparece en la línea a continuación.



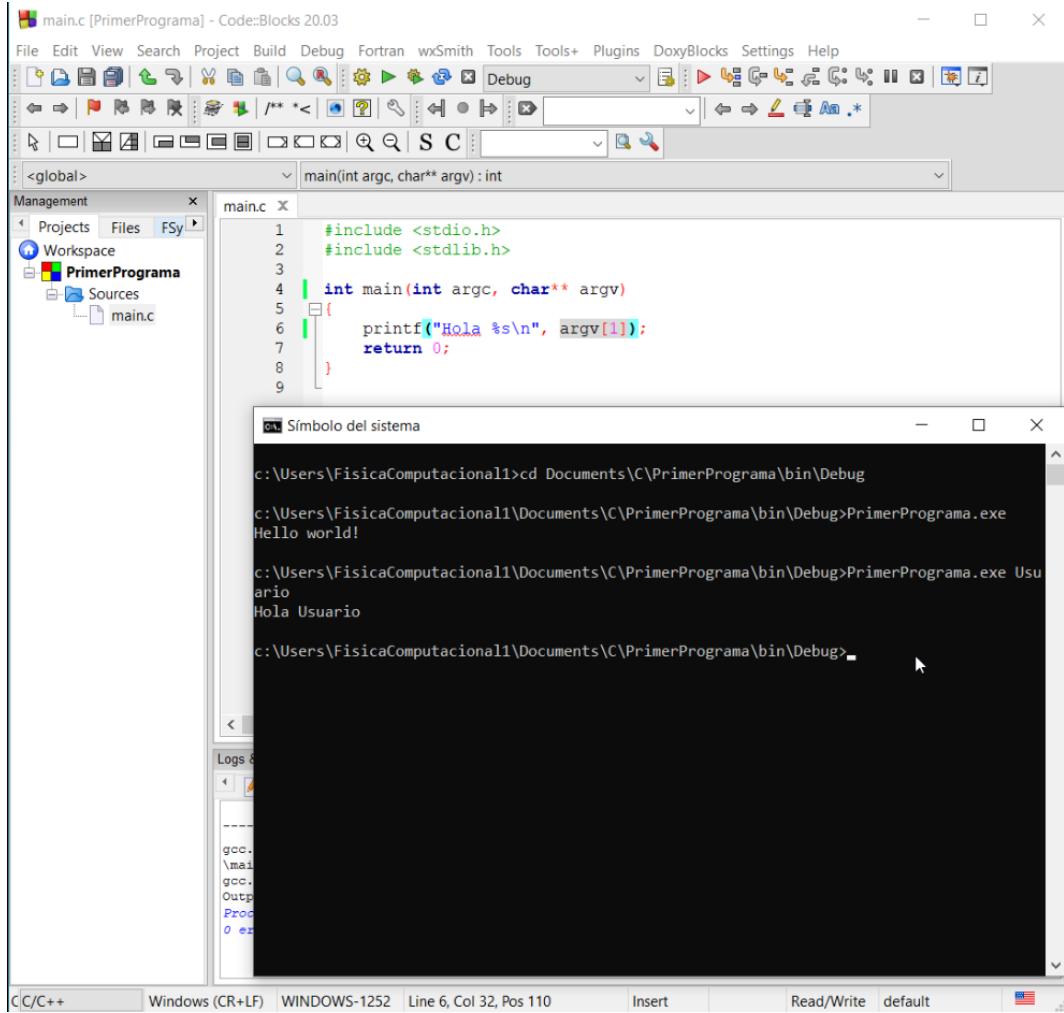
Se suele decir que el programa “Hello world” es el programa más sencillo que interactúa con el usuario diciéndole algo cuando éste lo llama. Pero esta interacción es muy limitada (siempre dice lo mismo). Vamos a complicarlo un poco modificando el código:

```
printf("Hola %s\n", argv[1]);
```

Con este cambio, el programa ya no imprimirá un genérico “Hola mundo”, sino que escribirá “Hola” seguido de la palabra que escribamos nosotros a continuación del nombre del ejecutable; es decir, hará algo diferente cuando nosotros lo llamemos de maneras diferentes. Construimos el ejecutable (como antes “Build > Build”) y volvemos a llamar al programa desde la línea de comandos, pero escribiendo después nuestro nombre): el programa nos saluda.

## 2B.3. ERRORES DE COMPILACIÓN

2b-7



¿Qué ha sucedido? Nosotros hemos llamado al programa desde la consola escribiendo su nombre, un espacio y luego nuestro nombre. La consola es, a su vez, un programa que interpreta esto que hemos tecleado y se lo pasa al sistema operativo con estos significados: la primera palabra es el nombre del programa (que puede incluir la ruta de directorios y subdirectorios para llegar a él), la segunda palabra es un argumento que se debe pasar al programa. A continuación podrían añadirse otros argumentos, separados por espacios; el programa que hemos escrito los ignorará (sólo usa el primero).

## 2b.3. Errores de compilación

Rescribamos completamente el “PrimerProgramma” para que haga algo más interesante y también interactivo: una cuenta atrás. Al escribir un programa es muy frecuente cometer algún error de sintaxis. La sintaxis de C es muy simple, pero también muy exigente. Al compilarlo, veremos que Code::Blocks marca la línea en la que hemos cometido el error más frecuente del C: falta el punto y coma (“;”) al final de la instrucción. En el panel de información aparece una descripción de este error que incluye también el nombre del archivo y el número de línea donde el compilador lo ha detectado.

```

main.c [PrimerPrograma] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran Tools Tools+ Plugins Doxygen Settings Help
Management x
Projects Files FSY
Workspace
PrimerProgramma
Sources main.c
main.c x
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv)
5 {
6     int N, n;
7
8     N=atoi(argv[1]);
9
10    printf("Cuenta atrás:\n");
11
12    for(n=N; n>=0; n--) {
13        printf("%d\n", n)
14    }
15
16    return 0;
17
18
Logs & others x
Code::Blocks x Search results x Cccccc x Build log x Build messages x CppCheck/Vera+
File Line Message
==== Build: Debug in PrimerProgramma (compiler: GNU GCC Compiler) ====
C:\Users\Fi... In function 'main':
C:\Users\Fi... 13   error: expected ';' before ')' token
==== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ====

```

Corregido el error y vuelto a compilar, ejecutamos el programa indicando el número desde el que queremos empezar la cuenta atrás. Si no indicamos ningún número, el resultado es impredecible; en este ejemplo afortunado, el programa interpreta que la falta de argumento representa el número cero, pero es realmente una coincidencia afortunada.

```

Símbolo del sistema
c:\Users\FisicaComputacional1>cd Documents\C\PrimerProgramma\bin\Debug
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma\bin\Debug>PrimerProgramma.exe
Hello world!
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma\bin\Debug>PrimerProgramma.exe Usu
ario
Hola Usuario
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma\bin\Debug>cd ..
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma\bin>cd ..
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma>bin\Debug\PrimerProgramma.exe 5
Cuenta atrás:
5
4
3
2
1
0
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma>bin\Debug\PrimerProgramma.exe
Cuenta atrás:
0
c:\Users\FisicaComputacional1\Documents\C\PrimerProgramma>

```

## 2B.4. RESUMEN

2b-9

### 2b.4. Resumen

Siguiendo los pasos descritos arriba, construyamos un proyecto llamado “SegundoPrograma”. El código de este segundo proyecto está escrito para que calcule el número  $\pi$  iterando un número de veces indicado por el usuario la serie alternada de Leibnitz. Construyendo el ejecutable y llamándolo desde la línea de comandos seguido de un número (hay que cambiar antes, con “cd”, al directorio de este nuevo ejecutable), obtendremos aproximaciones al número  $\pi$ , mejores cuanto mayor sea ese número.

The screenshot shows the Code::Blocks IDE interface. On the left, the Project Manager shows the SegundoPrograma project with a main.c file selected. The main.c code is as follows:

```

main.c [SegundoPrograma] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Management Projects Files Fsy
<global> main.c
1 main(int argc, char** argv)
2 {
3     int Niter, n;
4     double suma, flip, ePi;
5
6     if( argc <= 1 ) {
7         printf("El programa %s necesita saber el n-mero de iteraciones\n"
8             "exit(1);")
9     }
10
11     Niter=atoi(argv[1]);
12     suma=0.0;
13     flip=1;
14     for(n=0; n<Niter; n++) {
15         suma += flip/(2*n+1);
16         flip=-flip;
17     }
18     ePi=4*suma;
19
20     printf("PI = %f\n", ePi);
21
22     return 0;
23
24 }
25
26
27

```

The terminal window at the bottom shows the command-line interface and the resulting output for different iteration counts:

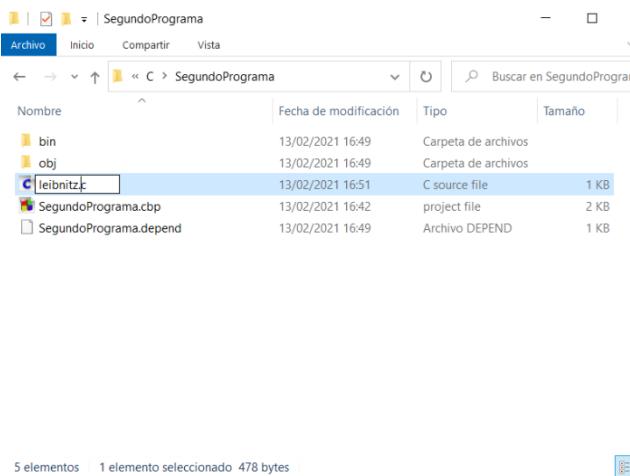
```

C:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>cd ..\SegundoPrograma
C:\Users\FisicaComputacional1\Documents\C\SegundoPrograma> SegundoPrograma.exe
El programa bin\Debug\SegundoPrograma.exe necesita saber el n-mero de iteraciones
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 1
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 1000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 10000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 100000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 1000000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 10000000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 100000000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>bin\Debug\SegundoPrograma.exe 1000000000
PI = 3.141593
c:\Users\FisicaComputacional1\Documents\C\SegundoPrograma>

```

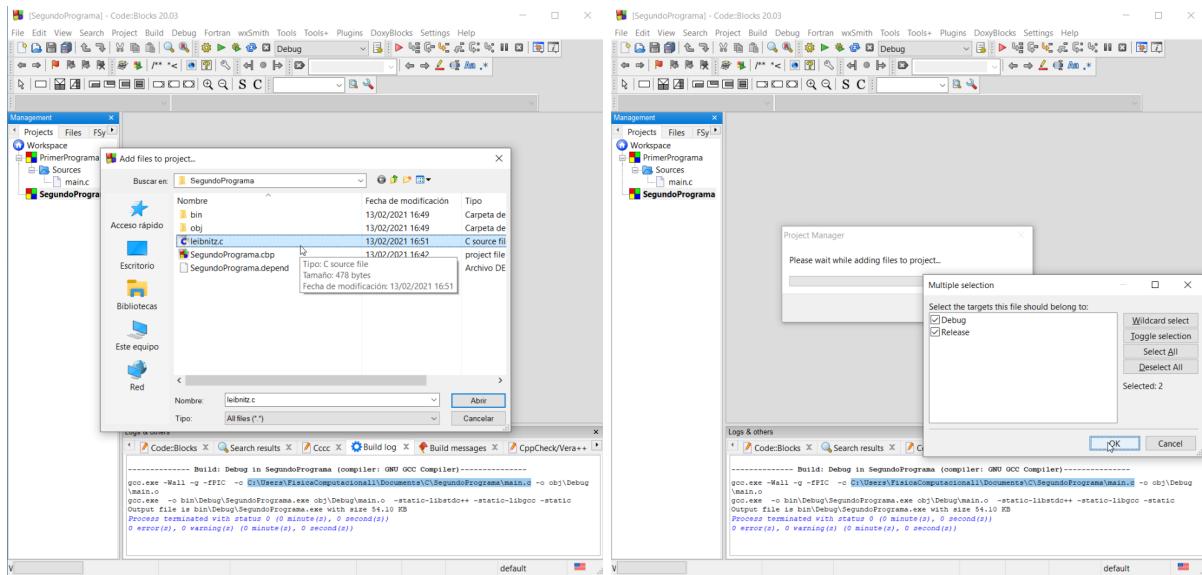
### Apéndice: Cómo añadir archivos fuente al proyecto

Que todos los códigos fuente de los proyectos se llamen “main.c” no es una buena idea a la larga. Lo más correcto es darles un nombre descriptivo, a menudo el del proyecto, pero también es frecuente darles un nombre acorde con aquello que calculan. Renombremos, en el directorio del proyecto, el archivo “main.c” como “leibnitz.c”, ya que se calcula la serie atribuida a este matemático.



Al cambiar el nombre del archivo, el ícono de “main.c” aparecerá “roto” en el proyecto y no se abrirá este archivo cuando hagamos doble click en él. Seleccionemos eliminarlo del proyecto (“Remove file from project”) en el menú emergente que se abre cuando pulsamos el botón secundario del ratón sobre el nombre “main.c”.

A continuación añadiremos el archivo “leibnitz.c” al proyecto. Para ello, hagamos click con el botón secundario del ratón sobre el nombre del proyecto (“SegundoPrograma”) y elijamos añadir archivos fuente (“Add files...”). Entonces seleccionemos en el diálogo abrir el archivo con el nuevo nombre. Luego, en el siguiente menú, aceptemos que ese archivo se incluya en las dos versiones de nuestro proyecto: “Debug” y “Release”.



Ahora aparecerá “leibnitz.c” en el árbol del proyecto y cuando hagamos doble click sobre él se abrirá en el editor.

## 2B.4. RESUMEN

2b-11

```

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int Niter, n;
    double suma, flip, ePi;

    if( argc <= 1 ) {
        printf("El programa &#x2019;s necesita saber el n”mero de iteraciones\n");
        exit(1);
    }

    Niter=atoi(argv[1]);
    suma=0.0;
    flip=1;
    for(n=0; n<Niter; n++) {
        suma += flip/(2*n+1);
        flip=-flip;
    }
    ePi=4*suma;

    printf("PI = %f\n", ePi);

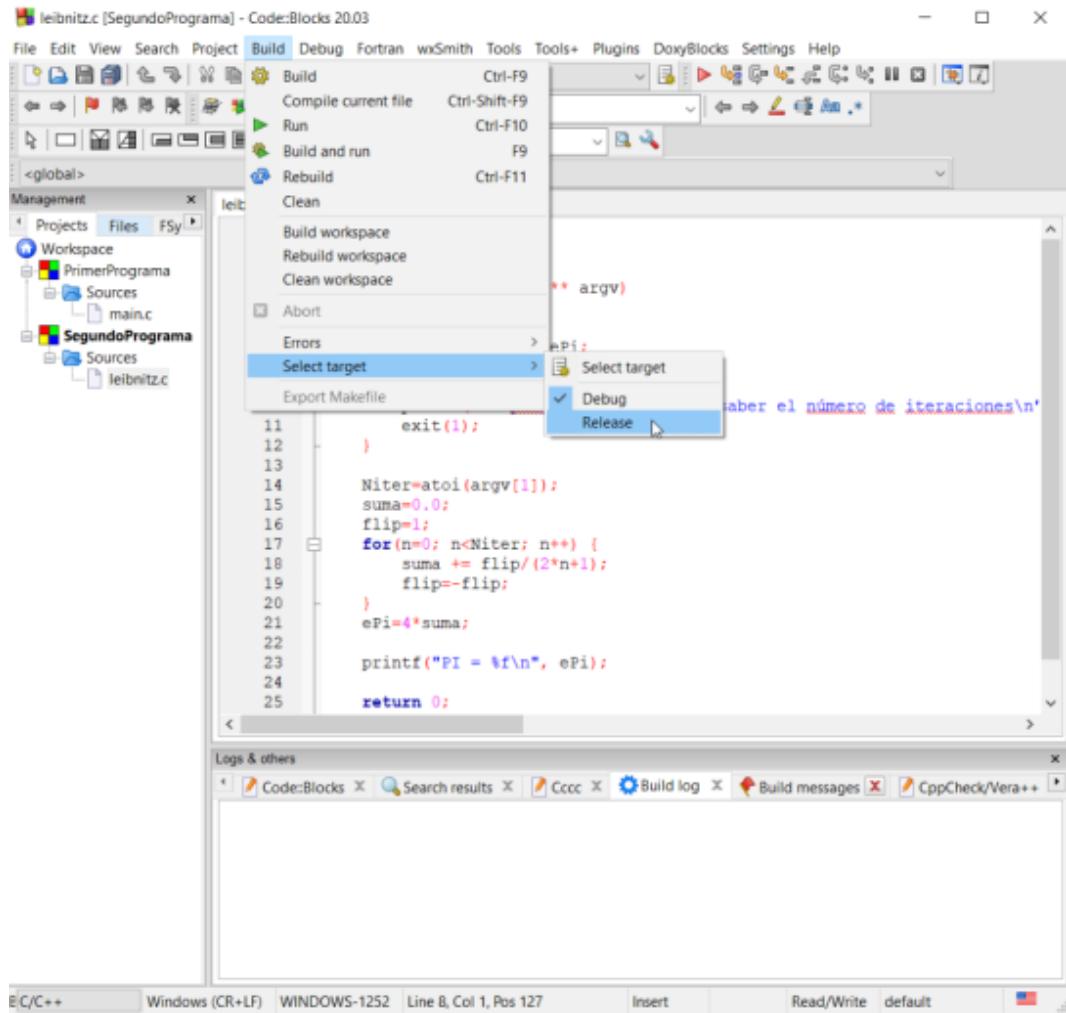
    return 0;
}

```

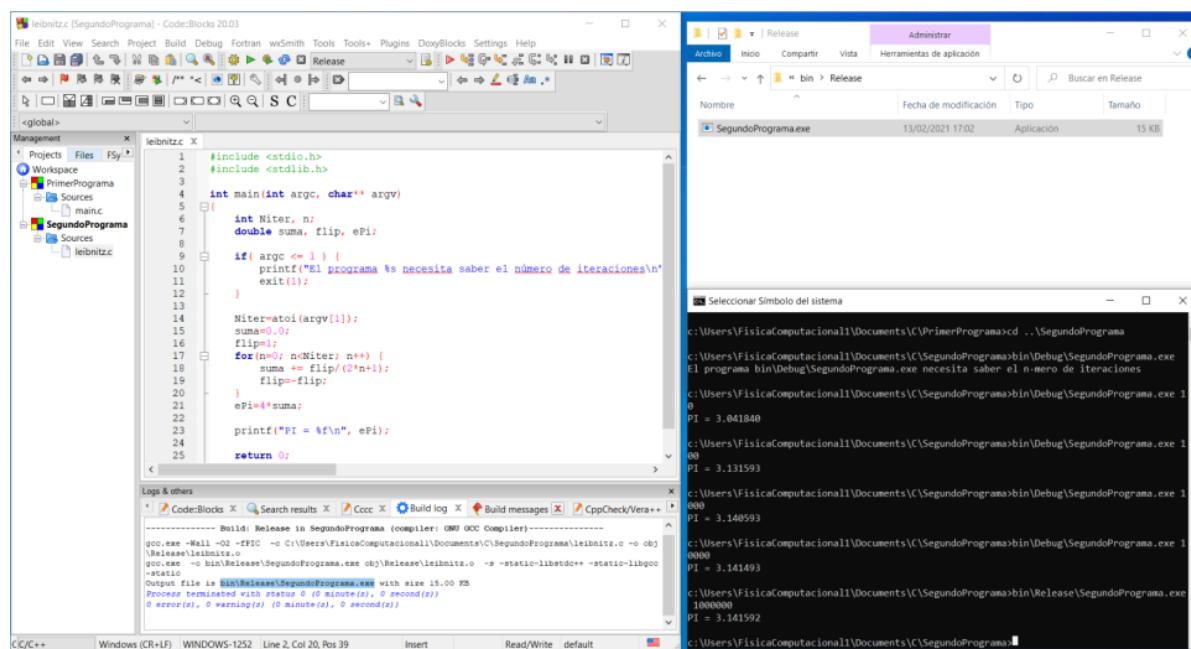
## Apéndice: Versiones “Debug” y “Release” del ejecutable

Como hemos visto hasta ahora, Code::Blocks ha creado el ejecutable de nuestros proyectos en un subdirectorio llamado “bin/Debug”. El nombre “bin” sugiere que son archivos “binarios”, que es como nos solemos referir a los que no son de texto plano, o sea, que no son archivos de código que se puedan abrir con un editor (no están pensados para que los lea un humano). El nombre “Debug” se refiere a que este código está generado para depurar (literalmente, “quitar los bichos”) el programa. El archivo ejecutable que contiene es un poco más grande y un poco menos eficiente de lo que podría ser; a cambio, incluye información para que, si se produce algún error en él durante su ejecución, Code::Blocks pueda pararlo y decirnos en qué instrucción de nuestro código se ha producido dicho error y, así, podamos corregirlo.

Podemos cambiar la versión del ejecutable que genera Code::Blocks seleccionando la otra opción: “Release”.



Ésta es la versión del ejecutable optimizada (tamaño y velocidad) que usaríamos para llevar a cabo cálculos masivos en grandes simulaciones y para copiar en otros ordenadores (p.ej. un cluster de ordenadores) que no necesitan tener copias del código fuente de nuestros cálculos.



## Tema 2c

# El lenguaje C mediante ejemplos

Este capítulo pretende repasar las construcciones más habituales del lenguaje C. No pretende ser un tutorial del mismo: para eso hay excelentes obras que realizan este recorrido paso a paso, como las citadas en el capítulo anterior.

A lo largo del capítulo se mostrarán muchos ejemplos y se propondrán ejercicios para que el lector trabaje los conceptos que se introducen. Al final del capítulo se dan las soluciones de todos estos ejercicios.

Un programa en C tiene una estructura estándar bien definida, requerida por el compilador: comienza con declaraciones de lo que se va a emplear (directivas “`#include`”), declaraciones propias particulares del programa, declaraciones y, por último, definiciones de las funciones del programa. En particular, todo programa C tiene una función, llamada “`main`” que es, como su nombre indica, la función principal del programa: la primera que es ejecutada. Más en lo particular, cada instrucción de C acaba en “;” si es una instrucción autónoma, o va seguida de un “bloque” de instrucciones abrazado por llaves “{” y “}”: así es como, por ejemplo, se indican las instrucciones que corresponden a una función.

### ¡Así no se debe escribir un programa en C!

Fuera de esta estructura general impuesta, un programa C debe ser legible por alguien más que el que lo escribe. El siguiente programa es correcto en su sintaxis y compila sin errores (salvo algún “Warning”):

```
long k=4e3,p,a[337],q,t=1e3;
main(j){for(;a[j=q=0]+=2,-k;)
for(p=1+2*k;j<337;q=a[j]*k+q%p*t,a[j++]=q/p)
k!=j>2?:printf(" %.3ld",a[j-2] %t+q/p/t);}
```

Es más, al ejecutarlo produce un resultado (¡sorprendente!). Sin embargo, cuesta unos cuantos minutos hacerse una idea de cómo lo hace: el ordenador lo va a entender, pero si tuviéramos que modificarlo, no sabríamos por dónde empezar.

### ¡Así se debería escribir un programa en C!

El programa anterior, sin pérdida de efectividad, pero más fácilmente legible y con una salida más adecuada se escribiría como se muestra en el listado 2c.1. La diferencia con el anterior es evidente: el sangrado de las líneas indica (además de las llaves) los bloques de

instrucciones que se ejecutan secuencialmente, los comentarios (entre “/\*” y “\*/”) facilitan la comprensión de qué se está haciendo, ...

**Listado 2c.1:** Versión bien documentada y formateada de un programa que calcula el número  $\pi$ .

```
1 #include <stdio.h>
2
3 /** Programa PI
4 * Calcula 1002 cifras decimales del número PI
5 * Algoritmo: pi3
6 * Referencia: (buscarla)
7 * Autor: (anónimo); Adaptación: DRP
8 * Fecha: 20/10/2010
9 * Observaciones: antes era así
10 * long k=4e3,p,a[337],q,t=1e3;
11 * main(j){for(;a[j=q=0]+=2,--k;)
12 *   for(p=1+2*k;j<337;q=a[j]*k+q%p*t,a[j++]=q/p)
13 *     k!=j>2?:printf("%.3d",a[j-2]%t+q/p/t);}
14 */
15 int main(int argc, char** argv) {
16     long k=4000; /* cuatro mil */          */
17     long t=1000; /* mil */                */
18     long a[337]; /* array de resultados parciales */
19     long p;      /* denominador de la serie modular */
20     long q;      /* numerador de la serie modular */
21     long d;      /* variable auxiliar */        */
22     int j;       /* contador */             */
23
24     /* PI=3. . . */
25     printf("PI=3.");
26
27     /* Bucle principal
28     * Nota: sólo imprime ternas de cifras en los
29     *         últimos dos pasos de cálculo
30     */
31     while(k>1)
32     {
33         k--; /* paso del bucle de impresión */
34         q=0; /* inicializar contador modular */
35
36         a[0]+=2; /* avanzar en a[0] por cada k */
37         p=1+2*k; /* inicializar el denominador
38                     de la serie modular */      */
39
40         /* bucle de cálculo de a[] */
41         for(j=0; j<337; j++)
42         {
43             /* condición de cálculo completado:
```

## 2C.1. ESTRUCTURA BÁSICA DE UN PROGRAMA: LA FUNCIÓN “MAIN”

2c-3

```

44         dos últimos pasos en k */
45     if( (j>2 && k==1) || k==0 )
46     {
47         /* todas las divisiones
48             son enteras! */
49         d=a[j-2] %t+(q/p)/t;
50
51         /* imprimir las siguientes
52             3 cifras */
53         printf("%.3ld",d);
54     }
55     /* actualizar el numerador de
56         la serie */
57     q=a[j]*k+q%p*t;
58     /* guardar el valor en a (se
59         utilizará dos j-pasos más
60         adelante) */
61     a[j]=q/p;
62 }
63
64 /* salto de línea final */
65 printf("\n");
66
67 /* Informar al S.O. de que no ha habido error
68 * al finalizar el programa
69 */
70 return 0;
71 }

```

---

En la realidad, nunca se hace ni como en el primer y oscuro ejemplo, ni como en éste tan bien documentado: parte de la explicación del programa y de su documentación se entiende que la proporciona el propio código C.

## 2c.1. Estructura básica de un programa: la función “main”

Aunque ya la hemos venido viendo, la estructura básica de un programa en C es la del listado 2c.2.

**Listado 2c.2:** El programa “Hola mundo” (versión estándar ANSI C).

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     printf("Hola mundo\n");
6
7     return 0;

```

8 }

La diferencia principal con el listado anterior es que aquí la función “main” recibe argumentos. El primero es un valor entero (tipo “int”); el segundo un vector de cadenas. Son dos variables usualmente llamadas así, “argc” y “argv”. Otra sintaxis alternativa es:

```
int main(int argc, char* argv[])
```

El tipo de variable “char\*” se refiere a una cadena, mientras que “[ ]” quiere decir que lo que recibe la función “main” es un vector (o “array”) de dicho tipo de variables. A cada uno de los valores en el vector se accederá indicando el nombre de éste, seguido de los corchetes y, entre éstos, el número de componente del vector. Así, por ejemplo, el nombre del ejecutable es recibido como el primer elemento del vector “argv”; nos referiremos a dicho valor como “argv[0]”.

**Ejercicio 3.1.** Modifíquese el programa anterior cambiando la llamada a “printf” por la siguiente:

```
printf("Hola\u mundo,\u me\u llamo\u %s\n", argv[0]);
```

Aprovéchese esta oportunidad para comprobar el significado de la cadena de formato (primer argumento) de la función “printf”.

**Ejercicio 3.2.** Compruébese que argc es un entero que contiene el número de argumentos que pasamos por línea de comandos cuando ejecutamos el programa (contando también el propio nombre del programa).

```
printf("El\u número\u de\u argumentos\u es\u %d\n", argc);
```

El primer argumento de línea de comandos se recibirá en el valor “argv[1]”. Este valor es el que escribimos a continuación del nombre del programa cuando lo llamemos. Por ejemplo, si hacemos

```
./holamundo Pepito
```

el valor de “argv[1]” será la cadena “Pepito”.

**Ejercicio 3.3.** Modifíquese el programa para que salude, en lugar de “al mundo entero”, al usuario que le indica su nombre como primer argumento en la línea de comandos (como en el ejemplo anterior). En este caso, “printf” será llamado como:

```
printf("Hola\u %s,\u me\u llamo\u %s\n", argv[1], argv[0]);
```

## 2c.2. Declaración de variables, asignación de valores, operaciones básicas e impresión de resultados

La declaración de variables, indicando su tipo y su nombre, es obligatoria en C. En el listado 2c.3 se declaran tres variables de tipo “float”, se asignan valores a dos de ellas y se calcula la tercera como suma de las dos primeras.

Nótese la sintaxis de los números (notación decimal y científica) y la de la asignación con el signo “=”.

### 2C.3. CONTROL DE FLUJO: “IF... ELSE”

2c-5

Además de la operación de la suma “+”, se pueden emplear otros operadores binarios (que actúan sobre el valor dado por la expresión anterior y posterior a él): “-”, “\*”, “/”<sup>1</sup>. Estos operadores binarios se aplican usando las “reglas de precedencia” habituales en álgebra. Por ejemplo, “ $a*b+c$ ” equivale a “ $(a*b)+c$ ”.

El operador “-” también actúa como operador “unario”. Así, “-a” significa (sorprendentemente), “el valor de la variable a con signo ( $\pm$ ) cambiado”<sup>2</sup>.

---

#### Listado 2c.3: El programa que suma dos números reales.

---

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     float a, b, c;
6
7     a=1.5;
8     b=-3.2e-2;
9     c=a+b;
10
11    printf(" %g+ %g=%g\n", a, b, c);
12
13    return 0;
14 }
```

---

**Ejercicio 3.4.** Búsquese información sobre la función “atof” que interpreta una cadena como un valor real (archivo H en el que está declarada, argumentos que recibe y su tipo). Utilícese para modificar el programa anterior en un programa que recibe por línea de comandos los dos números y muestra la suma y su resultado.

**Ejercicio 3.5.** Declárense las variables “a”, “b” y “c” como enteras y utilícese la función “atoi” para asignarle valores (búsquese el archivo H en el que está declarada, los argumentos que recibe y su tipo). Cámbiese la operación de suma (“+”) a división (“/”), y compruébese el comportamiento del programa.

## 2c.3. Control de flujo: “if... else”

Hasta ahora la ejecución de los programas ha sido lineal: una instrucción tras otra. El “flujo” de un programa consiste en el orden en que sus instrucciones van siendo ejecutadas y si lo son o no, en función de lo que indique el “usuario” de dicho programa. La modificación más sencilla al flujo consiste en decidir si se ejecuta una instrucción u otra según una condición lógica.

En el listado 2c.4 se calcula el valor absoluto de la diferencia entre dos números reales. Se utiliza la instrucción “if... else” para ejecutar un bloque de instrucciones u otro (bloques,

<sup>1</sup>A diferencia de otros lenguajes, en C no existe un operador de exponentiación; para ello se utilizará una función: “pow”.

<sup>2</sup>En C también se puede utilizar “+” como operador “unario”. Obviamente “+a” (valor de la variable a sin cambio de signo) indica lo mismo que “a”. Sin embargo, a veces, puede ser conveniente esta notación para enfatizar lo que se está escribiendo.

entre “{ . . . }”, con una única instrucción en ellos) de modo que se garantice el signo adecuado en el resultado.

La condición se construye con operadores de comparación: “<”, “>”, “<=”, “>=”, “!=”, “==”. Nótese la notación peculiar del C para el operador “*distinto de*” (“!=”) y para el operador “*igual a*” (“==”), que no se debe confundir con el operador “=” que sirve para asignar a la variable a su izquierda el valor de la expresión a su derecha. Los operadores de comparación tienen menos precedencia que los de suma o resta; esto no debe parecer evidente, ya que en C los resultados de operaciones lógicas se tratan como si fueran valores enteros. Es decir, se puede calcular algo así como “(4 || 6) + (7 || 8)”... otra cosa es el significado que tenga, pero en principio es legal.

Los resultados de las comparaciones se pueden combinar posteriormente con los operadores lógicos: “||” (*disyunción*, OR), “&&” (*conjunción*, AND), “^” (*disyunción exclusiva*, XOR). También se pueden “negar” con el operador unario “!”. Estos operadores son los últimos en ser evaluados.

---

**Listado 2c.4:** El programa que calcula la distancia entre dos números reales.

---

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     float a, b, c;
6
7     a=1.5;
8     b=3.2;
9
10    if( a<b ) {
11        c=b-a;
12    } else {
13        c=a-b;
14    }
15
16    printf(" | %g - %g | = %g \n", a, b, c);
17
18    return 0;
19 }
```

---

**Ejercicio 3.6.** Modificar el listado anterior usando la función “atof” para leer los valores de “a” y “b” desde línea de comandos. Eliminar también la alternativa “else” haciendo primero la operación “a-b”, comprobando el signo de “c” y, si no es adecuado, cambiándolo con el operador unario “-”.

## 2c.4. Bucle “for”

La mayor utilidad de un programa de ordenador es realizar una o varias operaciones tipo muchas veces, variando los valores de las variables involucradas. Una forma de hacer esto es usar bucles.

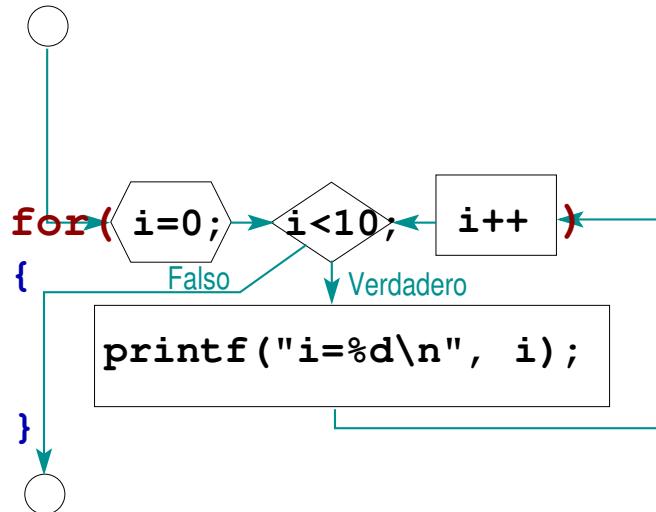


Figura 2c.1: Flujo de ejecución del bucle “for”.

El más completo de los bucles es el bucle “for”. El flujo de ejecución del bucle “for” es como se muestra en la figura 2c.1. Un ejemplo sencillo (cálculo de los seis primeros números pares), se muestra en el listado 2c.5.

---

**Listado 2c.5:** El programa que calcula los pares menores o iguales que 10.
 

---

```

1 #include <stdio.h>
2
3
4 int main(int argc, char** argv)
5 {
6
7     int n;
8
9     for(n=2; n<=10; n+=2)
10    {
11        printf("%d\n", n);
12    }
13
14    return 0;
15 }
```

---

Es habitual emplear en este tipo de bucles la sintaxis incremental, “ $n++$ ”, que indica que la variable “ $n$ ” (usualmente de tipo entero) sea incrementada en una unidad. Otra sintaxis habitual es la de incrementos no unitarios, de la forma “ $n+=2$ ”; en este caso, la variable “ $n$ ” es incrementada en dos unidades (como en el listado 2c.5). Éste es un caso particular de las “autoasignaciones” que tienen operadores de la forma “ $-=$ ”, “ $*=$ ”, “ $/=$ ”. Por ejemplo, un productorio se programaría a base de operaciones de la siguiente forma:

`productorio*=factor;`

**Ejercicio 3.7.** Calcular el factorial de un número entero indicado por línea de comandos. Empléese para guardar este factorial un “`long int`”, para tener posibilidad de valores

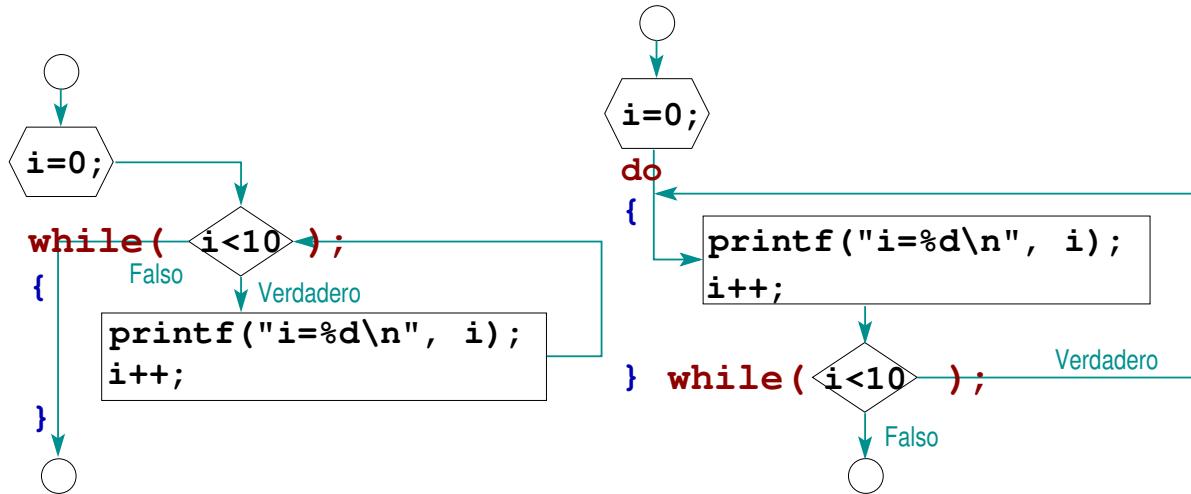


Figura 2c.2: Flujo de ejecución de los bucles “while” y “do...while”.

más grandes en el resultado<sup>3</sup> ¿Qué sucede para números enteros grandes (mayores que 17)?

Por supuesto, todo lo anterior no se ve limitado (como sucede en Fortran) a valores enteros de la variable del bucle. Para ello, pruébese el siguiente ejercicio.

**Ejercicio 3.8.** Calcúlense los puntos que dividen el intervalo  $[0; 1]$  en 100 subintervalos.

## 2c.5. Bucles “while” y “do...while”

El flujo de ejecución de los bucles “while” y “do...while” es como se muestra en la figura 2c.2. Estos bucles no añaden demasiado al bucle “for” ya visto.

**Ejercicio 3.9.** Rescribir el programa anterior para calcular los números pares menores que 10, usando un bucle “while” o “do...while”.

**Ejercicio 3.10.** Rescribir el programa anterior para calcular los puntos que dividen el intervalo  $[0; 1]$  en 100 subintervalos, usando un bucle “while” o “do...while”.

## 2c.6. Selección con “switch...case...default”

En muchas ocasiones habrá que comparar una variable entera con ciertos valores que ésta adquiera, y ejecutar unas instrucciones u otras en función del significado que se les dé a estos valores<sup>4</sup>.

Un pedazo de código que podría hacer esto se basaría en una serie de “if” encadenados:

<sup>3</sup>La efectividad de esta declaración “long int” dependerá de la “arquitectura” de nuestro procesador: 32 o 64 bits

<sup>4</sup>Una variable de este tipo se llamaría en estadística una variable “categórica”, ya que cada valor que toma pertenece a una categoría, de entre un número finito de ellas.

## 2C.6. SELECCIÓN CON “SWITCH... CASE... DEFAULT”

2c-9

```
if( x == 1 ) {
    ...
} else if( x==2 ) {
    ...
} else if( x==3 ) {
    ...
} else {
    ...
}
```

Sin embargo, es más elegante sustituir este código por una serie de “casos” en la siguiente forma equivalente al código anterior:

```
switch( x ) {
case 1:
    ...
    break;
case 2:
    ...
    break;
case 3:
    ...
    break;
default:
    ...
}
```

Este ejemplo, además de para explicar una *estructura de control* (de flujo) de C, sirve también para introducir la instrucción “break”. Esta instrucción obliga a salir de la estructura de control de flujo en la que se esté<sup>5</sup>. En “switch...case”, sale del ámbito (las “llaves”) del “switch”. Si se utilizara en un bucle “for”, saldría del bucle “for”. Si se utilizara en un bucle “while”, finalizaría en ese punto el bucle, etc.

Fuera de “switch...case”, el uso de “break” se debe evitar porque oscurece la lógica del programa. Sin embargo, muchas veces es la mejor manera de dar por terminado un bucle desde dentro...

El ejemplo de uso de “switch...case” más tradicional, es el uso en menús interactivos. Los menús interactivos más simples utilizan la función “printf” primero para mostrar distintas opciones de un menú, y después para pedir que se seleccione una opción. La opción seleccionada se lee en tiempo de ejecución usando la función “scanf”.

La función “scanf” se parece a “printf”: tiene una cadena de formato (qué datos se van a leer) y un argumento por cada dato que se leerá. La diferencia con “printf” es que, en lugar de indicar la variable “sel”, por ejemplo, se indica su dirección de memoria “&sel”. Como veremos más adelante, esta dirección de memoria se representa en C por un puntero (algo declarado, por ejemplo, como “int \*ptr”), pero en “scanf” casi siempre aparece con el operador de dirección “&” precediendo al nombre de la variable en la que se guardará el valor.

Como ejemplo, véase el listado 2c.6.

<sup>5</sup>Puede estar en todos los bucles, en “switch...case”, pero no en “if”.

**Listado 2c.6:** El programa que ofrece un menú y comenta la opción elegida (una y otra vez, hasta que se pide salir).

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     int sel;
6
7     printf("1. Opción A\n");
8     printf("2. Opción B\n");
9     printf("3. Opción C\n");
10    printf("0. Salir\n");
11
12    do {
13        printf("Opción: ");
14        scanf("%d", &sel);
15
16        switch(sel)
17        {
18            case 0:
19                break;
20            case 1:
21                printf("Buena opción la 1\n");
22                break;
23            case 2:
24                printf("No me parece tan buena la"
25                      " 2\n");
26                break;
27            case 3:
28                printf("La 3, definitivamente,"
29                      " no me gusta\n");
30                break;
31            default:
32                printf("Esta opción no se ha"
33                      " ofrecido...\n"
34                      "Pruebe de nuevo\n");
35        }
36
37    } while(sel!=0);
38
39    return 0;
40 }
```

## 2c.7. Vectores y matrices

Ya hemos visto un vector en C: los argumentos de línea de comandos, que se declaran como “`char* argv[]`”, esto es, un vector de punteros a caracteres. Los vectores numéricos son mucho más familiares y útiles.

Un vector se declara en un programa indicando entre los corchetes que siguen al nombre que le damos el número de elementos que contiene dicho vector (la dimensión del espacio al que pertenece, que diríamos en terminología matemática). Así, un vector de 7 números enteros, se declarará como

```
int vector[7];
```

y a su *quinto* elemento se accederá como en

```
x=vector[4];
```

porque en C se empieza a contar por el 0 (el quinto elemento irá precedido de los 0, 1, 2 y 3).

En el listado 2c.7 se muestra un ejemplo sencillo de uso de vectores de números reales.

---

**Listado 2c.7:** El programa que suma dos vectores de números reales.

---

```

1 #include <stdio.h>
2
3
4 int main(int argc, char** argv)
5 {
6     int i;
7     float x[]={1.0, 2.0, 3.0};
8     float y[3];
9     float z[3];
10
11    y[0]=3.0;
12    y[1]=2.0;
13    y[2]=1.0;
14
15    for(i=0; i<3; i++)
16    {
17        z[i]=x[i]+y[i];
18    }
19
20    printf("La suma de vectores da: ");
21
22    printf("[%g", z[0]);
23    for(i=1; i<3; i++)
24        printf(", %g", z[i]);
25    printf("]\n");
26
27    return 0;
28 }
```

---

Igual que se declaran vectores, se declaran matrices. Por ejemplo, una matriz de  $2 \times 2$  números reales (con doble precisión, tipo “`double`”), se declarará como

```
double matriz[2][2];
y a sus elementos se accederá por fila y columna, como en
int i, j;
double M_ij;
...
M_ij=matriz[i][j];
```

**Ejercicio 3.11.** Calcular el producto de dos matrices de números reales  $3 \times 3$ . Una de ellas codificada en el programa, y la otra pedida por consola, elemento a elemento, usando “scanf”. Mostrar el resultado debidamente formateado (en una matriz).

## 2c.8. Punteros

En este capítulo se han utilizado dos operadores que precedían a nombres de variables y que no han sido debidamente explicados, “\*” y “&”. Hemos visto al declarar los argumentos de la función “main” que “char\*\* argv” representa un vector de cadenas<sup>6</sup>, y que también puede expresarse como “char\* argv[]” o<sup>7</sup> “char \*argv[]”, que leído “literalmente” representa un vector de punteros a caracteres. Las tres opciones declaran lo mismo y después de esta sección esperamos que se entienda el porqué. También se ha utilizado el operador de dirección “&” precediendo al nombre de una variable, de modo que “&var” indica la dirección de memoria que ocupa la variable “var”, es decir, donde está. En esta sección vamos a ver que estos dos operadores están relacionados con una de las herramientas más interesantes del lenguaje C, los punteros.

En C la dirección de memoria de una variable es, en sí, un tipo de dato. Un puntero es una variable del lenguaje C que puede contener una de esas direcciones de memoria. Por esta razón se denominan punteros, porque “apuntan” o “referencian” a otras variables. Los punteros se declaran de acuerdo con el tipo de dato al que apuntan, de modo general tenemos que su declaración es “tipo \*ptr”, donde “tipo” es el tipo de la variable cuya dirección se guardará en “ptr”. Por ejemplo, si declaramos el puntero “int \*q”, esto quiere decir que el puntero “q” apunta a una dirección de memoria en la que se guarda una variable entera. El valor de la variable se obtiene como “\*q”. Dos formas alternativas de leer la declaración anterior son: 1) lo que hay en “q” (esto es “\*q”) es una variable de tipo “int”; 2) “q” es el puntero a un valor “int”.

Los operadores de indirección “\*” y de dirección “&” son operadores inversos. Esto quiere decir que si tenemos una variable “var”, se cumple que “\*(&var)==var”, que es como decir que lo que hay en la dirección de memoria ocupada por “var” es el valor de la propia “var”. Por otro lado, si tenemos un puntero “q”, se cumple que “&(\*q)=q”, que viene a decir que la posición en memoria de la variable a la que apunta el puntero “q” es la propia posición indicada por “q”. En el ejemplo del listado 2c.8 se muestra cómo se puede utilizar un puntero para modificar el valor de una variable a través de su dirección de memoria.

---

**Listado 2c.8:** Uso de un puntero para cambiar el valor de una variable a través de su dirección de memoria.

<sup>6</sup>Recordamos que una cadena (del inglés *string*) es un vector de caracteres y que se define como “char\* cadena”.

<sup>7</sup>Recuérdese que en C los espacios entre operadores y operandos pueden ser ignorados.

```

1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     double a=43.4;
6     double *q;
7     q=&a; //asignamos al puntero q la dirección de memoria de la variable a
8
9     printf("El puntero q apunta al valor %lg\n",*q); //resultado el valor de a
10
11    *q=0.3; //modificamos el valor de la variable a la que apunta el puntero q
12
13    printf("El puntero q apunta al valor %lg\n",*q); //resultado en nuevo valor
14    printf("El nuevo valor de a es %lg\n",a); //como la variable "a" está en la
15        misma dirección de memoria
16
17    return 0;
}

```

---

## Punteros y vectores

Supongamos que un puntero “q” apunta a un entero; por tanto ha sido declarado como “int \*q”. El valor “q+10” mueve la dirección de la memoria cuatro elementos hacia la derecha, es decir, si un entero ocupa 4 bytes, el nuevo puntero “q+10” apunta a una posición de la memoria 40 bytes más allá del punto inicialmente apuntado por “q”. Es importante darse cuenta de que los valores de las variables a las que apuntan los punteros se pueden expresar como vectores, así podemos escribir indistintamente “\*(q+10)” o “q[10]”.

Esta característica hace que los punteros estén muy relacionados con los vectores y las matrices, ya que en estos últimos, los valores se almacenan consecutivamente en memoria, de modo que, sabiendo la posición en la memoria que ocupa la primera entrada, podemos desplazarnos a lo largo de toda la matriz o el vector. De hecho, si declaramos un vector como “int v[5]”, tenemos que, en realidad, “v” es un puntero que apunta a “v[0]”, es decir, “v==&v[0]” y “\*v==v[0]”. Por lo tanto, “v+1” contendrá la dirección de memoria en la que está almacenado “v[1]” y apuntará a él: “v+1=&v[1]” y “\*(v+1)=v[1]”, y así sucesivamente: “v+n=&v[n]” y “\*(v+n)=v[n]”. Nótese que la segunda expresión se obtiene de aplicar el operador “\*” en ambos lados de la primera expresión:

$$*(v+n) \leftrightarrow *(&v[n]) \leftrightarrow v[n]$$

Queda ya claro que podemos trabajar con vectores o punteros indistintamente. En el siguiente listado se muestra este hecho asignando valores a los elementos de un vector, previamente declarado, mediante un puntero.

---

### Listado 2c.9: Uso de punteros para asignar valores a los elementos de un vector.

```

1 #include <stdio.h>
2
3

```

```

4 int main(int argc, char** argv)
5 {
6     int v[6];
7     int *q;
8
9     q=v;      //asignamos al puntero q la dirección de memoria del primer elemento
10    del vector
11   /* también podríamos haber hecho q=&v[0] */
12
13   *v=0;      //v[0]=0
14   v[1]=1;    //v[1]=1
15   q[2]=2;    //v[2]=2
16   *(q+3)=3; //v[3]=3
17   *(v+4)=4; //v[4]=4
18   *(&v[5])=5; //v[5]=5
19
20   printf("El vector es { %d, %d, %d, %d, %d, %d}\n", v[0], v[1], v[2], v[3], v
21   [4], v[5]);
22 }
```

---

## Punteros y matrices

Una matriz no es más que un vector de vectores. Por lo tanto, si un vector es un puntero, una matriz será un puntero a puntero. Vamos a ver qué significa esto.

Una matriz declarada como “int M[i][j]” se almacena en memoria por filas consecutivas: M[0][0], M[0][1], M[0][2], …, M[0][j-1], M[1][0], M[1][1], …, M[i-1,j-1]. En el caso de las matrices, el nombre de la matriz “M” es un puntero a la dirección de memoria de la primera entrada de la matriz, esto quiere decir que su valor es la dirección de memoria en la que está almacenada la dirección de memoria de la primera entrada de la matriz, por eso se dice que es un puntero a puntero:

$$*M = \&M[0][0] \quad **M = M[0][0].$$

En realidad, “M” es un puntero al primer elemento de un vector de punteros llamado “M[]”, cuyos elementos (que son punteros) contienen las direcciones de memoria del primer elemento de cada fila de la matriz. Esto quiere decir que

$$M == \&M[0] \quad M[0] == \&M[0][0],$$

o lo que es lo mismo (aplicando el operador “\*” en ambos lados de las igualdades)

$$*M == M[0] \quad *M[0] == M[0][0],$$

que dicho verbalmente significa que lo que hay en la dirección “M” es “M[0]”, que apunta a la primera fila de la matriz, y que es, a su vez, la dirección de memoria en la que se encuentra almacenado el elemento de su primera columna.

## 2C.8. PUNTEROS

2c-15

Por consiguiente, al igual que ocurría con los vectores, podremos desplazarnos a lo largo de matriz incrementando adecuadamente el valor de “M”.

Por ejemplo, si incrementamos el puntero “M” en una unidad, “M+1”, nos iremos a la dirección de memoria en la que está contenido el segundo elemento del vector “M[]”, “M[1]”, el cual contiene a su vez la dirección de memoria en la que está almacenado el primer elemento de la segunda fila de la matriz. En general podemos escribir:

$$M+n == \&M[n] \quad M[n] == \&M[n][0],$$

o

$$*(M+n) = M[n] = \&M[n][0],$$

o

$$**(M+n) = *M[n] = M[n][0].$$

Por lo tanto, al elemento “M[i] [j]” de la matriz podremos acceder de muchas formas, aquí tenemos algunos ejemplos:

$$M[i][j] = *(*(M+i)+j)$$

$$M[i][j] = *(M+i)[j]$$

$$M[i][j] = *(M[i]+j)$$

Es interesante darse cuenta de que, si aplicamos la regla general “\*(q+i)=q[i]” en la primera de las expresiones anteriores, obtenemos las dos siguientes. En el listado siguiente se muestran diferentes modos de asignar valores a los elementos de una matriz.

---

**Listado 2c.10:** Uso de la definición de puntero para asignar valores a los elementos de una matriz.

---

```

1 #include <stdio.h>
2
3
4 int main(int argc, char** argv)
5 {
6     int i,j,*q;
7     int M[3][3];
8
9
10    //diferentes formas de definir los elementos de la matriz M mediante punteros
11    //y vectores
12
13    q=&M[1][0];
14
15    **M=1;      //M[0][0]=1
16    *(*M+1)=2; //M[0][1]=2
17    (*M)[2]=3; //M[0][2]=3
18    **(M+1)=4; //M[1][0]=4

```

```

18  *(M[1]+1)=5; //M[1][1]=5
19  *(*(M+1)+2)=6; //M[1][2]=6
20  *(M+2)[0]=7; //M[2][0]=7
21  *(q+4)=8; //M[2][1]=8
22  q[5]=9; //M[2][2]=9
23
24
25  for(i=0; i<3; i++)
26  {
27      for(j=0; j<3; j++)
28      {
29          printf(" %d", M[i][j]);
30      }
31      printf("\n");
32  }
33
34  return 0;
35 }
```

---

Todo lo que acabamos de ver también se aplica, obviamente, a las cadenas de caracteres o *strings*. Una cadena de caracteres es un vector de tipo “char”. Consideremos por ejemplo la siguiente declaración:

```
char cadena[]="hola mundo";
```

Según lo que acabamos de ver, “cadena” es un puntero al primer elemento de la cadena, de modo que por ejemplo `*cadena+3=='a'`. Para imprimir en pantalla la cadena sólo tendremos que escribir

```
printf ("El contenido de la cadena es:%s\n", cadena);
```

Ahora podemos definir un puntero a carácter “char \*a” y realizar la asignación “a=cadena” u otra asignación como por ejemplo

```
a="adiós mundo";
```

También se puede declarar una cadena utilizando el tipo de variable “char\*” del siguiente modo:

```
char* cadena="hola mundo";
```

Ahora podemos entender el significado del segundo argumento de la función main, declarado como “char \*\*argv”. En este caso “argv” es un puntero a puntero de caracteres, lo que nos permite definir una matriz de caracteres, o lo que es lo mismo, un vector de vectores de caracteres (esto es, un vector de cadenas). Por eso podemos utilizar la sintaxis equivalente “char \*argv []”, que significa vector de punteros a carácter. En este caso “argv[n]” contendrá la dirección de memoria del primer carácter del argumento “n” pasado por línea de comandos (le apuntará). Otra forma de verlo es escribir “char\* argv []”. Como el tipo de variable “char\*” se refiere a una cadena, de este modo también estamos declarando un vector de cadenas.

## 2c.9. Lectura y escritura de datos

En lo anterior toda la entrada de datos al programa venía dada, o bien, desde el sistema operativo (encargado de cargar los contenidos de la variable “`*argv[]`” de la función “`main`”), o bien, leyéndolos interactivamente con la función “`scanf`”. Por otro lado, toda la salida de datos se ha realizado con “`printf`”, que construía cadenas y las mostraba en el terminal.

Aunque no lo sepamos, hemos estado haciendo lectura y escritura de “corrientes de datos”, que procederán de los archivos de datos (pero también podrán proceder de los “sockets” que conectan un programa a un dispositivo de adquisición de laboratorio, u otro programa a un servicio en otro ordenador o “sitio” de Internet).

Antes de abrir el primer archivo de datos, una prueba.

**Ejercicio.** La instrucción “`echo`” (en Unix) hace una cosa muy simple: escribe en el terminal todos los argumentos que se le pasan por línea de comandos (¿sería capaz de escribir un programa semejante con lo que sabe de C? si está usando Windows, debería hacerlo). Lo que escribe en el terminal, se puede dirigir a otro programa, usando el indicador “`|`” (“tubería”), como si éste lo estuviera leyendo desde la consola. Pruebe lo siguiente con el programa resultante de compilar [2c.6](#), al que llamaremos “`scanmenu`”:

```
echo 0 | ./scanmenu
```

Explique qué ha sucedido.

Un archivo en C se abre con la función “`fopen`” y retorna un puntero a un dato de tipo “FILE” (véase la documentación sobre esta función). Este dato contiene mucha información sobre el archivo: su nombre, su estado y posición de lectura/escritura, etc. Es algo habitual que, cuando lo que se necesita conocer es un conjunto de informaciones a las que el programador no va a acceder directamente, lo que se devuelva es un puntero con la dirección de memoria en la que empieza toda esa información (la información en sí puede variar, por ejemplo, de un sistema operativo a otro, o incluso entre compiladores, en el caso de Windows).

Como ya se ha dicho, la lectura o escritura de datos que se ha hecho hasta ahora era un caso particular de lectura y escritura de corrientes como la que describen los “FILE”. En particular, se ha estado leyendo de “`stdin`” y escribiendo en “`stdout`”, que figuran declarados en “`stdio.h`” como

```
extern FILE* stdin;
extern FILE* stdout;
```

En “`printf`” se sobreentiende que el destino es `stdout`. Para hacerlo explícito, o para escribir en un archivo abierto con “`fopen`”, utilizaríamos la función “`fprintf`”. Igualmente, para leer explícitamente de la entrada estándar “`stdin`”, utilizaríamos la función “`fscanf`”. Por si no es evidente, la “`f`” inicial en estas funciones se refiere a “FILE”. Como ejemplo:

```
float x;
...
fscanf(stdin, "%f", &x);
fprintf(stdout, "%g", &x);
```

Un ejemplo con más contenido, se muestra en el siguiente listado [2c.11](#). Este programa lee desde consola un número arbitrario de pares de números ( $X, Y$ ), y los guarda en un archivo que se podrá abrir y representar con “Gnuplot”.

**Listado 2c.11:** El programa que guarda pares de coordenadas en un archivo de texto (que puede interpretar “Gnuplot”).

```

1 #include <stdio.h>
2
3
4 int main(int argc, char** argv)
5 {
6     int npts, n;
7     double x, y;
8     FILE *fout;
9
10    fout=fopen("datos.dat", "w");
11
12    printf("Número de puntos: ");
13    scanf("%d", &npts);
14
15    for(n=0; n<npts; n++) {
16        printf("X_ %d= ", n);
17        scanf("%lf", &x);
18        printf("Y_ %d= ", n);
19        scanf("%lf", &y);
20
21        fprintf(fout, "%g\t%g\n", x,y);
22    }
23
24    fclose(fout);
25
26    return 0;
27 }
```

**Ejercicio 3.12.** Escriba un programa que lea desde un archivo de texto  $N$  pares de coordenadas ( $X, Y$ ) y calcule la recta de regresión correspondiente,  $Y = m_y X + y_0$ , así como el coeficiente de correlación,  $r$ , de dicha recta. Para ello, empléense las fórmulas:

$$m_y = \frac{N \sum x_n y_n - (\sum x_n)(\sum y_n)}{N \sum x_n^2 - (\sum x_n)^2}$$

$$m_x = \frac{N \sum x_n y_n - (\sum x_n)(\sum y_n)}{N \sum y_n^2 - (\sum y_n)^2}$$

$$y_0 = \frac{1}{N} \left( \sum y_n - m_y \sum x_n \right)$$

$$r^2 = m_x m_y$$

## 2c.10. Definición de funciones: paso de argumentos por valor y por referencia

En muchas ocasiones, deberemos calcular una función que no es posible expresar en forma de una expresión matemática cerrada. En ese caso nuestros cálculos involucrarán bucles que, como en el ejercicio anterior de regresión, proporcionarán finalmente uno o más valores.

Para calcular estas funciones, arbitrariamente complicadas, tendremos que declararlas y definirlas. Como se explicó en el primer tema, en C, cuando se llega a un punto en que se emplea una función, el compilador tiene que saber de qué forma es dicha función: debe haber sido declarada. Una función está declarada si se indica su tipo (el del valor que retorna) y los tipos de sus argumentos. Si a continuación de esto vienen unas llaves con la definición de la función (las instrucciones que debe seguir el ordenador para acabar dando el resultado), bien. Si no, también. En el proceso de enlazado, será cuando se busque la función por su nombre; como ya se explicó el “linker” no funciona dentro de la cadena continua de procesado en que se hallan el preprocesador y el compilador de C.

En el listado 2c.12 se muestra un ejemplo de declaraciones de funciones que son llamadas desde funciones.

Una de ellas es “`sqr`” que se declara como “`static inline`”. De los dos modificadores, sólo explicaremos el segundo; una función es `inline` cuando es tan pequeña que el compilador la puede “sustituir” a la hora de generar el objeto. Esto hace que se salte el paso de *llamar a la función*, realizando sólo las operaciones que contiene ésta, ahorrando así tiempo de ejecución. Esta es una optimización del código máquina sobre la que podemos decidir.

La otra función hace un cálculo matemático sencillo: calcula el valor de una función llamada lorenciana dados sus parámetros ( $x_0, y_0, w$ ) y su argumento ( $x$ ). Con la instrucción “`return y`” se indica cuál es el valor que devuelve la función a partir de sus argumentos.<sup>8</sup>

La función “`main`” se encarga de abrir un archivo para escritura, pedir un intervalo de valores y los parámetros, y guardar los valores de la función en un archivo que puede representar “Gnuplot”.

---

**Listado 2c.12:** El programa calcula la función lorenciana en un intervalo y guarda los puntos en un archivo para representarlos con “Gnuplot”.

---

```

1 #include <stdio.h>
2
3 static inline
4 double sqr(double x)
5 {
6     return x*x;
7 }
8
9 double lorenciana(double x, double x0,
10                     double y0, double w)
11 {
```

---

<sup>8</sup>La función “`main`” retorna también un valor. Este valor indica al sistema operativo cuál es el error que ha causado la finalización del programa. Retornar un valor de 0, equivale a decir que no ha habido ningún error. Cualquier otro valor indica un error; cuál depende del gusto del programador, no hay ningún estándar. Por lo general, el intérprete de comandos del sistema operativo (“`bash`” en Linux, “`cmd.exe`” en Windows) no indicará nada acerca de este valor retornado por “`main`”, pero otro programa sí podría hacerlo.

```
12     double y, den;
13
14     den=1+sqr(x-x0)/(w*w);
15
16     y=y0/den;
17
18     return y;
19 }
20
21
22 int main(int argc, char** argv)
23 {
24     double x, y, xa, xb, dx;
25     double x0, y0, w;
26     int npts;
27     FILE *fout;
28
29     fout=fopen("funcion.dat", "w");
30
31     printf("Intervalo [a, b]: ");
32     printf(" a = ");
33     scanf("%lf", &xa);
34     printf(" b = ");
35     scanf("%lf", &xb);
36     printf("Número de puntos en el intervalo: ");
37     printf(" N = ");
38     scanf("%d", &npts);
39
40     printf("Parámetros de la lorenziana:"
41           " Y(X) = Y0/(1+(X-X0)^2/W^2)\n");
42     printf(" Y0 = ");
43     scanf("%lf", &y0);
44     printf(" X0 = ");
45     scanf("%lf", &x0);
46     printf(" W = ");
47     scanf("%lf", &w);
48
49     dx=(xb-xa)/npts;
50
51     for(x=xa; x<=xb; x+=dx)
52     {
53         y=lorenciana(x, x0, y0, w);
54
55         fprintf(fout, "%g\t%g\n", x, y);
56     }
57
58     fclose(fout);
59
```

```

60         return 0;
61 }
```

---

Las funciones habituales en matemáticas (y en C) devuelven un único valor en respuesta a una serie de argumentos (y parámetros). Sin embargo, en C es posible que uno de estos argumentos sea una dirección de memoria, de una variable, en la que queremos que se “deposité” una información. Este es un truco muy usado en C para devolver muchos valores desde una única función.

Se dice que los argumentos de la función que son direcciones de memoria de variables son pasados *por referencia* (el puntero “apunta” o “referencia” a las variables) y que las otras, las que hemos usado hasta ahora, se pasan *por valor*. Ya hemos visto un ejemplo de variables pasadas por referencia en el caso de las funciones “scanf” y “fscanf”: los argumentos “&variable” son pasos por referencia.

Dentro de la función, a los valores de estas variables pasadas por referencia se accede precediendo el nombre de la variable por un “\*”. Es fácil entender esto: hemos visto que los punteros (las referencias) se declaraban como “int \*ptr”. Esto quiere decir que “\*ptr” es una variable de tipo “int”, cuando “ptr” es el puntero a un valor “int”.

En el listado 2c.13 mostramos un ejemplo de cómo podemos pasar variables, vectores y matrices, por referencia a una función. El programa calcula el producto de una matriz y un vector definidos en la función “main”. Para ello los pasamos por referencia a la función “producto”, que además calcula el módulo del vector resultante. Como todas estas variables son pasadas por referencia, la función “producto” no necesita devolver ningún valor (por eso se utiliza el tipo “void” y por eso al “return” de la función no se le asigna ningún valor). Obsérvese que el paso por referencia de las matrices exige la declaración de la dimensión de los vectores que las componen. Esto es lógico pues el puntero “M[i]” apunta al primer elemento de la fila “i”, pero no “sabe” cuántos elementos hay en ella. Comprobar que la función “producto” podría haber sido declarada también del siguiente modo sin que el resto del programa mude:

```
void producto (double (*M)[3], double *V, double *MxV, double *modulo)
```

En este caso “(\*M) [3]” representa un puntero a un vector de tamaño 3, esto es, un puntero a puntero, y no debe ser confundido con “\*M [3]”, que es un vector de punteros de tamaño 3. El operador subíndice o componente de un vector “[ ]” es el operador con mayor precedencia; se podría decir que se añade al nombre de la variable y se tratan variable y operador como un todo.

---

**Listado 2c.13:** Programa que calcula el producto de una matriz por un vector definidos pasándolos por referencia a una función que realiza todas las operaciones.

---

```

1 #include <stdio.h>
2
3
4 void producto(double M[] [3], double V[], double MxV[], double *modulo)
5 {
6     int i, j;
7
8     //calculamos el producto de la matriz y el vector
9     for (i=0; i<2; i++)
10    {
```

```

11         MxV[i]=0;
12         for (j=0; j<3; j++)
13             MxV[i]+=M[i][j]*V[j];
14     }
15
16 //calculamos el módulo del vector resultante
17 for (i=0; i<2; i++)
18     *modulo+=MxV[i]*MxV[i];
19
20 *modulo=sqrt(*modulo);
21
22 return;
23 }
24
25
26 int main(int argc, char** argv)
27 {
28     double M[2][3]={{1.,1.,1.},{1.,1.,1.}};
29     double V[3]={1.,2.,3.};
30     double MxV[2], modulo;
31
32 producto(M,V,MxV,&modulo);
33
34 printf("El producto de la matriz M por el vector V es (%g, %g)\n", MxV[0],
35        MxV[1]);
36 printf("El modulo del vector es %f\n", modulo);
37
38 return 0;
}

```

---

**Ejercicio 3.13.** Modificar el programa del listado 2c.13 para que la matriz sea pasada por referencia en forma de puntero simple, es decir, que la función “producto” esté declarada del siguiente modo:

```
void producto (double *p, double *V, double *MxV, double *modulo)
```

**Ejercicio 3.14.** Modificar el programa de cálculo de la recta de regresión definiendo (antes que “main”) la función

```

double regresionLineal(int npts,
                      double x[], double y[],
                      double *_m, double *_y0)
{
    ...
    *_m=m_y;
    *_y0=y0;
    return r2;
}

```

a la que se pasan dos vectores “`x []`” e “`y []`” con las coordenadas de los “`npts`” puntos y las variables “`*_m`” y “`*_y0`” por referencia. La función calculará como lo hacía el programa anterior los valores de la pendiente  $m_y$ , la ordenada en el origen  $y_0$  y el cuadrado del coeficiente de regresión  $r^2$ , y los retornará como se indica en el “esqueleto” anterior.

Vamos a concluir esta sección viendo que también podemos aplicar las ventajas de los punteros a las funciones. Al igual que ocurría con los vectores y las matrices, los nombres de las funciones son también punteros. Esto nos permite pasar por referencia funciones a otras funciones, es decir, pasar como argumento a una función el nombre de otra función. Para ello se utilizan declaraciones como:

```
int (*func)(double a, int b, ...)
```

que representa un puntero (“`func`”) a una función que devuelve un entero y tiene como argumentos “`a`”, “`b`”, ... Esta declaración no debe ser confundida con

```
int *func(double, int, ...)
```

que declara una función “`func`” que devuelve un puntero a entero y tiene como argumentos “`a`”, “`b`”, ...

En el listado 2c.14 se muestra un programa que toma dos números reales introducidos por línea de comandos y realiza entre ellos la operación (“multiplicar” o “dividir”) indicada en el tercer argumento introducido por línea de comandos. Como se puede comprobar, para comparar dos cadenas “`s1`” y “`s2`” se utiliza la función “`strcmp(s1,s2)`”, declarada en la librería “`string.h`”, que devuelve un entero: 0 si las dos cadenas son iguales y 1 si no lo son (por eso es necesario introducir el operador “`!`”). La operación deseada es pasada por referencia desde “`main`” a la función “`oper`”.

---

**Listado 2c.14:** Programa que toma dos números por línea de comandos y ejecuta la operación indicada por el usuario (“multiplicar” o “dividir”).

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5
6 double division(double a, double b)
7 {
8     return (a/b);
9 }
10
11 double multiplicacion(double a, double b)
12 {
13     return (a*b);
14 }
15
16 double oper(double a, double b, double (*operacion) (double, double))
17 {
18     return operacion(a,b);
19 }
```

```

20
21
22 int main(int argc, char** argv)
23 {
24     double a,b;
25     char *c;
26
27
28     a=atof(argv[1]);
29     b=atof(argv[2]);
30     c=argv[3];
31
32
33     if(!strcmp(c,"multiplicar"))
34         printf("el resultado de %s %g por %g es %g", c, a, b, oper(a,b,&
35             multiplicacion));
36     else
37         if(!strcmp(c,"dividir"))
38             printf("el resultado de %s %g por %g es %g", c, a, b, oper(a,b,&division)
39             );
40     else
41         printf("La operación no ha sido debidamente introducida");
42 }
```

---

**Ejercicio 3.15.** Modificar el programa del listado 2c.14 para que haga lo mismo, pero declarando la función “oper” dentro de “main” como

```
double (*oper)(double, double);
```

para luego asignarle la función correspondiente a la operación introducida por línea de comandos:

```
oper=multiplicacion;
```

si el usuario tecleó la palabra “multiplicar”.

## 2c.11. Utilidad: guardar datos como una imagen

Alguna vez querremos prescindir del “Gnuplot” para representar datos. Este programa es muy bueno, y puede hacer representaciones gráficas de toda clase, pero nosotros también podemos.

Ahora que hemos revisado las operaciones básicas en lenguaje C, vamos a sintetizar todo lo visto escribiendo un programa que guarda en un fichero una imagen en blanco y negro.

## 2C.11. UTILIDAD: GUARDAR DATOS COMO UNA IMAGEN

2c-25

El brillo de la imagen vendrá dado por una expresión matemática y puede ser un medio de representar funciones en forma de imágenes.

Una imagen digital es un conjunto de elementos de matriz llamados “píxeles” (*picture elements*, elementos de la imagen). En C una imagen es una matriz bidimensional. Si la imagen es en blanco y negro (en grises, para ser más exactos), los elementos de esa matriz bidimensional serán números que indican lo cerca que el píxel se encuentra del blanco o lo lejos que se encuentra del negro; es decir, guardarán números proporcionales al brillo de la imagen en ese píxel. En una imagen en color, cada píxel contendrá la información de brillo de cada uno de los colores (de luz) primarios: rojo, verde y azul. En esta sección sólo trataremos de imágenes en escala de grises, aunque es muy fácil pasar a imágenes RGB (de Red, Green, Blue, nombres de los colores primarios en Inglés).

Para que un programa de procesamiento de imágenes como el “Gimp”<sup>9</sup> sepa cómo leer una imagen, es necesario proporcionarle tres datos: el número de filas y de columnas de la matriz, y cuánto vale el *nivel de gris* en un píxel totalmente blanco (los píxeles negros se asume siempre que tienen valor de brillo cero, por razones obvias).

El formato de imagen que vamos a usar, el PGM (Portable Gray Map, mapa de grises portable), proporciona, precisamente esos tres datos: el ancho y alto de la imagen, y el valor del blanco en ella; a continuación, sigue toda la ristra de valores enteros que indican los niveles de gris de los píxeles, en el orden de lectura habitual de los elementos de una matriz (de arriba a abajo, de izquierda a derecha, o lo que es lo mismo, en orden creciente de filas, recorriendo todas las columnas en cada fila).

Todos los archivos que contienen imágenes comienzan con unos cuantos “bytes mágicos”, valores que indican el formato en que se ha guardado la imagen en el archivo. En el caso del PGM, este “magic” es una línea de texto que dice “P2” (el formato 2 de la familia PNM; véanse las páginas de manual, “man pgm” y “man ppm” para más detalles). Así, un ejemplo de archivo de imagen PGM es el icono , que se muestra a continuación:

```
P2
24 7
15
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 3 3 3 0 0 7 7 7 7 0 0 0 11 11 11 11 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0
0 3 3 3 0 0 0 7 7 7 0 0 0 0 11 11 11 0 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 0 11 0 0 0 0 15 0 0 0 0
0 3 0 0 0 0 0 7 7 7 7 0 0 0 11 11 11 11 0 0 15 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Los valores 24 y 7 son las columnas y filas de la imagen, respectivamente; en la imagen, el negro vale 0, y el blanco vale 15.

El siguiente ejemplo (listado 2c.15) genera una imagen de  $512 \times 512$  píxeles que representa el cuadrado  $[-1; +1] \times [-1; +1]$ . Su brillo viene dado por la función

$$f(x, y) = \frac{1}{1 + 2x^2 + 3y^2}$$

<sup>9</sup>Hablaremos del “Gimp” (el GNU Image Manipulation Program, o programa de manipulación de imágenes de GNU) porque es el programa estándar de retoque fotográfico en Linux y porque también está disponible en Windows. Otros programas válidos para nuestro propósito (es decir, capaces de abrir imágenes en formato PGM) son “ImageMagick” (en Linux), “PaintShop Pro” (distribuido como “shareware”) y el “Adobe PhotoShop” (comercial). Este formato no está soportado por los programas del Office de Microsoft.

donde  $x$  e  $y$  varían en dicho cuadrado  $[-1; +1] \times [-1; +1]$ .

Como novedades de este programa [2c.15](#), destacamos:

1. El tamaño de la imagen se ha definido usando una directiva del preprocesador, "#define" que lo que hace es que, cuando se ejecuta el preprocesador, las palabras "ALTURA" y "ANCHURA" sean sustituidas por el número 512. "ALTURA" y "ANCHURA" no son variables del programa; no se pueden asignar, simplemente porque cuando llegan al compilador ya no son literales, sino el número 512, como se han "#definido".
2. El tipo de la función "guardaPGMd", es "void". Este tipo denota la falta de tipo y valor de retorno: basta ver que al "return" de la función no le sigue ninguna expresión. Usaremos el tipo "void" en C cuando algo no tenga tipo. Una variable nunca será "void", porque no tiene sentido guardar algo que es nada (o está indefinido lo que es). Sin embargo sí será habitual tener punteros a "void", declarados como "void \*ptr": apuntan a una dirección de memoria, en la que no se sabe qué es lo que hay.
3. La matriz de píxeles en la función "main", llega a la función "guardaPGMd" como un vector. Esto se debe a que los elementos de una matriz se guardan en memoria uno tras otro, precisamente en el orden en que los leeríamos (y en el que los va a guardar "guardaPGMd"). Al llamar a la función, esta conversión de una matriz "double f [ALTURA] [ANCHURA]" a un vector "double pixels[]" o, equivalentemente, "double \*pixels", se indica con un *retipado* (un "cast", en Inglés): "(double\*)f". Esta expresión, un tipo de dato entre paréntesis precediendo a una variable, quiere decir: sea del tipo que sea la variable, trátala como si fuera de este tipo. Siempre que sea posible, el compilador C tomará las medidas oportunas y la tratará así. Otro ejemplo en este programa son las conversiones de las variables enteras "i" y "j" a tipo "double" antes de ser divididas por la "ALTURA" y la "ANCHURA", respectivamente (¿qué pasaría si no se hiciese este retipado?).

**Listado 2c.15:** El programa que guarda una imagen en escala de grises (formato PGM) calculada a partir de una expresión matemática.

```

1 #include <stdio.h>
2
3 #define ANCHURA 512
4 #define ALTURA 512
5
6 void guardaPGMd(char* nombre,
7                 int anchura, int altura,
8                 double *pixels,
9                 double pixel_min, double pixel_max)
10 {
11     int i, j, ij, p;
12     FILE* imagen;
13
14     imagen=fopen(nombre, "wb");
15     fprintf(imagen, "P2\n");
16     fprintf(imagen, "#guardaPGMd %s\n", nombre);
17     fprintf(imagen, "%d %d\n", anchura, altura);
18     fprintf(imagen, "255\n");
19

```

## 2C.11. UTILIDAD: GUARDAR DATOS COMO UNA IMAGEN

2c-27

```
20     ij=0;
21     for(i=0; i<altura; i++)
22     {
23         for(j=0; j<anchura; j++)
24         {
25             p=(int)(255*(pixels[ij]-pixel_min))
26                 /(pixel_max-pixel_min);
27             if( p<0 ) p=0;
28             if( p>255) p=255;
29
30             fprintf(imagen, " %d", p);
31
32             ij++;
33         }
34         fprintf(imagen, "\n");
35     }
36
37     fclose(imagen);
38
39     return;
40 }
41
42 int main(int argc, char **argv)
43 {
44     int i, j;
45     double x, y, z;
46     double f [ALTURA] [ANCHURA];
47
48     for(i=0; i<ALTURA; i++)
49     {
50         for(j=0; j<ANCHURA; j++)
51         {
52             x=2*(double)j/(ANCHURA-1)-1.0;
53             y=2*(double)i/(ALTURA-1)-1.0;
54             z=1.0/(1+(2*x*x+3*y*y));
55
56             f [i] [j]=z;
57         }
58     }
59
60     guardaPGMd("prueba.pgm",
61                 ANCHURA, ALTURA, (double*)f, 0, 1.0);
62
63     return 0;
64 }
```

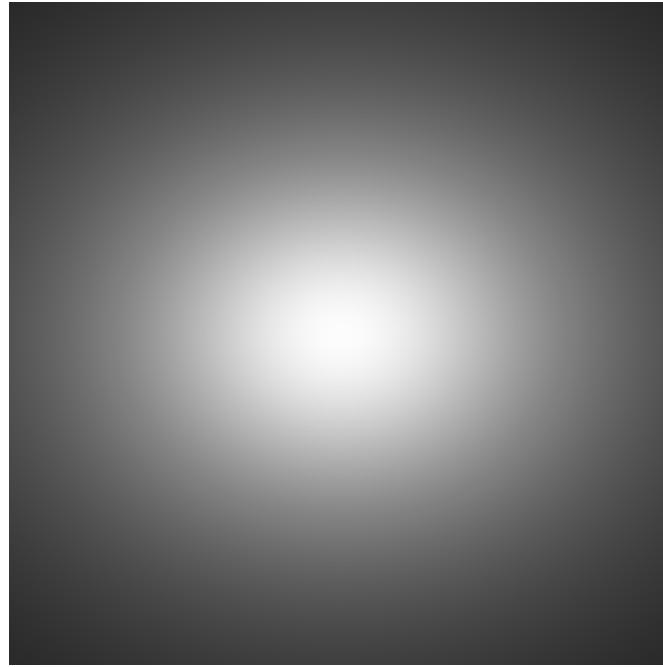


Figura 2c.3: Imagen de ejemplo generada por el programa del listado 2c.15.

## 2c.12. Estructuras de datos y definición de tipos

Una estructura de datos es un conjunto de datos que el programa debe tratar siempre juntos y que, por ello, es conveniente agrupar o encapsular en una única entidad, una estructura<sup>10</sup>. En C, esta estructura de datos se declara usando “struct”. Por ejemplo, la siguiente estructura define un vector bidimensional, formado por dos componentes reales: “x”, “y”.

```
struct Vector2d {  
    double x, y;  
};
```

En un programa, se puede operar entre dos vectores explícitamente como:

```
struct Vector2d u, v;  
...  
u.x=1.0; u.y=2.0;  
v.x=2*u.x; v.y=2*u.y;  
...
```

o creando funciones que retornen el resultado, exactamente igual a como si la operación se realizase entre dos tipos de datos usuales (primitivos):

```
struct Vector2d porescalar(struct Vector2d u, double l)  
{  
    struct Vector2d v;  
    v.x=l*u.x;
```

<sup>10</sup>En otros lenguajes a las estructuras se las llama “record”, porque cuando se lee o guarda el conjunto de datos se hace de forma agrupada (atómica). En algún contexto se llamará a una estructura de datos un objeto, aunque este concepto es más genérico. También se emplea el término “estructura de datos” para indicar un modo de jerarquizar los datos, dotándolos de relaciones entre ellos (formando listas, árboles, grafos, etc.). Aquí nos quedaremos en el primer concepto, más sencillo, aunque en la base de todos los demás.

```

    v.y=l*u.y;
    return v;
}

```

Esta notación obliga a utilizar siempre la palabra clave “struct” a la hora de declarar el uso de una estructura de tipo “Vector2d”. Cuando el uso es muy frecuente, puede ser interesante instruir al compilador para que trate a las estructuras como otro tipo de datos más (ya que así lo estamos haciendo ya, por otro lado). Para ello utilizaremos la instrucción “typedef”.

Por ejemplo, para trabajar con números reales, nos puede interesar definir el tipo “Real” como:

```
typedef double Real;
```

Del mismo modo, para definir el tipo “Vector2d”, haríamos los dos pasos siguientes<sup>11</sup>:

```

struct _Vector2d
{
    double x, y;
};
typedef struct _Vector2d Vector2d;

```

Obsérvese que, en lo que sigue a “typedef” es como si estuviéramos declarando una variable llamada “Real”, en el primer ejemplo, o “Vector2d”, en el segundo.

**Ejemplo.** Escribir una función que, dado un vector del plano (variable de tipo “Vector2d”), lo rote en un ángulo dado. Para ello, se pasará la variable por referencia a una función “void rotar(Vector2d, double)”, donde el segundo argumento es el ángulo en radianes: esto se muestra en el listado 2c.16. Obsérvese en esta función la notación para acceder a los campos de una variable pasada a través de un puntero (por ejemplo, “v->x”). Explíquese el uso de la variable auxiliar “double aux” empleada en la función “void rotar()”.

**Listado 2c.16:** El programa que rota un vector en un ángulo de  $+10^\circ$ .

```

1 #include <stdio.h>
2 #include <math.h>
3
4 struct _Vector2d
5 {
6     double x, y;
7 };
8
9 typedef struct _Vector2d Vector2d;
10
11 void rotar(Vector2d *u, double theta)
12 {

```

<sup>11</sup>También se podrían usar...  
la sintaxis abreviada:

```

typedef struct _Vector2d
{
    double x, y;
} Vector2d;

```

la sintaxis compacta (con una estructura anónima):

```

typedef struct
{
    double x, y;
} Vector2d;

```

```

13     double aux;
14
15     aux=u->x;
16     u->x=u->x*cos(theta)-u->y*sin(theta);
17     u->y=aux*sin(theta)+u->y*cos(theta);
18
19     return;
20 }
21
22 #define ROTACION (M_PI*10/180)
23
24 int main(int argc, char** argv)
25 {
26     int n;
27     Vector2d vec;
28     vec.x=1.0;
29     vec.y=0.0;
30
31     for(n=1; n<=9; n++)
32     {
33         rotar(&vec, ROTACION);
34         printf("%d grados: (%g, %g)\n", 10*n, vec.x, vec.y);
35     }
36
37     return 0;
38 }
```

---

## Notación.

Es muy conveniente adherirse a un estilo de escritura de los programas, porque ayuda mucho a comprender el propósito y el funcionamiento de los mismos.

Por ejemplo, en todo esta unidad didáctica nos adheriremos al convenio de abrir las llaves en la línea siguiente a la instrucción que las rige, y a cerrarlas en una línea después de la última instrucción. Esta forma de estructurar el texto le es indiferente al preprocesador o al compilador, pero a nosotros nos facilita la lectura.

Otra norma a la que conviene adherirse es a nombrar las variables o funciones con la primera letra minúscula (por ejemplo “double coordX;”, o “extern double sin(double x);”) o a las #definiciones de símbolos con todas las letras mayúsculas (por ejemplo “#define M\_PI 3.14159265354”).

Nosotros usaremos una norma más referida a los nombres de las estructuras y nuevos tipos que definamos: comenzarán por una letra mayúscula. De este modo, cuando veamos “Vector2d”, sabremos que es una estructura o nuevo tipo definido por nosotros (no un tipo primitivo del lenguaje), y no una función o variable. En los ejemplos anteriores hemos visto ya un contraejemplo de esta norma autoimpuesta: “\_Vector2d”, que empieza por “\_”.

## 2c.13. Soluciones a los ejercicios del capítulo

---

### Listado 2c.17: Ejercicio 3.16.

---

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5
6     printf("Hola mundo, me llamo %s\n", argv[0]);
7
8     return 0;
9 }
```

---

---

### Listado 2c.18: Ejercicio 3.17.

---

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5
6     printf("Hola %s, soy el programa %s\n", argv[1], argv[0]);
7
8     return 0;
9 }
```

---

---

### Listado 2c.19: Ejercicio 3.18.

---

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5
6     printf("El nºmero de argumentos es %d\n", argc);
7
8     return 0;
9 }
```

---

---

### Listado 2c.20: Ejercicio 3.19.

---

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv)
5 {
6     float a, b, c;
```

```

7
8     a=atof(argv[1]);
9     b=atof(argv[2]);
10    c=a+b;
11
12    printf(" %g+%g=%g\n", a, b, c);
13
14    return 0;
15 }
```

---

**Listado 2c.21:** Ejercicio 3.20.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv)
5 {
6     int a, b, c;
7
8     a=atoi(argv[1]);
9     b=atoi(argv[2]);
10    /* Al ser a y b enteros, el resultado
11       será la parte entera de la división */
12    c=a/b;
13
14
15    printf(" %d+%d=%d\n", a, b, c);
16
17    return 0;
18 }
```

---

**Listado 2c.22:** Ejercicio 3.21.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv)
5 {
6     float a, b, c;
7
8     a=atof(argv[1]);
9     b=atof(argv[2]);
10
11    c=a-b;
12
13    if( c<0 ) {
14        c=-c;
```

## 2C.13. SOLUCIONES A LOS EJERCICIOS DEL CAPÍTULO

2c-33

```
15     }
16
17     printf(" | %g - %g | = %g \n", a, b, c);
18
19     return 0;
20 }
```

---

**Listado 2c.23:** Ejercicio 3.22.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv)
5 {
6     int i, n;
7     long int f;
8
9     n=atoi(argv[1]);
10
11    f=1;
12    for(i=2; i<=n; i++)
13        f*=i;
14
15    printf(" %d ! = %ld\n", n, f);
16
17    return 0;
18 }
```

---

**Listado 2c.24:** Ejercicio 3.23.

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     int n;
6     float f, df;
7
8     n=100;
9     df=1.0/n;
10
11    for(f=df; f<1.0; f=f+df)
12        printf(" %g\n", f);
13
14    return 0;
15 }
```

---

**Listado 2c.25:** Ejercicio 3.24.

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     int n;
6
7     n=2;
8     do
9     {
10         printf("%d\n", n);
11         n+=2;
12
13     } while(n<=10);
14
15     return 0;
16 }
```

---

---

**Listado 2c.26:** Ejercicio 3.25.

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     int n;
6     float f, df;
7
8     n=100;
9     df=1.0/n;
10
11     f=df;
12     while(f<1.0)
13     {
14         printf("%g\n", f);
15         f=f+df;
16     }
17
18     return 0;
19 }
```

---

---

**Listado 2c.27:** Ejercicio 3.26.

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     int i, j, k;
```

## 2C.13. SOLUCIONES A LOS EJERCICIOS DEL CAPÍTULO

2c-35

```

6     double A[3][3]={ {1.0, 2.0, 3.0},
7                 {2.0, 3.0, 1.0},
8                 {3.0, 1.0, 2.0} };
9     double B[3][3];
10    double C[3][3];
11
12    for(i=0; i<3; i++)
13    {
14        for(j=0; j<3; j++)
15        {
16            printf("B[%d, %d]= ", i, j);
17            scanf("%lf", &B[i][j]);
18        }
19    }
20
21    for(i=0; i<3; i++)
22    {
23        for(j=0; j<3; j++)
24        {
25            C[i][j]=0.0;
26            for(k=0; k<3; k++)
27            {
28                C[i][j]+=A[i][k]*B[k][j];
29            }
30        }
31    }
32
33    for(i=0; i<3; i++)
34    {
35        for(j=0; j<3; j++)
36        {
37            printf(" %4.4lf", C[i][j]);
38        }
39        printf("\n");
40    }
41
42    return 0;
43 }
```

---

**Listado 2c.28: Ejercicio 3.27.**

```

1 #include <stdio.h>
2
3
4 int main(int argc, char** argv)
5 {
6     int npts, n;
7     double x, y;
```

```

8      double Sx, Sy, Sxy, Sxx, Syy;
9      double m_y, y0;
10     double m_x, x0;
11     double r2;
12     FILE *fin;
13
14     fin=fopen("datos.dat", "r");
15
16     npts=0;
17     Sx=0.0;
18     Sy=0.0;
19     Sxy=0.0;
20     Sxx=0.0;
21     Syy=0.0;
22
23     do
24     {
25         /* la función fscanf devuelve el número
26            de datos leídos, que deben ser 2 */
27         n=fscanf(fin, "%lf%lf", &x, &y);
28         if( n==2 )
29         {
30             npts++;
31             Sx+=x;
32             Sy+=y;
33             Sxy+=x*y;
34             Sxx+=x*x;
35             Syy+=y*y;
36         }
37     } while( n==2 );
38
39     fclose(fin);
40
41     /* y=m_y*x+y0 */
42     m_y=(npts*Sxy-Sx*Sy)/(npts*Sxx-Sx*Sx);
43     y0 =(Sy-m_y*Sx)/npts;
44
45     /* x=m_x*y+x0 */
46     m_x=(npts*Sxy-Sx*Sy)/(npts*Syy-Sy*Sy);
47     /* y0 =(Sx-m_x*Sy)/npts; */
48
49     r2=m_x*m_y;
50
51     printf("Resultados de la regresion:\n");
52     printf("Y(X) = m_y * X + Y0 \n");
53     printf(" m_y= %g\n", m_y);
54     printf(" Y0 = %g\n", y0 );
55

```

## 2C.13. SOLUCIONES A LOS EJERCICIOS DEL CAPÍTULO

2c-37

```
56     printf("Coeficiente de correlacion r^2 = %g\n",
57             r2);
58
59     return 0;
60 }
```

---

**Listado 2c.29:** Ejercicio 3.28.

---

```
1 #include <stdio.h>
2
3 #define NMAXPTS 100
4
5 int leePuntos(char* arch, double x[], double y[])
6 {
7     int n, npts;
8     double xi, yi;
9
10    FILE *fin;
11
12    fin=fopen(arch, "r");
13
14    npts=0;
15    do
16    {
17        n=fscanf(fin, "%lf %lf",
18                  &xi, &yi);
19        if( n==2 && npts<NMAXPTS )
20        {
21            x[npts]=xi;
22            y[npts]=yi;
23        }
24        npts++;
25
26    } while( n==2 && npts<=NMAXPTS );
27
28    fclose(fin);
29
30    /* error */
31    if( npts==NMAXPTS+1 && n==2 )
32    {
33        npts=-1;
34    }
35
36    return npts;
37 }
38
39 double regresionLineal(int npts, double x[], double y[],
40                         double *_m, double *_y0)
```

```
41 {
42     int n;
43     double Sx, Sy, Sxy, Sxx, Syy;
44     double m_x, m_y, y0;
45     double r2;
46
47     Sx=0.0;
48     Sy=0.0;
49     Sxy=0.0;
50     Sxx=0.0;
51     Syy=0.0;
52
53     for(n=0; n<npts; n++)
54     {
55         Sx+=x[n];
56         Sy+=y[n];
57         Sxy+=x[n]*y[n];
58         Sxx+=x[n]*x[n];
59         Syy+=y[n]*y[n];
60     }
61
62     if( npts>0 )
63     {
64         /*  $y=m_y*x+y_0$  */
65         m_y=(npts*Sxy-Sx*Sy)/(n*Sxx-Sx*Sx);
66         y0 =(Sy-m_y*Sx)/npts;
67
68         /*  $x=m_x*y+x_0$  */
69         m_x=(npts*Sxy-Sx*Sy)/(n*Syy-Sy*Sy);
70
71         r2=m_x*m_y;
72
73         /* valores retornados */
74         *_y0=y0;
75         *_m =m_y;
76     }
77     else
78     {
79         r2=-1.0;
80     }
81
82     return r2;
83 }
84
85
86 int main(int argc, char** argv)
87 {
88     int npts, n;
```

## 2C.13. SOLUCIONES A LOS EJERCICIOS DEL CAPÍTULO

2c-39

```

89     double x[NMAXPTS],
90         y[NMAXPTS];
91     double m_y, y0, r2;
92
93     /* usa la función para leer los puntos */
94     npts=leePuntos("datos.dat", x, y);
95
96     /* manejar el error */
97     if( npts<0 )
98     {
99         fprintf(stderr, "Error: el número de puntos"
100             " excedió la memoria reservada\n");
101         exit(-1);
102     }
103
104    /* usa la función para calcular la regresión */
105    r2=regresionLineal(npts, x, y, &m_y, &y0);
106
107    /* manejar el error */
108    if( r2<0 )
109    {
110        fprintf(stderr, "Error: no se han pasado "
111            "puntos a la función de "
112            "regresión\n");
113        exit(-1);
114    }
115
116    printf("Resultados de la regresión:\n");
117    printf("Y(X) = m_y * X + Y0 \n");
118    printf(" m_y= %g\n", m_y);
119    printf(" Y0 = %g\n", y0 );
120
121    printf("Coeficiente de correlación r^2 = %g\n",
122           r2);
123
124    return 0;
125 }
```

---

**Listado 2c.30: Ejercicio 3.29.**

```

1 #include <stdio.h>
2
3
4 void producto(double *p, double V[], double MxV[], double *modulo)
5 {
6     int i, j;
7
8     //calculamos el producto de la matriz y el vector
```

```

9
10    for (i=0; i<2; i++)
11    {
12        MxV[i]=0;
13        for (j=0; j<3; j++)
14            MxV[i]+=p[3*i+j]*V[j];
15    }
16
17 //calculamos el módulo del vector resultante
18 for (i=0; i<2; i++)
19     *modulo+=MxV[i]*MxV[i];
20
21 *modulo=sqrt(*modulo);
22
23 return;
24 }
25
26
27 int main(int argc, char** argv)
28 {
29     double M[2][3]={{1.,1.,1.},{1.,1.,1.}};
30     double V[3]={1.,2.,3.};
31     double MxV[2], modulo;
32
33     producto(*M,V,MxV,&modulo);
34
35     printf("El producto de la matriz M por el vector V es (%g, %g)\n", MxV[0],
36           MxV[1]);
37     printf("El modulo del vector es %f\n", modulo);
38
39     return 0;
}

```

**Listado 2c.31: Ejercicio 3.30.**

```

1 #include <stdio.h>
2
3
4 void producto(double *p, double V[], double MxV[], double *modulo)
5 {
6     int i, j;
7
8 //calculamos el producto de la matriz y el vector
9
10    for (i=0; i<2; i++)
11    {
12        MxV[i]=0;
13        for (j=0; j<3; j++)

```

## 2C.13. SOLUCIONES A LOS EJERCICIOS DEL CAPÍTULO

2c-41

```
14         MxV[i] += p[3*i+j]*V[j];
15     }
16
17 //calculamos el módulo del vector resultante
18 for (i=0; i<2; i++)
19     *modulo+=MxV[i]*MxV[i];
20
21 *modulo=sqrt(*modulo);
22
23 return;
24 }
25
26
27 int main(int argc, char** argv)
28 {
29     double M[2][3]={{1.,1.,1.},{1.,1.,1.}};
30     double V[3]={1.,2.,3.};
31     double MxV[2], modulo;
32
33     producto(*M,V,MxV,&modulo);
34
35     printf("El producto de la matriz M por el vector V es (%g, %g)\n", MxV[0],
36           MxV[1]);
37     printf("El modulo del vector es %f\n", modulo);
38
39     return 0;
}
```



## Tema 2d

# Algunas herramientas útiles para la programación en C

Este capítulo revisa algunas utilidades de las bibliotecas estándar del lenguaje C. Como el anterior, no pretende ser un tutorial, sino mostrar cómo se usan a través de algunos ejemplos, ya que serán empleadas en los próximos capítulos de simulaciones en física computacional.

El capítulo incluye al final una interesante sección sobre cómo organizar el código de nuestros proyectos en bibliotecas de funciones propias (de una forma semejante a como está organizada la biblioteca estándar de C) de modo que puedan ser reutilizadas de unos proyectos a otros, sin necesidad de escribir el código cada vez y, además, depurando las funciones al usarlas en diferentes contextos y con diferentes argumentos de entrada.

### 2d.1. Gestión dinámica de la memoria

Cuando declaramos en C cualquier tipo de variable, el sistema operativo reserva de forma automática la memoria necesaria para almacenar esa variable. La cantidad de memoria reservada dependerá del tipo de variable, y en el caso de vectores y *arrays* también de su longitud, independientemente de que luego se vayan, o no, a utilizar todos los elementos de estos objetos. Por ejemplo, al declarar el vector “A” mediante “int A[10]” reservamos un bloque de memoria de tamaño 10 enteros, lo que equivale en muchos sistemas a 40 bytes.<sup>1</sup> Eso nos permitirá trabajar con vectores de hasta 10 componentes: “A[i]” con  $i = 0, \dots, 9$ . Sin embargo, si tratamos de escribir o leer el valor de la componente “A[13]”, estaremos intentando acceder a un espacio de memoria *no reservado* y es muy probable que se produzca un error de ejecución, aunque el programa se haya compilado correctamente. Si trabajamos con el sistema

<sup>1</sup>Conviene recordar que muchos aspectos concretos de la programación en C –como el tamaño de los diferentes tipos de variables– dependen esencialmente del compilador. Sin embargo, como los compiladores se diseñan y/o se adaptan al sistema operativo y a la arquitectura del procesador, se podría decir que esas características dependerán en último extremo del SO y del hardware de nuestra máquina. En cualquier caso, hay unos rangos mínimos que se deben cumplir y que están establecidos por el denominado *C standard*. Por ejemplo, el tamaño de un “int” debe ser lo suficientemente grande como para poder representar valores entre  $-32767$  y  $32767$ , lo que significa un tamaño mínimo de 2 bytes. Por lo tanto, es conveniente no hacer ninguna suposición sobre el tamaño de los tipos de datos en C, y si queremos saber el tamaño de un cierto tipo de dato o de una variable concreta “x”, lo mejor será preguntárselo al compilador utilizando la función “`sizeof()`”. De este modo “`sizeof(int)`” y “`sizeof(x)`” nos devolverá un entero que nos indica el tamaño en bytes de las variables tipo “int” y de la variable “x”, respectivamente. En sistemas operativos de 32 y 64 bits, lo habitual es que las variables tipo “int” sean de 32 bits (4 bytes).

operativo Linux aparecerá el mensaje *segmentation fault ('core' dumped)* (o *violación de segmento ('core' generado)*, en castellano), mientras que en SO Windows el programa se quedará “colgado” sin responder. Excepcionalmente el programa continuará su ejecución y finalizará, pero desde el punto de vista del programador eso es muy peligroso ya que no sabremos qué se ha producido ese error ni cómo ha afectado a los resultados del programa.

Los sistemas operativos establecen por defecto un límite al tamaño de la memoria reservada para las variables locales y globales, como por ejemplo vectores y matrices. Esta zona de la memoria se llama pila o *stack*. En Windows, por ejemplo, este límite es generalmente de 1 Mb. En Linux suele ser mucho mayor. Esto supone que si declararamos vectores o matrices más grandes que este límite (por ejemplo “`int A[1000][1000]`”) se producirá lo que se denomina *desbordamiento de pila (stack overflow)* y tendremos el mismo problema de ejecución que acabamos de ver más arriba. La compilación no dará errores, pero cuando el programa intente leer o escribir datos fuera del espacio reservado (que en este caso está dado por el máximo que el sistema asigna por defecto) se producirá un error de ejecución manifestándose de la misma forma.

Por estas razones resulta muy útil: (1) poder pedir al sistema operativo toda la memoria que el programa vaya a necesitar, y (2) poder “adaptar” la memoria de las variables en *tiempo de ejecución*, es decir, poder reservar más o menos memoria en función de las necesidades que requiera el programa cuando se ejecuta. Esto es lo que se denomina *gestión dinámica de la memoria* y lo vamos a estudiar en esta sección.

Antes, conviene aclarar que se puede reservar más memoria para estas variables sin recurrir al uso de memoria dinámica. Por ejemplo, si estamos trabajando en Windows con “Code::Blocks”, deberemos hacer lo siguiente:

1. Ir a *Project* (Proyecto) > *Build options* (Opciones de construcción)
2. Seleccionar la pestaña *Linker settings* (Ajustes del Linker)
3. En *Other linker options* (Otras opciones del linker), escribir (sin espacios), por ejemplo:  
-WI,-stack,4096000

La ruta es, por tanto, *Project > Build Options > Linker Settings > Other linker options*. Esto le dirá al sistema operativo que para ese ejecutable le asigne una memoria a la pila de 4096000 bytes (unos 4 MB). Evidentemente se puede variar este tamaño en función de las necesidades.

De este modo le podemos pedir al sistema operativo que le asigne al programa toda la memoria que vaya a necesitar, lo que implica que debemos haberla calculado previamente. El sistema se la asignará automáticamente, la vaya a utilizar el programa o no (y no la liberará hasta que el programa termine).

Como hemos comentado anteriormente, la forma más “elegante” y eficiente de gestionar la memoria de los programas en C es mediante lo que se denomina asignación dinámica de memoria. Esto nos permitirá pedirle al sistema operativo “pedazos” (*chunks*) de memoria añadidos, del tamaño que necesitemos, y todo en tiempo de ejecución. Para esto, en C, se usan las funciones “`malloc()`”, “`calloc()`” y “`realloc()`”, que están declaradas en la librería “`stdlib.h`”. Para devolver estos pedazos de memoria al sistema operativo, de modo que puedan ser usados por otros programas, se usa la función “`free()`”.

### 2d.1.1. La función `malloc()`

La función “`malloc()`” está declarada como:

## 2D.1. GESTIÓN DINÁMICA DE LA MEMORIA

2d-3

```
void *malloc(int N_bytes)
```

y reserva en la memoria un bloque de tamaño dado por “N\_bytes” (número entero de bytes que se quieren reservar), devolviendo un puntero de tipo “void” al primer elemento de la zona reservada. Por ejemplo, si queremos reservar espacio para una lista de  $N$  valores reales, en lugar de declarar el vector “double A[N]” podemos escribir:

```
double *A;
A=(double *)malloc(N*sizeof(double));
```

Vemos que para indicarle a “malloc()” la cantidad en bytes de memoria que queremos reservar, hemos multiplicado la longitud de nuestra lista por el tamaño en bytes de cada elemento de la lista, que al tratarse de reales vendrá dado por “sizeof(double)” (8 bytes, por lo general). La sintaxis “(double \*)” antes de “malloc()” es lo que se denomina un “retipado” (o *cast*), y se utiliza muy frecuentemente en programación en C para asegurarse de que el tipo de dato que resulta de la operación que hay a su derecha, coincide con el tipo de dato de la variable que hay a la izquierda de la igualdad. En nuestro caso, esto nos sirve para asegurarnos de que el puntero devuelto por “malloc”, que por defecto es de tipo “void \*”, se va a asignar a una variable “A” que es un puntero de tipo “double \*”, o sea, que la memoria que hemos reservado va a usarse para almacenar doubles. El resto del programa sería igual que si hubiésemos utilizado la instrucción habitual. A los elementos del vector se accede con la sintaxis de siempre, tanto para leer sus valores como para escribir valores en ellos, de modo que “A[n]” es el  $n$ -ésimo elemento del array “A”.

Cuando hayamos terminado de usar esa memoria (p. ej. al finalizar la función o el programa dentro del que la reservamos), es de “buena educación” dejar libre la memoria reservada utilizando la función “free()”, declarada del siguiente modo:

```
void free(void *q);
```

Esta función libera el bloque de memoria al que apunta<sup>2</sup> el puntero “\*q” introducido como argumento, y que será un puntero devuelto por “malloc()” o “calloc()”. Esto es importante ya que la memoria no se libera por defecto, hay que hacerlo explícitamente. Siguiendo con el ejemplo anterior tendremos que escribir:

```
free(A);
```

La operación que acabamos de describir para los vectores (*arrays unidimensionales*) puede ser generalizada a arrays multidimensionales como por ejemplo las matrices. Supongamos que en nuestro programa vamos a trabajar con matrices de  $N \times S$  números reales, donde  $N$  es el número de filas y  $S$  el de columnas. En lugar de la conocida declaración “double B[N][S]” podemos escribir:

```
double **B;
B=(double **)malloc(N*sizeof(double *));
for (i=0; i<N; i++)
    B[i]=(double *)malloc(S*sizeof(double));
...
for (i=0; i<N; i++)
    free(B[i]);
free(B);
```

---

<sup>2</sup>Cuando decimos que un puntero “apunta” a una determinada variable, queremos decir que guarda la dirección de memoria en la que está almacenada esa variable. Como sabemos, los punteros son variables cuyos valores son direcciones de memoria.

Esta estructura sigue la misma lógica que ya estudiamos para las matrices. Como ya vimos, una matriz puede ser interpretada como un vector de vectores. También sabemos que, en términos de programación, un vector es representado mediante un puntero. Por lo tanto, una matriz será un vector de punteros y se representará mediante un *puntero a puntero*. Esto se implementa en C mediante la declaración “`double **B`”. Con la primera llamada a la función “`malloc()`” reservamos la memoria necesaria para almacenar ese vector de longitud  $N$  y cuyos elementos son punteros a “`double`” (y por tanto con un tamaño dado por “`sizeof(double *)`”). El elemento  $i$ -ésimo de ese vector, “`B[i]`” (con  $i=0, \dots, N-1$ ), es un puntero que apunta al comienzo de la fila “ $i$ ”, la cual es un vector de reales de longitud  $s$ . La reserva de memoria para cada uno de estos bloques (de ahí el bucle “`for`”) la realizamos con la segunda llamada a la función “`malloc()`”. Obsérvese que antes de liberar la memoria mediante “`free(B)`”, hemos liberado previamente las filas de la matriz mediante “`free(B[i])`”.

El proceso de generalización a dimensiones mayores es inmediato, aunque puede resultar un tema más avanzado y no será objeto de evaluación en este curso. Por ejemplo, aumentando en 1 la dimensión tendremos arrays tridimensionales de la forma “`C[N1][N2][N3]`”, que son vectores de matrices, o vectores de vectores de vectores. Por lo tanto, estas variables se representarán mediante *punteros a puntero a puntero* y se declararán como variables tipo “`double ***`” (si el array es de reales). Después de la declaración, lo primero que debemos hacer es reservar la memoria necesaria para almacenar el vector de longitud  $N_1$  y cuyos elementos representan matrices. Por lo tanto, el elemento  $i$ -ésimo de ese vector, “`C[i]`” (con  $i=0, \dots, N_1-1$ ), será un puntero a puntero y podemos aplicar lo que ya sabemos de las matrices. Para cada uno de estos elementos deberemos hacer una segunda llamada a la función “`malloc()`” y asignarle memoria al correspondiente vector de punteros de longitud  $N_2$  (número de “filas” de cada matriz). Por último, cada elemento “`C[i][j]`” (con  $i=0, \dots, N_1-1$  y  $j=0, \dots, N_2-1$ ) será un puntero que presenta cada vector de reales en la tercera dimensión y que tiene una longitud  $N_3$  (número de “columnas”). Tendremos que recorrer todos los valores de  $i, j$  y hacer la correspondiente reserva de memoria mediante una tercera llamada a “`malloc()`”. El código final será:

```
double ***C;
C=(double ***)malloc(N1*sizeof(double **));
for (i=0; i<N1; i++)
{
    C[i]=(double **)malloc(N2*sizeof(double *));
}
for (i=0; i<N1; i++)
{
    for (j=0; j<N2; j++)
    {
        C[i][j]=(double *)malloc(N3*sizeof(double));
    }
}
```

Como en el caso de vectores y matrices, las componentes de este objeto tendrán la forma “`C[i][j][k]`” con  $i=0, \dots, N_1-1$ ,  $j=0, \dots, N_2-1$  y  $k=0, \dots, N_3-1$ . Por ejemplo, podemos escribir la matriz asociada a “`k=a`” donde “`a`” es una constante menor que “`N3`” (por ejemplo “`a=0`”) haciendo:

```
int a=0;
for (i=0; i<N1; i++)
```

## 2D.1. GESTIÓN DINÁMICA DE LA MEMORIA

2d-5

```
{
    for (j=0; j<N2; j++)
        printf(" %g\t",C[i][j][a]);
    printf("\n");
}
```

De nuevo, antes de liberar la memoria con “`free(C)`”, debemos liberar la memoria de las filas “`C[i][j]`” y de las matrices “`C[i]`”:

```
for (i=0; i<N1; i++)
{
    for (j=0; j<N2; j++)
    {
        free(C[i][j]);
    }
}
for (i=0; i<N1; i++)
{
    free(C[i]);
}
free (C);
```

### 2d.1.2. La función `calloc()`

La función “`calloc()`” es completamente equivalente a “`malloc()`”, pero además inicializa todos los elementos a 0. Está declarada como:

```
void *calloc(int N_datos, int tamaño_dato)
```

y observamos que tiene dos argumentos: el número de datos y el tamaño en bytes de cada dato. La equivalencia con “`malloc()`” es inmediata si tenemos en cuenta que `int N_bytes=N_datos*tamaño_dato`. Al igual que “`malloc()`”, devuelve un puntero al primer elemento de la zona reservada. Siguiendo con el ejemplo anterior para el vector “`double A[N]`”, debemos escribir:

```
double *A;
A=(double *)calloc(N, sizeof(double));
...
free (A);
```

### 2d.1.3. La función `realloc()`

Por último, también podemos modificar en tiempo de ejecución el tamaño de la memoria previamente reservada con “`malloc()`” o “`calloc()`”, mediante la función “`realloc()`”, declarada como:

```
void *realloc(void *q, int N_bytes)
```

Esta función cambia el tamaño del bloque de memoria al que apunta el puntero “`*q`” (previamente asignado con “`malloc()`”, “`calloc()`” o la propia “`realloc()`”) y le asigna un nuevo tamaño en bytes dado por “`N_bytes`”.

## Ejemplos

En el siguiente ejemplo mostramos de forma muy sencilla la gestión dinámica de memoria mediante las funciones “malloc()”, “calloc()” y “realloc()”.

**Listado 2d.1:** Uso de las funciones “malloc()”, “calloc()” y “realloc()”.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5
6 int main(int argc, char **argv)
7 {
8     int i;
9
10    /* empezaremos con un ejemplo aplicado a cadenas de caracteres*/
11    char *asignatura;
12    asignatura = (char *)malloc(4*sizeof(char));
13    strcpy(asignatura, "FCI");
14    printf("El nombre abreviado de nuestra asignatura es %s\n\n", asignatura);
15    /* Es interesante señalar que hemos reservado espacio para
16       cuatro caracteres, y no tres. Esto es debido a que en el lenguaje C
17       las cadenas de texto se definen con un carácter nulo al final,
18       por lo que la cadena "FCI" ocupa cuatro caracteres en memoria: {'F', 'C', 'I'
19       , '\0'}.
20   Es fácil comprobarlo viendo que el resultado de sizeof("FCI") es 4. */
21
22    /* reasignamos la memoria para que nuestra cadena sea más grande */
23    asignatura = (char *)realloc(asignatura, 23*sizeof(char));
24    asignatura="Fisica_Computacional_I";
25    printf("El nombre completo de nuestra asignatura es %s\n", asignatura);
26
27    /* ahora vamos a trabajar con vector de reales de tamaño 5*/
28    int N=5;
29    double *A;
30    A=(double *)calloc(N, sizeof(double));
31
32    /* calloc por defecto inicializa todas las componentes del vector A a 0.
33       Construimos el vector A[i]=i y lo imprimimos por pantalla */
34    for (i=0; i<N; i++)
35    {
36        A[i]=i; printf("%g ", A[i]);
37    }
38    printf("\n\n");
39
40    /* llegados a este punto, queremos aumentar el número de elementos
41       del vector A en 3 unidades, pero manteniendo los datos que ya teníamos */
42    A=(double *)realloc(A, (N+3)*sizeof(double));

```

## 2D.1. GESTIÓN DINÁMICA DE LA MEMORIA

2d-7

```

42
43     /* ahora podemos asignar valores a los nuevos elementos del vector: */
44     for (i=N; i<N+3; i++)
45         A[i]=2*i;
46
47     /* si lo imprimimos al completo veremos que se han guardado
48     los valores que ya teníamos */
49     for (i=0; i<N+3; i++)
50         printf("%g ",A[i]);
51
52     /* IMPORTANTE: al final del programa hay que liberar la memoria */
53     free(asignatura);
54     free(A);
55
56     return 0;
57
58 }
```

---

A continuación mostramos otro ejemplo en el que hemos modificado el programa del listado 2c.3, que calculaba el producto de una matriz por un vector, para que toda la gestión de la memoria sea dinámica.

**Listado 2d.2:** Programa que calcula el producto de una matriz por un vector utilizando asignación dinámica de la memoria.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 2 /* número de filas de la matriz */
5 #define S 3 /* número de columnas de la matriz */
6
7
8 void producto(int nfilas, int ncolumnas, double **matriz,
9                 double *vector, double *resultado)
10 {
11     int i,j;
12
13     /* calculamos el producto de la matriz y el vector */
14     for (i=0; i<nfilas; i++)
15     {
16         for (j=0; j<ncolumnas; j++)
17             resultado[i]+=matriz[i][j]*vector[j];
18     }
19
20     return;
21 }
22
23 int main(int argc, char **argv)
24 {
```

## 2d-8 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```
25 int i,j;
26
27 /* realizamos las declaraciones */
28 double **M, *V, *MxV;
29
30 /* reservamos memoria dinámica para la matriz de N x S números reales */
31 M=(double **)calloc(N, sizeof(double *));
32 for (i=0; i<N; i++)
33     M[i]=(double *)calloc(S, sizeof(double));
34
35 /* reservamos memoria dinámica para los vectores V y MxV */
36 V=(double *)calloc(S, sizeof(double));
37 MxV=(double *)calloc(N, sizeof(double));
38
39 /* introducimos por teclado los valores de la matriz M y del vector V*/
40 printf("Introduzca los valores de la matriz M:\n");
41 for (i=0; i<N; i++)
42 {
43     for (j=0; j<S; j++)
44     {
45         printf("M[%d] [%d]=",i,j);
46         scanf("%lf",&M[i][j]);
47     }
48 }
49 printf("Introduzca los valores del vector V:\n");
50 for (i=0; i<S; i++)
51 {
52     printf("V[%d]=",i);
53     scanf("%lf",&V[i]);
54 }
55 printf("\n");
56
57 /* llamamos a la función producto definida arriba */
58 producto(N, S, M, V, MxV);
59
60 /* imprimimos en pantalla la matriz, el vector, y el producto de ambos*/
61 printf("El producto de la matriz:\n");
62 for (i=0; i<N; i++)
63 {
64     for (j=0; j<S; j++)
65     {
66         printf("%g ",M[i][j]);
67     }
68     printf("\n");
69 }
70
71 printf("por el vector:\n");
72 for (i=0; i<S; i++)
```

## 2D.1. GESTIÓN DINÁMICA DE LA MEMORIA

2d-9

```

73     printf("%g\n",V[i]);
74
75     printf("es el vector:\n");
76     for (i=0; i<N; i++)
77         printf("%g\n",MxV[i]);
78
79     printf("\n");
80
81     for (i=0; i<N; i++)
82         free(M[i]);
83     free(M);
84     free(V);
85     free(MxV);
86
87     return 0;
88
89 }
```

---

Del mismo modo que hemos utilizado “realloc()” para reasignar en tiempo de ejecución el tamaño en memoria de un vector, podemos hacer lo mismo con una matriz, aunque en este caso sólo se puede ampliar el número de filas ya que las matrices se almacenan en filas consecutivas de modo que no hay espacio “físico” para añadir más elementos a cada fila, es decir, para aumentar el número de columnas. En el siguiente listado mostramos cómo hacerlo en el caso de una matriz. Obsérvese que hemos omitido el *retipado*, confiando en que el compilador realizará la asignación correctamente (lo cual ocurre en la mayoría de los casos, aunque de la otra forma estaremos seguros).

**Listado 2d.3:** Programa que cambia el número de filas de una matriz en tiempo de ejecución.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define N 2 /* número de filas de la matriz */
5 #define S 3 /* número de columnas de la matriz */
6
7 int main(int argc, char **argv)
8 {
9     int i,j;
10
11     double **M;
12     /* reservamos memoria dinámica para la matriz de N x S números reales */
13     M=calloc(N, sizeof(double *));
14     for (i=0; i<N; i++)
15         M[i]=calloc(S, sizeof(double));
16
17     /* asignamos valores a los elementos de la matriz y los imprimimos por
18      pantalla */
19     for (i=0; i<N; i++)
{
```

## 2d-10 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```

20         for (j=0; j<S; j++)
21         {
22             M[i][j]=(i*S)+j; printf("%g\t",M[i][j]);
23         }
24         printf("\n");
25     }
26     printf("\n");
27
28     /* ahora ampliamos el número de filas en 2 unidades,
29     pasando a una matriz de (N+2)x(S) */
30     M=realloc(M,(N+2)*sizeof(double *));
31     for (i=N; i<N+2; i++)
32         M[i]=calloc(S, sizeof(double));
33
34     /* si imprimimos toda la matriz comprobaremos que mantiene los elementos
35     de la matriz anterior y que los nuevos elementos son 0 */
36     for (i=0; i<N+2; i++)
37     {
38         for (j=0; j<S; j++)
39             printf("%g\t",M[i][j]);
40         printf("\n");
41     }
42
43     for (i=0; i<N; i++)
44         free(M[i]);
45     free(M);
46
47     return 0;
48
49 }
```

---

**2d.2. Generación de números aleatorios: función rand()**

Los métodos de Monte Carlo (que veremos en el tema ??) se basan en la generación de números aleatorios, esto es, de números que siguen una secuencia desordenada e imposible de predecir exactamente aunque sí obedezcan una ley de probabilidad.

Para ilustrar lo que esto significa, el ejemplo más sencillo de proceso aleatorio es la tirada de un dado: esto genera números aleatorios entre 1 y 6. Son aleatorios, puesto que no se puede predecir cual será el resultado de una tirada antes de realizar esta. Sin embargo, si el dado no está “cargado”, uno esperaría que todos los números saliesen la misma cantidad de veces, cuando se han hecho muchas pruebas, esto es, siguiendo una ley de probabilidad.

La fracción de veces que se obtiene un resultado se denomina probabilidad de obtenerlo y las probabilidades de todos los posibles resultados (la distribución de probabilidad) es lo máximo que se puede conocer de un proceso aleatorio. En el ejemplo del dado, la probabilidad de que salga el número 5 será 1/6; igual para todos los demás resultados.

Matemáticamente, el resultado de la tirada es una variable aleatoria  $X$ , que puede tomar

## 2D.2. GENERACIÓN DE NÚMEROS ALEATORIOS: FUNCIÓN `RAND()`

2d-11

los valores 1, 2, 3, 4, 5, 6 (que es su espacio muestral). Diremos que existe una función de probabilidad  $P(x)$ , que asigna a cada elemento  $x$  del espacio muestral un valor de probabilidad: para nuestro dado,  $P(x) \equiv 1/6$ . Esta distribución se denomina uniforme, porque todas las probabilidades son iguales.

Otras distribuciones asignan probabilidades según una función que depende de  $x$ . Por ejemplo, la distribución de Poisson que describe la probabilidad de que  $x$  de núcleos de una muestra de  $N$  átomos se desintegren durante un tiempo fijo  $T$ , viene dada por

$$P(x) = \frac{\tau^x}{x!} e^{-\tau}$$

donde  $\tau = \lambda NT$  es el producto del tiempo considerado,  $T$ , por la constante de desintegración del isótopo radiactivo,  $\lambda$ , y por el número total de átomos en la muestra.

Dado que los ordenadores son máquinas “deterministas”, puesto que siguen un programa que determina el resultado de sus operaciones, se han desarrollado métodos que simulan este comportamiento aleatorio, que se busca, mediante secuencias de operaciones matemáticas. Los más sencillos, son los más utilizados porque, como se vio anteriormente, se necesita un gran número de lanzamientos para llegar a un resultado más o menos preciso de un problema. Entre los métodos más simples se encuentra el método congruente lineal, que se trata a continuación.

### 2d.2.1. Un programa que genera números aleatorios uniformemente distribuidos en el intervalo $[0, 1)$

El método congruente lineal está basado en el siguiente algoritmo recursivo

$$X_{n+1} = (AX_n + B) \bmod C$$

donde  $X_n$  es el valor del número (entero) aleatorio generado en el paso  $n$ -ésimo. Los parámetros enteros  $A$  y  $B$  definen la expresión lineal y el entero  $C$  es el módulo de congruencia. Para un valor inicial entero (o semilla),  $X_0$ , el algoritmo genera en cada paso un valor entero entre 0 y  $C - 1$ , que es el resultado de la función mod (resto de la división). Siempre que  $A$ ,  $B$  y  $C$  se elijan adecuadamente, se generará, partiendo de la semilla inicial, una sucesión de números enteros aleatorios uniformemente entre 0 y  $C - 1$ .

La implementación del método congruente lineal se muestra en el listado 2d.4.

En el preámbulo del programa se incluyen los dos archivos H estándar, `<stdio.h>` y `<stdlib.h>` (en el que se declara la función “exit()” que emplearemos para finalizar el programa cuando no se llame correctamente). También se #definen algunos literales que se emplean en el cálculo de los números aleatorios: los coeficientes de la transformación lineal (`LINEAL_A` y `LINEAL_B`), el módulo de congruencia (`CONGRUENCIA`) y la semilla, el número anterior en la serie al primer pseudoaleatorio.

El programa emplea aritmética entera, pero dado el orden de magnitud de los números, usaremos los enteros más grandes que pueda manejar nuestro procesador: los de tipo “`long long int`”. Por lo general, un sistema con un procesador de 16 bits tendrá “`long int`” de 16 bits y “`long long int`” de 32 bits; los sistemas actualmente más comunes, de 32 bits, tienen “`long int`” de 32 bits, y “`long long int`” de 64 bits. Elegimos un tipo con el máximo número de bits para que pueda albergar el entero resultante de multiplicar `LINEAL_A` por el número pseudoaleatorio anterior. (Haga la prueba de degradar la variable “`r`” a simple “`int`”).

## 2d-12 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

Como se ha visto, el primer número “*r*” se ha elegido “a ojo”: asignado como “SEMILLA”. Para evitar que nuestras preferencias por los dígitos sesguen nuestro generador pseudoaleatorio, se suele incluir una etapa de “calentamiento” (o termalización) del generador de números aleatorios, durante la que se ejecutan ciclos de cálculo de números pseudoaleatorios que se descartan. Después de este transitorio, el número que siga, aunque calculado determinísticamente a partir de nuestra elección inicial, tendrá poco en común con éste.

A partir del número entero calculado por el método de congruencias lineales, obtendremos un número aleatorio en el intervalo  $[0, 1)$  dividiendo por el valor de la variable de CONGRUENCIA. Llamaremos, a la distribución de probabilidad de estos números,  $U[0, 1)$ : la distribución uniforme en el intervalo  $[0, 1)$ .

Como resultado, el programa del listado 2d.4 proporciona  $N$  números pseudoaleatorios reales en el intervalo  $[0, 1)$ , tantos como se indiquen en el primer argumento de la línea de comandos<sup>3</sup>.

---

**Listado 2d.4:** El programa que genera números aleatorios por el método de congruencias lineales.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* período 4294967296 */
5 #define SEMILLA 1234567891LL
6 #define LINEAL_A 1664525LL
7 #define LINEAL_B 1013904223LL
8 #define CONGRUENCIA 4294967296LL
9
10 int main(int argc, char** argv)
11 {
12     int nnums, n;
13     /* el entero más largo que se pueda:
14        64 bits en sistemas de 32 */
15     long long int r;
16     double f;
17
18     if( argc!=2 )
19     {
20         printf("Uso del programa:\n %s <num aleatorios>\n",
21                argv[0]);
22         exit(0);
23     }
24
25     nnums=atoi(argv[1]);
```

---

<sup>3</sup>Todos los programas que requieran que se les pase algún valor como argumento de línea de comandos deben comprobar que dicho argumento se pasa efectivamente y, en caso contrario, informar al usuario del uso que debe hacer del programa, mediante un mensaje como:

Uso del programa:  
./aleatorios <num. aleatorios>

Que indica que el programa (que se ha llamado aleatorios) requiere un único argumento de línea de comandos que es el número de valores aleatorios que se quieren generar.

## 2D.2. GENERACIÓN DE NÚMEROS ALEATORIOS: FUNCIÓN RAND()

2d-13

<pre>/* período 509 */ #define SEMILLA 123L #define LINEAL_A 65L #define LINEAL_B 0L #define CONGRUENCIA 509L</pre>	<pre>/* período 32749 */ #define SEMILLA 12345L #define LINEAL_A 1944L #define LINEAL_B 0L #define CONGRUENCIA 32749L</pre>
<pre>/* período 2147483647 */ #define SEMILLA 12345678L #define LINEAL_A 16807L #define LINEAL_B 0L #define CONGRUENCIA 2147483647L</pre>	<pre>/* período 4294967296 */ #define SEMILLA 1234567891LL #define LINEAL_A 1664525LL #define LINEAL_B 1013904223LL #define CONGRUENCIA 4294967296LL</pre>

Tabla 2d.1: Posibles (buenas) definiciones de generadores de números pseudoaleatorios por congruencias lineales.

```

26     if(nnums<1)
27     {
28         nnums=1;
29     }
30
31     /* "calentamiento" */
32     r=SEMILLA;
33     for(n=0; n<2*nnums; n++)
34     {
35         /* recurrencia de congruencia-lineal */
36         r=(LINEAL_A*r+LINEAL_B) % CONGRUENCIA;
37     }
38
39     /* generación */
40     for(n=0; n<nnums; n++)
41     {
42         /* recurrencia de congruencia-lineal */
43         r=(LINEAL_A*r+LINEAL_B) % CONGRUENCIA;
44         f=(double)r/CONGRUENCIA;
45
46         printf("%g\n", f);
47     }
48
49     return 0;
50 }
```

Para modificar el generador de números pseudoaleatorios se pueden probar los cambios sugeridos en la tabla 2d.1.

**Ejercicio 4.1.** Modifíquese el programa 2d.4 creando una función “double uniforme()” que

retorne números reales (pseudo)aleatorios distribuidos según una distribución  $U[0, 1]$ . Úsese una variable global, definida fuera de “main()” (y de “uniforme()”)

```
/* el entero más largo que se pueda */
long long semilla_uniforme=SEMILLA;
```

para guardar el último número generado.

## 2d.2.2. Números aleatorios continuos: generación de variables (pseudo)aleatorias distribuidas gaussianamente

Los ejemplos que acabamos de ver corresponden a variables aleatorias discretas, es decir, aquellas cuyo espacio muestral está formado por un conjunto numerable de elementos. Cuando el espacio muestral de la variable un intervalo de números reales (por tanto, un conjunto no numerable), se dice que es una variable aleatoria continua.

El paso de una variable aleatoria discreta a una continua es análogo al paso de un sumatorio a una integral. El intervalo continuo se partitiona en un subintervalos (que ya forman un conjunto numerable) y se define sobre ello una distribución de probabilidad discreta. Luego, los subintervalos se hacen tan pequeños como se quiera.

A continuación vamos a ver el ejemplo de la distribución aleatoria continua más frecuente en la naturaleza: la distribución gaussiana.

La distribución gaussiana o normal es ubicua: aparece en, prácticamente, todos los problemas de las ciencias naturales. Esto se debe a una importante propiedad que cumple esta distribución, llamada el “Teorema Central del Límite”. Este teorema dice (simplificando):

**Teorema.** La variable aleatoria  $Y$  construida como la suma de  $N$  variables aleatorias  $X_i$  distribuidas, a su vez, según distribuciones de probabilidad con medias ( $\mu_i$ ) y varianzas ( $\sigma_i^2$ ) acotadas, sigue una distribución que se acerca a la de Gauss con media  $\mu$  el la suma de las medias, y varianza  $\sigma^2$  la suma de las varianzas, según  $N$  crece. Esta distribución de Gauss viene descrita por la densidad de probabilidad

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Usando este teorema, la Naturaleza “construye” variables aleatorias distribuidas gaussianamente, y usando este teorema, podemos construir un generador de variables (pseudo)aleatorias distribuidas gaussianamente (bastará que  $N > 30$ ).

El objetivo de este ejercicio es usar un generador de números (pseudo)aleatorios uniformemente distribuidos para construir un generador de números (pseudo)aleatorios gaussianamente distribuidos. Buscaremos que esta distribución sea la  $N(0, 1)$ , que tiene valor medio  $\mu = 0$  y varianza  $\sigma^2 = 1$ . Para ello, definiremos una función “double normal()”. En ella, un bucle calculará la suma de  $N_{\text{uniformes}}$  valores  $x_i$  de las variables  $X_i$  aleatorias uniformemente distribuidas en  $[0, 1]$

$$S = \sum x_i$$

Para terminar teniendo una distribución  $N(0, 1)$ , centraremos esta variable en el cero, restándole  $N_{\text{uniformes}}$  veces el valor medio de las  $X_i$  ( $\mu_i = 1/2$ ), y dividiendo el resultado por la raíz cuadrada de  $N_{\text{uniformes}}$  veces la varianza de las  $X_i$  ( $\sigma_i^2 = 1/12$ ):

$$y = \frac{\sum x_i - \frac{1}{2}N_{\text{uniformes}}}{\sqrt{\frac{1}{12}N_{\text{uniformes}}}}$$

---

**Listado 2d.5:** El programa con la función que genera números aleatorios normalmente distribuidos según  $N(0, 1)$ .

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* período 2147483647 */
5 #define SEMILLA      12345678L
6 #define LINEAL_A     16807L
7 #define LINEAL_B     0L
8 #define CONGRUENCIA 2147483647L
9
10 /* Definimos el número de variables U(0, 1)
11 * para construir la variable gaussiana */
12 #define NUNIFORMES 100
13
14 /* el entero más largo que se pueda:
15   64 bits en sistemas de 32 */
16 long long semilla_normal=SEMILLA;
17
18 double normal()
19 {
20     int n;
21     double f, s, g;
22
23     s=0.0;
24     for(n=0; n<NUNIFORMES; n++)
25     {
26         semilla_normal=( LINEAL_A*semilla_normal
27                         +LINEAL_B) % CONGRUENCIA;
28         f=(double)semilla_normal/CONGRUENCIA;
29         s=s+f;
30     }
31     g=(s-0.5*NUNIFORMES)/sqrt((1.0/12)*NUNIFORMES);
32
33     return g;
34 }
35
36 int main(int argc, char** argv)
37 {
38     int nnums, n;
39     double f;
40
41     if( argc!=2 )
42     {
43         printf("Uso del programa:\n %s <num aleatorios>\n",
44                argv[0]);
45         exit(0);
```

## 2d-16 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```

46     }
47
48     nnums=atoi(argv[1]);
49     if(nnums<1)
50     {
51         nnums=1;
52     }
53
54     /* "calentamiento" */
55     for(n=0; n<2*nnums; n++)
56     {
57         (void)normal();
58     }
59
60     /* generación */
61     for(n=0; n<nnums; n++)
62     {
63         f=normal();
64
65         printf("%g\n", f);
66     }
67
68     return 0;
69 }
```

---

### 2d.2.3. Obtención de la distribución de probabilidad

Para comprobar que las funciones anteriores “double uniform()” y “double normal()” distribuyen sus valores uniformemente y según una distribución  $N(0, 1)$ , respectivamente, hay que construir un histograma con los valores.

Un histograma es una representación gráfica de la frecuencia de ocurrencias de los resultados por intervalos en que se divide el espacio muestral. Por ejemplo, para una variable aleatoria continua que varía entre 0 y 1, si se divide este intervalo en 10 subintervalos disjuntos (pero no necesariamente de igual longitud), el histograma será una representación como la de la figura 2d.1. En esa representación, el área de cada rectángulo es proporcional al número de resultados (o a la frecuencia) en ese intervalo. Si la distribución hubiese sido uniforme, el histograma de la figura sería plano: cada rectángulo tendría la misma altura.

Este ejercicio consiste en, partiendo de los datos en un archivo de datos numéricicos, calcular el histograma de su distribución. Para ello, se leerán los valores de la variable a estudiar desde un archivo.

Por comodidad, se creará una función “int leeNumeros(char\* arch, double x[])” que recibe el nombre del archivo de datos y un “array” de números reales, que se va a llenar; el número de valores que guarde en él la función será el que retorne ésta (así que, si no se pudiera leer ningún dato, el valor sería 0). Para leer los valores de la variable aleatoria  $x$  se usará la función “fscanf”, y para saber cuándo se acaba de leer el fichero, se usa el hecho de que esta función retorna el número de valores que ha sido capaz de leer desde el fichero: la condición de final de archivo se traduce en que “fscanf” no lea ningún dato. Por otro lado, el

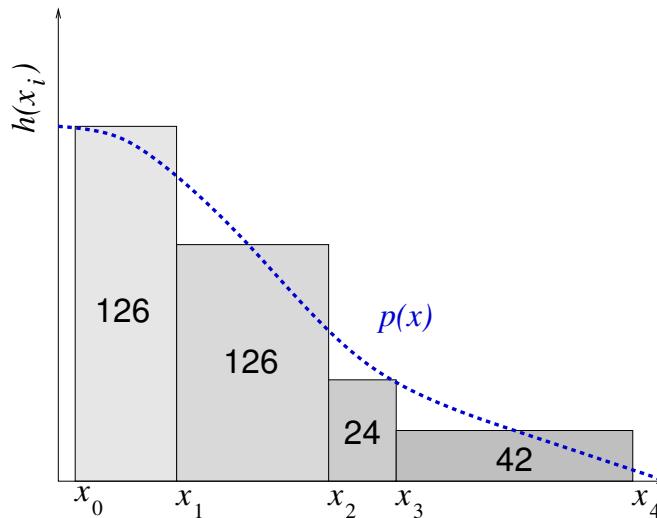


Figura 2d.1: Ejemplo de histograma  $h(x_i)$  comparado con la función de densidad de probabilidad  $p(x)$ .

tamaño del vector a llenar será finito; para ello #definimos NMAXPTS (con un valor de 10000): el tamaño reservado para el array, que no se deberá rebasar.

La función que calcule el histograma será “int histograma(int npts, double x[], int nhist, int histograma[], double x\_min, double x\_max)”. Para el cálculo necesita los valores de la variable aleatoria  $x$  (su número, “npts”, y sus valores, “x[]”), también el intervalo que contiene los valores de  $x$  (el intervalo  $[x_{\min}, x_{\max}]$ ), el número de subintervalos en que se dividirá éste ( $n_{\text{hist}}$ ) y el array en el que se retornará el histograma. El procedimiento para calcularlo es sencillo: cada valor de  $x$  hace incrementar la cuenta de la “caja”  $i$ -ésima del histograma

$$i = \text{floor} \left( n_{\text{hist}} \frac{x - x_{\min}}{x_{\max} - x_{\min}} \right)$$

donde  $\text{floor}(\xi)$  representa la parte entera de  $\xi$ , y el valor de  $i$  es un entero entre 0 y  $n_{\text{hist}} - 1$ , esto es, que puede corresponder a cualquiera de las  $n_{\text{hist}}$  “cajas” del array del histograma.

---

**Listado 2d.6:** El programa que lee valores de un archivo y genera el histograma de su distribución.

---

```

1 #include <stdio.h>
2 #include <math.h>
3
4 #define X_MIN 0.0
5 #define X_MAX 1.0
6 #define NHISTOGRAMA 100
7
8 #define NMAXPTS 10000
9
10 int leeNumeros(char* arch, double x[])
11 {
12     int n, npts;
13     double xa;
```

## 2d-18 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```
15     FILE *fin;
16
17     fin=fopen(arch, "r");
18
19     npts=0;
20     do
21     {
22         n=fscanf(fin, "%lf", &xa);
23         if( n==1 && npts<NMAXPTS )
24         {
25             x[npts]=xa;
26         }
27         npts++;
28
29     } while( n==1 && npts<=NMAXPTS );
30
31     fclose(fin);
32
33     /* error */
34     if( npts==NMAXPTS+1 && n==1 )
35     {
36         npts=-1;
37     }
38
39     return npts;
40 }
41
42 void histograma(int npts, double x[],
43                  int nhist, int histo[],
44                  double x_min, double x_max)
45 {
46     int i, n;
47
48     for(i=0; i<nhist; i++)
49     {
50         histo[i]=0;
51     }
52
53     for(n=0; n<npts; n++)
54     {
55         i=(int)floor(
56             (nhist*(x[n]-x_min))
57             /(x_max-x_min) );
58         if( 0<=i && i<NHISTOGRAMA )
59         {
60             histo[i]++;
61         }
62     }
63 }
```

```

63
64     return;
65 }
66
67 int main(int argc, char** argv)
68 {
69     int nAleatorios, i;
70     double aleatorios[NMAXPTS], dx, x, f;
71     int histo[NHISTOGRAMA];
72
73     nAleatorios=leeNumeros("aleatorios.dat",
74                           aleatorios);
75
76     histograma(nAleatorios, aleatorios,
77                 NHISTOGRAMA, histo,
78                 X_MIN, X_MAX);
79
80     dx=(double)(X_MAX-X_MIN)/NHISTOGRAMA;
81
82     for(i=0; i<NHISTOGRAMA; i++)
83     {
84         x=X_MIN+(double)i*dx;
85         f=(double)histo[i]/(nAleatorios*dx);
86
87         printf("%g\t%d\t%g\n",
88                x, histo[i], f);
89     }
90
91     return 0;
92 }
```

---

**Ejercicio 4.2.** Comprobar que el histograma de una variable aleatoria  $X$  distribuida según una distribución uniforme en el intervalo  $[0, 1]$  (listado 2d.4) es, aproximadamente, constante. Obsérvese cómo este resultado es más aproximado cuantos más valores de la variable aleatoria se consideran en el cálculo.

**Ejercicio 4.3.** Comprobar que el histograma de una variable aleatoria  $X$  distribuida según una distribución normal  $N(0; 1)$  (listado 2d.5) viene dado, aproximadamente, por la expresión

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right)$$

que es la distribución gaussiana de varianza  $\sigma^2 = 1$  y valor medio  $\mu = 0$ . Obsérvese, de nuevo, cómo este resultado es más aproximado cuantos más valores de la variable aleatoria se consideran en el cálculo.

## 2d.2.4. La función rand()

Acabamos de ver cómo podemos generar números enteros pseudoaleatorios mediante el método congruente lineal, a partir de los cuales podemos obtener números reales también pseudoaleatorios uniformemente distribuidos en el intervalo  $[0, 1)$ . En la práctica esto es muy importante ya que es posible muestrear cualquier distribución de probabilidad, es decir, es posible obtener números aleatorios distribuidos según una cierta función de probabilidad, utilizando exclusivamente números aleatorios con una distribución uniforme  $U[0, 1)$ . Bastará con aplicar la transformación adecuada. Hemos visto un ejemplo de ello con la obtención de números distibuidos *normalmente* (es decir, según la distribución gaussiana).

Afortunadamente no va ser necesario programar el método congruente lineal cuando queramos generar números aleatorios (lo cual ocurre con mucha frecuencia) pues el lenguaje C cuenta con una función que ya lo hace por nosotros, la función “rand()”, declarada en la librería “`stdlib.h`” del siguiente modo:

```
int rand(void)
```

Esta función no tiene argumentos y devuelve un número entero positivo pseudoaleatorio uniformemente distribuido en el intervalo  $[0, \text{RAND\_MAX}]$  (obsérvese que se incluyen los dos extremos del intervalo), donde `RAND_MAX` es una macro del C definida en “`cstdlib.h`” que representa el mayor valor generado por “`rand()`”. `RAND_MAX` es por tanto una constante numérica y su valor depende de las librerías empleadas, pero como mínimo es de 32767. En Linux suele ser `RAND_MAX`=  $2^{31} - 1 = 2147483647$  mientras que en Windows es habitual encontrarse con el valor anterior: `RAND_MAX`= 32767. Para conocer el valor de `RAND_MAX` podemos escribir:

```
printf("El valor de RAND_MAX en nuestro sistema es: %d\n", RAND_MAX);
```

Cada llamada a “`rand()`” devolverá, por tanto, un “`int`” dentro del intervalo anterior y con la misma probabilidad. Al igual que en el método congruente lineal, previamente se tiene que haber definido (al comienzo de la función “`main`”, por ejemplo) la semilla del generador de números aleatorios que emplea C. Es lo que se denomina *inicializar el generador de números aleatorios*. Esto se hace mediante la función “`srand()`”, declarada también en “`stdlib.h`” con la siguiente sintaxis:

```
void srand (unsigned int semilla)
```

Si esta “semilla” se ha definido a partir de una constante entera (p. ej. si escribimos “`srand(20)`”), cada vez que se ejecute el programa la secuencia de números pseudoaleatorios generada por “`rand()`” será siempre la misma. Esto es debido a que se parte siempre de la misma semilla y los números no son estrictamente aleatorios. Por lo tanto, podemos reproducir la misma secuencia de números pseudoaleatorios (por grande que sea) cuantas veces queramos, lo cual puede ser útil en estudios comparativos. Como los valores generados por “`rand()`” están limitados por `RAND_MAX`, si llamamos a “`rand()`” un número suficientemente grande de veces llegará un momento en el que se repetirá el primer valor generado (que no se corresponde con “semilla”), y a partir de ahí obtendremos la misma secuencia. Al número de valores que hay que generar hasta que se repita un cierto valor se le denomina *periodo* del generador. Este periodo dependerá de las propiedades matemáticas del método utilizado y del propio valor inicial, y está acotado superiormente por `RAND_MAX`. El hecho de utilizar generadores con periodos bajos –como los que nos encontramos cuando se trabaja en Windows– puede provocar sesgos numéricos que afecten dramáticamente a los resultados. Cuando se trabaja en Física Computacional es muy importante tener esto presente.

Para evitar tener que trabajar siempre con la misma secuencia de números pseudoaleatorios podemos definir esa semilla a partir de una variable que cambie con el tiempo, de modo que cada vez que sea llamada obtengamos una semilla diferente. Esto se consigue escribiendo:

```
rand(time(NULL));
```

De este modo, la semilla del generador de números aleatorios estará dada por el tiempo (en segundos) que marca el reloj de la CPU.<sup>4</sup> Por lo tanto, cada vez que se ejecute el programa la semilla será distinta y también lo será la secuencia de números pseudoaleatorios generados. Es importante señalar que esto será así siempre que el intervalo entre dos inicializaciones consecutivas sea mayor que 1 s. Si no lo es, obtendremos la misma semilla y por tanto la misma secuencia de números. Este también es otro aspecto muy importante a tener en cuenta cuando se realizan simulaciones computacionales. Para poder hacer uso de la función “time()” hemos de incluir la línea “#include <time.h>”.

A continuación mostramos dos ejemplos. En el primero de ellos el programa imprime en pantalla los diez primeros números pseudoaleatorios de la secuencia generada a partir de la semilla 20. Podemos comprobar que cada vez que se ejecuta el programa, la secuencia obtenida es la misma. En el segundo, el programa realiza la misma tarea con la diferencia de que la semilla cambia dinámicamente, por lo que cada vez que se ejecuta el program la secuencia varía (siempre que haya pasado más de un segundo entre cada ejecución).

---

**Listado 2d.7:** Programa que genera números pseudoaleatorios con una semilla fija.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main (int argc, char** argv)
5 {
6     int i, semilla=20;
7     srand(semilla);
8     for (i=0; i<10; i++)
9         printf("%d,\t%d\n", i, rand());
10
11    return 0;
12 }
```

---

**Listado 2d.8:** Programa que genera números pseudoaleatorios con semilla dependiente del tiempo.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main (int argc, char** argv)
6 {
```

<sup>4</sup>Concretamente, la llamada “time(NULL)” devuelve el número de segundos que han pasado desde el 1 de enero de 1970 (UTC) según el reloj del sistema. Este valor es devuelto en forma de un tipo concreto de variable denominado como “time\_t”. Todas las funciones, variables y macros relacionadas con el tiempo están incluidas en la librería “time.h”.

## 2d-22 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```

7     int i;
8     srand(time(NULL));
9     for (i=0; i<10; i++)
10        printf(" %d,\t%d\n", i, rand());
11
12    return 0;
13 }
```

---

A partir de “rand()” podemos generar números aleatorios con otro tipo de distribución de probabilidad. Como hemos comentado anteriormente, es especialmente importante poder generar números reales uniformemente distribuidos en el intervalo  $[0, 1)$ . Esto lo conseguimos haciendo

$$\frac{(\text{double}) \text{rand}()}{\text{RAND\_MAX} + 1}$$

La sintaxis “(double)rand()” representa un *retipado* del resultado de “rand()” para que lo trate como un “double”, y así haga la división con decimales. De lo contrario, al ser el numerador y el denominador enteros, C realizaría la división entera dando como resultado 0. También podríamos haber considerado simplemente “ $1,0 * \text{rand}() / (\text{RAND\_MAX} + 1)$ ” ó “ $\text{rand}() / (\text{RAND\_MAX} + 1,0)$ ”.

## 2d.2.5. Ejemplos

Vamos a comenzar los ejemplos con un código que, dada una semilla introducida por consola, calcula el número de iteraciones necesarias para que se repita el primer valor aleatorio obtenido a partir de la semilla (para información del usuario también imprime por pantalla el valor de “RAND\_MAX”). Además, imprime los cinco primeros números aleatorios obtenidos, y los cinco siguientes después de que se produzca la repetición. Si ejecutamos este programa utilizando diferentes semillas observaremos algunas propiedades interesantes del generador “rand()”. Por ejemplo, podemos comprobar que la secuencia de números obtenidos después de la repetición no coincide con la secuencia inicial. Esto quiere decir que el número de iteraciones necesarias hasta obtener la primera repetición no representa el periodo de este generador, ya que el periodo de un generador se define como el número de iteraciones necesarias hasta que se vuelve a repetir *toda* la secuencia. Esta es una diferencia importante con respecto al método congruente lineal, en el que ambas secuencias forzosamente coincidirían y el número de iteraciones hasta la repetición sí que representa el periodo del generador. Otro resultado interesante es que el número de iteraciones necesarias hasta encontrar la primera repetición puede ser muy superior a “RAND\_MAX”.

---

**Listado 2d.9:** Programa que calcula el número de iteraciones necesarias para que se repita el primer valor aleatorio obtenido después de una semilla concreta, e imprime en pantalla los 5 primeros números y los 5 siguientes después de la repetición.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     int i, ciclo, semilla, aleat, aleat_ini;
7
```

## 2D.2. GENERACIÓN DE NÚMEROS ALEATORIOS: FUNCIÓN RAND()

2d-23

```

8   printf("Introduzca la semilla:");
9   scanf("%d",&semilla);
10  srand(semilla);

11
12  printf("\nEl valor de RAND_MAX es: %d\n",RAND_MAX);

13
14  aleat_ini=rand();

15
16  printf("\nLos primeros 5 numeros aleatorios generados despues de la semilla
17    son:\n");
18  printf("%d\n",aleat_ini);
19  ciclo=0;
20  do
21  {
22      aleat=rand();
23      ciclo++;
24      if (ciclo<5) printf("%d\n",aleat);
25  }
26  while(aleat!=aleat_ini);

27  printf("\nEl primer valor aleatorio %d se repite despues de %d iteraciones \
28    \n\n",aleat_ini,ciclo);

29  printf("Los primeros 5 numeros aleatorios despues de la repeticion (incluida
30    son:\n");
31  printf("%d\n", aleat);
32  for(i=1; i<=4; i++)
33  {
34      printf("%d\n",rand());
35  }

36
37  return 0;
38
39 }

```

A continuación mostramos un código que genera una matriz A compuesta de unos y ceros de forma aleatoria, es decir, cada elemento de la matriz tiene la misma probabilidad de ser 1 que de ser 0. Para ello basta con asignar a cada elemento de la matriz el resultado de la siguiente comparación:

$$A[i][j] = \frac{(\text{double})\text{rand}()}{\text{RAND\_MAX} + 1} < 0,5$$

El tamaño de la matriz y el nombre del archivo en el que se van a guardar los datos son introducidos por línea de comandos.

---

**Listado 2d.10:** Programa que genera una matriz aleatoria de 0s y 1s.

---

```

1 #include <stdio.h>
2 #include <stdlib.h>

```

## 2d-24 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```

3 #include <time.h>
4
5 int main(int argc, char** argv)
6 {
7     int i,j,N,M;
8     FILE *file;
9
10    N=atoi(argv[1]); // número de filas
11    M=atoi(argv[2]); // número de columnas
12    int A[N][M];
13
14    file=fopen(argv[3], "wb");
15
16    srand(time(NULL));
17
18    /* rellenamos la matriz con 0s y 1s de forma aleatoria */
19    for (i=0; i<N; i++)
20    {
21        for (j=0; j<M; j++)
22        {
23            A[i][j]=(double)rand()/(RAND_MAX+1)<0.5;
24            fprintf(file,"%d ",A[i][j]);
25        }
26        fprintf(file,"\n");
27    }
28
29    fclose(file);
30    return 0;
31 }
```

---

**Ejercicio 4.4.** Modificar el programa anterior para que la probabilidad de que cada elemento de la matriz sea 1 es  $p$ , y  $1 - p$  de que sea 0, con  $p$  introducida por línea de comandos.

**Ejercicio 4.5.** Modificar el programa anterior para que los elementos de la matriz puedan tomar los valores 0, 1 ó 2, con probabilidades  $p$ ,  $q$  y  $1 - p - q$ , respectivamente (teniendo en cuenta que  $p + q \leq 1$ ), donde  $p$  y  $q$  son introducida por línea de comandos.

El objetivo del próximo código es obtener números *enteros* pseudoaleatorios uniformemente distribuidos en el intervalo  $[a, b]$ , siendo ambos enteros. Este objetivo se puede conseguir de muchas maneras. La más sencilla e intuitiva es

$$a + \text{rand}() \% (b-a+1)$$

donde hemos utilizado la *operación módulo “%”* que proporciona el resto de la división entre dos enteros (p. ej.  $7\%2=1$ ). Por lo tanto, el resultado de “ $\text{rand}() \% (b-a+1)$ ” será un entero en el intervalo  $[0, b-a]$ . Esta fórmula proporcionará una distribución uniforme dentro del intervalo siempre que el tamaño del intervalo sea mucho menor que el máximo valor devuelto por “ $\text{rand}()$ ”, es decir, siempre que  $(b-a+1) << \text{RAND\_MAX}$ . En caso contrario los valores más bajos

## 2D.2. GENERACIÓN DE NÚMEROS ALEATORIOS: FUNCIÓN RAND()

2d-25

del intervalo serán más probables. Podemos entender esto con un ejemplo muy sencillo. Supongamos que `RAND_MAX=10` y que nuestro intervalo es  $[0, 7]$ . Entonces los valores 0, 1 y 2 tendrán más probabilidad de aparecer que el resto de valores: 2, 4, 5, 6, 7.

La forma de “repartir” los valores generados por “`rand()`” lo más uniformemente posible en el intervalo de enteros  $[a, b]$  es mediante:

$$a + (\text{int}) \left( \frac{1,0 \times (b-a+1) \times \text{rand}()} {\text{RAND\_MAX} + 1,0} \right)$$

donde se ha procedido de nuevo a un “retipado” para que el resultado del segundo término (entre paréntesis) sea un entero. A continuación mostramos el código para este segundo método. Los extremos del intervalo, el número de números aleatorios que se quieren generar y el nombre del archivo de datos en el que se van a guardar serán introducidos por línea de comandos:

---

**Listado 2d.11:** Programa que genera números enteros aleatorios distribuidos uniformemente en el intervalo  $[a, b]$ .

---

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main(int argc, char** argv)
6 {
7     int a, b, Nvalores, i;
8     FILE *file;
9
10    a=atoi(argv[1]); // extremo inferior del intervalo
11    b=atoi(argv[2]); // extremo superior del intervalo
12    Nvalores=atoi(argv[3]); // número de valores a generar
13
14    file=fopen(argv[4] , "wb");
15
16    srand(time(NULL));
17
18    /* generamos los valores y los exportamos al archivo */
19    for (i=0; i<Nvalores; i++)
20        fprintf(file, "%d\n", a+(int)((1.0*(b-a+1)*rand())/(RAND_MAX+1.0)));
21
22    fclose(file);
23    return 0;
24 }
```

---

**Ejercicio 4.6.** Modificar el programa anterior para que implemente el método basado en la operación módulo.

**Ejercicio 4.7.** Utilizar el código que calcula el histograma de un conjunto de valores para comparar el resultado de ambos métodos. Hacer esto en dos casos:  $(b-a+1) \approx \text{RAND\_MAX}/1000$  y  $(b-a+1) \approx \text{RAND\_MAX}/5$ . Para simplificar el análisis podemos trabajar con un `RAND_MAX`

efectivo más pequeño que el utilizado por el generador (definámolo p. ej. como `RAND_MAX_eff`). Para ello bastará con descartar los valores generados por `rand()` que sean mayores que `RAND_MAX_eff`.

## 2d.3. Bibliotecas de funciones

Como hemos visto, el lenguaje C cuenta con una *biblioteca estándar* compuesta por archivos de cabecera<sup>5</sup> en los que se declaran las funciones estándar del C y se definen algunos tipos de datos, constantes numéricas y otras macros, y por un conjunto de librerías que incluyen rutinas en las que se implementan estas funciones.

Sin embargo, el lenguaje C también nos permite crear nuestras propias bibliotecas, las cuales normalmente estarán compuestas por un fichero “.h” con las *declaraciones* de las funciones (también denominadas *prototipos*), que nos indica su tipo y los argumentos que reciben, y por un fichero “.c” con sus *definiciones*, es decir, con el código que desarrolla las tareas que realizan. En este punto pueden surgir las siguientes preguntas: ¿qué ventajas tiene esa estructura frente a escribir directamente el código de las funciones en el archivo “.h”, e incluir después este archivo en mi programa? O también, ¿qué necesidad tengo de crearme una biblioteca e incluirla en mi programa si puedo “incluir” esas funciones copiándolas directamente en mi código? Al fin y al cabo, durante el preprocesamiento, el preprocesador básicamente “sustituye” en el programa todo el código contenido en los archivos de cabecera.

Hay varias razones que justifican esta estructura y el esfuerzo que puede suponer crear estas bibliotecas. Esas razones responden a necesidades fundamentales en programación.

- Claridad y portabilidad. Si en algún momento tenemos que usar las bibliotecas de funciones creadas por otro programador, normalmente no nos interesaría tanto cómo las ha implementado sino cómo utilizarlas, y para ello sólo necesitamos la información que aparecerá en el “.h” (en ese sentido también se agradecerá que haya comentarios sobre cómo llamarlas).
- Privacidad y seguridad. Si por ejemplo trabajamos profesionalmente escribiendo bibliotecas de funciones, puede que queramos dejar que otro usuario las utilice en sus proyectos sin que conozca el detalle de su implementación. En ese caso le daremos el archivo “.h” y compilaremos como un objeto o una biblioteca de enlace dinámico el módulo donde están definidas las funciones, o sea, el archivo “.c”. Esto es muy frecuente con los controladores de dispositivos, p. ej. de tarjetas de adquisición de datos usadas en el laboratorio, que se suelen distribuir como archivos “.so” o como “.dll” junto con el archivo “.h”, y todo ello junto con el hardware.
- Eficiencia y coste computacional. Supongamos que las definiciones de las funciones incluyen miles de línea de código, por lo que su compilación tardará mucho tiempo. En ese caso resulta mucho más eficiente que las definiciones de las funciones vayan de modo separado en otro archivo “.c”, que además puede ser compilado una única vez generando un objeto “.o” que puede ser añadido a cualquier programa sin tener que ser compilado de nuevo. Un problema similar puede surgir cuando nuestro programa contenga varios módulos. Si hubiese que compilar las funciones de la biblioteca (incluidas

<sup>5</sup>Los archivos con extensión “.h” que incluimos en la cabecera de los códigos mediante la instrucción “#include”, p. ej. “stdio.h”, “stdlib.h”, “math.h”, etc.

## 2D.3. BIBLIOTECAS DE FUNCIONES

2d-27

desde el “.h”) al compilar cada módulo, se tardaría mucho tiempo(además de los problemas de símbolos duplicados que seguramente surgirían en el enlazado final).

Por todo esto concluimos que:

1. Es mejor mantener las dos partes separadas: el “.h” para las declaraciones y las instrucciones de uso, y el “.c” para la implementación de las funciones en C.
2. Es muy recomendable recopilar en forma de bibliotecas todas aquellas funciones generales que puedan ser utilizadas por diferentes programas.

Supogamos que hemos creado la biblioteca “mibiblioteca” compuesta por los archivos “mibiblioteca.h” y “mibiblioteca.c”. Ahora podemos compilarla generando un objeto “.o” mediante:

```
gcc -c -o mibiblioteca.o mibiblioteca.c
```

Como hemos comentado anteriormente, esto nos ahorrá tiempo ya que una vez compilada, el objeto “mibiblioteca.o” puede ser añadido a cualquier programa sin tener que ser compilado de nuevo.

Supongamos ahora que hemos implementado un código (“micodigo.c”) en el que queremos usar funciones de esa biblioteca. En primer lugar deberemos incluir la biblioteca en nuestro código del siguiente modo:

```
#include <stdlib.h>
#include <stio.h>
...
#include "mibiblioteca.h"
...
```

Esto es válido cuando la librería se encuentra en nuestro directorio de trabajo, es decir, en el mismo directorio en el que se encuentra el código fuente y desde el que estamos compilando y ejecutando el programa. Si no es así, dentro de las comillas deberemos indicar el path del archivo. Por ejemplo en Windows puede ser:

```
#include <stdlib.h>
#include <stio.h>
...
#include "C:\Mis_estudios\Grado_en_Fisica\FCI\...\mibiblioteca.h"
...
```

Ahora ya sólo tenemos que compilar el código, añadiendo, antes del nombre del programa que se quiere compilar, el nombre del objeto de la biblioteca:

$\underbrace{\text{gcc } -\text{o } \text{micodigo}}_{\text{ejecutable}} \underbrace{-\text{lm}}_{\text{biblio. matemat.}} \underbrace{\text{mibiblioteca.o}}_{\text{nuestra biblioteca}} \underbrace{\text{micodigo.c}}_{\text{programa fuente C}}$

Si no hubiéramos creado previamente el objeto “mibiblioteca.o”, podemos compilar a la vez la biblioteca y el código fuente del siguiente modo:

```
gcc -o micodigo -lm mibiblioteca.c micodigo.c
```

En esta sección vamos a ver dos ejemplos, una biblioteca de funciones relacionadas con la generación y análisis de números aleatorios, que denominaremos “libprobabilidad”, y otra biblioteca para crear imágenes en escala de grises a partir de matrices de datos, denominada “libguardaimagen”.

## 2d-28 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

**2d.3.1. La biblioteca libprobabilidad.h**

En los ejemplos y ejercicios anteriores, habremos programado una serie de funciones que hacen uso de números (pseudo)aleatorios o que ayudan a analizarlos. Estas funciones son:

- double normal()
- double uniforme()
- int histograma(int npts, double x[], int nhist, int histograma[], double x\_min, double x\_max)

Guardaremos el código de estas funciones en un archivo, que llamaremos “libprobabilidad.c”, y sus declaraciones en un archivo H, que llamaremos “libprobabilidad.h”. Los códigos de ambos se pueden ver en los listados 2d.12 y 2d.13.

Es de destacar la presencia de “#ifndef”. Esta directiva del preprocesador comprueba si está #definido el símbolo que sigue y, si no lo está, deja que el preprocesador procese y pase al compilador todo el contenido del archivo hasta la directiva “#endif”. Del mismo modo existe la directiva “#ifdef”, que hace justo lo contrario: procesa sólo si está #definido el símbolo del preprocesador.

En los archivos H, se utiliza este “truco” de comprobar que un símbolo no ha sido #definido para evitar #incluir el archivo H varias veces y evitar así, tanto el tiempo de procesamiento, como los errores que ello causaría.

---

**Listado 2d.12:** Las declaraciones de funciones en la biblioteca de probabilidad (libprobabilidad.h).

---

```

1 #ifndef _LIBPROBABILIDAD_H_
2 # define _LIBPROBABILIDAD_H_
3
4 /* período 509
5 #define SEMILLA    123L
6 #define LINEAL_A   65L
7 #define LINEAL_B   0L
8 #define CONGRUENCIA 509L
9 */
10
11 /* período 32749
12 #define SEMILLA    12345L
13 #define LINEAL_A   1944L
14 #define LINEAL_B   0L
15 #define CONGRUENCIA 32749L
16 */
17
18 /* período 2147483647
19 #define SEMILLA    12345678L
20 #define LINEAL_A   16807L
21 #define LINEAL_B   0L
22 #define CONGRUENCIA 2147483647L
23 */
24
```

## 2D.3. BIBLIOTECAS DE FUNCIONES

2d-29

```

25 /* periodo 4294967296 */
26 #define SEMILLA 1234567891LL
27 #define LINEAL_A 1664525LL
28 #define LINEAL_B 1013904223LL
29 #define CONGRUENCIA 4294967296LL
30
31 #define NUNIFORMES 100
32 #define NHISTOGRAMA 1000
33
34
35 double normal();
36 double uniforme();
37 void histograma(int npts, double x[],
38                  int nhist, int histo[],
39                  double x_min, double x_max);
40 #endif

```

---

**Listado 2d.13:** Las definiciones de las funciones de la biblioteca de probabilidad (libprobabilidad.c).

```

1 #include <math.h>
2 #include "libprobabilidad.h"
3
4
5 static
6 long long semilla_normal=SEMILLA;
7
8 double normal()
9 {
10     int n;
11     double f, s, g;
12
13     s=0.0;
14     for(n=0; n<NUNIFORMES; n++)
15     {
16         semilla_normal=(LINEAL_A*semilla_normal+LINEAL_B)
17                         % CONGRUENCIA;
18         f=(double)semilla_normal/CONGRUENCIA;
19         s=s+f;
20     }
21     g=(s-0.5*NUNIFORMES)/sqrt((1.0/12)*NUNIFORMES);
22
23     return g;
24 }
25
26
27 static
28 long long semilla_uniforme=SEMILLA;

```

```

2d-30    TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

29
30 double uniforme()
31 {
32     double f;
33     semilla_uniforme=(LINEAL_A*semilla_uniforme+LINEAL_B) % CONGRUENCIA;
34
35     f=(double)semilla_uniforme/CONGRUENCIA;
36
37     return f;
38 }
39
40
41
42 void histograma(int npts, double x[],
43                  int nhist, int histo[],
44                  double x_min, double x_max)
45 {
46     int i, n;
47
48     for(i=0; i<nhist; i++)
49     {
50         histo[i]=0;
51     }
52
53     for(n=0; n<npts; n++)
54     {
55         i=(int)floor(
56             (nhist*(x[n]-x_min))
57             /(x_max-x_min) );
58         if( 0<=i && i<NHISTOGRAMA )
59         {
60             histo[i]++;
61         }
62     }
63
64     return;
65 }

```

---

**Nota:** Para ahorrar tiempo, lo mejor es, compilar la biblioteca libprobabilidad como

```
gcc -c -o libprobabilidad.o libprobabilidad.c
```

Cada vez que sea necesaria una función de la biblioteca (y se use `#include "libprobabilidad.h"` por ello), habrá que añadir, antes del nombre del programa que se quiere compilar, el nombre del objeto de la biblioteca. Así, por ejemplo, compilaremos el próximo programa, “browniano1d.c” como

```
gcc -o browniano1d -lm libprobabilidad.o browniano1d.c
```

ejecutable
biblio. matem.
biblio. probabilidad
programa fuente C

### 2d.3.2. La biblioteca libguardaimagen.h

En la sección 2c.11 del capítulo anterior, se presentó en el Listado 2c.15 la función “guardaPGMd()”, que construye y guarda una imagen en escala de grises (formato PGM) a partir de una matriz de valores reales que es pasada por referencia. Este tipo de datos y de imágenes son muy habituales en programación, así que nos proponemos crear una librería de funciones que nos permitan pasar de matrices de datos a imágenes en formato PGM. La llamaremos “libguardaimagen” y estará compuesta de dos archivos: el archivo “libguardaimagen.h” que contiene las declaraciones de las funciones, y el archivo “libguardaimagen.c” con las definiciones. Respecto a las funciones, consideraremos la propia función “guardaPGMd()” que recibe una matriz de números reales (tipo “double”), y una versión adaptada para matrices de enteros (tipo “int”), que denominaremos “guardaPGMi()”. A continuación mostramos los códigos de los dos archivos.

---

**Listado 2d.14:** Archivo libguardaimagen.h. Contiene las declaraciones de las funciones que guardan una imagen en un archivo con formato PGM.

---

```

1 #ifndef _LIBGUARDAIMAGEN_H_
2 # define _LIBGUARDAIMAGEN_H_
3
4 #include <stdio.h>
5
6 /* Guarda en el archivo de nombre dado una imagen
7  * PGM de dimensiones anchura X altura que contiene en
8  * sus píxeles valores enteros entre pixel_min y pixel_max
9  */
10 void guardaPGMi(char* nombre, int anchura, int altura,
11                  int *pixels, int pixel_min, int pixel_max);
12
13 /* Guarda en el archivo de nombre dado una imagen
14  * PGM de dimensiones anchura X altura que contiene en
15  * sus píxeles valores reales entre pixel_min y pixel_max
16  */
17 void guardaPGMd(char* nombre, int anchura, int altura,
18                  double *pixels, double pixel_min, double pixel_max);
19
20#endif

```

---

**Listado 2d.15:** Archivo libguardaimagen.c. Contiene las definiciones de las funciones que guardan una imagen en un archivo con formato PGM.

---

```

1 #include <stdio.h>
2
3 void guardaPGMi(char* nombre, int anchura, int altura,
4                  int *pixels, int pixel_min, int pixel_max)
5 {
6     int i, j, ij, p;
7     FILE* imagen;
8

```

## 2d-32 TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C

```
9     imagen=fopen(nombre,"wb");
10    fprintf(imagen,"P2\n");
11    fprintf(imagen,"#guardaPGMi %s\n",nombre);
12    fprintf(imagen,"%d %d\n",anchura,altura);
13    fprintf(imagen,"255\n");

14
15    ij=0;
16    for(i=0; i<altura; i++)
17    {
18        for(j=0; j<anchura; j++)
19        {
20            p=(255*(pixels[ij]-pixel_min))
21                /(pixel_max-pixel_min);
22            if(p<0) p=0;
23            if(p>255) p=255;

24
25            fprintf(imagen, " %d",p);

26
27            ij++;
28        }
29        fprintf(imagen, "\n");
30    }

31    fclose(imagen);

32
33    return;
34 }
35 }

36 void guardaPGMd(char* nombre, int anchura, int altura,
37                   double *pixels, double pixel_min, double pixel_max)
38 {
39
40
41    int i, j, ij, p;
42    FILE* imagen;

43
44    imagen=fopen(nombre,"wb");
45    fprintf(imagen,"P2\n");
46    fprintf(imagen,"#guardaPGMd %s\n",nombre);
47    fprintf(imagen,"%d %d\n",anchura,altura);
48    fprintf(imagen,"255\n");

49
50    ij=0;
51    for(i=0; i<altura; i++)
52    {
53        for(j=0; j<anchura; j++)
54        {
55            p=(int)(255*(pixels[ij]-pixel_min))
56                /(pixel_max-pixel_min);
```

## 2D.3. BIBLIOTECAS DE FUNCIONES

2d-33

```

57         if(p<0) p=0;
58         if(p>255) p=255;
59
60         fprintf(imagen, " %d",p);
61
62         ij++;
63     }
64     fprintf(imagen, "\n");
65 }
66
67 fclose(imagen);
68
69 return;
70 }
```

---

**2d.3.3. Ejemplo: el triángulo de Sierpinski**

El triángulo de Sierpinski es uno de los fractales deterministas más conocidos y seguro que todos hemos visto alguna representación suya en algún momento. En el tema ?? veremos que una manera de generarlo es la siguiente. Partimos de una matriz bidimensional T de dimensiones ALTURA×ANCHURA con todos sus elementos inicializados a cero (se debe cumplir que ANCHURA es impar y que  $\text{ALTURA}=(\text{ANCHURA}+1)/2$ ). En la mitad de la primera fila de la matriz se coloca una semilla (valor 1) que dará lugar al fractal:  $T[0][(\text{ANCHURA}-1)/2]=1$ . A continuación recorremos la matriz fila a fila, asignando a cada elemento el resultado de la operación XOR (simbolizada por el operador  $\wedge$  en C) de los elementos anterior y posterior en la fila anterior:

$$A[i][j] = A[i-1][j-1] \wedge A[i-1][j+1]$$

XOR significa “eXclusive OR”, esto es, la disyunción lógica “O pero no Y”. La tabla de verdad de esta operación lógica es:  $0 \wedge 0 = 0$ ,  $1 \wedge 0 = 1$ ,  $0 \wedge 1 = 1$  y  $1 \wedge 1 = 0$ . En C, los operadores del álgebra booleana se construyen con un único carácter, así “O” es “|”, “Y” es “&”, “O pero no Y” es “ $\wedge$ ”. Estos operadores actúan al nivel de los bits de los números; por eso aquí emplearemos 0 (todos sus bits son cero) y 1 (todos sus bits son cero menos uno, que es 1).

El siguiente listado muestra el código que implementa este algoritmo. Cuando ha acabado de generar el fractal, llama a la función “guardaPGMi()” y le pasa por referencia la matriz de ceros y unos. La función crea un archivo con la imagen en formato PGM (“Triangulo\_Sierpinsky.pgm”), que abierto con el programa Gimp muestra la imagen que se muestra en la Figura 1.

---

**Listado 2d.16:** Programa que genera el triángulo de Sierpinski mediante el método XOR.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "libguardaimagen.h"
4
5 #define ANCHURA 512*2-1
6 #define ALTURA 512
7
8
9 int main(int argc, char** argv)
```

## 2D.3. BIBLIOTECAS DE FUNCIONES

```

10 {
11     int i,j;
12     int T[ALTURA] [ANCHURA];
13
14     for(i=0; i<ALTURA; i++)
15     {
16         for(j=0; j<ANCHURA; j++)
17         {
18             T[i] [j]=0;
19         }
20     }
21
22     T[0] [(ANCHURA-1)/2]=1;
23
24     for(i=1; i<ALTURA; i++)
25     {
26         for(j=1; j<ANCHURA-1; j++)
27         {
28             T[i] [j]=(T[i-1] [j-1]^T[i-1] [j+1]);
29         }
30     }
31
32     guardaPGMi("Triangulo_Sierpinsky.pgm", ANCHURA, ALTURA, *T, 1, 0);
33
34     return 0;
35 }
```

---

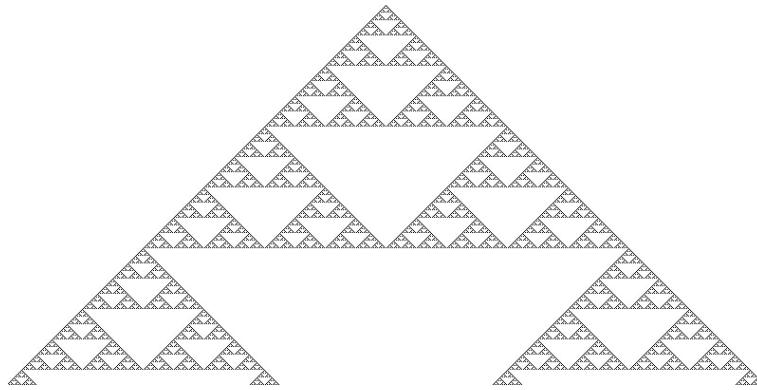


Figura 2d.2: Imagen del archivo generado con el código del listado 2d.16 y que muestra el triángulo de Sierpinski.

## *TEMA 2D. ALGUNAS HERRAMIENTAS ÚTILES PARA LA PROGRAMACIÓN EN C*