Cuaderno de C

Programación para Sistemas

Ángel Herranz
aherranz@fi.upm.es
Universidad Politécnica de Madrid

2025-2026

Índice

1. Primer contacto con C

Ejercicio 1. Algunos ejercicions tiene un icono en el margen. Aquí tienes un pequeño diccionario con su significado:

- *peligro, prestar mucha atención*
- *adelantarse*
- *leer, convenciones*
- *buscar en internet*
- *pensar, observar*

Ejercicio 2. Busca en internet qué es GNU y qué es la FSF.

Ejercicio 3. Tu primera tarea es ser capaz de compilar y ejecutar un *hola mundo* en C. Para ello, necesitas un editor de texto que ya deberías tener instalado y un compilador de C.

Lo más normal es que instales el compilador de C de GNU: GCC. En Ubuntu, el paquete a instalar es `gcc`. Puedes instalarlo con la siguiente línea de comandos en tu *shell*:

```
$ sudo apt-get install gcc
```

Pero te recomiendo instalar un paquete que además te va a instalar Make, una herramienta que usaremos de forma masiva. El paquete en cuestión es `build-essential`:

```
$ sudo apt-get install build-essential
```

Ejercicio 4. Observa con atención cada uno de los mensajes que puedes leer cuando ejecutas una línea de comando. En el caso anterior, si lo repasas, podrás ver que al instalar un *paquete* también se instalan otros de los que el primero **depende**.

Ejercicio 5. Ya deberías tener instalado el compilador de C. Un compilador no deja de ser un programa. Y un programa no deja de ser un fichero. Todos los ficheros tienen un nombre que dice en qué sitio del disco vive. El nombre *absoluto* del compilador de C suele ser `/usr/bin/gcc` aunque en tu instalación puede ser otro. Vamos a preguntar al sistema operativo a través de nuestro *shell* (Bash):

- Ejecuta en la línea de comandos `which gcc`
- Ejecuta en la línea de comandos `which cc` (a veces el compilador de C se llama simplemente `cc`)
- ¿Qué hace `which`? (usa `man which`)
- Ejecuta en la línea de comandos `ls -l /usr/bin/gcc`

Ejercicio 6. Ahora deberías estar en condiciones de seguir las instrucciones de las transparencias:

1. Crea un directorio en el que dejar el trabajo de la asignatura, `clases_pps` por ejemplo:

   ```
   $ mkdir clases_pps
   ```

2. Entra en el directorio:

   ```
   $ cd clases_pps
   ```

3. Crea el fichero `hola.c` con tu editor de texto favorito.

4. Comprueba los ficheros que hay en tu directorio:

   ```
   $ ls -l
   ```

5. Compila:

   ```
   $ gcc hola.c
   ```

6. Comprueba los ficheros que hay en tu directorio:

   ```
   $ ls -l
   ```

7. Ejecuta:

   ```
   $ ./a.out
   ```

Ejercicio 7. ¿Te cuadran las transparencias? ¿Entiendes el proceso de compilación?

Ejercicio 8. Dale un nombre más razonable al ejecutable, en vez de `a.out`, haz que se llame `hola` (usa el programa `mv`, para saber cómo puedes pedir ayuda al manual).

Ejercicio 9. No permitas que `gcc` te oculte lo que está haciendo internamente. Compila primero y *linka* después. Deberás ver el fichero *objeto* intermedio: `hola.o`.

Ejercicio 10. Recuerda que tienes disponible el manual de GCC con la línea

```
$ man gcc
```

Ejercicio 11. No permitas que `gcc` te oculte nada de lo que está haciendo internamente. Descubre cuál es el significado de **`#include`** usando la opción `-E` del compilador.

Ejercicio 12. ¿Qué crees que está pasando? ¿Puedes encontrar el fichero `stdio.h`? Quizás puedes buscarlo con `locate`.

`#include` simplemente expande, como si fuera un Copy & Paste el fichero nombrado, en este caso `stdio.h`. Nada más.

Ejercicio 13. Ejecuta el manual de `gcc` o busca en internet hasta descubrir de qué forma podrías generar un fichero con código ensamblador a partir del fichero `hola.c`.

2. Ejecutando programas C

Ejercicio 14. Repasa las transparencias de clase. Observa los detalles, especialmente los relacionados con GDB.

Ejercicio 15. Busca en internet GDB, explora.

Ejercicio 16. Experimenta con `argc` y `argv`. Por ejemplo, puedes escribir un programa que diga cuál es el valor de `argc`, supongamos que n y que luego vaya imprimiendo los valores de `argv[0]`, `argv[1]`, `argv[2]`, ..., `argv[`$n-1$`]`. ¿Qué ocurre si intentas imprimir también el dato `argv[argc]`? ¿Y `argv[argc+1]`? ¿Y `argv[argc+2]`? ¿Y ...?

Ejercicio 17. Experimenta con `getenv`. Lee la página del manual de `getenv`: `man getenv`. Haz un programa para mostrar diferentes variables de entorno.

Ejercicio 18. Corrige los bugs del programa `fact.c` de las transparencias y haz una ejecución paso a paso con GDB sin entrar en las funciones de biblioteca y mirando cómo cambian las variables en cada paso.

Ejercicio 19. Cuando un programa termina su ejecución, además de todo lo que imprime en la salida estándar o en la salida de error, nos ofrece otra información: el código de terminación. El código de terminación de un programa C es el valor que devuelve su función `main`.

Para saber el valor que ha devuelto el último programa ejecutado desde Bash podemos usar el mandato **`echo`** `$?`.Prueba lo siguiente:

```
$ ls hola.c
hola.c
$ echo $?
0
$ ls supercalifragilisticoespialidoso.c
ls: cannot access 'supercalifragilisticoespialidoso.c': No such file or directory
$ echo $?
2
```

En este ejercicio tienes que inspeccionar el código de terminación de tu programa `fact.c`.

Ejercicio 20. En C se pueden definir macros apoyándonos en el preprocesador. Una de las directivas del preprocesador de C más usada es **`#define`**. Dicha directiva sustituye en tiempo de compilación[1] un nombre por una expresión.

Prueba el siguiente código (puedes llamar al fichero `prueba_define.c`):

```
#include <stdio.h>
#define N 5
int main() {
  int x = N;
```

[1] Es decir, antes de ejecutar.

```
  printf("El valor de x es %i\n", x);
  return 0;
}
```

Puedes ver el efecto de **#define** ejecutando `gcc -E prueba_define.c`

⏭ **Ejercicio 21.** Parece que la directiva **#define** sirve para definir constantes pero... Consideremos este código:

```
double x = 18.0 / squared( 2 + 1 );
```

¿Cuál será el valor de `x` para cada una de las siguientes macros?

- `#define squared(x) x*x`
- `#define squared(x) (x*x)`
- `#define squared(x) (x)*(x)`
- `#define squared(x) ((x)*(x))`

Ejercicio 22. Escribe un programa para comprobar tu respuesta a la última pregunta.

Ejercicio 23. Invoca el preprocesador (`gcc -E MI_PROGRAMA`) para ver el efecto que tiene usar la directiva **#define**. Evita usar la directiva **#include**: *cuanto menos bulto más claridad*.

⏭ **Ejercicio 24.** Durante la clase hemos visto varias declaraciones de variables, todas ellas de tipo entero (**int**). En este ejercicio deberás ir al mítico libro *The C Programming Language* de Brian W. Kernighan y Dennis M. Ritchie y explorar qué otros tipos hay, declara varias variables, inicialízalas e imprime sus valores con `printf`.

⏭ **Ejercicio 25.** Siguiendo con el libro, durante la clase hemos visto varios usos de la función `printf`. En este ejercicio deberás explorar las conversiones básicas que es capaz de realizar la función: `%d` y `%i` para enteros en notación decimal, `%x` para enteros en notación hexadecimal, `%s` para strings, etc.

🔍 **Ejercicio 26.** Realiza el siguiente tutorial de GDB de Andrew Gilpin (profesor de la universidad Carnegie Mellon):

`https://www.cs.cmu.edu/~gilpin/tutorial/`

3. Tipos básicos

📖 **Ejercicio 27.** Repasa las transparencias de clase. Ten siempre en cuenta que en cada variable sólo puede haber `0`s y `1`s.

Ejercicio 28. Asegúrate de realizar todos los ejercicios de las transparencias antes de continuar.

Ejercicio 29. Ya has visto que es complicado fiarse de lo que `printf` nos enseña. Utiliza `gdb` para ver el contenido *exacto* de la variable `mi_double` en el programa `basicos.c`.

Este ejercicio es especialmente complicado, así que aquí tienes una sesión de `gdb` que puede ayudarte:

```
angel@T440p: /03-basicos (master)$ gdb basicos
GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
[...]
Reading symbols from basicos...done.
(gdb) break main
Breakpoint 1 at 0x652: file basicos.c, line 3.
(gdb) run
Starting program: /home/angel/Asignaturas/undergit/pps-src/03-basicos/basicos

Breakpoint 1, main () at basicos.c:3
3       char mi_char = 'a';
(gdb) step
4       int mi_int = 42;
(gdb) step
5       float mi_float = 1000000.0;
(gdb) step
6       double mi_double = 0.0000001;
(gdb) step
7       printf("El char es: %c\n", mi_char);
(gdb) x/tg &mi_double
0x7fffffffdfb8: 0011111100111101011010011111111001101001101011111001011111010001000
(gdb) q
angel@T440p: /03-basicos (master)$
```

Ejercicio 30. Los programadores somos muy vagos. No soportamos tener que teclear

```
$ gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

cada vez que necesitamos compilar el programa `basicos.c` para generar el ejecutable `basicos`. Para evitar este tipo de tareas tan tediosas, y evitar también errores a la hora de escribir esas líneas, tenemos herramientas que nos ayudan.

Una de estas herramientas es `make`. Cuando ejecutamos `make`, la herramienta busca un fichero `Makefile` en el que hay una serie de **reglas** con las que automatizamos ciertas tareas. Veamos un ejemplo:

```
1 basicos: basicos.c
2     gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

La primera línea dice: para conseguir el fichero `basicos` necesitas el fichero `basicos.c`. La segunda línea dice: para construir el fichero `basicos` hay que ejecutar el mandato

```
$ gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
```

A partir de ahí, basta con ejecutar la línea

```
$ make basicos
```

Quizás lo más importante: **`make` *sabe* que sólo es necesario ejecutar el mandato cuando `basicos` es *más viejo* que `basicos.c`**.

Ejercicio 31. Los ficheros `Makefile` admiten que pongas tantas reglas como quieras. Eso

significa que para practicar con los programas de cada clase puedes hacerte un `Makefile` con las reglas que *fabriquen* todos los programas. Por ejemplo:

```
1 basicos: basicos.c
2     gcc -ansi -Wall -Werror -pedantic -o basicos basicos.c
3
4 sizeof: sizeof.c
5     gcc -ansi -Wall -Werror -pedantic -o sizeof sizeof.c
```

No dejes de crear ese `Makefile` y de probarlo.

Ejercicio 32. Seguro que has oido hablar del "código ASCII". Ha llegado el momento de entenderlo `;)`. Busca en Internet lo que es el código ASCII, lo necesitas para el siguiente ejercicio.

Ejercicio 33. Escribe un programa (`caracteres.c`) que escriba en la salida estándar los caracteres imprimibles del código ASCII junto con su código en decimal.

```
Tu programa debería imprimir algo como
' ', 32
'!', 33
...
'}', 125
'~', 126
```

Ejercicio 34. Escribe un programa y haz las modificaciones necesarias para conocer los valores máximo y mínimo de los siguientes tipos (no uses la biblioteca `limits.h`): **`char`**, **`unsigned char`**, **`int`**, **`unsigned int`**, **`long int`**, **`unsigned long int`**, **`long long int`**, **`unsigned long long int`**.

Ejercicio 35. A la luz de lo que has visto en el ejercicio anterior, deduce el tamaño en bytes de los tipos (no uses el operador **`sizeof`**).

Ejercicio 36. Contrasta que tus respuestas a los ejercicios anteriores son correctas: escribe un programa que imprima en la salida estándar el nombre de cada tipo, el tamaño de sus valores en bytes y los valores máximo y mínimo.

```
Tu programa debería imprimir algo como
char 1 -128 127
unsigned 1 0 255
...
```

Ejercicio 37. Asegúrate que entiendes todas tus deducciones sobre límites y tamaños de los tipos.

Ejercicio 38. Ejecuta `man 3` **`printf`** y muévete por el manual de *printf* entendiendo su estructura.

Ejercicio 39. Lee las secciones *Conversion specifiers* y *Length modifier* del manual anterior. Presta especial atención a los formatos

%i, %u, %d, %o, %x, %e, %E, %f, %F, %s, %c

y a los modificadores

%l, %h.

Conversion specifiers
A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

d, i The int argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

o, u, x, X
The unsigned int argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation. The letters abcdef are used for x conversions; the letters ABCDEF are used for X conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

e, E The double argument is rounded and converted in the style [-]d.ddde±dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

f, F The double argument is rounded and converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

(SUSv2 does not know about F and says that character string representations for infinity and NaN may be made available. SUSv3 adds a specification for F. The C99 standard specifies "[-]inf" or "[-]infinity" for infinity, and a string starting with "nan" for NaN, in the case of f conversion, and "[-]INF" or "[-]INFINITY" or "NAN" in the case of F conversion.)

g, G The double argument is converted in style f or e (or F or E for G conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style e is used if the exponent from its

conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

a, A (C99; not in SUSv2, but added in SUSv3) For a conversion, the double argument is converted to hexadecimal notation (using the letters abcdef) in the style [-]0xh.hhhhp±; for A conversion the prefix 0X, the letters ABCDEF, and the exponent separator P is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type double. The digit before the decimal point is unspecified for nonnormalized numbers, and nonzero but otherwise unspecified for normalized numbers.

c If no l modifier is present, the int argument is converted to an unsigned char, and the resulting character is written. If an l modifier is present, the wint_t (wide character) argument is converted to a multibyte sequence by a call to the wcrtomb(3) function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

s If no l modifier is present: the const char * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

If an l modifier is present: the const wchar_t * argument is expected to be a pointer to an array of wide characters. Wide characters from the array are converted to multibyte characters (each by a call to the wcrtomb(3) function, with a conversion state starting in the initial state before the first wide character), up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null byte. If a precision is specified, no more bytes than the number specified are written, but no partial multibyte characters are written. Note that the precision determines the number of bytes written, not the number of wide characters or screen positions. The array must contain a terminating null wide character, unless a precision is given and it is so small that the number of bytes written exceeds it before the end of the array is reached.

C (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for lc. Don't use.

S (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for ls. Don't use.

p The void * pointer argument is printed in hexadecimal (as if by %#x or %#lx).

n The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. (This specifier is not supported by the bionic C library.) The behavior is undefined if the conversion specification includes any flags, a field width, or a precision.

m (Glibc extension; supported by uClibc and musl.) Print output of strerror(errno). No argument is required.

% A '%' is written. No argument is converted. The complete conversion specification is '%%'.

Length modifier

Here, "integer conversion" stands for d, i, o, u, x, or X conversion.

hh A following integer conversion corresponds to a signed char or unsigned char argument, or a following n conversion corresponds to a pointer to a signed char argument.

h A following integer conversion corresponds to a short int or unsigned short int argument, or a following n conversion corresponds to a pointer to a short int argument.

l (ell) A following integer conversion corresponds to a long int or unsigned long int argument, or a following n conversion corresponds to a pointer to a long int argument, or a following c conversion corresponds to a wint_t argument, or a following s conversion corresponds to a pointer to wchar_t argument.

ll (ell-ell). A following integer conversion corresponds to a long long int or unsigned long long int argument, or a following n conversion corresponds to a pointer to a long long int argument.

q A synonym for ll. This is a nonstandard extension, derived from BSD; avoid its use in new code.

L A following a, A, e, E, f, F, g, or G conversion corresponds to a long double argument. (C99 allows %LF, but SUSv2 does not.)

j A following integer conversion corresponds to an intmax_t or uintmax_t argument, or a following n conversion corresponds to a pointer to an intmax_t argument.

z A following integer conversion corresponds to a size_t

```
              or  ssize_t argument, or a following n conversion corre-
              sponds to a pointer to a size_t argument.

       Z      A nonstandard synonym for z that predates the appearance
              of z.  Do not use in new code.

       t      A  following  integer  conversion  corresponds  to  a
              ptrdiff_t argument, or a following n  conversion  corre-
              sponds to a pointer to a ptrdiff_t argument.
```

Ejercicio 40. Una vez conoces los tamaños de los tipos y los límites, puedes forzar *overflows*. Escribe código para ver qué pasa cuando intentas ir más allá del límite (por ejemplo sumando o restando `1` a un límite.

Ejercicio 41. No dejes de leer el artículo de David Goldberg: *What every computer scientist should know about floating-point arithmetic*

Ejercicio 42. Sorprendéntemente puedes pedirle a C que te imprima un dato como si fuera de otro tipo. Por ejemplo, imprimir un caracter como si fuera un entero o un entero negativo como si fuera **`unsigned`** o incluso un número en coma flotante como si fuera un entero. El compilador se va a quejar y no podrás compilar alguno de esos intentos, especialmente si usas los *flags* `-Werror`, no lo uses para compilar este código:

```
int i = -42;
float f = -3.0;
printf("%d\n", i);
printf("%u\n", i);
printf("%f\n", i);
printf("%d\n", f);
printf("%u\n", f);
printf("%f\n", f);
```

El compilador debería decir algo como:

```
In function ‘main’:
warning: format ‘%f’ expects argument of type ‘double’,
   but argument 2 has type ‘int’ [-Wformat=]
   printf("%f\n", i);
           ~^
           %d
warning: format ‘%d’ expects argument of type ‘int’,
   but argument 2 has type ‘double’ [-Wformat=]
   printf("%d\n", f);
           ~^
           %f
warning: format ‘%u’ expects argument of type ‘unsigned int’,
   but argument 2 has type ‘double’ [-Wformat=]
   printf("%u\n", f);
           ~^
           %f
```

Y la salida del programa debería ser:

```
-42
4294967254
0.000000
-124652960
4170314336
-3.000000
```

Ejercicio 43. ¿Puedes darle una interpretación a esos datos?

Ejercicio 44.

No nos gustan nada los *warnings*

Son un indicador de que algo no va mal o de que no somos conscientes de algo. Por eso usamos el *flag* `-Werror`. Vamos a adelantarnos un poco y hacer conversiones de tipos que eliminen las quejas del compilador. Por ejemplo, si C espera un **`float`** y le pasamos un entero `n`, podemos pedir al compilador que *convierta* (*cast*) `n` a **`float`** con esta expresión: `(float)n`. Prueba a eliminar los *warnings*.

Ejercicio 45. Ejecuta ahora el programa y compara los resultados con los del programa sin las conversiones.

Ejercicio 46. Transcribe el siguiente programa e interpreta su salida:

```
#include <stdio.h>
int main() {
  int x = 42;
  char *s = "Un string en C";
  printf("%p\n",(void *)&x);
  printf("%p\n",(void *)&s);
  printf("%p\n",s);
  return 0;
}
```

4. Funciones

Ejercicio 47. Repasa las transparencias de clase. Presta especial atención a los detalles sintácticos del fichero `Makefile`. Intenta entender el *guión* que lleva a solucionar cada problema que va surgiendo.

Ejercicio 48. En las transparencias puedes encontrar varias referencias, una al K&R, capítulo 4 y otras dos para la herramienta `make`. No dejes de tenerlas a mano y explorarlas:

Chapter 4. Functions and Program Structure
(*The C Programming Language*, K&R 2nd. edition)

GNU Make Manual
The Free Software Foundation (FSF)

`http://makefiletutorial.com/`
Chase Lambert

Ejercicio 49. Asegúrate de realizar todos los ejercicios de las transparencias antes de continuar. Deberías tener en un directorio los siguientes ficheros: `lcg1.c`, `sum1.c`, y un maravilloso `Makefile` con el que puedes *fabricar* los ejecutables `lcg1` y `sum1`.

Ejercicio 50. Recuerda que la herramienta `make` tiene algunas reglas por defecto que no es necesario que tú mismo escribas. Dependiendo de la instalación las reglas pueden variar (aunque las relacionadas con el lenguaje C son bastante estables en todas las distribuciones de Unix). Para ver las reglas predefinidas ejecuta:

```
$ make -p -f /dev/null
```

Ejercicio 51. Antes de continuar, vamos a enriquecer el `Makefile` para que con la simple ejecución de `make` se creen todos los ejecutables. Para ello basta con que añadas esta regla justo después de que establezcas la variable `CFLAGS`:

```
CFLAGS=-Wall -g -pedantic

todos: lcg1 sum1

...
```

Prueba ahora a ejecutar

```
$ make todos
```

o simplemente

```
$ make todos
```

Ejercicio 52. A veces, después de ejecutar `make`, el directorio en el que estás trabajando se llena de ficheros *feos* o inservibles. Vamos a añadir un par de reglas al final a nuestro el `Makefile` para realizar una limpieza:

```
...

limpio:
        rm -f *.o

muylimpio: limpio
        rm -f lcg1 sum1 core
```

Ahora, la ejecución de

```
$ make limpio
```

borrará todos los ficheros `.o` y

```
$ make muylimpio
```

hará lo mismo que `make limpio` y además borrará todos los ejecutables.

Nota: por convención, la mayor parte de los programadores usan `clean` y `veryclean` como objetivos así que es habitual ver:

```
$ make clean
```

y

```
$ make veryclean
```

Ejercicio 53. Deberías haber sido capaz de hacer el `Makefile` por ti mismo. Puedes compararlo con el siguiente:

```
CFLAGS=-Wall -g -pedantic

todos: lcg1 sum1

lcg1: lcg1.o
	$(CC) $(CFLAGS) -o lcg1 lcg1.o

sum1: sum1.o
	$(CC) $(CFLAGS) -o sum1 sum1.o

clean:
	rm -f *.o

veryclean: clean
	rm -f lcg1 sum1 lcg2 test_lcg core
```

Ejercicio 54. Ejecución paso a paso: `gdb lcg1`

- Poner en marchar el depurador `gdb`.
- Colocar un *breakpoint* en `main`.
- Ejecutar el programa paso a paso y explorar las variables[2]
- ¿Puedes ver el valor de `anterior` cuando está ejecutando `main`? ¿Y el valor de x?
- ¿Puedes ver el valor de `i`, variable de `main` cuando está ejecutando `generar_aleatorio`?

Ejercicio 55. Ejecución paso a paso: `gdb sum1` (usa un número bajito `;)`)

[2]Ver transparencias de la sesión 2

- Poner en marchar el depurador `gdb`.
- Colocar un *breakpoint* en `main`.
- Ejecutar el programa paso a paso y explorar las variables[3]
- ¿Puedes ver el valor de `n` cuando está ejecutando `main`? ¿Y el valor de i?
- ¿Puedes ver el valor de `n`, variable de `main` cuando está ejecutando `sum`?

Ejercicio 56. Poco a poco vamos viendo nueva sintaxis para las expresiones del lenguaje C. En esta sesión hemos descubierto dos sintaxis nuevas y muy importantes:

$$\star e \qquad \& e$$

Antes de continuar vuelve a asegurarte de que entiendes la semántica de ambas sintaxis:

- $\star e$: se evalúa e, su resultado es entonces interpretado como una dirección de memoria, y $\star e$ hace referencia al contenido de dicha dirección de memoria.
- $\& e$: en este caso e no puede ser cualquier expresión, sólo puede ser un *lvalue*[4], normalmente un identificador. El significado es «la dirección de memoria de e».

Ejercicio 57. No olvides realizar una implementación correcta de la función `intercambiar`.

Ejercicio 58. Ejecuta la función `intercambiar` bajo el control de `gdb` y vete explorando las variables.

Ejercicio 59. No olvides realizar una implementación correcta de la función `sum` no recursiva (fichero `sum2.c`).

Ejercicio 60. Hasta ahora has usado "masivamente" la función `printf` de la biblioteca estándar de C para poder escribir en la salida estándar. Dicha función forma parte de una familia de funciones entre las que están `printf`, `fprintf`, `dprintf`, `sprintf` y `snprintf`. Puedes verlas todas invocando la *sección 3* del manual desde Bash:

```
$ man 3 printf
```

Existe una versión dual de `printf` para poder leer datos de la entrada estándar: **`scanf`**. El siguiente ejemplo lee un entero de la entrada estándar:

```
int x;
scanf("%i", &x);
printf("El dato leído es: %i\n", x);
```

Como podrás observar es fundamental entender el uso del paso de parámetros por referencia para poder usar la función `scanf`. El resto del ejercicio, además de probar esas líneas, consiste en leer diferentes tipos de datos y entender cómo funcionan las funciones de la familia: `scanf`, `fscanf` y `sscanf`. Para ello tienes a tu disposición el manual:

```
$ man 3 scanf
```

[3] Ver transparencias de la sesión 2

[4] Todo aquello que puede aparecer en la izquierda de una asignación.

5. Arrays y Strings

Ejercicio 61. Repasa las transparencias de clase. No dejes de transcribir y realizar todos los programas sugeridos en las mismas.

Arrays

Ejercicio 62. Para realizar los siguientes ejercicios vamos a utilizar la función *inversa* a `printf`. Se llama `scanf` y en vez de imprimir datos en la salida estándar lee datos de la entrada estándar. Puedes probar tú mismo el manual de `scanf`: `man 3 scanf` o incluso puedes echar un ojo al libro K&R (sección 7.4 de la segunda edición en inglés).

Ejercicio 63. Aunque para entender bien el funcionamiento de la familia de funciones `scanf` es necesario saber un poquito de punteros, vamos a usarla con el siguiente patrón:

```
1  #include <stdio.h>
2
3  int main() {
4    int x;
5    int leidos;
6    leidos = scanf("%i", &x);
7    while (leidos == 1) {
8      printf("%i\n", 2*x);
9      leidos = scanf("%i", &x);
10   }
11   return 0;
12 }
```

La línea 6 intenta leer **un** entero (eso se indica con `"%i"`) de la entrada estándar y lo deja en la variable `x` (eso se indica con la expresión `&x`). Si en la entrada estándar se lee un entero la función `scanf` devuelve **1** que se asigna a `leidos`, si lo que hay en la entrada estándar es algo diferente a un entero entonces devuelve un `0` que se asigna a `leidos`.

Ese programa lee enteros en la entrada estándar y los multiplica por dos antes de imprimirlo en la salida estándar. El bucle para cuando lo que se lee de la entrada estándar no es un entero. Compila y ejecuta para que lo puedas ver.

La verdad es que un programador de C pura zepa *jamás escribirá el código anterior, cualquier pensará que lo ha hecho un programador de Java. Veamos un programar equivalente en un estilo más C:*

```
1 #include <stdio.h>
2
3 int main() {
4     int x;
5     while (scanf("%i", &x))
6         printf("%i\n", 2*x);
7     return 0;
8 }
```

Se puede ver cómo se aprovecha que la función `scanf` *devuelve el número de argumentos que encajan con el formato, que dicho número se aprovecha directamente como* condición del bucle*: si distinto de* `0`.

Ejercicio 64. Con lo que has aprendido en el ejercicio anterior y en las transparencias, escribe un programa que lea enteros y los almacene en un array para imprimirlos en el orden inverso. Tu programa admitirá un máximo de `M` enteros (pongamos 1000). Si en la entrada estándar hay más de `M` enteros sólo se invertirán los `M` primeros. En el momento en el que haya algo diferente a un entero, el programar invertirá todos los enteros leídos hasta ese momento.

Para no tener que teclear números desde el teclado os sugerimos que pongáis los números en un fichero, digamos `numeros.txt`:

```
1
8
4000
2
23
51
87
fin
```

Si suponemos que has llamado a tu programa `invertir_enteros`, esperamos que este sea el comportamiento:

```
$ ./invertir_enteros < numeros.txt
87
51
23
2
4000
8
1
$ |
```

Ejercicio 65. En las transparencias hemos visto cómo podemos calcular la longitud de un array usando esta expresión:

```
sizeof(a)/sizeof(a[0])
```

Escribe una macro (**#define**) que realice ese trabajo dado un array cualquiera[5].

⚠ Recuerda que dicha **técnica es errónea**

Ejercicio 66. Escribir una función que ordena un array de enteros entre dos posiciones. La *cabecera* de dicha función será:

```
void ordenar(int a[], int min, int max);
```

Ejercicio 67. Utiliza la función anterior para implementar un programa que lea enteros de la entrada estándar y los imprima en la salida estándar ordenados. Tu programa admitirá un máximo de `M` enteros (pongamos 1000). Si en la entrada estándar hay más de `M` enteros sólo se ordenarán los `M` primeros. En el momento en el que haya algo diferente a un entero, el programar ordenará todos los enteros leídos hasta ese momento.

Si suponemos que has llamado a tu programa `ordenar_enteros`, esperamos que este sea el comportamiento[6]:

```
$ ./ordenar_enteros < numeros.txt
1
2
8
23
51
87
4000
$ |
```

Ejercicio 68. Puedes intentar varios algoritmos de ordenación (*buble sort, insert sort, merge sort,* y *quick sort* por ejemplo) y probar el tiempo que tarda en ejecutar cada uno de ellos.

Para ello puedes generar 1000 números aleatorios y almacenarlos en el fichero `numeros.txt` usando este mandato de Bash:

```
for i in $(seq 1000); do echo $RANDOM >> numeros.txt; done
```

Con este otro mandato de Bash puedes comprobar el tiempo de ejecución:

```
time ./ordenar_enteros < numeros.txt
```

💬 **Ejercicio 69.** ¿Se puede combinar una inicialización de un array con el hecho de que sea un array de longitud fina? ¿Se puede hacer esto en C?

[5] Mira la sección 4.11.2 del K&R: *Macro Substitution*

[6] El fichero `numeros.txt` es el mismo que el del ejercicio 64

```
int a[21] = {1, 2, ,3};
```

¿Qué significa? ¿Cuál es su longitud? ¿Cuántos enteros caben?

Ejercicio 70. Sorprendéntemente C admite hacer declaraciones de argumentos de tipo array con longitud fija. ¿Qué significado tiene?

Ejercicio 71. ¿Puedes almacenar un string de longitud 5 (como por ejemplo `"adios"`) en un array de longitud 4? ¿Y en uno de longitud 5? ¿Y en uno de longitud 6? ¿Y en uno de longitud 10? ¿Entiendes la diferencia entre la longitud de un string y la del array que lo contiene?

Ejercicio 72. Escribe un programa que lea de la entrada estándar dos matrices y las multiplique. Puedes asumir que las matrices están codificadas de la siguiente forma en la entrada estándar:

- Un entero positivo m que indica el número de filas de la primera matriz.
- Un entero positivo n que indica el número de columnas de la primera matriz (que coincide con el número de filas de la segunda).
- Un entero positivo p que indica el número de columnas de la segunda matriz.
- $m \times n$ *floats*: los n primeros son la primera fila de la primera matriz, los n segundos la segunda fila, etc.
- $n \times p$ *floats*: los p primeros son la primera fila de la segunda matriz, los p segundos la segunda fila, etc.

Se puede asumir que m, n y p están entre 1 y 1000. El resultado serán $m \times p$ *floats* donde los p primeros son la primera fila de la matriz multiplicación, los p segundos la segunda fila, etc.

Para no tener que teclear las matrices desde el teclado os sugerimos que las pongáis en un fichero, digamos `matrices.txt`:

```
3
2
3
1 2
-1 0
-3 -1
2 0 1
-5 2 3
```

Si suponemos que has llamado a tu programa `multiplicar_matrices`, esperamos que este sea el comportamiento:

```
$ ./multiplicar_matrices < matrices.txt
-8 4 7
-2 0 -1
-1 -2 -6
$ |
```

Strings

Ejercicio 73. Explora el módulo (*header*) `<string.h>` de la **biblioteca estándar de C**. En el K&R puedes encontrar el listado de funciones de cada *header*.

Ejercicio 74. Explora el módulo (*header*) `<ctype.h>` de la **biblioteca estándar de C**. Dicho módulo tiene funciones que hablan sobre caracteres. En el K&R puedes encontrar el listado de funciones de cada *header*. Escribe código que use las siguientes funciones:

- `int isalnum(int c);`
- `int isalpha(int c);`
- `int iscntrl(int c);`
- `int isdigit(int c);`
- `int isgraph(int c);`
- `int islower(int c);`
- `int isprint(int c);`
- `int ispunct(int c);`
- `int isspace(int c);`
- `int isupper(int c);`
- `int isxdigit(int c);`
- `int toupper(int c);`
- `int tolower(int c);`

Ejercicio 75. A continuación tienes una lista de funciones incluídas en los headers `strings.h` y `string.h`: `stpcpy`, `strcasecmp`, `strcat`, `strchr`, `strcmp`, `strcoll`, `strcpy`, `strcspn`, `strdup`, `strfry`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strncasecmp`, `strpbrk`, `strrchr`, `strsep`, `strspn`, `strstr`, `strtok`, `strxfrm`, `index`, `rindex`. Te sugerimos que intentes implementarlas todas. Si no tienes tanto tiempo, debes intentarlo con las siguientes al menos:

`strcpy`, `strcat`, `strchr`, `strcmp`, `strdup` y `strlen`.

Nota: observa que para comprobar que tu implementación es correcta puedes compararla con los resultados de la biblioteca estándar.

6. Punteros

Ejercicio 76. Implementa por ti misma la función de la biblioteca estándar `memcpy` (usa el manual).

Ejercicio 77. Implementa por ti misma la función de la biblioteca estándar `memset` (usa el manual).

Ejercicio 78. Repasa las transparencias de clase. Las transparencias de este tema están repletas de pruebas que tienes que entender en detalle. La memoria dinámica, especialmente cuando se viene de lenguajes con recolección automática de basura, se hace muy complicado, mucho más al combinarlo con la estrecha relación entre arrays y punteros de C. Entender cada transparencia se hace esencial.

Ejercicio 79. Poco a poco vamos viendo nueva sintaxis para las expresiones del lenguaje C. En esta sesión hemos descubierto dos sintaxis nuevas y muy importantes:

$$\star e \qquad \& e$$

Antes de continuar vuelve a asegurarte de que entiendes la semántica de ambas sintaxis:

- $\star e$: se evalúa e, su resultado es entonces interpretado como una dirección de memoria, y $\star e$ hace referencia al contenido de dicha dirección de memoria.
- $\& e$: en este caso e no puede ser cualquier expresión, sólo puede ser un *lvalue*[7], normalmente un identificador. El significado es «la dirección de memoria de e».

Ejercicio 80. No olvides realizar una implementación correcta de la función `intercambiar`.

Ejercicio 81. Ejecuta la función `intercambiar` bajo el control de `gdb` y vete explorando las variables.

Ejercicio 82. RPN (*Reverse Polish Notation*)

The following algorithm evaluates postfix expressions using a stack, with the expression processed from left to right:

```
for each token in the postfix expression:
  if token is an operator:
    operand_2 = pop from the stack
    operand_1 = pop from the stack
    result = evaluate token with operand_1 and operand_2
    push result back onto the stack
  else if token is an operand:
    push token onto the stack
result = pop from the stack
```

[7] Todo aquello que puede aparecer en la izquierda de una asignación.

Reverse Polish notation (Wikipedia)

En este ejercicio tendrás que programar una calculadora *polaca inversa*:

- Los *tokens* serán floats, operadores (`'+'`, `'-'`, `'*'`, `'/'`) y el caracter de *fin de expresión* (`'='`)
- Leemos la expresión de la entrada estándar con `scanf`[8]:

```
float operando;
char operador[2];
scanf("%f", &operando);
scanf("%s", operador);
```

- Utilizaremos un array para implementar la pila de operandos
- Asumiremos que la pila no puede crecer en más de 1000 elementos

Así deberá comportarse tu programa:

```
$ ./rpn
15 7 1 1 + - / 3 * 2 1 1 + + - =
5
$ |
```

Ejercicio 83. Si no lo has hecho ya, modifica tu programa `rpn` para acceder a la pila de operandos utilizando punteros.

Ejercicio 84. Transcribe el siguiente programa `nodina.c`.

```
#include <stdio.h>

int main() {
  int n;
  scanf("%d",&n);
  int a[n];
  printf("%p\n",a);
  return 0;
}
```

Como ves, el programador pretende que el usuario introduzca un número para luego *crear* un array con tantos elementos como los indicados por el usuario. Compílalo con

```
cc -Wall -Werror -g -ansi -pedantic -o nodina nodina.c.
```

- ¿Entiendes el mensaje del compilador?
- ¿Cómo debes hacerlo para ser *puro C*?
- ¿Qué ocurre si eliminas el *flag* `-pedantic`?

Ejercicio 85. Consultar el manual `man 3 malloc`. En dicha página del manual encontrarás la documentación de varias operaciones funciones:

- **void** *malloc(size_t size);
- **void** free(**void** *ptr);
- **void** *calloc(size_t nmemb, size_t size);
- **void** *realloc(**void** *ptr, size_t size);
- **void** *reallocarray(**void** *ptr, size_t nmemb, size_t size);

```
MALLOC(3)              Linux Programmer's Manual              MALLOC(3)

NAME
       malloc,  free, calloc, realloc - allocate and free dynamic mem-
       ory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);

   Feature  Test  Macro  Requirements  for  glibc  (see  fea-
   ture_test_macros(7)):

       reallocarray():
           _GNU_SOURCE

DESCRIPTION
       The  malloc()  function  allocates  size  bytes  and  returns a
       pointer to the allocated memory.  The memory  is  not  initial-
       ized.   If  size  is  0, then malloc() returns either NULL, or a
       unique pointer value that can later be successfully  passed  to
       free().

...
```

Ejercicio 86. La llamada a la función `calloc(`n`, `s`)` de la biblioteca estándar devuelve un puntero a n objetos de tamaño s cada uno con toda la memoria inicializada a 0. En este ejercicio tendrás que implementar tú mismo la función `calloc`, puedes llamarla `micalloc ;)`, llamando a la función `malloc`.

[8] Utiliza el manual (`man 3 scanf`), o mejor el libro K&R, apéndice B, sección1.3.

Ejercicio 87. Para poder probar tu implementación de ordenación de enteros propuesta en las transparencias, te sugerimos usar la potencia de Bash.

Las siguientes órdenes de Bash generan un fichero `enteros.txt` con una entrada aleatoria apropiada para tu programa:

```
R=$RANDOM
echo $R > enteros.txt
for ((i = 0 ; i < $R ; i++)); do
  echo $RANDOM >> enteros.txt;
done
```

A mi me ha salido esto:

```
$ cat enteros.txt
5
29022
957
13682
10575
22773
$ |
```

Ahora, si has llamado a tu programa `ordenar_enteros`, puedes ejecutarlo así:

```
$ ./ordenar_enteros < enteros.txt
957
10575
13682
22773
29022
$ |
```

Ejercicio 88. Modifica el programa que ordena ristras de enteros (versión final de las transparencias) para que la ordenación la realice una función `bubble_sort`. Este ejercicio está muy orientado a que puedas entender los siguientes puntos:

- Como los punteros y los arrays *son lo mismo* en C, se puede declarar un array como argumento de la función. Como es un puntero, lo que modificamos es aquello a lo que apunta, que resulta ser el contenido del array.
- En la sesión anterior ya vimos que no era posible conocer la longitud de un array cuando es argumento de una función. Por ello es necesario que la función de ordenación reciba como argumento la longitud del array.

Ejercicio 89. Modifica el programa anterior para usar un algoritmo de ordenación más rápido que *bubble sort*: *quick sort* o *merge sort* por ejemplo.

7. Estructuras

Ejercicio 90. Repasa las transparencias de clase. El trabajo con *structs* es muy natural, gran parte de su sintaxis es idéntica a la de otros lenguajes de programación y otra parte es muy parecida. Sin embargo, la memoria dinámica, especialmente cuando se viene de lenguajes con recolección automática de basura, se hace muy complicado. De nuevo, entender cada transparencia se hace fundamental.

Ejercicio 91. La biblioteca estándar de C introduce innumerables *structs*. Entre ellos cabe destacar `FILE`. Aunque en general usamos el tipo `FILE *` a través de las funciones de `stdio.h` (`fopen`, `fgets`, etc.), puedes intentar perseguir la definición del *struct* buscand ficheros como `stdio.h`, descubriras que incluye `FILE.h`, etc. ¿Puedes encontrar su definición en tu sistema operativo? ¿Puedes entender los campos? ¿Les encuentras sentido?

```
struct _IO_FILE
{
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */

/* The following pointers correspond to the C++ streambuf protocol. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
/* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;
/* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  __off_t _old_offset; /* This used to be _offset but it's too small.
*/
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  _IO_lock_t *_lock;
}
```

Ejercicio 92. Utilizando la definición de **struct** rectangulo_s, tienes que escribir un programa que lea rectángulos de la entrada estándar, calcule su área y los imprima en orden, de mayor área a menor área.

La entrada estándar tendrá el siguiente formato:

- Un entero n que indica el número de rectángulos que habrá que leer. El dato n estará entre 1 y 1000.
- n filas con cuatro enteros A_x, A_y, B_x, B_y cada una que indican los puntos (A_x, A_y) y (B_x, B_y) que definen el rectángulo.

La salida estándar tiene que sacar n filas, cada una con un rectángulo, estando las filas ordenadas de menor a mayor área.

Veamos un ejemplo de entrada:

```
2
0 0 1 1
-1 -2 3 1
```

Y su correspondiente salida:

```
-1 -2 3 1
0 0 1 1
```

Ejercicio 93. Siguiendo las indicaciones del ejercicio sobre RPN de la sesión anterior, tu trabajo es elaborar una implementación basada en cadenas enlazadas.

Ejercicio 94. Lo cierto es que las pilas hacen un uso muy restringido de las cadenas enlazadas. Por ello, en este ejercicio, te proponemos elaborar un tipo abstracto de datos *listas*. Las funciones podrían ser algo como:

- `crear_vacía`
- `insertar_por_el_principio`
- `insertar_por_el_final`
- `insertar_por_posición`
- `longitud`
- `primero`
- `último`
- `iésimo`
- `borrar_primero`
- `borrar_último`
- `borrar_por_posición`

⏭ **Ejercicio 95.** En este ejercicio vamos a combinar conocimiento de varios temas y nos vamos a adelantar un poco para crear un tipo *tipo abstracto de datos*[9]. El tipo abstracto de datos que tendrás que implementar será el de las pilas usando cadenas enlazadas como estructura de datos.

De esta forma, las pilas serán **punteros a nodos de una cadena enlazada**. Vamos a construir un módulo para dicho tipo abstracto. Empezaremos con el *header*:

[9] Aunque en C cuesta ser realmente abstracto, lo vamos a intentar.

Listado 1: `pila.h`

```
#ifndef PILA_H
#define PILA_H

/* Las pilas serán punteros a struct nodo */
struct nodo {
  int dato;
  struct nodo *siguiente;
};

/* Crea una nueva pila vacía */
extern struct nodo *crear();

/* Decide si la pila está vacía */
extern int vacia(struct nodo *pila);

/* Coloca x en la cima de la pila y devuelve la "nueva" pila */
extern struct nodo *apilar(struct nodo *pila, int x);

/* Elimina la cima de la pila y devuelve la "nueva" pila */
extern struct nodo *desapilar(struct nodo *pila);

/* Devuelve la cima de una pila no vacía */
extern int cima(struct nodo *pila);

#endif
```

Tu labor será completar e implementar correctamente las funciones en el fichero `pila.c`. Te ofrecemos aquí el esqueleto:

Listado 2: `pila.c`

```
#include <stdlib.h>
#include "pila.h"

struct nodo *crear() {
  return NULL;
}

struct nodo *apilar(struct nodo *pila, int dato) {
  struct nodo *nuevo;
  /* TODO: corregir y completar */
  return nuevo;
}

struct nodo *desapilar(struct nodo *pila) {
  struct nodo *siguiente;
  /* TODO: corregir y completar */
```

```
  return siguiente;
}

int cima(struct nodo *pila) {
  /* TODO: corregir y completar */
  return 0;
}

int es_vacia(struct nodo *pila) {
  return pila == NULL;
}
```

Para ayudarte a saber que vas por el buen camino hemos elaborado el siguiente programar de test:

Listado 3: `pila_test.c`

```
#include <assert.h>
#include <stdio.h>
#include "pila.h"

#define N 20

int main() {
  int i;
  struct nodo *pila;

  pila = crear();

  /* Tras inicializar la pila debería estár vacía */
  assert(vacia(pila));

  pila = apilar(pila,42);

  /* Tras apilar un elemento la pila no debería estar vacía */
  assert(!vacia(pila));
  /* y la cima debería ser el 42 apilado */
  assert(cima(pila) == 42);

  pila = desapilar(pila);

  /* Tras desapilar el único elemento la pila debería estar vacía */
  assert(vacia(pila));

  /* Este bucle llena la pila con N enteros */
  for (i = 0; i < N; i++) {
    pila = apilar(pila, 2*i);
    /* Tras cada apilado la pila no debería estar vacía */
```

```
    assert(!vacia(pila));
    /* y la cima debería ser lo último apilado */
    assert(cima(pila) == 2*i);
  }

  /* Este bucle casi vacía la pila */
  for (i = N - 1; i > 0; i--) {
    /* Tras cada desapilado la pila debería tener los elementos
       apilados en orden inverso */
    assert(cima(pila) == 2*i);

    pila = desapilar(pila);

    /* Tras cada desapilado la pila no deberia estar vacía */
    assert(!vacia(pila));
  }

  pila = desapilar(pila);

  /* Tras el último desapilado, la pila deberia estar vacía */
  assert(vacia(pila));

  fprintf(stderr, "¡Todos los tests pasados!\n");

  return 0;
}
```

Y para que lo tengas aún más fácil, puedes utilizar el siguiente `Makefile`:

```
CFLAGS=-Wall -Werror -g -pedantic

pila_test: pila.o pila_test.o
	$(CC) $(CFLAGS) -o $@ $^

pila.o: pila.c pila.h

test: pila_test
	./pila_test

clean:
	rm -f *.o pila_test
```

Cuando todo esté correcto deberías experimentar la siguiente sesión Bash:

```
$ make
cc -Wall -Werror -g -pedantic   -c -o pila.o pila.c
cc -Wall -Werror -g -pedantic   -c -o pila_test.o pila_test.c
cc -Wall -Werror -g -pedantic -o pila_test pila.o pila_test.o
$ ./pila_test
```

```
¡Todos los tests pasados!
$ |
```

Ejercicio 96. En el ejercicio anterior (ejercicio 95), intenta reescribir el módulo introduciendo un nuevo tipo *pila* usando **#typedef** para así evitar tener que escribir el tipo

```
struct nodo_s *
```

de forma continuada.

Ejercicio 97. En este ejercicio te proponemos realizar una implementación de listas implementadas con un array redimensionable. El *header* `alist.h` de este tipo para listas podría ser el siguiente:

```
#ifndef _ALIST_H
#define _ALIST_H

struct alist_s {
  /*
   * Máxima cantidad de datos a almacenar, cuando last llegue a max
   * habrá que redimensionar data y modificar max
   */
  size_t max;
  /* El array redimensionable en cuestión */
  int *data;
  size_t last;
}

typedef struct alist_s *alist_t;

/* Crea una nueva lista vacía con capacidad inicial 1024 */
alist_t create_empty();

/* Decide si la lista l está vacía */
int is_empty(alist_t l);

/*
 * Anade el dato d en el índice i de la lista l, si no caben más
 * datos en el array redimensionable lo redimensiona al doble de
 * max.
 */
void add(alist_t l, int i, int d);

/*
 * Modifica el dato en la posición i de la lista l con el nuevo
 * dato d.
 */
void set(alist_t l, int i, int d);
```

```
/*
 * Elimina el dato en la posición i. Si el número de datos baja
 * por debajo de la mitad del máximo se redimensiona el array a
 * la mitad.
 */
void remove(alist_t l, int i);

/*
 * Devuelve el índice que ocupa el elemento d en la lista,
 * -1 si no está.
 */
int index_of(alist_t l, int d);

#endif
```

Implementa las funciones del *header* en el fichero `alist.h` así como un programa principal `test_alist.c` con las pruebas. Sugerencia: empieza por las pruebas.

8. *Módulos* en C

Ejercicio 98. Repasa las transparencias de clase. Además de explorar los nuevos elementos (**`enum`** y **`union`**), **el aspecto más importante es la combinación de tantas cosas vistas en la sesiones anteriores y la exploración a fondo** de ciertas partes de la sintaxis de C.

Ejercicio 99. Tu primera tarea va a consistir en implementar un programa que lea figuras geométricas de la entrada estándar e imprima el área de cada una de ellas en la salida estándar.

Cada línea de la entrada estándar representa una de estas figuras. Cada línea empieza con un string que indica el tipo de figura, a saber: `circulo`, `triángulo`, `rectángulo`. Dependiendo del tipo, le seguiran ciértos parámetros:

- Para `circulo`: El punto central con dos coordenadas **`int`** x e y separadas por espacios [10] y el radio (**`float`**) también separado por espacios.
- Para `triangulo`: Los tres puntos que definan el triángulo, seis enteros separados por espacios. Dos de los puntos siempre coincidirán en la coordenada y y forman parte de la base del triángulo.
- Para `rectángulo`: Los puntos sudoeste y noreste, cuatro enteros separados por espacios.

Veamos algunos ejemplos:

```
circulo 0 0 2
triangulo -1 0 2 1 0 3
rectángulo -1 -2 3 2
```

[10] Todos los puntos se representarán así.

Un poquito de ayuda:

- *Tendrás que declarar un struct por cada tipo de figura:* `círculo`,
- *Tendrás que utilizar union para describir que en la variable puedes tener cualquiera de las tres figuras.*
- *Necesitarás un último struct para poder discriminar el datos que tienes almacenado en el unuión*

Ejercicio 100. En la sesión de hoy, en clase, tienes múltiples ejercicios que han quedado propuestos. Te los recordamos aquí:

- Funciones sobre el tipo **`enum`** `mes`.
- Adaptar tu código a la convención de código `_t`, `_e`, `_s`, , `_u`.

Ejercicio 101. En la sesión de hoy, en clase, se ha dejado propuesto un ejercicio de ordenación de enteros en ristras. Recordatorio:

- Escribe un programa que ordene ristras de enteros de menor a mayor
- Cada ristra se representa de la siguiente forma en la entrada estánda:
  - Un entero positivo n en la primera línea
  - n enteros en las n siguientes líneas
- Por cada ristra, la salida de tu programa tiene los n enteros de la ristra ordenados de menor a mayor
- Tu programa debe parar cuando lea una ristra de 0 enteros

Veamos de nuevo un ejemplo de entrada (con dos ristras) y la salida esperada:

Entrada

```
2
7
1
3
1
4
2
0
```

Salida esperada

```

1
7

1
2
4

```

Las restricciones para este ejercicio son:

- **Usa el módulo de árboles binarios**

- **Evita consumir más memoria de la necesaria**
- **Presta especial atención a los *memory leaks***

La función de inserción en orden ya la implementaste en clase. En este ejercicio tendrás que escribir una función que haga un recorrido apropiado del árbol e imprima los enteros en orden. También tendrás que recorrer el árbol adecuadamente para liberar toda la memoria consumida.

Ejercicio 102. Entre los ejercicios de sesiones anteriores estaba la implementación del tipo abstracto de datos de las pilas: acotadas y no acotadas. Una vez que conocemos **`typedef`** y **`struct`** podemos hacerlo mucho mejor.

- En pilas acotadas podemos agrupar el array y el tamaño de la pila en un *struct*:

```
struct pila_acotada_s {
  int data[MAX];
  int top;
}

typedef struct pila_acotada_s pila_acotada_t;
```

- En pilas usando cadenas enlazadas, podemos usar las orientaciones de las transparencias:

```
/* Declaración de un struct, sólo el nombre */
struct nodo_pila_s;

/* Definición del tipo pila_t */
typedef struct nodo_pila_s *pila_t;

/* Definición del struct */
struct nodo_pila_s {
  int cima;
  pila_t resto;
};
```

Adapta las implementaciones de los ejercicios de sesiones anteriores.

Ejercicio 103. Implementar el tipo abstracto de datos de las colas (`cola_fun.c`),

- usando como estructura de datos una cadena enlazada,
- en la que el primer elemento de la cola sea el primero de la cadena,
- y respetando un interfaz *funcional* como el que indica el siguiente header:

Listado 4: `cola_fun.h`

```
#ifndef COLA_FUN_H
#define COLA_FUN_H

struct nodo_cola_fun_s;

typedef struct nodo_cola_fun_s *cola_fun_t;

struct nodo_cola_fun_s {
  int primero;
  cola_fun_t resto;
}

/* Crea una nueva cola vacía */
extern cola_fun_t crear();

/* Decide si la cola está vacía */
extern int vacia(cola_fun_t cola);

/* Inserta una nuevo elemento en la cola */
extern cola_fun_t insertar(cola_fun_t cola, int dato);

/* Elimina el primero de la cola */
extern cola_fun_t borrar(cola_fun_t cola);

/* Devuelve el primero de la cosa */
extern int primero(cola_fun_t cola);

#endif
```

Ejercicio 104. Implementar el tipo abstracto de datos de las colas (`cola.c`),

- usando como estructura de datos una cadena enlazada,
- en la que el primer elemento de la cola sea el primero de la cadena,
- y respetando un interfaz *procedural*[11]) como el que indica el siguiente header:

Listado 5: `cola.h`

```
#ifndef COLA_H
#define COLA_H

struct nodo_cola_s;

typedef struct nodo_cola_s *cola_t;
```

[11] Observa cómo cuando se quiere modificar la cola, ésta se pasa por referencia.

```
7
8  struct nodo_cola_s {
9    int primero;
10   cola_t resto;
11 }
12
13 /* Crea una nueva cola vacía */
14 extern void crear(cola_t *cola);
15
16 /* Decide si la cola está vacía */
17 extern int vacia(cola_t cola);
18
19 /* Inserta una nuevo elemento en la cola */
20 extern void insertar(cola_t *cola, int dato);
21
22 /* Elimina el primero de la cola */
23 extern void borrar(cola_t *cola);
24
25 /* Devuelve el primero de la cosa */
26 extern int primero(cola_t cola);
27
28 #endif
```

Ejercicio 105. En este ejercicio vamos a combinar conocimiento de varios temas para crear un tipo *tipo abstracto de datos*[12]. El tipo abstracto de datos que tendrás que implementar será el de las pilas acotadas de enteros.

Para implementar dicho tipo abstracto de datos tendrás que usar un array de enteros, **y nada más**. Como necesitas llevar la cuenta del número de elementos que hay en la pila y la capacidad máxima, la estructura a utilizar va a ser la siguiente:

- En la posición 0 del array guardarás la capacidad de la pila.
- En la posición 1 del array guardarás el número de elementos en la pila.
- En la posición 2 del array guardarás el primer elemento en ser apilado.
- En la posición 3 del array guardarás el segundo elemento en ser apilado.
- etc.

Vamos a construir un módulo para dicho tipo abstracto. Empezaremos con el *header*:

Listado 6: `pila_acotada.h`

```
1 #ifndef PILA_ACOTADA_H
2 #define PILA_ACOTADA_H
3
4 /* Inicializa una pila dejándola vacía */
```

[12] Aunque en C cuesta ser realmente abstracto, lo vamos a intentar.

```
5  extern void inicializar(int pila[],
6                          int capacidad);
7
8  /* Decide si la pila está vacía */
9  extern int vacia(int pila[]);
10
11 /* Decide si la pila está vacía */
12 extern int llena(int pila[]);
13
14 /* Coloca x en la cima de la pila (si la pila no está llena */
15 extern void apilar(int pila[], int x);
16
17 /* Devuelve la cima de la pila (si la pila no está  vacía) */
18 extern int cima(int pila[]);
19
20 /* Elimina el elemento en la cima de la pila */
21 extern void desapilar(int pila[]);
22
23 #endif
```

Tu labor será completar e implementar correctamente las funciones en el fichero `pila_acotada.c`. Te ofrecemos aquí el esqueleto:

Listado 7: `pila_acotada.c`

```
#include "pila_acotada.h"

void inicializar(int pila[],
                 int capacidad){
  /* Implementar correctamente */
}

int vacia(int pila[]) {
  /* Implementar correctamente */
  return 0;
}

int llena(int pila[]) {
  /* Implementar correctamente */
  return 0;
}

void apilar(int pila[], int x) {
  /* Implementar correctamente */
}

int cima(int pila[]) {
  /* Implementar correctamente */
```

```
  return 0;
}

void desapilar(int pila[]) {
  /* Implementar correctamente */
}
```

Para ayudarte a saber que vas por el buen camino hemos elaborado el siguiente programar de test:

Listado 8: `pila_acotada_test.c`

```
#include <assert.h>
#include <stdio.h>
#include "pila_acotada.h"

#define MAX 1000
#define N 20

int main() {
  int pila[MAX];
  int i;

  /* Para no cagarla con los parámetros de la prueba */
  assert(N < MAX);
  assert(N > 1);

  inicializar(pila, N);

  /* Tras inicializar la pila debería estár vacía */
  assert(vacia(pila));
  /* y por tanto no llena */
  assert(!llena(pila));

  apilar(pila,42);

  /* Tras apilar un elemento la pila no debería estar vacía */
  assert(!vacia(pila));
  /* tampoco llena (N es mayor que 1) */
  assert(!llena(pila));
  /* y la cima debería ser el 42 apilado */
  assert(cima(pila) == 42);

  desapilar(pila);

  /* Tras desapilar el único elemento la pila debería estar vacía */
  assert(vacia(pila));
  /* y no llena */
```

```
assert(!llena(pila));

/* Este bucle casi llena la pila */
for (i = 1; i < N; i++) {
  apilar(pila, 2*i);
  /* Tras cada apilado la pila no debería estar vacía */
  assert(!vacia(pila));
  /* tampoco llena */
  assert(!llena(pila));
  /* y la cima debería ser lo último apilado */
  assert(cima(pila) == 2*i);
}

apilar(pila, 0);

/* Tras apilar un elemento más, la pila no debería estar vacía */
assert(!vacia(pila));
/* y ahora sí, ya debería estar llena */
assert(llena(pila));
/* Y la cima debería ser lo último apilado */
assert(cima(pila) == 0);

desapilar(pila);

/* Este bucle casi vacía la pila */
for (i = N - 1; i > 1; i--) {
  /* Tras cada desapilado la pila debería tener los elementos
     apilados en orden inverso */
  assert(cima(pila) == 2*i);

  desapilar(pila);

  /* Tras cada desapilado la pila no deberia estar vacía */
  assert(!vacia(pila));
  /* pero tampoco llena */
  assert(!llena(pila));
}

desapilar(pila);

/* Tras el último desapilado, la pila deberia estar vacía */
assert(vacia(pila));
/* y no llena */
assert(!llena(pila));

fprintf(stderr, "¡Todos los tests pasados!\n");
```

```
  return 0;
}
```

Y para que lo tengas aún más fácil, puedes utilizar el siguiente `Makefile`:

```
CFLAGS=-Wall -Werror -g -pedantic

pila_acotada_test: pila_acotada.o pila_acotada_test.o
	$(CC) $(CFLAGS) -o $@ $^

pila_acotada.o: pila_acotada.c pila_acotada.h

test: pila_acotada_test
	./pila_acotada_test

clean:
	rm -f *.o pila_acotada_test
```

Cuando todo esté correcto deberías experimentar la siguiente sesión Bash:

```
$ make
cc -Wall -Werror -g -pedantic   -c -o pila_acotada.o pila_acotada.c
cc -Wall -Werror -g -pedantic   -c -o pila_acotada_test.o pila_aco...
cc -Wall -Werror -g -pedantic -o pila_acotada_test pila_acotada.o ...
$ ./pila_acotada_test
¡Todos los tests pasados!
$ |
```

Ejercicio 106. Modifica tu implementación de *polaca inversa* `rpn` para usar tus recién estrenadas pilas acotadas.

Ejercicio 107. ¿Recuerdas las órdenes bash que generaban listas de enteros? Para probar el ejercicio anterior, te sugerimos que las modifiques para generar más de una ristra de enteros.

Ejercicio 108. Tu labor en este ejercicio será modificar el módulo `generador_lcg` y enriquecerlo de la siguiente forma.

- La idea es tener un módulo desde el que poder cambiar en tiempo de ejecución los parámetros a, c y m del generador (ahora no es posible porque se han definido como macros).
- El módulo va a contener tres variables: `lcg_a`, `lcg_c`, y `lcg_m`.
- Además, se expondrá una operación para establecer la *semilla* del generador, es decir, el valor de X_0.

Para facilitar la tarea dispones del *header* aquí:

generador_lcg.h

```
/* Parámetros a, c, y m del generador */
extern int lcg_a = 1;
extern int lcg_c = 1;
extern int lcg_m = 1;

/* Resetea el generador colocando el valor de X_0 al valor semilla */
extern void lcg_resetear(int semilla);

/* Devuelve el valor de X y genera el siguiente */
extern int lcg_generar();
```

Para que puedas ver si lo que has hecho funciona correctamente puedes usar este *tester*:

test_lcg.h

```
#include <stdio.h>
#include <assert.h>
#include "generador_lcg.h"
int main() {

  lcg_a = 7;
  lcg_c = 1;
  lcg_m = 11;

  lcg_resetear(9);

  assert(lcg_generar() == 9);
  /* 7 * 9 + 1 % 11 == 9 */
  assert(lcg_generar() == 9);

  lcg_resetear(0);

  assert(lcg_generar() == 0);
  /* 7 * 0 + 1 % 11 == 1 */
  assert(lcg_generar() == 1);
  /* 7 * 1 + 1 % 11 == 8 */
  assert(lcg_generar() == 8);
  /* 7 * 8 + 1 % 11 == 2 */
  assert(lcg_generar() == 2);
}
```

9. Ejercicios extra (GNU/Linux y libc)

Ejercicio 109. En el sistema operativo GNU/Linux hay infinidad de programas extraordinariamente inspiradores y retadores. Explóralos a través del manual. Hablamos de programas como `cat`, `comm`, `cut`, `diff`, `echo`, `find`, `grep`, `head`, `join`, `ls`, `mkdir`, `paste`, `rmdir`, `rm`, `sort`, `tail`, `tee`, `uniq`, o `wc`.

Nota: observa que para comprobar que tu implementación es correcta puedes usar el propio programa en cuestión que seguro que está instalado en tu ordenador `;)`.

Ejercicio 110. Intenta implementar los programas mencionados en el ejercicio anterior. Obviamente todos esos programas tienen una gran cantidad de *flags* que los hace bastante complejos por lo que te sugerimos hacer sólo una versión simplificada de los mismos, por poner dos ejemplos de simplificación:

- `grep [PALABRA] [FICHERO]` busca la PALABRA en el FICHERO.
- `find [NOMBRE_FICHERO] [DIRECTORIO]` busca el fichero con nombre NOMBRE_FICHERO en el directorio DIRECTORIO recursivamente.

Ejercicio 111. Otra fuente de ejercicios muy interesantes es la implementación de algunas funciones de la Libc[13]. Algunas propuestas:

- `strdup` usando sólo `malloc`.
- `strdup` usando sólo `malloc` y `strlen`.
- `strdup` usando sólo `malloc`, `strlen` y `memcpy`.
- `memset` sin usar ninguna otra función de la libc.
- `calloc` usando sólo `malloc`.
- etc.

[13] Puedes implementar la función usando la misma cabecera pero como nombre de la función usa el prefijo `mi_`, ej. `mi_strdup`