240204 – Estructuras de datos

Enunciados de prácticas de laboratorio

Federico Fariña[1]

Dpto. de Estadítica, Informática y Matemática
Universidad Pública de Navarra
Campus de Arrosadía
31006 Pamplona

February 4, 2020

[1]Aunque muchas ideas se han obtenido de prácticas más antiguas cuya definición corrió a cargo de José Ramón Garitagoitia. Merece un agradecimiento especial José Ramón González de Mendívil por la profunda revisión que ha hecho a la versión preliminar de este documento. También es destacable la participación de José Enrique Armendariz que permitió seguir eliminando erratas y mejorando el contenido.

Práctica 1

Recapitulación y novedades

1.1 Introducción

En la asignatura de Programación ha recibido una introducción a la codificación de algoritmos en lenguaje de programación C. En esta asignatura seguirá aprendiendo a codificar en C algoritmos más complejos. Por lo tanto, todo lo que aprendió en Programación sigue siendo necesario para seguir el laboratorio de Estructuras de Datos. En esta práctica se repasan y amplían algunos conceptos impartidos en Programación. Descubrirá que, además, la práctica incluye algunas novedades. El laboratorio de Programación está pensado para que usted adquiera buenas prácticas para crear código legible y fácilmente verificable. No obstante, en ocasiones, es mucho más eficiente el código que se salta algunas de esas buenas prácticas. En este laboratorio jugamos con la idea de que estamos codificando código verificado formalmente, por lo que daremos más importancia a la eficiencia.

A pesar de lo dicho, las normas de estilo que se utilizaron en Programación, a excepción de las que le indique, siguen siendo de obligado cumplimiento. Y se añaden algunas nuevas (quizá las considere pesadas, pero tienen que ver con el concepto de calidad de software, que es uno de los que se tienen en cuenta a la hora de evaluar la asignatura):

- Todo programa debe comenzar, en tiempo de ejecución, identificando a su autor y la fecha en la que se creó. A continuación debe explicar para qué sirve.
- Siempre que un programa solicite un dato al usuario debe explicar los requisitos que debe cumplir dicho dato. Sería deseable que también se compruebe que el dato leído realmente cumple sus requisitos y que se pida un nuevo valor si el introducido no es correcto.
- Todo fichero de código fuente debe comenzar indicando quién es su autor, en que fecha se ha creado y un histórico de versiones y modificaciones, si es que éstas han existido.
- En lo que sigue, no se hará referencia a la especificación de acciones y funciones. A pesar de ello, no debería olvidar que la especificación es una parte de algoritmos, acciones y funciones. Añádala siempre en su código mediante comentarios.
- Es recomendable que el programa siempre permita que sea ejecutado múltiples veces para que se pueda comprobr su funcionamiento con diferentes entradas.

1.2 El lenguaje de programación C

En este apartado vamos a analizar todo lo que le contaron en Programación. Se puntualizarán algunos aspectos y ampliarán otros. En algunos apartados le pido que pruebe algunas cosas. Obviamente, no es obligatorio, pero sí muy conveniente.

Historia y estándares

Como le han contado Brian Kernighan y Dennis Ritchie desarrollaron el lenguaje C entre 1969 y 1972. Lo que no le han dicho es que, en muy poco tiempo, aparecieron muchas empresas que vendían compiladores de C. El problema fue que, aunque en esencia todas seguían la especificación informal del lenguaje creado por Kernighan y Ritchie, las diferentes variantes del lenguaje no eran compatibles entre sí. Esto llevó a que en 1983, el Instituto Nacional Estadounidense de Estándares (ANSI) formalizase un estándard, denominado ANSI C o C89 de obligado cumplimiento (al menos, en teoría). Ese mismo estándard, con pequeñas modificaciones, fue adoptado por la Organizacin Internacional para la Estandarización (ISO) dando lugar al denominado C90. En 1999 la ISO generó un nuevo estándard, C99, que fue adoptado por ANSI en el 2000. Posteriormente, en 2011 ISO publicó un nuevo estándard, C11 (también conocido como C1X) y uno nuevo en 2018 denominado C18. En cada modificación se han incluido características nuevas que hacen, en algunos casos, que el código deje de ser compatible con versiones anteriores.

Para acabar de liar el tema, cada fabricante de compiladores ha seguido *tan sólo hasta cierto punto* los estándares. En realidad, muchas veces ISO se ha limitado a hacer oficiales modificaciones que eran habituales en algunos compiladores de C comerciales.

A pesar de esto, no se preocupe, las diferencias entre versiones son mínimas y se centran fundamentalmente en las librerías. No suele haber problemas compilando código (y si los hay, se le puede decir a los compiladores que estándard se está usando). Eso sí, conviene que sepa en que estándard está usted trabajando.

1.2.1 Estructura típica de un programa C

En Programación se ha explicado que la estructura de un programa en C es la siguiente:

```
declaración de importaciones
definición de constantes
definición de nuevos tipos
definición de funciones
int main(void) {
    declaración de variables locales
    instrucciones ejecutables
    return 0;
}
```

La realidad es que esa es la estructura deseable, pero no todos los programas de C la cumplen. Hay varias razones para ello. En primer lugar, las evoluciones históricas han hecho surgir características que rompen esa estructura. Pero hay una razón mucho más importante. Cuando se creó C se buscaba un lenguaje de nivel medio (cercano al ensamblador) y no se tenían claros los conceptos de Ingeniería del Software, lo que llevó a tomar algunas malas decisiones. Una de las más utilizadas son las denominadas variables

globales. Son variables que se definen fuera del `main` (como el resto de definiciones) o, incluso, fuera de nuestro programa (tranquilo, ya hablaremos de esto más adelante). A esto hay que añadir que todo lo que aparece antes del main no tiene porqué seguir el orden que aparece arriba. En realidad, el orden es el que usted elija, al menos si quiere que nadie entienda su código.

A modo de ejemplo, observe el fichero `contrajemplo.c` que le he suministrado a través de miAulario. Compílelo, verá que no hay ningún problema. Es correcto. Vamos a analizarlo en detalle.

1. Líneas 1..13: Ahí comienza un comentario que da toda la información que le he indicado arriba que todo programa debe incluir. Es, y perdón por la chulería, perfecto. Comience de igual modo sus programas.

2. Líneas 14..18: Este nuevo comentario indica que comienza la zona de prototipos de las funciones. Puede que recuerde que en C no puede usar nada hasta que no se haya *definido*. En realidad, es falso. La norma indica que no puede usar nada que no haya sido definido o *declarado* previamente. Declarar algo consiste sólo en decir que existe y explicar su aridad (los tipos que tienen que ver con ese algo). La diferencia es muy poco evidente en las variables (al menos, de momento), pero es fundamental en las funciones. Declarar una función significa escribir su cabecera con los tipos de los parámetros (de hecho, el nombre no es necesario... en algunos estándares) y el tipo de salida. En el fondo le estamos indicando al compilador que existe en algún lugar una definición completa de una función con las características indicadas. De este modo, el compilador puede compilar cualquier aparición del nombre declarado de modo que se comunique correctamente con el código que se creará al leer la definición de la función.

3. Líneas 19..24: Es la especificación, nuevamente perfecta, de la función `mcdEduc`.

4. Línea 25: Es el prototipo de la función `mcdEduc`.

5. Línea 26: Una definición de un tipo (realmente un alias). Obviamente aunque puede estar aquí, el hecho de que no se encuentre en el sitio esperado hace que el código sea difícil de leer y mantener.

6. Línea 27: La definición de una variable global. Es decir, una variable accesible por cualquier función que se escriba en el fichero. Las variables globales permiten ahorrar mucho código, pero son una gran fuente de problemas. Es mejor no usarlas, pero son muy usadas en la comunidad de programadores no profesionales de C.

7. Línea 28: Como puede ver, los `#include` pueden no estar al principio. Nuevamente, el hecho de que no estén en su sitio habitual confiere malas características al código, pero al compilador no le parece mal que esté ahí.

8. Líneas 30..36: Otro comentario perfecto que indica la especificación de la función cuyo prototipo aparece a continuación. Se comprueba nuevamente que la ordenación no es necesaria, pero debe admitir que esto es un lío.

9. Líneas 38..44: Especificación del programa principal.

10. Línea 45: Comienza la definición del programa principal.

11. Líneas 46..47: Declaración (o definición, en variables es, de momento, lo mismo) de las variables locales del programa principal. Note que utilizamos el tipo `numero` anteriormente definido.

12. Líneas 49..54: Información que todo programa con salida por pantalla debe ofrecer.

13. Líneas 55..70: Bucle que asegura que el programa se pueda repetir múltiples veces.

14. Líneas 56..64: Bucle que permite leer los dos valores solicitados. Fíjese que dentro del bucle se declara una nueva variable. Esta construcción es válida desde C90, pero desde las primeras versiones de C se pueden declarar variables entre las sentencias del código.

15. Líneas 60..63: Bucle que fuerza a que la entrada verifique sus precondiciones.

16. Línea 65: Llamada a una función. Note que el compilador aún no sabe cómo es esa función, pero como ha leído su prototipo se fia del programador y no genera ningún problema.

17. Líneas 66..68: Salida de resultados.

18. Líneas 69: Fíjese en lo que está leyendo esta sentencia. El formato indica `''\n%c''`. Hasta ahora habíamos leído enteros y `scanf` avanzaba en la entrada buscando un entero y obviando otras cosas como blancos y saltos de línea. Pero ahora queremos leer un carácter y hay que tener cuidado. Lo último que escribió el usuario, si siguió nuestras instrucciones, fue un número y después pulso enter. La función `scanf` leyó el entero pero dejó el salto de línea en el buffer de entrada. Si no leemos dicho salto, la variable `quiereSalir` se cargaría con `\n` y estaríamos en un bucle infinito. Tenga siempre mucho cuidado con ajustar las lecturas con lo que le pide al usuario que escriba.

19. El resto del programa no tiene nada demasiado raro. Tan sólo que, como le habíamos prometido al compilador, al final le damos la definición de las funciones que habíamos declarado. Si se fija en la última, le hemos cambiado el nombre a uno de sus parámetros, pero eso no importa. Importan los tipos.

Como ve, la estructura puede diferir mucho de la conoce. A pesar de ello, conviene, por diversas razones, que sus programas sigan el esquema indicado al principio de este apartado y que no usen variables globales.

1.2.2 Ámbito de validez de las declaraciones/definiciones

Se ha visto en el programa de ejemplo que el programa principal usa la variable global `m` sin problema. También podrían utilizarla sin problema las funciones del final. Ahora bien, si se fija en la última función, hay algo raro. También se accede a una variable `m`. Pero ¿Es la misma? La respuesta es compleja y tiene que ver con el concepto que abordamos en este apartado.

Existen múltiples lenguajes que se organizan en bloques. Un bloque es una zona del archivo dónde se pueden definir/declarar nombres de cosas o poner sentencias. C se organiza en bloques, o casi. Un bloque de C es cualquier cosa entre { y }. Hemos visto que las definiciones de tipos, prototipos de funciones y definiciones de funciones aparecen en el archivo fuera de cualquier otra estructura, pero no ocurre lo mismo con todas las variables. En C podemos entender que hay dos tipos de variables, las globales y las automáticas. Las variables globales son las que se declaran fuera de cualquier función y son accesibles desde cualquier función de un programa. Las variables automáticas son los parámetros de las funciones y sus variables locales. Dentro de una función podemos crear bloques (si se fija, podemos ver que una función no es más que un bloque con nombre y parámetros) y dentro de esos bloques podemos declarar más variables automáticas. El ámbito de validez (es decir, la zona de código en la que realmente existe una variable automática) es el bloque en el que se declara.

Si lo piensa, lo dicho es un problema. Dentro de un bloque son accesibles las variables globales, las variables declaradas dentro del bloque y, si el bloque está dentro de otro bloque, las declaradas en dicho bloque externo... ¿Qué ocurre si alguna variable declarada en el bloque se llama igual que una externa (global o de bloques en los que está el bloque)?. Se aplica la regla de anidación más próxima, que viene a decir que la declaración correcta es la más próxima de las ya realizadas[1].

Para clarificar la idea, veamos un ejemplo. Suponga un programa que contenga el siguiente código:

```
1)  int m;
2)  {
3)      float m;
4)      {
5)          char m;
6)          m:= ...;
7)      }
8)      {
9)          m:= ...;
10)     }
11)     m:= ...;
12) }
13) {
14)     m:= ...;
15) }
```

En la línea 6 estamos accediendo al caracter declarado en la línea 5. En las líneas 9 y 11 accedemos al real declarado en la línea 3. Por último, en la línea 14 accedemos al entero declarado en la 1. En muchos sitios leerá que una declaración esconde cualquier declaración anterior con el mismo nombre.

Ah, y falta aclarar un punto. El bucle `for` de C es un bloque en si mismo. Es decir, si declaramos algo entre los paréntesis del `for`, su ámbito de validez es todo el `for`.

[1] Una vez que hemos eliminado, claro está, las declaradas de bloques ya cerrados

1.2.3 El Preprocesador de C

Como ya sabe, el preprocesador (denominado a veces macroprocesador) de C es un programa que el compilador invoca antes de que comience la compilación real. Realmente es el preprocesador el que construye la entrada al compilador.

El preprocesador de C (como ya debería saber, estamos hablando de `cpp`) es un programa que trabaja transformando ficheros de texto. Localiza alguna de las directivas del preprocesador, que tienen que estar al principio de alguna línea y aplica la orden dada por dicha directiva. Algunas de las directivas del preprocesador de C útiles para este curso son:

- `#define`. Como sabe se puede utilizar para definir constantes, aunque en realidad se utiliza para definir macros. Una macro es una representación de un valor o (grupo de) sentencia(s) mediante un nombre. Cada vez que el preprocesador encuentra el nombre de una macro definida, lo sustituye textualmente por el valor o sentencia que representa. Las macros de valores son muy sencillas. Responden a la estructura

  ```
  #define NOMBRE valor
  ```

  donde el `NOMBRE` de la macro se escribe siempre (por convenio) en mayúsculas. Se utilizan como parte de las buenas prácticas de programación para evitar el uso de los denominado números mágicos, pero, como veremos después, pueden tener otros usos. El `valor` de la macro puede ser cualquier cosa (pero mejor que no exceda de una línea). Podemos incluso tener macros que no tienen `valor`, es decir, simplemente se usan para indicar que se ha definido el nombre de una macro.

  Pero, como hemos dicho las macros también pueden definir sentencias. Se usan, a veces, para acelerar el acceso a las funciones de modo que no se pierda tiempo en las llamadas. Se diferencian de las macros normales en que su nombre finaliza con unos parámetros entre paréntesis. Su uso es complejo, por lo que merece la pena ver algún ejemplo: suponga que quiere crear una macro para tener a su disposición la función de conversión de grados celsius a fahrenheit. Puede crear la macro:

  ```
  #define FAH_A_CELS(x) (1.8*(x)+32)
  ```

  A partir de ahí, cada vez que escriba `FAH_A_CELS(x)` dando un valor a x, la macro se expandirá. Es decir, si pone `FAH_A_CELS(5)`, el preprocesador lo convertirá en (1.8 * (5)+32). Es fundamental poner todos los paréntesis que ve para evitar problemas. Piense en el resultado que obtendría en los siguientes casos:

  ```
  FAH_A_CELS(x) * 5 // 1.8*x+32 * 5 sin los parénteis externos.
  FAH_A_CELS(x + y) // 1.8*x + y+32 sin los parénteis de la variable.
  ```

  Los valores admiten operadores de caracteres, pero eso lo dejo ya a su curiosidad. No lo va a usar este curso.

- `#include`. Esta directiva le dice al preprocedador que debe copiar literalmente el fichero que se le indica a continuación en el lugar dónde aparezca la directiva[2]. Los ficheros a incluir pueden estar en el directorio estándard de los ficheros de cabecera

[2]Le habrán dicho que es para incluir librerías. Esa afirmación es una simplificación no muy correcta.

de C (que se define al instalar el compilador), lo que se indica rodeando el nombre del fichero por los símbolos `<` y `>` o bien en el directorio actual[3], lo que se indica rodeando el nombre del fichero por los s símbolos `''` y `''`.

- `#undef`. Permite eliminar la definición de una macro. Una vez se ha eliminado la definición, si el preprocesador encuentra el nombre de la macro ya no la expande.

- `#if, #elif, #else, #endif, #ifdef, #ifndef`. Este grupo de directivas permite decidir que algunas partes del código se compilan y otras no en función de condiciones establecidas sobre constantes y macros. A `#if` le debe seguir una expresión que se evalúa y si es distinta de cero (verdadera) las lineas siguientes se conservan. Obviamente, los `#elif` nos permiten establecer una condición múltiple y `#else` indica el código que debe permanecer si todo lo demás es falso. Por último `#endif` indica que se acaba la parte de código cuya inclusión depende de condiciones. Muchas veces la condición que se mira es simplemente si una macro está definida o no. Para ello, en lugar de `#if` se emplean `#ifdef` y `#ifndef`.

  Estas directivas permiten hacer lo que se denomina compilaciones condicionales de modo que, por ejemplo, se usen unas instrucciones u otras en función del sistema operativo y para evitar las inclusiones múltiples de un mismo archivo.

Podemos indicarle al compilador que sólo queremos que preprocese un archivo dándole la opción `-E`. El resultado saldrá a través del fichero estándar de salida.

1.2.4 Tipos de datos simples

Como ya sabe, un tipo de dato es una definición de un conjunto de valores y un conjunto de operaciones que se pueden realizar con ellos. Aunque esto es cierto, en C hay una gran diferencia entre la teoría y la realidad. Si somos estrictos, en C existen sólo dos tipos de datos simples: el entero y el real. Pero C permite utilizar muchas otras denominaciones que acotan el conjunto de valores

Tipos enteros

A continuacin le indico el conjunto completo de tipos que son internamente enteros[4]:

- El tipo `char`: Una variable `char` es una variable que ocupa 1 byte (8 bits) de memoria. Su contenido codifica un entero que se puede decodificar como potencias simples de 2 si es `unsigned` o utilizando complemento a 2 si es `signed`[5]. Usualmente se entiende que una variable `char` es un carácter. No es del todo cierto. Lo que ocurre es que el compilador de C utiliza variables `char` en varias funciones que trabajan directamente con las tablas ASCII que asocian a cada carácter un valor entero. Recuerda, cuando escribes `'a'` en C, estás escribiendo un número entero.

[3]En realidad pueden estar en cualquier sitio. Podemos compilar con la opción `-I lista_directorios` para indicarle al compilador dónde están los ficheros de cabecera que necesita. De hecho, la diferenciación entre los estándard y no estándard ya no es necesaria, aunque se mantiene como norma de estilo por claridad.

[4]Puede observar en el header <limits.h> las definiciones de las macros que indican los valores mayores y menores de cada tipo en cada implementación/arquitectura de C

[5]Si escribimos simplemente `char` el compilador entiende que la variable es `signed char`, por lo que no es muy habitual ver `signed char`

- El tipo `int`: Una variable `int` es una variable que ocupa 1 palabra de memoria[6]. Su contenido codifica en binario un entero positivo si es `unsigned` o lo codifica en complemento a 2 si es `signed`[7]. Si vamos a emplear enteros más pequeños podemos emplear `short int` o `unsigned short int`[8] [9] y si queremos emplear enteros más grandes, podemos usar `long int` o `unsigned long int`[10].

  Un aspecto importante a tener en cuenta es que `int` es el tipo por defecto. Es decir, ante una variable que no se ha declarado, el compilador la tratará como `int`. Esto se aprovechó hace tiempo para escribir código de forma más rápida, pero se consideró muy pronto una mala práctica de programación. De hecho, a partir de C90 se considera un error. A pesar de ello, muchos compiladores siguen funcionando de esa manera dando, en todo caso, un `warning`.

- El tipo `bool`. Como ya sabe, en C no existe el tipo booleano como tipo básico. Sin embargo, dada la comodidad expresiva que proporciona dicho tipo, a partir del stándard C99 se definió el header `stdbool.h` que define el tipo `bool` y las constantes `true` y `false`. Obviamente, no son más que macros. Internamente siguen siendo enteros[11].

- Los tipos enumerados: como ya sabe puede crear un enumerado simplemente escribiendo `enum` {`const`$_1$`, const`$_2$`, ..., const`$_n$}. En realidad lo que está es creando un conjunto de constantes a las que el compilador va a asociar un número (del 0 hasta el n-1). Por tanto, siguen siendo enteros.

- Los punteros son variables cuyo contenido representa una dirección de memoria. Dado que la memoria no es más que una tabla de palabras, en el fondo un puntero no es más que un entero (`unsigned`). Eso sí, un entero con el que es mejor no operar sin saber lo que se hace.

Tipos reales

El panorama con los tipos reales es similar, aunque menos complejo. En C existen:

- El tipo `float`: Una variable `float` es una variable que almacena un valor real codificado en coma flotante.

- El tipo `double`: Una variable `double` es una variable que almacena un valor real codificado en coma flotante con precisión doble. Obviamente ocupa, al menos el doble que un float. Si se desea aún más precisión, se puede emplear `long double`, que, obviamente, ocupa aún más espacio.

[6] Según el estándar ANSI deberían ser 2 bytes (16 bits) de memoria, pero hace tiempo que ese límite se rompió. En realidad el tamaño concreto depende del compilador que esté usando e incluso de la máquina en que esté instalado

[7] Nuevamente, sii escribimos `int` el compilador entiende que la variable es `signed int`, por lo que no es muy habitual ver `signed int` escrito. También es cierto que si escribimos `unsigned` el compilador entiende que es `unsigned int`

[8] Si escribimos sólo `short` el compilador entiende que es `short int`

[9] Según ANSI, un short int ocupa lo mismo que un char, pero en la malloría de compiladores actuales esto no es cierto

[10] Nuevamente si escribimos sólo `long` el compilador entiende que es `long int`

[11] En concreto `false` es 0 y `true` es 1

El tipo `void`

En C todo tiene tipo. Por lo tanto toda función tiene que *devolver* un valor de algún tipo. Incluso aquellas que no devuelven nada. Es por ello que aparece el tipo `void`. Un tipo sin valores ni ocupación de memoria.

Aunque puede parecer un tipo absurdo se emplea en múltiples situaciones. Ya habrá comprobado que se utiliza para indicar que una función no devuelve nada o para indicar en su prototipo que no tiene parámetros formales. Pero tiene un uso más interesante. Podemos declarar cosas como `void * p`. Un puntero a void, también llamado puntero genérico puede después apuntar a cualquier cosa haciendo la asignación a través de una operación de cast (`q=(int*)p;`).

1.2.5 Operadores, precedencia y asociatividad

C dispone de multiples operadores, algunos redundantes. En Programación se le limitó el conjunto de operadores a utilizar para facilitar la verificación de los algoritmos. En Estructuras de Datos vamos a relajar estas limitaciones. A continuación le presento una tabla en la que aparecen todos los operadores de C por orden de prioridad. En ausencia de paréntesis, tienen preferencia los operadores de menor nivel (los de la parte de arriba de la tabla) frente a los de mayor nivel[12]. Si tenemos que elegir el orden para operar con operadores de la misma línea, viene dado por su asociatividad: izquierda quiere decir que se operan de izquierda a derecha y, obviamente, derecha lo contrario.

Nivel	Descripción	Operador	Asociatividad
1	Paréntesis y operadores de miembro de estructura	() [ ] . ->	izquierda
2	Incremento, decremento, unarios aritméticos unarios lógicos, puntero, dirección, cast y tamaño	++ – + - ! ~ * & (tipo) sizeof()	derecha
3	Multiplicación, división y módulo	* / %	izquierda
4	Suma y resta	+ -	izquierda
5	Desplazamiento de bits	>> <<	izquierda
6	Comparaciones de superioridad	< <= >= >	izquierda
7	Comparaciones de igualdad	== !=	izquierda
8	Y bit a bit	&	izquierda
9	O exclusivo (exor)	^	izquierda
10	O bit a bit	|	izquierda
11	Y lógico	&&	izquierda
12	O Lógico	||	izquierda
13	Ternario condicional	?:	derecha
14	Asignaciones	= *= /= %= += -= >>= <<= &= ^= |=	derecha
15	Coma	,	izquierda

Un detalle que quizá le haya resultado curioso es que le diga que el simbolo `=` es un operador. Es una de las particularidades de C más extrañas y quizá menos usada. La sentencia

[12] Algunos de estos operadores no los conoce porque no hemos llegado a estudiar las partes de C dónde son necesarios. Llegaremos pronto.

`a = b = 3*c;`es perfectamente correcta. En el fondo es otra forma, probablemente mucho menos clara, de escribir `b = 3*c; a=b;`

1.2.6 Evaluación de expresiones booleanas en C

Existen dos formas de evaluar expresiones booleanas en diferentes lenguajes de programación. La primera, probablemente la que le resulta más natural, consiste en evaluar la expresión completa calculando un valor (`verdadero` o `falso`). Esta es la que se supone en lenguaje algorítmico. C emplea otra que es más eficiente pero hay que entender bien. Para que la entienda quizá merezca la pena un ejemplo: La forma habitual de cálculo ante la expresión `(5 < 5) && (5 > 3)` sería pasar por los resultados parciales `falso && (5 > 3)` y `falso && verdadero` para obtener el resultado `falso`. Sin embargo, C lo que hace es calcular en primer lugar `falso && (5 > 3)` tras lo cual, como la y lógica de `falso` con cualquier cosa es `falso`, al leer la `&&` no calcula la siguiente comparación, con lo que llega ya al resultado `falso`. Esta diferencia tiene múltiples implicaciones. La primera tiene que ver con el optimizador de código del compilador. El de C, al contrario de otros, no reordena nuestras expresiones lógicas, las calcula en el mismo orden en que nosotros las hemos escrito. Lo cual lleva a una implicación que tiene mucho que ver con la codificación de algoritmos. Podemos incluso cambiar algunos esquemas conocidos, como los de búsqueda, si tenemos cuidado en el orden en el que escribimos las expresiones booleanas. Debe quedar claro que obtendremos código más eficiente, pero corremos más riesgos de equivocarnos.

1.2.7 Otras cosas que afectan al control de flujo

Además de las sentencias de control de flujo que conoce, C tiene otras. No se le han explicado porque, en muchos casos, no son recomendables. Sin embargo, por completitud, debería saber al menos algo sobre ellas:

- La sentencia `switch`: En algunos casos tenemos que programar una composición alternativa en la que todas las condiciones son comparaciones de una variable o expresión con un grupo de constantes. Para esos casos C nos ofrece la sentencia `switch`. Esta sentencia tiene la siguiente estructura:

```
switch (expresión) {
    case constante_1:  sentencias_1
    case constante_2:  sentencias_2
...
    default:  sentencias_n
}
```

  Teniendo en cuenta que las sentencias pueden ser una composición secuencial de otras sentencias e incluso una cadena vacía. Eso sí, debe saber que el comportamiento de `switch` es peculiar. C evalúa la expresión y compara el resultado con todas las constantes. Si no coincide con ninguna ejecuta las sentencias del `default`, si es que existe. Pero, si coincide con alguna constante ejecutará todas las sentencias que aparezcan después hasta que se encuentre bien una sentencia `break` o el final del

`switch`. Este comportamiento, aparentemente extraño permite conectar multiples valores con las mismas sentencias.

- La sentencia `break`: Como acabamos de ver, esta sentencia se puede usar para marcar el final del grupo de sentencias a ejecutar en una de las opciones del `switch`, pero tiene otro uso. Si aparece en un bucle, fuerza a que el programa salga del bucle, independientemente de las condiciones de salida. Obviamente, este comportamiento hace que esta sentencia sea muy poco recomendable (salvo dentro del `switch`).
- La sentencia `continue`: Esta es otra sentencia de control de flujo cuyo uso es aún más conflictivo. Se usa dentro de un bucle para indicar que se ha finalizado la ejecución de la presente iteración del bucle. Es decir, fuerza un salto al punto donde se comprueba si se dan las condiciones para salir del bucle o no.
- La sentencia *maldita* `goto`. C admite la sentencia `goto`. Esta sentencia permite ir a ejecutar cualquier línea de código que tenga una etiqueta. Mejor no le doy más detalles porque se considera un tremendo error de programación utilizarla (salvo que se crea que sabe más que nadie de esto de C, claro).

1.3 Actividades a realizar

Aunque no le he contado nada muy nuevo, puede merecer la pena que *perdamos* un poco de tiempo haciendo un par de programas de repaso. Recuerde que todos sus programas deben comenzar mostrando por pantalla su nombre, el nombre de su programa y la fecha de creación del programa. Esa misma información debe estar escrita como comentario al principio de todos los ficheros fuente que escriba.

La mayoría de estas actividades consisten en implementar los algoritmos que se han diseñado para resolver los ejercicios del tema 1. Use las soluciones que se obtuvieron en clase. Obviamente sus programas van a consistir en una fase de comunicación con el usuario para solicitarle datos con los que operar, la llamada a un conjunto de acciones y funciones que trabajarán con esos datos y una nueva fase de comunicación para mostrar los resultados. En la primera fase tenga en cuenta que algunos de sus algoritmos tienen requisitos. Sus programas deben comprobar que los datos que introduce el usuario verifican dichos requisitos y en caso contrario volver a pedir un nuevo dato. También deberían encargarse de explicarle al usuario cuáles son esos requisitos (para que no se equivoque).

1. **Algoritmo Vectores** (`vectores.c`)

   Créese un programa que solicite al usuario 10 valores reales, los inserte en un vector `v1`, copie `v1` en otro vector `v2`, calcule el producto escalar de `v1` consigo mismo, ordene los elementos de `v1` de menor a mayor, calcule el producto escalar del nuevo vector con `v2` y saque por pantalla la diferencia, en valor absoluto, entre ambos resultados. Los algoritmos necesarios para el producto escalar y la ordenación deben ser los que se han diseñado en la clase de problemas del tema 1. Dichos algoritmos deben implementarse en forma de acciones o funciones.

2. **Suma de matrices cuadradas 3 $\times$ 3** (`sumMat.c`)

   Créese un programa que solicite al usuario los elementos de 2 matrices 3 $\times$ 3 de reales y saque por pantalla la matriz resultado de sumar ambas y sus respectivas

traspuestas. Para ello deben emplearse los algoritmos diseñados en los problemas del tema 1.

3. **Algoritmo Capicúas** (`capicuas1.c`).

   Créese un programa que solicite al usuario un número menor que 100.000 y que indique si el número es capicúa. El programa debe utilizar la función `esCapicúa` que aparece abajo en pseudocódigo. Nótese que esta función utliza la función `reverso` (cuyo código también se indica).

```
Entrada:  n, un número entero
Requisitos:  n >= 0
Salida:  el booleano b sera verdadero si n era capicua
funcion esCapicua (n:  entero) dev b:  booleano;
    b:= (n = reverso(n));
    dev b
ffuncion

Entrada:  el entero n = n_1 n_2 ... n_l,
Requisitos:  n >= 0
Salida:  m contiene los dígitos de n invertidos (m = n_l ... n_2 n_1)
funcion reverso (n:  entero) dev m:  entero;
    m:= 0;
    mientras n > 0 hacer
        m:= 10 * m + n mod 10;
        n:= n div 10
    fmientras;
    dev m
ffuncion;
```

   Pruebe a programar la función `esCapicua` mediante una macro. Utilice la opción `-E` de `gcc` o bien `cpp` para ver el resultado de su macro.