Implementación de Conjuntos en C

El lenguaje C

Presentaremos en este epígrafe una serie de reglas para la codificación de programas escritos en C que se utilizarán a lo largo de las prácticas. Estos convenios no pertenecen de por sí al lenguaje C, pero se usan para hacer más legibles los programas, sobre todo para programadores que no han realizado la implementación. Contrariamente a lo que se suele pensar, los programadores no pasan la mayor parte de su tiempo de trabajo escribiendo programas. Se gasta mucho más tiempo en mantener, actualizar y depurar programas ya existentes cuyo autor con frecuencia no es quien trabaja sobre ellos.

También introduciremos dos herramientas de apoyo al desarrollo de programas: make y lint. Make es una utilidad que permite automatizar en cierta medida el proceso de compilación de los diferentes módulos que componen un programa, mientras que lint se utiliza para detectar errores e inconsistencias en el código que el compilador de C pasa por alto.

La práctica finaliza con el desarrollo de una librería de funciones para operar con conjuntos y el planteamiento de un programa para ejercitar el uso de esta librería de funciones.

Convenios a la hora de codificar en C

Los convenios que aquí exponemos los utilizaremos a lo largo de esta obra para dar una apariencia uniforme a los códigos que desarrollaremos, acostumbrando también a los alumnos a la utilización de un estándar concreto de codificación de programas. Las indicaciones que presentaremos son sólo una guía, que no pretende ser exhaustiva, de cómo los alumnos deben codificar sus programas para que sean legibles, fáciles de depurar y de mantener. Para una revisión más profunda respecto a un buen estilo de programación en C, recomendamos las referencias [Can92] y [Oua97].

Un buen programador debería codificar aprovechando toda oportunidad que asegure que su código es claro y fácil de comprender. No debe utilizar "trucos inteligentes" a los cuales el lenguaje C se presta con facilidad. Considérese a modo de ejemplo los fragmentos de código de la Figura 1.1

Aunque la segunda versión del código es más larga, también es más clara y fácil de comprender. Incluso un programador sin demasiada experiencia en C puede entender que la segunda parte del código tiene algo que ver con el movimiento de datos desde un punto fuente hacia un destino. Al compilador no le importa cuál de las dos versiones se utilice: un buen compilador generará exactamente el mismo código máquina para ambas. Es el programador quien se beneficia de la claridad del código.

```
1º:
          while ('\n' != *p++=*q++);

2º:
          while (1) {
           *destination_ptr = *source_ptr;
           destination_ptr++;
           source_ptr++;
           if (*destination_ptr == '\n')
            break; /* abandonar el bucle si se ha acabado */
          }
```

Figura 1.1 Legibilidad del código

Reglas de tipo general

- Comentar, comentar, comentar. Documentar convenientemente los programas es imprescindible en cualquier lenguaje. Una tendencia generalizada de los programadores inexpertos consiste en comentar lo obvio. Veamos un ejemplo de este tipo de comentarios:

```
a = a + 3;  /* Se suma 3 al valor de a */
```

Este comentario es absolutamente inútil: cualquiera que conozca el lenguaje C entenderá sin necesitar el comentario lo que éste dice.
La regla que se ha de seguir es comentar el porqué se hace algo, no lo que se hace, que ya deberá resultar evidente a través de la lectura del código. Un comentario cada diez líneas de código fuente es un buen promedio, aunque obviamente esta proporción no tiene porqué mantenerse constante en todo el fichero fuente: habrá partes del código que precisen más explicaciones que otras.

- Utilizar siempre la regla del beso (KISS: Keep It Simple, Stupid). Claro y sencillo es siempre preferible a complejo y maravilloso.
- Evitar efectos laterales. Utilizar los operadores ++ y −− siempre en una única línea por sí mismos (no incluídos en otras sentencias complejas).
- Nunca colocar asignaciones dentro de los condicionales. Nunca colocar una asignación dentro de otra sentencia.
- Considere la diferencia entre = y ==. Utilizar = en lugar de == es un error muy común y difícil de encontrar, especialmente entre programadores procedentes de otros lenguajes.
  Consulte la referencia [Dye99] para obtener más información sobre este tipo de problemas.
- Nunca haga "nada" de forma "silenciosa":

```
/* Nunca programe de este modo */
for (index = 0; data[index] < key; index++);
/* ¿Ha notado el punto y coma al final de la línea anterior? */

/* Forma correcta para hacer lo mismo: */
for (index = 0; data[index] < key; index++)
       ;       /* No hacer nada */
```

Figura 1.2 Deje constancia expresa de lo que se hace

Reglas que atañen al diseño

- Cuando diseñe sus programas, tenga en mente la "ley del mínimo asombro", que establece que su programa se debe comportar de modo que asombre lo menos posible a quien lo lea.
- Haga la interface de usuario tan simple y consistente como sea posible.
- Provea al usuario final de tanta ayuda como le sea posible.
  En concreto, identifique claramente todos los mensajes de error con la palabra "error" y si es posible, trate de dar al usuario alguna idea de cómo corregir el problema.

Legibilidad

Haga todo lo posible para incrementar la legibilidad de sus programas. En concreto:

- Coloque siempre un espacio después de las comas y puntos y comas. Es preferible

```
for (i = 1; i <= MAX; i++)
```

que

```
for(i=1;i<MAX;i++)
```

En cuanto a la colocación de espacios, el código escrito debería seguir las mismas reglas que utilizamos en la escritura de textos que no son programas. Así por ejemplo, al escribir en castellano o inglés no colocamos un espacio después de un paréntesis abierto ni tampoco antes de uno cerrado. Tampoco lo haremos al escribir en C, siguiendo la ley del mínimo asombro.

- Coloque siempre espacios a ambos lados de un operador binario:
  Es preferible

```
a + b
```

que

```
a+b
```

Declaraciones

- Coloque una declaración de variable por cada línea, y comente el significado de cada variable significativa para comprender el código. Obviamente no será necesario comentar absolutamente todas las variables: sólo aquellas que son especialmente significativas para entender el código.
- Utilice identificadores lo suficientemente significativos como para comprender su significado y lo suficientemente cortos como para que sean fáciles de escribir.
- Nunca utilice declaraciones por defecto: si una función retorna un entero, declárela de tipo int. Todos los parámetros de una función deben ser declarados y comentados.
- Los identificadores de las constantes y macros se suelen escribir en mayúscula y el resto de identificadores en minúscula, y en esta obra trataremos de seguir este convenio.

Sentencia switch

- Coloque siempre una opción default en las sentencias switch (incluso en caso de que no haga nada.
- Todas las opciones de una sentencia switch deben finalizar con su correspondiente break.

Reglas de estilo

- Los programas de un cierto tamaño se suelen dividir en varios ficheros fuente. Cada uno de los ficheros realiza una determinada parte del trabajo, pudiéndose compilar por separado para posteriormente enlazarlos juntos y crear el ejecutable.
- Un único bloque de código delimitado por { } no debe extenderse más de tres pantallas. Si se da ese caso, el código debería estar dividido en funciones más pequeñas y simples.
- Cuando el código comienza a salirse de la pantalla por la parte derecha de la misma es el momento de dividirlo en módulos más pequeños y simples.

Disposición de las llaves

Las llaves izquierdas (abiertas) se suelen añadir a otras líneas de manera que no ocupen ellas solas una línea. Por ejemplo, se suelen colocar en la misma línea que un *if, for, while,* etc., cerrarla justo antes de *else* y abrirla justo después, etc.

Sangrado

El sangrado que se utiliza es el normal de cualquier lenguaje de programación, sangrando más a la derecha el contenido de los bloques (*if, for, while, {, ...*) con anidamientos más profundos.

La función main()

Normalmente la función main no realiza funciones importantes, es decir, en ella no se codifica la solución del problema. Sólo se encarga de invocar a la funciones que implementan la solución del problema.

Normas de carácter general aplicables a todas las prácticas

La documentación de cabecera de cada subprograma incluirá al menos lo siguiente:

- Finalidad: Descripción suficiente de la función que lleva a cabo el módulo.
- Significado de las variables más importantes utilizadas en el módulo (comentando cada una de ellas).
- Comentarios prólogo indicando las ideas generales de cómo trabaja la rutina.
- Comentarios relevantes.

Además de esta información de cada rutina, todos los ficheros de programas incluirán una documentación de cabecera suficientemente explicativa de las diferentes funcionalidades así como de las rutinas más relevantes que lo componen. En cada fichero constará siempre el autor del código y la fecha de la implementación, así como cualquier otra información que el programador considere significativa.

Un aspecto al que los alumnos no suelen prestar la debida atención e importancia es el diseño del conjunto de pruebas para los programas que se desarrollan. Con frecuencia es imposible determinar de forma totalmente exhaustiva si un programa funcionará correctamente para unos datos de entrada. Lo que sí es posible realizar es un conjunto suficientemente grande de pruebas sobre el programa que se ha diseñado, que nos permitan asegurar su corrección con ciertas garantías. Junto con cada práctica, el alumno deberá diseñar un conjunto de datos de prueba para comprobar que la misma funciona del modo correcto y que se comporta del modo especificado para un cierto número de entradas, elegidas de forma adecuada.

Proyectos de programación: la herramienta make

En entornos profesionales de desarrollo de software es frecuente dividir el código fuente de un programa en módulos pequeños que se compilan de forma separada. Esto permite al programador concentrarse en el desarrollo de un módulo específico sin interaccionar con el resto, o también que diferentes equipos de desarrollo trabajen de forma cooperativa minimizando sus interacciones. Para generar el código ejecutable hay que compilar por separado cada uno de los módulos y enlazarlos todos juntos, y con frecuencia no es fácil recordar (especialmente si se trabaja en una aplicación de envergadura) cuales son los módulos que han de ser compilados en cada momento porque se haya cambiado algo en ellos. Una solución consiste en compilar todos los módulos cada vez que se hace un cambio en alguno de ellos, pero esta posibilidad conlleva una pérdida de tiempo tanto mayor cuanto más grande sea el proyecto en el que se trabaja.

```
 1 testset: sets.o testset.o basic.o
 2   gcc −o testset sets.o testset.o basic.o −lm
 3
 4 testset.o: testset.c basic.h sets.h
 5   gcc −ansi −c testset.c
 6
 7 sets.o: sets.c sets.h
 8   gcc −ansi −c sets.c
 9
10 basic.o: basic.c basic.h
11   gcc −ansi −c basic.c
12
13 clean:
14   rm −f testset *.o
```

Figura 2.1 Un ejemplo de fichero makefile

La herramienta *make* permite automatizar algunas de las actividades que se realizan en el desarrollo y mantenimiento de un programa. Se almacena en un fichero de texto (que se suele denominar *makefile*) la información de dependencias y comandos necesarias para reconstruir toda la aplicación y cuando se quiere generar el programa ejecutable la herramienta se encarga de recompilar solamente los ficheros que han sufrido cambios desde la última compilación, para lo cual utiliza el reloj del sistema.

Consideremos un ejemplo en el que nuestro programa ejecutable se llamará *testset* y resulta de la compilación (y enlazado) de los módulos *testset.c*, *sets.c* y *basic.c*. Supondremos también que el programa utiliza alguna función definida en el fichero *math.h*, es decir, alguna función cuyo código se encuentra en las librerías matemáticas que por tanto habrá que enlazar con el programa. La Figura 2.1 muestra un ejemplo de fichero *makefile* para la compilación del programa, en el que los números de línea se han incluido sólo para facilitar su explicación (no figurarían en un fichero *makefile* real).

En el ejemplo se supone que los ficheros fuentes de código C *testset.c*, *sets.c* y *basic.c* incluyen cada uno de ellos los ficheros de definiciones *testset.h*, *sets.h*, y *basic.h* respectivamente. En esos ficheros de cabecera se encontrarían las definiciones pertinentes para cada uno de los módulos de código C. El fichero que aparece en la Figura 2.1 se denomina *makefile* o *Makefile*, aunque se le puede dar otro nombre. Si este fichero se almacena en el directorio de trabajo, la llamada a la utilidad make:

```
make
```

producirá las operaciones necesarias para recompilar *testset* después de que se realice cualquier cambio sobre los ficheros **.c* o **.h*. Si el fichero *makefile* se almacena con un nombre diferente de éste, la llamada deberá realizarse como

```
make −f <nombre_del_fichero>
```

La herramienta *make* trabaja con tres elementos: un fichero *makefile* de descripción suministrado por el usuario (como el de la Figura 2.1), los nombres de fichero que forman parte de la aplicación junto con la hora de sus últimas modificaciones (que son almacenadas por el sistema operativo) y ciertas reglas incorporadas en la herramienta *make* para realizar las operaciones necesarias.

Por ejemplo, el fichero *makefile* del ejemplo especifica en su línea 1 que el programa ejecutable *testset* depende de tres ficheros con código objeto (extensión .o) que son *sets.o*, *testset.o* y *basic.o*. Una vez que estos ficheros estén disponibles, la línea 2 del fichero especifica que el programa *testset* se crea enlazando juntos estos ficheros de código objeto junto con la librería matemática (opción −lm en la línea 2 del fichero).

De forma análoga, la línea 4 del fichero indica que el fichero *testset.o* depende de *testset.c*, *basic.h* y *sets.h* y por tanto será generado si se introduce algún cambio (aunque sea la hora de la última edición) en cualquiera de estos tres ficheros. La línea 5 indica que el código objeto *testset.o* se genera mediante la compilación del programa *testset.c*. De forma similar, las líneas 7 y 10 especifican las dependencias de los ficheros de código objeto *sets.o* y *basic.o* y las líneas 8 y 11 indican cómo producirlos a partir del código fuente (*sets.c* y *basic.c*).

Con frecuencia resulta útil incluir reglas con nombres fáciles de recordar que realicen ciertas tareas. Es el caso por ejemplo de las líneas 13 y 14 de nuestro ejemplo: si se ejecuta el comando

```
make clean
```

ello provocará que se borren del directorio actual los ficheros objeto (**.o*) y el programa ejecutable (*testset*) (ver la línea 15 del fichero *makefile*).

La herramienta *make* está dotada de un sencillo mecanismo de macros que permite sustituir elementos en las líneas de dependencia y de comandos. Una macro se define indicando su nombre seguida de un signo igual y el valor que toma, y la macro se invoca precediendo su nombre con el signo "$", con su nombre encerrado entre paréntesis. Un ejemplo de fichero *makefile* análogo al anterior pero utilizando macros es el que se presenta en la Figura 2.2.

```
OBJECTS=sets.o testset.o basic.o
LIBS=−lm
CC=gcc
COPTIONS=−ansi

testset: $(OBJECTS)
  gcc −o testset $(OBJECTS) $(LIBS)

testset.o: testset.c basic.h sets.h
  $(CC) $(COPTIONS) −c testset.c

sets.o: sets.c sets.h
  $(CC) $(COPTIONS) −c sets.c

basic.o: basic.c basic.h
  $(CC) $(COPTIONS) −c basic.c

clean:
  rm −f testset *.o
```

Figura 2.2 Un ejemplo de fichero makefile utilizando macros

En ese ejemplo se definen macros para los nombres de los ficheros con código objeto, la librería con la que hay que enlazar, el compilador que se ha de invocar y las opciones con que se invoca el compilador. Si en lugar de utilizar el compilador de gnu (gcc) se quisiera utilizar el estándar del sistema, bastaría con cambiar la línea 3 en la que se define el compilador que se usa en la compilación de los diferentes módulos.

Antes de comenzar a utilizar la herramienta make, recomendamos estudiar detenidamente la información del manual de unix acerca de ella (man make). Por otra parte, en el servidor ftp del CSI se encuentra disponible el fichero (postscript comprimido con gzip)

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/doc/make.ps.gz

que describe en detalle la herramienta *make* de Gnu.

Comprobación del código: lint

Una de las críticas más frecuentes que se hace al lenguaje C es que su comprobación de tipos no es tan exhaustiva como en otros lenguajes (como por ejemplo Pascal). Como consecuencia de ello, la mayoría de los compiladores de C compilan correctamente sentencias que no representan exactamente lo que el programador pretende realizar. Ante este tipo de sentencias el compilador "hace su interpretación" y las compila de determinado modo. En estos casos, a lo sumo obtendríamos un aviso (warning). Por otra parte, el número de situaciones en las que el compilador "actuaría por su cuenta" sin generar avisos, es muy grande.

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

int main(void) {
  char *c;
  float r = 3.5;
  int x, y = 7;

  x = *c + r;
  if (0)
    printf("Esto nunca se escribirá\n");
  else
    return 5;
}
```

Figura 2.3 Un código C sintácticamente correcto

Lint [Eva98] viene a aliviar este tipo de situaciones. Se trata de una herramienta disponible en Unix (y actualmente en algunos otros sistemas) que detecta situaciones presentes en el código fuente escrito en C que pueden ser potencialmente errores, sentencias sin utilidad o poco portables, avisando al programador de estas circunstancias. Podríamos decir que *lint* es un comprobador de tipos "más fuerte" que el que está incorporado en el propio compilador de C. Entre las situaciones que detecta se incluyen sentencias inalcanzables (que nunca se ejecutarán), bucles en los que no se entra, variables que han sido declaradas pero que no se utilizan en el código o expresiones lógicas con un valor constante. En todos estos casos la herramienta genera mensajes de aviso al usuario indicando la situación del posible error.

Consideremos el código que se presenta en la Figura 2.3. Si ese código se escribe en un fichero *mi_prog.c* y se compila podremos comprobar que el compilador lo compila correctamente, sin producir ningún aviso y generando el correspondiente código ejecutable. No obstante, si se ejecuta

```
lint mi_prog.c
```

se obtiene el resultado que se muestra en la Figura 2.4. El primer aviso indica que la variable *c* está siendo utilizada antes de ser inicializada, lo cual es un error (su valor estará indefinido). El segundo aviso indica que en la sentencia condicional se está usando una expresión de valor constante, lo cual es muy probable que también represente un error. La herramienta indica también que las variables *x* e *y* de la función *main()* han sido inicializadas, pero que no se utilizan en el código (no se hace nada con ellas). También notifica que es posible que la función finalice sin retornar valor alguno y por último, que en el código "no se hace nada" con el valor que devuelve la llamada a la función *printf()*. Todas estos puntos representan posibles situaciones de error, y el usuario debería reescribir su código de forma que *lint* no generara mensajes.

```
 (9) warning: variable may be used before set: c
(10) warning: constant in conditional context

set but not used in function
    (7) y in main
    (7) x in main

function falls off bottom without returning value
    (14) main

function returns value which is always ignored
    printf
```

Figura 2.4 Avisos generados por *lint* para el código de la Figura 2.3

En muchos casos el conseguir que *lint* no produzca mensajes de aviso conllevaría transformar el código de forma significativa. El programador deberá al menos entender bien los avisos que la herramienta proporciona y proceder en consecuencia: si realmente se trata de errores, corregirlos, y si se trata de situaciones que están controladas, se puede ignorar el error. En este sentido *lint* no es más que una herramienta de ayuda en el desarrollo de programas en C.

En las páginas del manual de *lint* se pueden consultar las opciones de la llamada, que posibilitan que la herramienta no genere mensajes ante algunas situaciones concretas, lo cual puede resultar interesante en ocasiones. Por ejemplo, en el caso anterior, la llamada a la función *printf()* devuelve un valor con el que "no se hace nada". El programador puede optar por reescribir esa llamada como:

```
(void)printf("Esto nunca se escribirá\n");
```

o bien utilizar la opción correspondiente de *lint* para que este tipo de situaciones sean ignoradas.

En el servidor ftp del CSI se encuentra disponible el fichero (postscript comprimido con gzip)

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/doc/lclint.ps.gz

Se trata del manual de usuario de la aplicación *lclint*, un lint desarrollado dentro del proyecto Gnu.

Una librería de funciones para operar con conjuntos

Muchas de las operaciones que realizaremos en las prácticas que se van a desarrollar tales como la representación de los estados de un autómata o los símbolos de un lenguaje se pueden implementar con mayor comodidad a través de la utilización de conjuntos. El lenguaje de programación C, al contrario que Pascal no posee los conjuntos como tipo de datos básico pero afortunadamente no es difícil implementar este tipo de datos en C a través por ejemplo de mapas de bits (bit maps) es decir, un vector de bits.

En este epígrafe introduciremos una implementación de una librería de funciones para operar con conjuntos, y en epígrafes subsiguientes plantearemos unas prácticas para que el alumno se familiarice con la utilización de estas funciones.

Los prototipos de las funciones que se diseñarán se incluirán en un fichero que llamaremos `sets.h` y para utilizar las funciones de la librería habrá que incluir dicho fichero en el código:

```
#include "sets.h"
```

El fichero `sets.c` contendrá el código fuente de las funciones de la librería. Veamos a continuación las funciones que se encuentran inicialmente en la librería de conjuntos:

```
set *set_ini(unsigned num_ele);
```

Esta función crea un conjunto y retorna un puntero al conjunto recién creado. El parámetro de la función indica el número máximo de elementos que podrá almacenar el conjunto que se crea. Los conjuntos se implementan utilizando memoria dinámica y si el sistema no dispone de memoria suficiente para crear el conjunto solicitado, la función devuelve un puntero NULL. Antes de operar con un conjunto será preciso haberlo declarado e inicializado con el tamaño adecuado mediante una llamada a `set_ini()`.

```
void set_vac(set *cj);
```

Esta función inicializa a vacío el conjunto que se le pasa como parámetro. Hay situaciones donde es conveniente inicializar a vacío el conjunto para su uso posterior (por ejemplo, en el caso en que el usuario desea reutilizar un conjunto ya inicializado para una finalidad diferente).

```
void set_lib(set *cj);
```

Como se ha señalado, los conjuntos se implementan utilizando memoria dinámica. Esta función se encarga de eliminar un conjunto `cj` creado previamente mediante una llamada a `set_ini()` y libera la memoria asociada al conjunto. Es conveniente invocar a esta función al finalizar las operaciones que se realizan sobre un conjunto, aunque si la memoria no se libera, a la finalización del programa siempre es liberada (como ocurre siempre al usar memoria dinámica).

```
void set_ins(set *cj, unsigned e);
```

Esta función, supuesto que el conjunto `cj` ha sido previamente creado con el tamaño adecuado, añade el elemento e al conjunto `cj`. Los elementos de los conjuntos han de ser siempre números enteros sin signo, aunque esto no supone una restricción, como veremos más adelante.

```
int set_per(set *cj1, unsigned e);
```

Esta función devuelve cero si el elemento e no pertenece al conjunto `cj1` y un valor distinto de cero en caso que el elemento pertenezca al conjunto.

```
int set_igu(set *cj1, set *cj2);
```

La función `set_igu()` devuelve un valor distinto de cero si los dos conjuntos que se le pasan como parámetros son iguales y devuelve un cero en otro caso.

```
void set_uni(set *cjdest, set *cjorg);
```

Esta función realiza la unión de conjuntos. El conjunto destino (`cjdest`) se ve modificado añadiéndole los elementos del conjunto `cjorg`. Se supone que ambos conjuntos tienen la misma capacidad de almacenamiento; es decir, fueron creados mediante llamadas a `set_ini()` con el mismo parámetro.

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "sets.h"
5
6  int main (int argc, char *argv[]) {
7    #define NUM_MESES 12
8    typedef enum {ENE=1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT,
9                  NOV, DIC} meses;
10   set *verano, *invierno;
11
12   verano = set_ini(NUM_MESES);
13   set_vac(verano);
14   invierno = set_ini(NUM_MESES);
15   set_vac(invierno);
16   set_ins(verano, JUN);
17   set_ins(verano, JUL);
18   set_ins(verano, AGO);
19   set_ins(invierno, NOV);
20   set_ins(invierno, DIC);
21   set_ins(invierno, ENE);
22   if (!set_per(verano, MAR))
23      printf("Marzo NO es un mes de verano\n");
24   else
25      printf("Marzo SI es un mes de verano\n");
26   printf("\nMeses de verano: \n");
27   set_print(verano);
28   set_lib(verano);
29   set_lib(invierno);
30   return 0;
}
```

Figura 2.5 Ejemplo de utilización de la librería de funciones de conjuntos

```
void set_cpy(set *cjorg, set *cjdest);
```

La función `set_cpy()` copia el conjunto `cjorg` en el conjunto `cjdest`, de modo que al final de su ejecución, ambos conjunto son iguales.

```
void set_print(set *s);
```

La función `set_print()` imprime en pantalla los elementos que forman parte del conjunto que se le pasa como parámetro.

El código que aparece en la Figura 2.5 es un ejemplo del modo de utilización de algunas de las funciones que hemos descrito. En la línea 4 del código se incluye el fichero `sets.h`, que contiene definiciones y los prototipos de las funciones. En la línea 10 se declaran dos conjuntos con identificadores `invierno` y `verano` que son creados en las líneas 12 y 14 mediante sendas llamadas a la función `set_ini()`. Las llamadas a `set_vac()` de las líneas 13 y 15 sirven para inicializar a vacío los conjuntos recién creados. El código entre las líneas 16−21 sirve para insertar en cada uno de los conjuntos los correspondientes meses de verano e invierno. En la línea 22 se muestra un ejemplo de utilización de la función `set_per()` para comprobar si el mes de marzo es uno de los meses de verano. La llamada `set_print(verano)` de la línea 27 imprimirá en pantalla los elementos del conjunto `verano`. Por último, las llamadas a `set_lib()` de las líneas 28 y 29 liberan la memoria que consumen los conjuntos.

La Figura 2.6 muestra el resultado de la ejecución del programa de la Figura 2.5.

```
Marzo NO es un mes de verano
Meses de verano:
{ 6  7  8 }
```

Figura 2.6 Resultado de la ejecución del programa de la Figura 2.5

Implementación de los conjuntos

Un conjunto se representa mediante una secuencia de bits (lo que se suele denominar un bitmap. Los bits que constituyen el conjunto son los correspondientes a una serie de palabras, entendiendo por palabra un dato de tipo `unsigned long`. Si un bit está a 1 indica que el elemento correspondiente pertenece al conjunto, mientras que si está a cero ello indica que el elemento no pertenece al conjunto. Los conjuntos en principio representan números, de modo que el bit N−ésimo de todos los utilizados corresponde al número N (indica su pertenencia o no al conjunto según la regla especificada). En la Figura 2.7 se presenta la estructura de datos que se utiliza para almacenar un conjunto. Se trata de un registro (struct) con un campo *longitud* que indica el número de palabras que ocupa el conjunto y un campo *palabra* de tipo puntero que apunta a las palabras cuyos bits representan los elementos del conjunto.

En una máquina en la que una palabra (entero largo sin signo) ocupe cuatro bytes (por ejemplo en las estaciones de trabajo Sun), el bitmap asociado con el conjunto `verano` de la Figura 2.5 tendría la representación que se muestra en la Figura 2.8. Este bitmap sería apuntado por el puntero *palabra* de la estructura asociada al conjunto. En este ejemplo, a nivel de representación interna, la estructura que representa el conjunto tendrá el campo longitud con un valor de 1 (el conjunto utiliza una sola palabra −32 bits en este caso− para su representación) y el campo palabra apuntaría a una entero largo sin signo como el que se muestra en la Figura 2.8. Dado que en el código de la Figura 2.5 a los meses de verano se les asigna los números 6, 7 y 8 (ver el resultado de la ejecución del programa en la Figura 2.6) todos los bits de esta palabra estarán a cero, salvo los bits números 6, 7 y 8 (6 y 7 del primer byte y 0 del segundo byte de la palabra).

```
typedef unsigned long palabra;
typedef struct {              /* Estructura de un conjunto:  */
   unsigned longitud;         /* Número de palabras que ocupa */
   palabra  *conj;            /* Puntero a las palabras del conjunto */
} set;
```

Figura 2.7 La representación interna de un conjunto

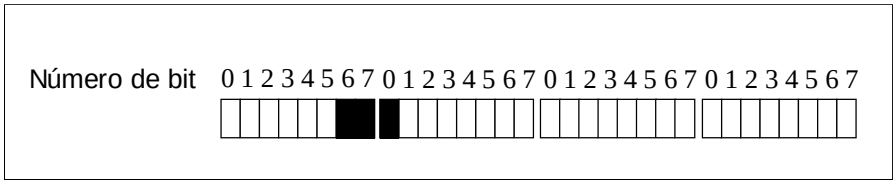

Figura 2.8 La representación del conjunto "verano" de la Figura 2.5

En definitiva, en la estructura que representa a un conjunto (Figura 2.7) el campo *longitud* indica el número de palabras (una palabra es un unsigned long) necesarias para representar el conjunto. Con esta representación, un conjunto de N palabras podrá contener un máximo de *N x 8 x sizeof(palabra)* elementos, teniendo en cuenta que *sizeof(palabra)* es el número de bytes de una palabra y 8 es el número de bits de un byte.

En el fichero `sets.h` se encuentra la definición de las macros `PAL(x)` y `BIT(x)`. Dado un elemento `x` de un conjunto representado del modo que hemos expuesto, la macro `PAL(x)` devuelve la palabra en la que se encuentra el bit correspondiente al elemento `x`, mientras que `BIT(x)` devuelve el número del bit correspondiente al elemento. Estas dos macros se utilizan profusamente en la implementación de las funciones de `sets.c`

Todas las funciones de manipulación de conjuntos que hemos presentado, asumen esta representación interna que se hace de los mismos. Así, por ejemplo, la llamada
`verano=set_ini(NUM_MESES);`
de la línea 12 del código de la Figura 2.5 aloja memoria dinámica para una estructura del tipo de la que se muestra en la Figura 2.7, calcula el número de palabras necesarios para almacenar un máximo de `NUM_MESES` elementos (en este caso 1 palabra, como hemos visto), coloca ese valor en el campo *longitud* de la estructura, y aloja memoria también dinámica para las palabras que precise el conjunto en su representación, haciendo que el puntero *palabra* apunte a dicha memoria.

El tener en cuenta la representación subyacente a los conjuntos permite en algunos casos una implementación eficiente de las funciones. Por ejemplo, en un conjunto vacío, todos los bits de su representación han de estar a cero. La función `set_vac()` que inicializa a vacío un conjunto se limita a recorrer todas las palabras del conjunto (cuyo número se obtiene a partir del campo *longitud*) y asignarles el valor cero a cada una de ellas, con lo cual todos los bits de cada una de las palabras (y por tanto del conjunto) quedarán a cero.

Para trabajar con las librerías de funciones de manipulación de conjuntos hay varias alternativas:

1. Incluir los ficheros allí donde sean necesarios
2. Usar la utilidad make
3. Convertir el fichero fuente (*.c) en una verdadera librería compilada

Comenzaremos en este capítulo por utilizar la segunda alternativa, y en capítulos posteriores introduciremos el modo de usar la tercera forma de trabajo. En el servidor ftp del csi se puede obtener el fichero

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p02_sets/makefile

Se trata de un fichero `makefile` de ejemplo que utiliza los ficheros `sets.c` y `sets.h` estableciendo las dependencias en su compilación. Ese fichero supone que el nombre del fichero ejecutable que se pretende crear es *testset* y supone también que el programa ejecutable *testset* se obtiene de compilar por separado (y luego enlazar) los ficheros *sets.c*, y *testset.c*. El fichero *testset.c* que también está disponible en el mismo directorio del servidor ftp contiene un programa principal que realiza llamadas a las funciones de la librería de conjuntos, para comprobar en cierta medida el correcto funcionamiento de la implementación de la práctica.

También en el servidor ftp del CSI se encuentran los ficheros

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p02_sets/sets.c

y

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p02_sets/sets.h

que han sido descritos en este epígrafe.

Práctica 3.1: Ampliación de la librería de funciones de conjuntos

La primera práctica que planteamos consiste en ampliar el fichero `sets.c` con una serie de funciones de manipulación de conjuntos que inicialmente no están diseñadas. Veamos a continuación las modificaciones que se pide realizar:

2.1.1.− Utilizar las funciones `set_ini()` y `set_print()` para comprobar experimentalmente que un conjunto recién creado (con `set_ini()`) no tiene porqué estar vacío. Dicho de otro modo, en la implementación inicial que se ha realizado, a la hora de crear un conjunto habrá que invocar sucesivamente a las funciones `set_ini()`y `set_vac()`.
Modifique el código de `set_ini()` para garantizar que un conjunto recién creado esté siempre vacío.

2.1.2.− Diseñar una función

```
set *set_int(set *cj1, set *cj2);
```

que calcule la intersección de los dos conjuntos que se le pasan como parámetro. La función alojará memoria para el conjunto intersección de cj1 y cj2 y devolverá un puntero al conjunto intersección.

2.1.3.− Diseñar una función

```
int set_cardinal(set *cj);
```

que devuelva el cardinal (número de elementos) del conjunto que se le pasa como parámetro.

2.1.4.− Implementar una función

```
set *set_complement(set *cj);
```

que calcule el complementario de un determinado conjunto. Se supone que el número máximo de elementos del conjunto original y de su complementario será el mismo.

2.1.5.− Implementar una función

```
int set_subconj(set *cj, set *sub);
```

que devuelva 1 si el conjunto `sub` es un subconjunto de `cj` o cero en caso contrario. La función asumirá que un conjunto vacío es subconjunto de cualquier otro, y la función también devuelve 1 si ambos conjuntos son vacíos o si son el mismo conjunto.

Para la realización de esta práctica se recomienda consultar la referencia [Hol90].

Práctica 3.2 El problema del cubrimiento

Utilizando libremente las funciones para operar con conjuntos, el alumno deberá implementar un algoritmo que resuelva el problema del cubrimiento de un conjunto.

El problema del cubrimiento puede describirse como sigue: dada una familia de conjuntos $F=\{S_1, S_2, ..., S_N\}$ y un conjunto C, hallar el mínimo número de conjuntos $S_i \in F$ tales que $C \subseteq \cup_i S_i$. Es decir, hallar el cubrimiento mínimo de C: el mínimo número de conjuntos de F en cuya unión está contenido el conjunto C.

El programa leerá en la línea de comandos el nombre de un fichero de texto que contendrá la descripción de la familia de conjuntos F y del conjunto C. La Figura 2.9 presenta un ejemplo del contenido de estos ficheros.

El primer número (10) indica el número máximo de elementos que habrá en los conjuntos que se van a considerar. El segundo número (5) indica el número de conjuntos de la familia F. A continuación vienen los números correspondientes a tantos conjuntos como se indicaba en la línea 2, que en nuestro caso son $S_1=\{2, 4, 6, 8, 0\}$, $S_2=\{1, 3, 5\}$, $S_3=\{7\}$, $S_4=\{1, 2, 3\}$, $S_5=\{7, 9\}$. Por último aparece el conjunto a cubrir, C, que en nuestro caso es $C=\{1, 3, 5, 7, 9\}$.

```
10
5
2, 4, 6, 8, 0
1, 3, 5
7
1, 2, 3
7, 9
1, 3, 5, 7, 9
```

Figura 2.9 Un ejemplo de fichero para el problema del cubrimiento

En este ejemplo, una solución del problema es S_2, S_5 puesto que $C=S_2 \cup S_5$.

La solución del problema se puede abordar como se describe a continuación:

- Almacenar la familia de conjuntos *F* en un vector (dinámicamente alojado) de conjuntos (el vector tendrá un conjunto S_i de *F* en cada componente.
- Almacenar en alguna estructura de datos la mejor solución alcanzada en cada momento (número de conjuntos utilizados en la unión y cuáles son los conjuntos utilizados).
- Calcular todas las uniones posibles de conjuntos de *F* mediante un bucle: uniones en las que sólo intervenga un conjunto, uniones en las que intervengan 2 conjuntos, uniones con 3 conjuntos, y así sucesivamente. En este proceso se estudiarán las soluciones que son válidas (aquellas cuya unión contiene a *C* y se actualizará la mejor solución hallada en cada momento.

En el servidor ftp del CSI se encuentran los programas

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p02_covering/dos/covering.exe

ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/p02_covering/linux/covering

(versiones DOS y linux) que tienen el comportamiento deseado. En el mismo directorio se encuentra el fichero *ejemplo.txt* que se muestra en la Figura 2.9.

Como título de referencia en esta práctica, se puede utilizar [Sed90].

Bibliografía

[Can92] Cannon, L. W. Et al *Recommended C Style and Coding Standards*
ftp://ftp.csi.ull.es/pub/asignas/AUTOMALF/doc/cstyle.ps.gz

[Dye99] Dyer, D. The Top 10 Ways to get screwed by the "C" programming language.
http://www.andromeda.com/people/ddyer/topten.html

[Eva98] Evans, D. Software Devices and Systems. *LCLint Home Page*.
http://www.sds.lcs.mit.edu/lclint/

[Hol90] Holub, A. *Compiler Design in C*. Prentice−Hall International, Inc., 1990. ISBN: 0−13−155151−5

[Oua97] Oualline, S. *Practical C Programming*. O'Reilly & Associates, Inc., 1997 ISBN: 1−56592−306−5

[Sed90] Sedgewick, R. *Algorithms in C*. Addison Wesley, 1988. ISBN: 0−201−51425−7