Sesión 6: Codificación de datos

XDR, Unicode

José Luis Díaz

Curso 2011-2012

En esta sesión se practicarán los conceptos teóricos sobre codificación de la información. En primer lugar, la codificación de datos binarios (enteros, reales, arrays, etc.) usando XDR como *middleware*, y en segundo lugar la codificación de caracteres, usando los estándares más comunes tales como ASCII, ISO-8859-15, Windows 1252 y Unicode, usando simplemente un editor de texto.

1. XDR

1.1. Ejemplo preliminar

Se suministra el fichero `tipos.x`, que contiene un conjunto de declaraciones de tipos en lenguaje XDR. Su contenido es el siguiente:

```
/* Damos nuevos nombres a algunos tipos básicos */
typedef int Entero;
typedef float Real;
typedef string Texto<10>;
typedef int ArrayEnteros[4];


/* Esta unión permite elegir el tipo en tiempo de ejecución
   entre cualquiera de los antes definidos */
union MiUnion switch(int tipo) {
 case 1: Entero        i;
 case 2: Real          x;
 case 3: Texto         t;
 case 4: ArrayEnteros  a;
 default: void;
};
```

Como vemos, se declaran algunos tipos básicos dándoles nuevos nombres (`Entero`, `Real`, `Texto` y `ArrayEnteros`, y seguidamente se declara una unión discriminada de los tipos anteriores.

1.1.1. Uso de `rpcgen`

Si "compilamos" el fichero anterior con la herramienta `rpcgen`, obtendremos dos ficheros llamados `tipos.h` y `tipos_xdr.c`. El primero contiene el equivalente en C de los tipos definidos en XDR y el segundo contiene una función de conversión (filtro) para cada uno de esos tipos.

Ejecuta el comando siguiente:

```
$ rpcgen tipos.x
```

y examina el fichero `tipos.h` resultante. Debes comprender claramente qué tipos C equivalen a cada tipo XDR, y en particular cómo la unión discriminada del XDR se ha convertido en una estructura con dos campos, uno de tipo entero llamado `tipo` y otro de tipo unión llamado `MiUnion_u`.

Abre el fichero `tipos_xdr.c` y examina brevemente su contenido. Localiza las funciones filtro para cada tipo. No es necesario que comprendas el código generado por `rpcgen` aunque comprobarás que en muchos casos no es tan difícil.

Para poder usar estos filtros en nuestra aplicación (que desarrollaremos seguidamente), debemos antes *compilar* este código fuente. Sin embargo, la compilación no podrá completarse todavía, porque el código está incompleto: falta la función `main()`. Para compilar un código incompleto debemos darle a `gcc` la opción `-c` (significa "*compile only*", y con esta opción `gcc` no intentará crear un ejecutable, sino sólo un *código objeto*).

```
$ gcc -Wall -c tipos_xdr.c
```

Por lo general, `gcc` generará gran cantidad de avisos (*warnings*), debido a que el código que `rpcgen` ha generado no es muy "limpio". Ya que no tiene sentido que intentemos retocar este código, simplemente ignoraremos los avisos. El resultado será un fichero llamado `tipos_xdr.o` que contiene la traducción a código máquina de `tipos_xdr.c`. Esto es lo que se denomina un *código objeto* y será añadido posteriormente a nuestro programa.

1.1.2. Escribiendo un productor de datos elemental

Seguidamente escribiremos un programa que declare una variable de cada uno de los tipos que hemos definido en `tipos.x`, que asigne un valor a dichas variables, y que use los filtros adecuados para volcar estas variables, codificadas en XDR, a un fichero llamado `datos.xdr`.

El código necesario se suministra parcialmente escrito en el fichero `productor-incompleto.c`, cuyo listado se muestra en el anexo 3.1. Se proporciona la declaración de una variable de cada tipo, y la inicialización de éstas variables con valores de ejemplo. La variable de tipo unión, en cambio, no se inicializa directamente en el código, sino que se le pide al usuario que, en tiempo de ejecución, elija el tipo y el valor que se asignará a esta variable[1]

Seguidamente se crea el fichero de datos, se asocia con la operación `XDR_DECODE` y se usa el filtro `xdr_int()` para volcar el primer dato. Se pide que completes el código para volcar los restantes datos (`x, t, a, u`).

Una vez hayas completado el código, para obtener el ejecutable debes, en primer lugar, compilar el código en cuestión con la opción `-c`, pues este código está también incompleto al faltarle las funciones filtro, y seguidamente juntar los *código objeto* que has generado en un único *código ejecutable*. La secuencia de comandos necesaria es:

```
$ gcc -Wall -c productor.c
$ gcc -o productor productor.o tipos_xdr.o
```

Prueba a ejecutar este programa y comprueba que genera un fichero de datos. Observa que el tipo de datos `Texto` se ha declarado con un tamaño máximo de 10 caracteres. ¿Qué pasará si el usuario teclea más? Compruébalo.

1.1.3. Examinando los datos generados

Calcula mentalmente (o en un papel) cuántos bytes ocupará la codificación de cada una de las variables que el productor ha volcado en el fichero. Súmalas todas y comprueba que la longitud del fichero de datos coincide con lo que te ha salido (la longitud de los ficheros es visible si usas el comando `ls -l`).

Con ayuda de la utilidad `hexdump` (recuerda poner la opción `-C`) examina el fichero de datos que el productor ha generado. Debes ser capaz de identificar los bytes que corresponden a cada variable del fichero. Encuentra el valor del discriminante y deduce el tipo de la unión allí almacenada.

1.2. Programación de un consumidor

Escribe un programa que, llamando a los filtros XDR en el mismo orden en que fueron usados en el productor, lea los datos del fichero `Datos.xdr` y muestre en pantalla los valores leidos.

Para ello deberás declarar una variable de cada uno de los tipos, abrir el fichero para lectura, asociarlo con la operación `XDR_DECODE` y llamar a cada función filtro. A la hora de mostrar por pantalla el contenido de cada variable, deberás usar `printf()` con la cadena de formato adecuada a cada tipo. En el caso de la unión, no sabemos de antemano su tipo, por lo que, una vez leída del fichero con el filtro apropiado, deberemos consultar el campo discriminante y en función de su valor usar un `printf()` adecuado al caso.

Gran parte del código puede ser reciclado del productor.

Recordar que, cuando el tipo de datos (en su versión C) que estemos leyendo contiene punteros, es necesario inicializar estos punteros con NULL antes de invocar el filtro. Esto ocurre por ejemplo en el caso del *string*. También ocurre en el caso de la unión, ya que la unión contiene un *string* y no sabemos de antemano si ese será el caso que encontraremos en el fichero.

[1] La lectura del caso `string` es un tanto compleja debido a la forma en que funciona `scanf()`, que no lee los retornos de carro que hay tras los datos cuando éstos son de tipo entero o flotante. Es necesario por tanto eliminar ese retorno de carro superfluo antes de llamar a `fgets()`, pues de lo contrario ésta retornaría inmediatamente una línea vacía.

1.3. XDR a través de sockets

Siguiendo el ejemplo visto en teoría, modificar el productor y el consumidor para que, en lugar de escribir y leer respectivamente en un fichero, lo hagan en un socket.

El productor, tras inicializar todas las variables, creará un socket de escucha, le asignará un número de puerto (por ejemplo, las cinco últimas cifras de tu DNI), y quedará a la espera de que un cliente se conecte (mediante `accept()`). Cuando esto ocurra, inicializará convenientemente la estructura XDR para que usen el socket de datos como destino de la codificación, y llamará a los filtros para codificar los datos. De este modo los datos serán enviados al cliente, codificados en XDR. Tras esta transferencia, cerrará el socket de datos y terminará.

El consumidor actuará como cliente, creando un socket (al que no necesita asignar dirección ni puerto) y conectando con el socket del productor. Seguidamente inicializará la estructura XDR para la operación `XDR_DECODE` sobre el socket y llamará a los filtros para recibir los datos decodificados, tras lo cual mostrará en pantalla la información recibida, cerrará el socket y terminará.

2. Codificación de caracteres

Existen varios estándares para codificar caracteres, y precisamente el problema es que no hay un único estándar. Para evitar sorpresas, cada máquina debería incluir junto con el texto qué estándar ha seguido para codificarlo. Por desgracia no siempre es así. Comenzaremos por escribir algunos archivos de texto "plano" en diferentes plataformas, para comprobar cómo cada una los almacena con una codificación diferente.

2.1. Escribiendo el texto con la codificación "por defecto"

2.1.1. Windows

Crea un documento nuevo de texto en Windows. Ponle el nombre "`texto-win.txt`" y escribe en su interior el siguiente texto[2] :

```
¿Qué codificación estoy usando?
Apostaría 1€ a que no se sabe.
```

Al guardar el archivo, el bloc de notas usará por defecto la codificación Windows 1252.

2.1.2. MS-DOS

Abre una interfaz de comandos. Usando el comando `cd`, "navega" hasta la carpeta en la que tienes el fichero anterior. Teclea el comando `EDIT TEXTO-DOS.TXT` para crear un nuevo archivo y escribe en él el mismo texto que habías escrito antes. Tecléalo de nuevo, no uses "copiar y pegar". En este caso es imposible obtener el símbolo del euro, pues no está presente en la tabla de códigos de MS-DOS. Escribe "EUR" en su lugar.

Al guardar el archivo, se almacenará con la *codepage* que esté activa en ese momento, que por defecto será la CP-850.

Si usas el comando `TYPE` para ver el contenido de los ficheros, ver que puedes leer correctamente el texto que has creado con `EDIT`, pero que no se ve bien el que has creado con el bloc de notas. Mediante el comando `chcp 1252` puedes cambiar la *codepage* que usa la interfaz de comandos, para que use la misma que Windows. En este caso podrás ver bien el texto escrito con el bloc de notas, pero ya no verás correctamente lo escrito con EDIT.

2.1.3. Linux

Para editar en Linux podemos usar diferentes editores. Cada uno tiene sus peculiaridades a la hora de introducir caracteres no-ASCII. Veremos tres de los editores más habituales.

emacs o xemacs Cuando este editor se ejecuta en "modo ventana" no suele tener problemas para admitir letras acentuadas u otros caracteres no-ascii. Sin embargo, si se ejecuta en "modo terminal" (con la opción `-nw` o cuando no hay un servidor X disponible), trata las vocales acentuadas y los caracteres no ASCII como controles especiales. Para evitar que esto ocurra, debemos crear un fichero y escribir en él el siguiente código:

[2]Para introducir el símbolo del euro, algunos teclados soportan la combinación AltGr-E. En otros funciona AltGr-5. En los que no sea así, siempre se puede obtener un símbolo si se sabe su código Windows. Para ello mantén pulsada la tecla Alt y escribe en el teclado numérico el código del carácter en cuestión, en base 10 y precedido por un cero. En este caso, 0128.

```
(set-input-mode (car (current-input-mode))
                (nth 1 (current-input-mode)) 0)
```

Este fichero debe estar guardado en un lugar y con un nombre concretos, pero este lugar y nombre depende de si el editor se llama `emacs` o `xemacs`:

emacs En este caso el fichero debe llamarse `.emacs` (observar el punto delante) y almacenarse en la carpeta principal del usuario (puedes "saltar" directamente a esta carpeta escribiendo el comando `cd` sin argumentos).

xemacs En este caso el fichero debe llamarse `init.el` y debe estar almacenado en la carpeta `.xemacs` (observar el punto inicial), que cuelga de la carpeta principal del usuario (a la que puedes "saltar" directamente escribiendo el comando `cd` sin argumentos).

Sin entrar en demasiados detalles, digamos que este código se ejecuta cada vez que se arranca `emacs` o `xemacs` y que causa que los caracteres con códigos mayores de 127 no se traten como controles, sino como letras normales.

Una vez configurado emacs como se ha descrito, se pueden escribir normalmente los caracteres españoles, como la eñe, el signo de abrir interrogación, las vocales acentuadas, etc. Sin embargo el signo del Euro no está disponible en todos los teclados. Se puede intentar la combinación AltGr-E, pero si esta no funciona, `emacs` tiene un mecanismo más general para introducir cualquier carácter si sabemos su código ISO-8859-15. Basta pulsar Ctrl-q y seguidamente el código del caracter, pero eso sí, en base 8. El código ISO del Euro en base 8 es el 244, por lo que, para introducir este carácter, hay que pulsar Ctrl-q y seguidamente escribir 244 y pulsar Enter.

vi No se requiere ninguna configuración especial para poder introducir directamente los caracteres existentes en el teclado (tales como el signo de abrir interrogación, la eñe, las vocales acentuadas, etc.) Si se quiere introducir un carácter que no esté presente en el teclado, también es posible siempre que conozcamos su código ISO-8859-15. Esto puede ser necesario para introducir el signo del Euro desde un teclado que no lo tenga.

Para introducir un carácter cualquiera, conocido su código, basta pulsar Ctrl-V y teclear el valor numérico del código, en base 10. También admite que el código se teclee en otras bases, si se pone el prefijo adecuado, que es `x` para hexadecimal y `o` para octal. De manera que para introducir el signo del Euro, cualquiera de las siguientes posibilidades es válida: `C-v 164`, `C-v o244` ó `C-v xA4`. En versiones modernas de `vi` es posible incluso introducir cualquier carácter Unicode, usando el prefijo `u` y el código en hexadecimal. Así el Euro sería `C-v u20AC`. Este mecanismo no funciona en la versión de `vi` disponible en las máquinas de prácticas, y por tanto no es posible introducir "directamente" caracteres de alfabetos no occidentales.

joe Este editor no requiere configuración especial para introducir los caracteres españoles existentes en el teclado. Sin embargo, si se quiere introducir un carácter que no esté en el teclado (como el Euro en algunos teclados), el editor no proporciona mecanismos alternativos para poder introducirlo. Por ello en estos casos se recomienda usar `emacs`, `xemacs` o `vi`.

Utilizando el editor que prefieras (recomendado el `vi` por permitir introducir cualquier caracter ISO-latin dado su código hexadecimal), escribe de nuevo el texto anterior. Si transfieres este archivo a Windows y lo abres con el bloc de notas, verás que se ve todo correctamente excepto el euro, que aparecerá como "¤" (símbolo de la divisa genérica). Compara las tablas de códigos ISO-8859-15 y Windows 1252 y entenderás por qué.

2.2. Escribiendo el texto en UTF-8

Ahora, en lugar de dejar que el editor elija la codificación por defecto, forzaremos a que use UTF-8. Para ello:

En Windows Creamos un nuevo documento de texto, llamado `texto-utf8-win.txt` y lo editamos con el bloc de notas, al igual que hicimos en el primer punto, pero en el momento de guardar, elegiremos codificación UTF-8 de una pequeña lista que aparece en el diálogo de guardar.

Windows erróneamente guarda al principio de fichero un carácter BOM, (U+FEFF) el cual realmente sólo es necesario cuando se usa codificación UTF-16. No hay forma de evitar esto, y el carácter tampoco se puede borrar abriendo de nuevo el fichero, ya que es un carácter invisible.

En Linux La forma más sencilla de crear un archivo que use UTF-8 en Linux es partir de otro que use ISO-8859-15 y convertirlo con la herramienta `iconv`. Esta herramienta permite cambiar la codificación de un archivo y soporta prácticamente todas las codificaciones existentes. La opción `-f` es para especificar la codificación del archivo origen (f de "from") y la opción `-t` la del archivo destino (t de "to"). Con la opción `-l` se obtiene la lista de codificaciones soportadas. Finalmente, la opción `-o` permite especificar el nombre del fichero donde se guardará la conversión (o de "output").

Usando esta herramienta tomaremos el archivo que ya habíamos escrito en la sección 2.1.3 y lo recodificaremos como UTF-8, dejando el resultado en un nuevo fichero que llamaremos `texto-utf8-linux.txt`. El comando necesario es:

```
$ iconv -f iso-8859-15 -t utf-8 -o texto-utf8-linux.txt texto-linux.txt
```

En versiones modernas de `emacs`, `xemacs` y `vi` es posible introducir cualquier carácter Unicode y decirle después al editor que guarde el fichero en UTF-8. Sin embargo, en las versiones instaladas en las máquinas de prácticas ésto no es sencillo por lo que se recomienda usar `iconv` como se ha descrito anteriormente. El inconveniente es que no podremos escribir textos de alfabetos no occidentales.

Si comparas las longitudes de los ficheros `texto-utf8-win.txt` y `texto-utf8-linux.txt` verás que el primero es mayor. Esto se debe, por una parte, a que Windows utiliza dos códigos para almacenar el retorno de carro, mientras que linux sólo utiliza uno. Por otro lado, Windows incorrectamente añade al principio del fichero el carácter BOM (U+FEFF), que no es necesario cuando el formato es UTF-8, ya que este formato no depende de la *endianity* de la máquina.

Si vuelcas con la herramienta `hexdump` el fichero generado por windows, verás que comienza así:

```
$ hexdump -C texto-utf8-win.txt
00000000  ef bb bf c2 bf 51 75 c3  a9 20 63 6f 64 69 66 69  |ï»¿Â¿QuÃ© codifi|
...
```

Intenta decodificar en un papel aparte los seis primeros bytes de este fichero (es decir, escribe qué códigos Unicode representan). Si lo haces correctamente verás que se trata de U+FEFF, U+00BF y U+0051. El primero es el BOM que Windows introduce incorrectamente, el segundo es el carácter "¿" (como puedes comprobar en la tabla de ISO-8859-15) y el tercero es el carácter "Q" (como puedes comprobar en la tabla ASCII). Es decir, son las primeras letras del texto que has escrito.

Volcando de la misma forma el fichero escrito en linux verás que no incluye al principio los bytes que codifican U+FEFF, y el resto es idéntico.

2.3. Visualizando el texto UTF-8

Los bytes concretos que están almacenados en el fichero UTF-8 son los que hemos visto con ayuda de la herramienta `hexdump`. Estos bytes representan diferentes caracteres. Algunos caracteres están codificados en un solo byte, mientras que otros requieren dos o más.

Si usamos el comando `cat` para volcar a la pantalla el contenido de un fichero, éste comando se limitará a enviar a la terminal los bytes que encuentre en el fichero. El programa `cat` no sabe nada de UTF-8, y si encuentra la secuencia de bytes `c2 bf` se limita a enviar estos bytes a la terminal, sin saber que en realidad codifican un único carácter (en este caso el carácter de abrir interrogación).

En realidad, es la terminal la que tiene que estar preparada para "comprender" UTF-8. La terminal interpretará la secuencia de bytes que le llega de los diferentes programas que se ejecutan, usando una tabla de codificación propia de la terminal. Por defecto, esta tabla será probablemente ISO-8859-1 o bien ISO-8859-15. Según esta tabla la secuencia `c2 bf` no representa un carácter de abrir interrogación, sino dos caracteres. El primero representa una 'Â', y el segundo, casualmente, un '¿'. Y así con todos los demás caracteres UTF-8 que reciba. Es por esto que veremos algo como lo siguiente:

```
$ cat texto-utf8-linux.txt
Â¿QuÃ© codificaciÃ³n estoy usando?
ApostarÃa 1â□¬ a que no se sabe.
```

Para que el texto se visualice correctamente, necesitamos una terminal que tenga capacidad de "comprender" UTF-8. Por ejemplo, podemos usar el programa `PuTTy` desde Windows, el cual permite especificar la codificación soportada por la terminal (entre las opciones de Ventana, una llamada "Traducción").

Otra posibilidad es conectarse a la máquina Linux usando `ssh` con la opción `-X`, y una vez allí lanzar otra terminal con el comando:

```
$ gnome-terminal &
```

Esto abrirá una nueva ventana en la que tendremos otro *shell* remoto. Esta terminal tiene una barra de menú en la parte superior, y en "Terminal→Establecer codificación de caracteres" se puede seleccionar UTF-8. Si hacemos esto, ya podremos ver correctamente el contenido del fichero, tanto el que hemos creado en Linux como el que habíamos ecreado en Windows.

```
$ cat texto-utf8-linux.txt
¿Qué codificación estoy usando?
Apostaría 1€ a que no se sabe.

$ cat texto-utf8-win.txt
□¿Qué codificación estoy usando?
Apostaría 1€ a que no se sabe.
```

Observa, no obstante, que al volcar el fichero Windows aparece un carácter extraño al principio de la primera línea. Se trata del carácter BOM, que en teoría debería ser invisible (ya que se trata de un espacio en blanco de ancho cero). Según la fuente que la terminal tenga instalada, puede mostrarse como un cuadrado vacío[3]. El programa PuTTY, por ejemplo, muestra correctamente este carácter. Es decir, no lo muestra.

De modo que configurando la terminal para que "decodifique" utf8, ya podemos visualizar correctamente las cadenas de texto almacenadas en ese formato. No obstante, ahora se mostrarán incorrectamente los ficheros no utf8, ya que la terminal, cuando reciba un carácter ISO-8859-15 como "á", entenderá que es parte de una secuencia multi-byte UTF8, y ya que el carácter siguiente habitualmente no comenzará por 10..., se producirá un error de decodificación. Cada vez que la terminal detecte un error de decodificación, generará el carácter Unicode U+FFFF, que es un carácter inexistente y que se mostrará normalmente como una caja vacía.

```
□Qu□ codificaci□n estoy usando?
Apostar□a 1□ a que no se sabe.
```

2.4. Experimento: escribiendo UTF-8 "a mano"

Puesto que las aplicaciones no saben nada de UTF-8, y es la terminal la que sí sabe, es posible editar un archivo de texto e introducir "a mano" los bytes que componen una secuencia UTF-8. Cuando este archivo sea volcado en una terminal configurada para "decodificar" UTF-8, se mostrará correctamente el carácter deseado.

Por ejemplo, supongamos que queremos escribir "TETЯIS", usando la 'Я' del alfabeto cirílico, cuyo código es U+042F. Aún si nuestro editor no sabe nada de UTF-8, siempre podemos averiguar "a mano" la codificación UTF-8 de ese carácter. Se recomienda que en este punto no sigas leyendo e intentes por tí mismo hallar esta codificación.

¿Ya lo tienes? El resultado correcto es `D0 AF`. Pues bien, ya que estos dos bytes son códigos ISO-8859-15 válidos (corresponden a la 'Ð' y al signo de subrayado superior '¯'), siempre será posible introducirlos en un editor como `vi`. En este caso, ya que ninguno de estos dos caracteres está disponible en el teclado, tendremos que usar el método de pulsar `Ctrl-V` y seguidamente el código hexadecimal, precedido por una `x`. De este modo, escribiremos en el editor el texto siguiente:

```
TETÐ¯IS
```

Cuando este texto sea volcado en una terminal UTF-8, si la fuente usada por esa terminal dispone de caracteres cirílicos, mostrará 'TETЯIS'.

3. Anexo: listados

3.1. `productor-incompleto.c`

```
1  /*
2     Este programa genera un fichero en el que se guarda un dato para
3     cada tipo declarado en "tipos.x"
```

[3] El cuadrado vacío suele ser el signo utilizado por la terminal para indicar que ese carácter Unicode no lo tiene disponible en la fuente utilizada.

```
4
5       Los valores de cada dato están prefijados en este código fuente,
6       excepto para el último dato (tipo MiUnion) en que se pide al
7       usuario el tipo deseado y el valor para ese caso.
8   */
9   #include "tipos.h"
10  int main(int argc, char *argv[])
11  {
12    /* Declaramos una variable de cada tipo */
13    Entero i;
14    Real   x;
15    Texto  t;          /* Es en realidad un puntero a char */
16    ArrayEnteros a;    /* Es un array de 4 */
17    MiUnion u;
18
19    /* Declaramos las variables necesarias para que los filtros
20       escriban en un fichero los datos */
21    FILE *fichero;
22    XDR  operacion;
23
24    /* Otras variables auxiliares */
25    int j;   /* Para bucles */
26    char txtaux[20];   /* buffer para leer texto del teclado */
27
28    /* Comienza la ejecución, inicializando las variables */
29    /* Inicializar el entero */
30    i = 10;
31
32    /* Inicializar el flotante */
33    x = 1.0;
34
35    /* Inicializar el texto, que es en realidad un puntero a char */
36    t = "Prueba";   /* Hacemos apuntar el puntero a la cadena "Prueba" */
37
38    /* Inicializar el array de 4 enteros */
39    for (j=0; j<4; j++)   a[j]=j*2;
40
41    /* Inicializar la union (preguntando al usuario) */
42    printf("Elija el tipo para la unión:\n");
43    printf(" 1: Entero\n");
44    printf(" 2: Real\n");
45    printf(" 3: Texto\n");
46    printf(" 4: ArrayEnteros\n");
47    printf(" otro: sin datos\n");
48    printf("Su elección: ");
49    scanf("%d", &u.tipo);
50    /* Pedir el dato, dependiendo del tipo elegido */
51    switch(u.tipo) {
52    case 1:
53      printf("Valor del entero: ");
54      scanf("%d", &u.MiUnion_u.i);
55      break;
56    case 2:
57      printf("Valor del real: ");
58      scanf("%f", &u.MiUnion_u.x);
59      break;
60    case 3:
61      /* Leer el retorno de carro que quedó tras el último scanf, para
62         evitar que fgets lo encuentre y lea una linea en blanco. */
63      getchar();
64      printf("Escriba un texto: ");
65      fgets(txtaux, 20, stdin); /* Leer la línea */
66      u.MiUnion_u.t=txtaux;   /* Hacemos apuntar el puntero al buffer auxiliar */
67      break;
68    case 4:
69      for (j=0; j<4; j++) {
70        printf("Valor del entero[%d]: ", j);
71        scanf("%d", &u.MiUnion_u.a[j]);
72      }
73      break;
74    default:
75      printf("No se requiere dato\n");
76    }
77
78    /* Creación del fichero para guardar los datos */
79    fichero=fopen("Datos.xdr", "w");
80    if (fichero==NULL) {
81      perror("Al crear el fichero");
```

```
82      exit(-1);
83    }
84    /* Preparación de la operación para codificar */
85    xdrstdio_create(&operacion, fichero, XDR_ENCODE);
86
87    /* Llamada a los filtros para volcar los datos */
88    if (xdr_Entero(&operacion, &i)!=TRUE) {
89      fprintf(stderr, "Error al escribir el Entero\n");
90    }
91    /* Completar lo que falta para que se vuelquen los restantes datos,
92       en el orden en que han sido declarados en main()
93    */
94
95
96    fflush(fichero);   /* Para forzar el volcado */
97    /* Una vez almacenado, podemos destruir la estructura XDR
98       y cerrar el fichero */
99    xdr_destroy(&operacion);
100   fclose(fichero);
101   printf("\nFichero generado\n");
102 }
```

Índice