8-4-2024

XDR- Tipos de conversión

Práctica 2.2

Julián Blanco González
UNIVERSIDAD DE EXTREMADURA

Contenido

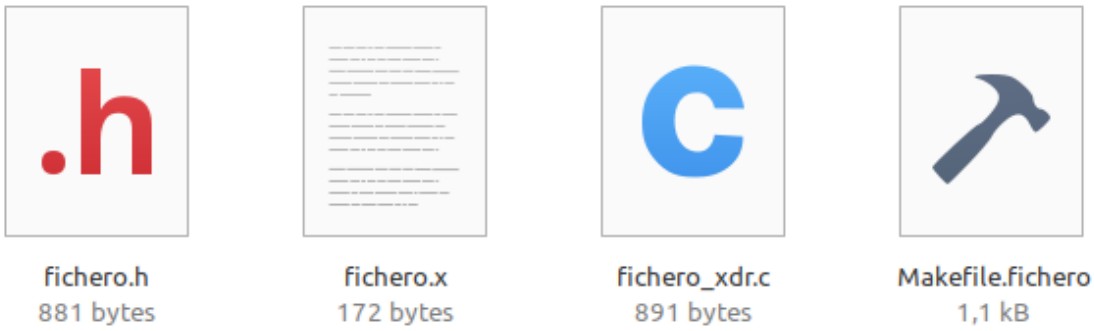
La práctica 2.2 consiste en ir viendo como funciona el archivo de cabecera (extensión ".h") y el archivo de filtro xdr con diferentes tipos de datos.

Ejercicio 1

Crear un fichero.x con este contenido:

```
const MAXNAMELEN = 255; /* longitud máxima */
typedef string nametype<MAXNAMELEN>; /* nombre del directorio */
typedef struct namenode *namelist; /* enlace de la lista */
```

¿Qué ficheros se generan?



¿Qué hay en el *.h y en el fichero *_xdr.c?

Cabecera (“.h”):

```
#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif

#define MAXNAMELEN 255

typedef char *nametype;

typedef struct namenode *namelist;

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_nametype (XDR *, nametype*);
extern  bool_t xdr_namelist (XDR *, namelist*);

#else /* K&R C */
extern bool_t xdr_nametype ();
extern bool_t xdr_namelist ();

#endif /* K&R C */
```

- Se incluye la librería rpc.h
- Se declara la constante MAXNAMELEN
- Se definen el tipo de datos char* de nametype (string)
- Se define la estructura de *namelist
- Se crean las funciones de filtro XDR que se usan en el archivo xdr.c

*Como curiosidad, también se generan estas funciones xdr con el estilo K&R (Kernighan and Ritchie).

XDR.c:

```c
#include "fichero.h"

bool_t
xdr_nametype (XDR *xdrs, nametype *objp)
{
    register int32_t *buf;

     if (!xdr_string (xdrs, objp, MAXNAMELEN))
         return FALSE;
    return TRUE;
}

bool_t
xdr_namelist (XDR *xdrs, namelist *objp)
{
    register int32_t *buf;

     if (!xdr_pointer (xdrs, (char **)objp, sizeof (struct namenode), (xdrproc_t) xdr_namenode))
         return FALSE;
    return TRUE;
}
```

- Se incluye la cabecera
- Se crean las dos funciones generadas en la cabecera de filtro xdr

Explica con tus palabras los tipos que se han creado.

#define MAXNAMELEN 255

Constante que no ocupa memoria que tiene un valor de 255

*typedef char *nametype*

Es una cadena de char, para representar un String de tamaño máximo 255 (MAXNAMELEN)

*typedef struct namenode *namelist*

Puntero a una estructura creada por el usuario.

La constante definida ¿dónde se la declara con un #define?

En el fichero de cabecera, para que así se pueda usar en el cliente o en el servidor siempre y cuando se añada la cabecera (“#include cabecera.h”).

¿Cómo se utilizarían en el código fuente del servidor?

Primero, incluir la cabecera:

```
#include “fichero.h”
```

Constante:

```
If (tamano < MAXNAMELEN);
```

typedef char *nametype:

```
nametype nombre;
nombre[0] = ‘A’;
```

typedef struct namenode *namelist:

```
namelist lista;
```

`lista->campo` = rand(); //Aleatorio de un campo entero que tuviera la estructura

Ejercicio 2

Analizar la definición del tipo enumerado.

Este es el contenido de mi fichero.x:

```
enum Estaciones {PRIMAVERA=1, VERANO=2, OTONO=3, INVIERNO=4};
```

En la cabecera, se genera esto:

```
enum Estaciones {
	PRIMAVERA = 1,
	VERANO = 2,
	OTONO = 3,
	INVIERNO = 4,
};
typedef enum Estaciones Estaciones;
```

Se genera una estructura con el nombre de Estaciones (variable del .x) y cada campo corresponde con el número de datos que tenga la enumeración. Por último, se declara un tipo de datos enum que referencia a esta estructura.

Y en el XDR, se genera el filtro necesario con el nombre de xdr_Estaciones y dentro de éste, se le pasa el filtro que ya tiene XDR creado, que se llama xdr_enum.

Para usar esta enumeración en el servidor, la mejor solución sería un switch:

```
printf("Estación: “);
switch (estacion) {
        case PRIMAVERA:
            printf("PRIMAVERA\n");
            break;
        case VERANO:
            printf("VERANO\n");
            break;
        case OTONO:
            printf("OTOÑO\n");
            break;
        default:
            printf("INVIERNO\n");
            break;
    }
```

Arrays de tamaño fijo vs Arrays tamaño variable

Aquí hay que comparar las formas con las que se crea un array, ya sea de tamaño fijo o variable.

Este es mi fichero.x:

```
/*4 formas de declaración de tamaño fijo */

struct VecFijo1{
    int v[20];
};

struct VecFijo2{
    int v<20>;
};

typedef int vectorFijo3 [20];
typedef int vectorFijo4 <20>;

/* 2 formas de declaración de tamaño variable */

typedef int vectorVariable1 <>;
struct vectorVariable2{
    int v<>;
};
```

La cabecera queda así:

```
struct VecFijo1 {
    int v[20];
};
typedef struct VecFijo1 VecFijo1;

struct VecFijo2 {
    struct {
        u_int v_len;
        int *v_val;
    } v;
};
typedef struct VecFijo2 VecFijo2;

typedef int vectorFijo3[20];

typedef struct {
    u_int vectorFijo4_len;
    int *vectorFijo4_val;
} vectorFijo4;

typedef struct {
    u_int vectorVariable1_len;
    int *vectorVariable1_val;
} vectorVariable1;

struct vectorVariable2 {
    struct {
        u_int v_len;
        int *v_val;
    } v;
};
typedef struct vectorVariable2 vectorVariable2;
```

Cuando el array se genera con “< >”, siempre se crea una estructura con dos campos: tamaño y valor, en cambio, cuando se usa “[ ]”, se crea la estructura o el tipo de datos que se haya puesto.

Y en el archivo XDR se diferencian 2 tipos de filtros:

- XDR_array: para arrays de tamaño variable.
- XDR_vector: para arrays de tamaño fijo.

Por último, en el server siempre hay que hacer 3 pasos antes de manipular cualquier array:

1. Liberar el espacio de memoria de antes
2. Darle el tamaño al vector (vector variable)
3. Reservar la memoria para el tamaño del vector

Ejercicio 3

Comparar cadenas y array entre string y char.

Este es el contenido de mi fichero.x:

```
typedef string cadena <>;
typedef char array_variable <>;
typedef char array_fijo [5];
```

La cabecera que se genera queda así:

```
typedef char *cadena;

typedef struct {
    u_int array_variable_len;
    char *array_variable_val;
} array_variable;

typedef char array_fijo[5];
```

La primera sustituye string, por un puntero a char.

La segunda, al ser variable, crear la estructura como en los vectores, solo que el campo valor, es otro puntero a char.

El tercero, es un vector de array con tamaño fijo, por lo que es igual.

En el archivo XDR, se genera un filtro diferente para cada uno:

- String cadena <>: xdr_string → Tamaño variable, pero tiene un tamaño máximo (2^32 – 1).

- Char array_variable <>: xdr_array → Tamaño variable

- Char array_fijo []: xdr_vector → Tamaño fijo

Ejercicio 4

Comprobar la cabecera y el xdr de una estructura de un “objeto” Persona.

La cabecera queda así:

```
struct Persona {
    int edad;
    char *nombre;
    char *apellidos;
};
typedef struct Persona Persona;
```

Se genera como cualquier estructura con sus campos correspondientes.

Luego, en el XDR:

```
bool_t
xdr_Persona (XDR *xdrs, Persona *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->edad))
         return FALSE;
     if (!xdr_string (xdrs, &objp->nombre, ~0))
         return FALSE;
     if (!xdr_string (xdrs, &objp->apellidos, ~0))
         return FALSE;
    return TRUE;
}
```

Se le pasa el filtro XDR según el tipo del campo de la estructura.

Ahora, como posible código de servidor sería así:

Argp -> edad;

Argp -> nombre;

Argp -> apellidos;

Y el cliente así:

Persona p;

p.edad = edad por teclado;

p.nombre = nombre por teclado;

p.apellidos = apellidos por teclado;

Ejercicio 5

Ver como funciona la “estructura” unión. Para ello, he realizado un pequeño programa de prueba para probarla. Este es mi fichero.x:

```
struct datos{
    int a;
    int b;
};

union Resultado switch (int tipo) {
    case 1: int entero;
    case 2: double real;
    default: string cadena <>;
};

program CalculadoraDivision {
    version Calc_1 {
        Resultado division (datos)=1;
    }=1;
}=0x20000003;
```

El programa pide dos números enteros (struct datos) y los divide. Si el divisor es 0, devolveré un String diciendo que no se puede dividir por 0; si el resto es 0 (división entera), devolveré un entero; y si no, haré la división devolviendo un real.

La cabecera queda así:

```
struct datos {
    int a;
    int b;
};
typedef struct datos datos;

struct Resultado {
    int tipo;
    union {
        int entero;
        double real;
        char *cadena;
    } Resultado_u;
};
typedef struct Resultado Resultado;
```

Se crea la estructura Resultado con dos campos:

- Tipo: entero para hacer referencia al campo que queramos acceder.
- Resultado_u: union que contiene los subcampos donde se almacenan los resultados de la calculadora dependiendo de los números introducidos.

En el fichero XDR:

```
bool_t
xdr_Resultado (XDR *xdrs, Resultado *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->tipo))
         return FALSE;
    switch (objp->tipo) {
    case 1:
         if (!xdr_int (xdrs, &objp->Resultado_u.entero))
             return FALSE;
        break;
    case 2:
         if (!xdr_double (xdrs, &objp->Resultado_u.real))
             return FALSE;
        break;
    default:
         if (!xdr_string (xdrs, &objp->Resultado_u.cadena, ~0))
             return FALSE;
        break;
    }
    return TRUE;
}
```

Se le pasa el filtro XDR_int al campo tipo y luego, para acceder a cada subcampo de unión, se usa un switch y dependiendo del tipo, se pasa un filtro XDR u otro.

La función del servidor queda así:

```
Resultado *
division_1_svc(datos *argp, struct svc_req *rqstp)
{
    static Resultado  result;

    int dividendo = argp->a;
    int divisor = argp->b;

    if(divisor == 0){
        result.tipo=3;
        result.Resultado_u.cadena = "Error. No se puede dividir por 0";
    }
    else{
        if(dividendo % divisor == 0) {
            result.tipo=1;
            result.Resultado_u.entero = (dividendo / divisor);
        }
        else{
            result.tipo = 2;
            result.Resultado_u.real = (double)dividendo / divisor;
        }
    }


    return &result;
}
```

- Si el divisor es 0, voy al campo de String (default) de la estructura Resultado, y le añado un mensaje de error.

- Si el resto es 0 (división entera), voy al campo entero (case: 1) y añado el resultado de la división entera.

- Si no, hago la división, convirtiendo el resultado a double (case: 2).

Cliente:

```
printf("Primer número: ");
scanf("%d", &division_1_arg.a);
printf("Segundo número: ");
scanf("%d", &division_1_arg.b);

result_1 = division_1(&division_1_arg, clnt);
if (result_1 == (Resultado *) NULL) {
    clnt_perror (clnt, "call failed");
}
else{
    switch (result_1->tipo)
    {
    case 1: printf("Resultado division entera: %d\n", result_1->Resultado_u.entero);
        break;
    case 2: printf("Resultado division: %lf\n", result_1->Resultado_u.real);
        break;
    default: printf("%s\n", result_1->Resultado_u.cadena);
        break;
    }
}
```

Recojo los dos números enteros por teclado y los agrego directamente a la estructura de datos. Llamo a la función del servidor y si se ejecuta bien, utilizando un switch, usando el campo entero de tipo, muestro el campo correspondiente.

Ejemplos de la ejecución:

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/S2_2/5$ ./f2_client localhost
Primer número: 6
Segundo número: 1
Resultado division entera: 6
```

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/S2_2/5$ ./f2_client localhost
Primer número: 6
Segundo número: 4
Resultado division: 1.500000
```

```
julian@julian-VirtualBox:~/Escritorio/Distribuidos/S2_2/5$ ./f2_client localhost
Primer número: 6
Segundo número: 0
Error. No se puede dividir por 0
```

Bibliografía

Hacer la práctica

PDF de la práctica: S2_ XDR - Filtros de conversión.pdf

Buscar información

K&R (Kernighan and Ritchie)

https://en.wikipedia.org/wiki/C_(programming_language)#K&R_C

Filtros XDR

https://linux.die.net/man/3/xdr_enum