Práctica 2. Primera parte: Calculadora Sun RPC

Noelia Escalera Mejías Grupo DSD1

22 de marzo de 2020

Índice

1. Introducción

El primer paso a seguir para implementar la calculadora es escribir el fichero .x (*calculadora.x* para nosotros) y luego generar los archivos con **rpcgen -NCa**. Se ha escrito un método para cada operación. He aquí una captura de pantalla del archivo:

```
typedef float t_vector<>;

struct potencia{
    float base;
    float exponente;
};

typedef struct potencia potencia;

program CALCULADORA {
    version CALCULADORAVERS{
        float SUMAR (float,float) = 1;
        float RESTAR (float,float) = 2;
        float MULTIPLICAR (float,float) = 3;
        float DIVIDIR (float,float) = 4;
        float PRODUCTO_ESCALAR (t_vector,t_vector) = 5;
        t_vector ESCALAR_POR_VECTOR (float,t_vector) = 6;
        t_vector SUMA_VECTORES (t_vector,t_vector) = 7;
        t_vector RESTA_VECTORES (t_vector,t_vector) = 8;
        t_vector PRODUCTO_VECTORIAL (t_vector,t_vector) = 9;
        float PRODUCTO_MIXTO (t_vector,t_vector,t_vector) = 10;
        float MEDIA (t_vector) = 11;
        float POTENCIA (potencia) = 13;
        potencia MULTIPLICAR_POTENCIAS (potencia,potencia) = 14;
        potencia DIVIDIR_POTENCIAS (potencia,potencia) = 15;
    } = 1;
} = 0x20000001;
```

Podemos clasificar las operaciones de la calculadora implementada en 4 grupos: Operaciones básicas, operaciones con vectores, media y operaciones con potencias. Nada más entrar en el programa aparecerá un menú para seleccionar una de ellas:

```
noelia@noelia-pc:~/Escritorio/Universidad/3º/DSD/Prácticas/Calculadora$ ./calculadora_server &
[1] 13280
noelia@noelia-pc:~/Escritorio/Universidad/3º/DSD/Prácticas/Calculadora$ ./calculadora_client localhost
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
```

Si introducimos una operación no especificada, nos saltará un mensaje de error y el programa parará su ejecución.

A continuación, hablaremos más en profundidad de cada uno de los distintos grupos de operaciones.

2. Operaciones básicas

En el guión se especificaba que la calculadora debía hacer operaciones básicas con enteros (sumar, restar, multiplicar y dividir). Sin embargo, en el presente caso se ha tomado la decisión de hacer las operaciones con float para tener más juego. Tenemos entonces, que las operaciones de este bloque son sumar, restar, multiplicar y dividir (ver en el fichero .x o en el del servidor).

Si seleccionamos operaciones básicas en el menú, nos aparece una pequeña ayuda para introducir el formato:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
1
Introduce la operación de la siguiente manera: <operando 1> <operador> <operando2>
3 + 2
El resultado es 5.000000
```

Si el formato no es correcto, nos mostrará un mensaje diciendo que la operación no está definida y mostrará en pantalla el resultado 0.

En el cliente simplemente se ha hecho un scanf de los dos operandos y posteriormente un switch del operador. Según el operador escrito, se llamará a una operación del servidor u otra. En el servidor simplemente cada función realiza la operación solicitada con los operandos escogidos. He aquí un ejemplo con la función suma, las diferentes se hacen de manera análoga:

```c
float *
sumar_1_svc(float arg1, float arg2,  struct svc_req *rqstp)
{
    static float  result;

    result = arg1+arg2;

    return &result;
}
```

3. Operaciones con vectores

Si seleccionamos esta opción, nos aparecerá otro menú con las distintas operaciones que podemos realizar.

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
```

El principal problema se ha presentado en esta parte era que no se podían devolver vectores como tal. Se solucionó consultando el foro de dudas de la práctica y usando el typedef t_vector que se puede observar en el archivo .x. También ha sido necesario crear una variable auxiliar en el cliente de tipo t_vector para las funciones que devolvían punteros de este tipo en la que copiar el resultado de las operaciones, ya que ha habido problemas a la hora de mostrar los resultados.

3.1. Producto escalar

El producto escalar consiste en multiplicar dos vectores de forma que el resultado sea un escalar. Multiplicamos la coordenada 0 del vector 1 con la coordenada 0 del vector 2, la 1 con la 1, etc y vamos sumando los resultados. He aquí la implementación:

```
float *
producto_escalar_1_svc(t_vector arg1, t_vector arg2,  struct svc_req *rqstp)
{
    static float  result;
    float r = 0;
    int contador = 0;

    while(contador<arg1.t_vector_len){
        r += arg1.t_vector_val[contador]*arg2.t_vector_val[contador];
        contador++;
    }

    result = r;

    return &result;
}
```

Cuando seleccionamos esta opción en el menú se pedirá una dimensión para los 2 vectores (ya que los 2 deben tener la misma) y a continuación se pedirán los componentes:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
1
Introduce la dimensión de los vectores: 3
Introduce el primer vector:
1 2 3
Introduce el segundo vector:
1 1 1
Resultado: 6.000000
```

Los problemas encontrados aquí han sido con respecto a leer los vectores. Es necesario reservar memoria con malloc al dato t_vector_val del vector que se va a leer y actualizar el dato t_vector_len. También ha habido problemas con la variable result, al acumular sobre ella cada ejecución (aunque tuviera los mismos datos) daba un valor distino. Para arreglar esto se ha usado la variable r y se ha acumulado sobre ella, copiando su valor final en result.

3.2. Suma de vectores

Para sumar vectores simplemente hay que sumar las coordenadas de los vectores una a una, de manera que la suma de las coordenadas 0 de los vectores será la coordenada 0 del nuevo vector y así sucesivamente. He aquí la implementación:

```c
t_vector *
suma_vectores_1_svc(t_vector arg1, t_vector arg2,  struct svc_req *rqstp)
{
    static t_vector  result;
    int contador = 0;
    result.t_vector_val = (float *) malloc(50*sizeof(float));
    result.t_vector_len = 0;

    while (contador < arg1.t_vector_len){
        result.t_vector_val[contador] = arg1.t_vector_val[contador] + arg2.t_vector_val[contador];
        result.t_vector_len++;
        contador++;
    }

    return &result;
}
```

El diálogo es prácticamente idéntico al del apartado anterior:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
2
Introduce la dimensión de los vectores: 4
Introduce el primer vector:
1 2 3 4
Introduce el segundo vector:
-1 7.5 3 4
Vector resultado: 0.000000 9.500000 6.000000 8.000000
```

3.3. Resta de vectores

Idéntica a la suma de vectores, cambiando sumar por restar.

```c
t_vector *
resta_vectores_1_svc(t_vector arg1, t_vector arg2,  struct svc_req *rqstp)
{
    static t_vector  result;
    int contador = 0;
    result.t_vector_val = (float *) malloc(50*sizeof(float));
    result.t_vector_len = 0;

    while (contador < arg1.t_vector_len){
        result.t_vector_val[contador] = arg1.t_vector_val[contador] - arg2.t_vector_val[contador];
        result.t_vector_len++;
        contador++;
    }

    return &result;
}
```

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
3
Introduce la dimensión de los vectores: 3
Introduce el primer vector:
1 4 5
Introduce el segundo vector:
-1 2 3
Vector resultado: 2.000000 2.000000 2.000000
```

3.4. Producto vectorial

A diferencia del producto escalar, el resultado del producto vectorial es un vector. Lo que hace es calcular un vector perpendicular a los 2 vectores dados, es por eso que no tiene sentido que los vectores tengan una dimensión mayor que 3. Se calcula resolviendo el siguiente determinante:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

He aquí la implementación:

```c
t_vector *
producto_vectorial_1_svc(t_vector arg1, t_vector arg2,  struct svc_req *rqstp)
{
    static t_vector  result;
    int tam = 3;

    result.t_vector_val = (float *) malloc(tam*sizeof(float));
    result.t_vector_len = tam;

    result.t_vector_val[0] = arg1.t_vector_val[1]*arg2.t_vector_val[2] - arg1.t_vector_val[2]*arg2.t_vector_val[1];
    result.t_vector_val[1] = arg1.t_vector_val[2]*arg2.t_vector_val[0] - arg1.t_vector_val[0]*arg2.t_vector_val[2];
    result.t_vector_val[2] = arg1.t_vector_val[0]*arg2.t_vector_val[1] - arg1.t_vector_val[1]*arg2.t_vector_val[0];

    return &result;
}
```

El diálogo es parecido a los anteriores, pero no pregunta por la dimensión ya que como hemos mencionado antes, no se puede hacer mayor que 3 (si se quiere

hacer de 2 basta con poner 0 en el último componente)

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
4
Introduce el primer vector (3 valores, con más el producto vectorial no tiene sentido)
2 3 4
Introduce el segundo vector (3 valores, con más el producto vectorial no tiene sentido)
-4 8.6 1
Vector resultado: -31.400002 -18.000000 29.200001
```

3.5. Producto mixto

Es una combinación del producto escalar y el vectorial, para ello necesitaremos 3 vectores de dimensión 3 máximo. Se calcula el producto vectorial de los dos últimos vectores y luego se hace el producto escalar del primer vector con el vector antes calculado.

```
float *
producto_mixto_1_svc(t_vector arg1, t_vector arg2, t_vector arg3,  struct svc_req *rqstp)
{
    static float  result;
    t_vector vector_intermedio;
    int tam = 3;
    float r;

    vector_intermedio.t_vector_val = (float *) malloc(tam*sizeof(float));
    vector_intermedio.t_vector_len = tam;

    vector_intermedio.t_vector_val[0] = arg2.t_vector_val[1]*arg3.t_vector_val[2] - arg2.t_vector_val[2]*arg3.t_vector_val[1];
    vector_intermedio.t_vector_val[1] = arg2.t_vector_val[2]*arg3.t_vector_val[0] - arg2.t_vector_val[0]*arg3.t_vector_val[2];
    vector_intermedio.t_vector_val[2] = arg2.t_vector_val[0]*arg3.t_vector_val[1] - arg2.t_vector_val[1]*arg3.t_vector_val[0];

    for (int i=0; i<tam; i++){
        r += arg1.t_vector_val[i]*vector_intermedio.t_vector_val[i];
    }

    result = r;

    return &result;
}
```

El diálogo es igual al del producto vectorial, pero ahora se piden 3 vectores:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
5
Introduce el primer vector (3 valores, con más el producto mixto no tiene sentido)
1 2 3
Introduce el segundo vector (3 valores, con más el producto mixto no tiene sentido)
-2 4 6
Introduce el tercer vector (3 valores, con más el producto mixto no tiene sentido)
8 11 1
Resultado: -124.000000
```

3.6. Producto de un escalar por un vector

Simplemente se multiplican todas las coordenadas del vector por dicho escalar.

```c
t_vector *
escalar_por_vector_1_svc(float arg1, t_vector arg2,  struct svc_req *rqstp)
{
    static t_vector  result;

    result.t_vector_val = (float *) malloc(arg2.t_vector_len);
    result.t_vector_len = arg2.t_vector_len;

    for (int i=0; i<arg2.t_vector_len; i++){
        result.t_vector_val[i] = arg2.t_vector_val[i]*arg1;
    }

    return &result;
}
```

Primero se introducirá la dimensión del vector, luego las componentes de este y por último el escalar.

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
2
Producto escalar: 1
Suma: 2
Resta: 3
Producto vectorial: 4
Producto mixto: 5
Escalar por vector: 6
6
Introduce la dimensión del vector
4
Introduce el vector
1 2 3 4
Introduce el escalar
3
Vector resultado: 3.000000 6.000000 9.000000 12.000000
```

4. Media

Esta sección solo contiene una operación, que es realizar la media de un conjunto de reales. Para ello, metemos todos los valores que nos proporciona el usuario en un vector para facilitar el trabajo. He aquí la implementación:

```
float *
media_1_svc(t_vector arg1,  struct svc_req *rqstp)
{
    static float  result;
    float r = 0;

    for (int i=0; i<arg1.t_vector_len; i++){
        r += arg1.t_vector_val[i];
    }

    r /= arg1.t_vector_len;

    result = r;

    return &result;
}
```

Al seleccionar esta opción primero se nos preguntará por el número de valores

y luego por los susodichos valores

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
3
¿Cuántos valores vas a introducir? 5
Introduce los valores
1 2 3 4 5
El resultado es 3.000000
```

5. Operaciones con potencias

Al entrar en esta opción se nos mostrará un pequeño menú con las operaciones a realizar:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
4
1: Potencia básica
2: Multiplicación
3: División
```

Para esta categoría he añadido el struct potencia, que está formado por dos float: base y exponente. Esto facilita el trabajo de base y exponente por separado.

5.1. Potencia básica

Resuelve una potencia, es decir, eleva la base al exponente, para ello se ha usado pow de math.h.

```c
float *
potencia_1_svc(potencia arg1,  struct svc_req *rqstp)
{
    static float  result = 0;

    result = pow(arg1.base,arg1.exponente);

    return &result;
}
```

El problema que se ha presentado en esta operación era que el compilador no reconocía pow, para solucionar esto simplemente había que añadir -lm al campo LDLIBS del makefile. He aquí un ejemplo de ejecución:

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
4
1: Potencia básica
2: Multiplicación
3: División
1
Introduce la base: 2
Introduce el exponente: 5
El valor de la potencia es: 32.000000
```

5.2. Multiplicación de potencias

Es necesario para esta función que las potencias tengan la misma base (por ello solo se pide una). Simplemente hay que sumar los exponentes.

```
potencia *
multiplicar_potencias_1_svc(potencia arg1, potencia arg2,  struct svc_req *rqstp)
{
    static potencia  result;

    result.base = arg1.base;
    result.exponente = arg1.exponente+arg2.exponente;

    return &result;
}
```

Se dará el resultado en forma de baseˆexponente y en forma de número real.

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
4
1: Potencia básica
2: Multiplicación
3: División
2
Introduce la base de las potencias (solo una, ya que debe ser la misma): 3
Introduce el exponente de la primera potencia: 2
Introduce el exponente de la segunda potencia: 1
La potencia resultante es 3.000000^3.000000 y su valor es 27.000000
```

5.3. División de potencias

Igual que la multiplicación pero restamos los exponentes en vez de sumarlos.

```c
potencia *
dividir_potencias_1_svc(potencia arg1, potencia arg2,  struct svc_req *rqstp)
{
    static potencia  result;

    result.base = arg1.base;
    result.exponente = arg1.exponente-arg2.exponente;

    return &result;
}
```

```
1: Operación básica
2: Operación con vectores
3: Media
4: Operación con potencias
4
1: Potencia básica
2: Multiplicación
3: División
3
Introduce la base de las potencias (solo una, ya que debe ser la misma): 2
Introduce el exponente de la primera potencia: 1
Introduce el exponente de la segunda potencia: 2
La potencia resultante es 2.000000^-1.000000 y su valor es 0.500000
```