Utilicemos nuestras propias funciones

Hace algunos días estábamos hablando de la modularización. Dijimos que cuando un problema es lo suficientemente complejo como para separarlo en partes, lo que se hace es modularizarlo y que la suma de las soluciones de las partes equivale a la solución del problema original. Esto lo conocíamos como "divide y vencerás".

También se les hacía notar que siempre hemos estado acudiendo a la utilización de módulos para resolver nuestros problemas, ya sea en el pasado o ahora que sabemos programar. Lo que pasa es que lo hacemos de manera tan natural que no nos habíamos percatado de ello. Por ejemplo, cada vez que resolvemos una expresión como *5 + **cos(**x**)** + 2*x*, u otra como *w + **pow(**a**, b**)*, estamos utilizando módulos. En el primer ejemplo, para resolver la expresión completa necesitamos de invocar la función coseno, la cual resolvemos por aparte y luego la sustituimos por el valor que hemos resuelto, para terminar de resolver la expresión. En el segundo caso hacemos lo mismo con la función potencia.

Lo que sí debemos resaltar es que esas funciones ya están pre elaboradas y solo las hemos usado, o invocado. Pero, de ahora en adelante, también las construiremos dentro de nuestros programas.

Un primer problema a resolver

Un problema típico para aprender a implementar funciones es el combinatorio de dos números, que se resuelve mediante la fórmula:

$$\binom{n}{k} = \frac{n!}{k!*(n-k)!}$$

Observemos que se requiere calcular tres factoriales, así que hay que aplicar el mismo procedimiento tres veces antes de calcular el valor definitivo.

Resolvamos primero el problema del factorial de un número, haciendo un programa sencillo:

El factorial de un número, $n!$, requiere de la multiplicación de todos los número naturales menores o iguales al número dado, pero es *1* en los casos particulares en que n es *0* o *1*:

$$n! = \begin{cases} 1, si \quad n = 0 \quad \vee \quad n = 1 \\ \prod_{k=1}^{n} k, \quad si \quad n > 1 \end{cases}$$

El programa es el siguiente:

```
#include "iostream"

using namespace std;

int main(void)
{
    int n, k, fact;

    cout << endl;
    cout << "CÁLCULO DEL FACTORIAL DE UN NÚMERO" << endl << endl;
    cout << "Digite el vaor de n: ";
    cin >> n;

    fact = 1;
    for(k = 1; k <= n; k++)
        fact = fact * k;
    cout << "*El factorial es: " << fact << endl;

    cout << endl;
    return 0;
}
```

Notar que la parte medular del problema consiste en un proceso iterativo cuyo contador toma, en cada ciclo, un valor que está involucrado en el producto y lo acumula en una variable acumulador.

Una versión del programa que resuelve el número combinatorio, pero sin modularizar, sería la siguiente:

```
#include "iostream"

using namespace std;

int main(void)
{
    int n, k, i, fact1, fact2, fact3, comb;

    cout << endl;
    cout << "CÁLCULO DEL COMBINATORIO DE N Y K" << endl << endl;
    cout << "Digite el vaor de n: ";
    cin >> n;
    cout << "Digite el vaor de k: ";
    cin >> k;

    // Cálculo del factorial de n:
    fact1 = 1;
    for(i = 1; i <= n; i++)
        fact1 = fact1 * i;

    // Cálculo del factorial de k:
    fact2 = 1;
    for(i = 1; i <= k; i++)
        fact2 = fact2 * i;

    // Cálculo del factorial de n-k:
    fact3 = 1;
    for(i = 1; i <= n-k; i++)
```

```
        fact3 = fact3 * i;

    // Operación que calcula el número combinatorio:
    comb = fact1 / (fact2 * fact3);

    cout << "El número combinatorio es: " << comb << endl;

    cout << endl;
    return 0;
}
```

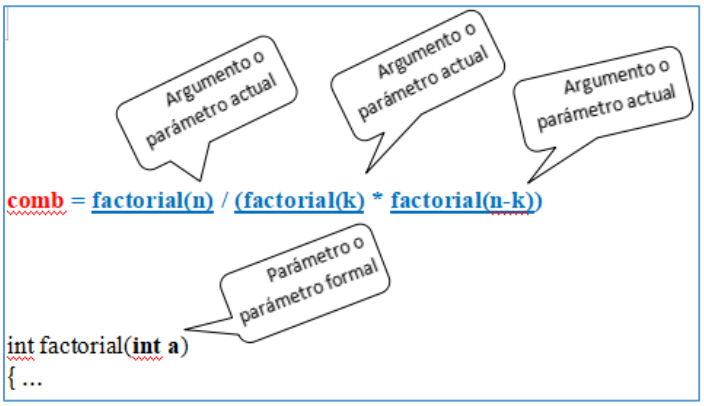
Podemos notar que este programa realiza exactamente las mismas operaciones para calcular tres factoriales. Así que, en lugar de escribir exactamente el mismo procedimiento tres veces, lo podemos escribir solo una vez e invocarlo las veces que consideremos necesario. Así que separaremos el cálculo del factorial como una función aparte y en el programa principal la invocaremos cuando sea necesario, enviándole en cada vez el parámetro correspondiente.

```
#include "iostream"

using namespace std;
int factorial(int);

int main(void)
{
int n, k, comb;

    cout << endl;
    cout << "CÁLCULO DEL COMBINATORIO DE N Y K" << endl << endl;
    cout << "Digite el valor de n: ";
    cin >> n;
    cout << "Digite el valor de k: ";
    cin >> k;

    // Operación que calcula el número combinatorio:
    comb = factorial(n) / (factorial(k) * factorial(n-k));

    cout << "El número combinatorio es: " << comb << endl;

    cout << endl;
    return 0;
}

// Función que calcula el factorial de un número:
int factorial(int a)
{
    int fact, i;

    fact = 1;
    for(i = 1; i <= a; i++)
        fact = fact * i;

    return fact;
}
```

Puede notarse que la función requiere de un valor que se le proporciona como entrada, a través del la variable que tiene declarada dentro de los paréntesis del

encabezado. A esta se le llama parámetro. La cantidad de parámetros depende de cuántos valores de entrada requiere para hacer su trabajo. Los parámetros son variables locales de una función, es decir, su ámbito solo es el par de llaves del cuerpo de la función y fuera de él es como si no existieran. De hecho, las variables locales de una función solo existen en el preciso instante que la función se comienza a ejecutar, porque hasta ese momento es alcanzada la instrucción de su declaración. Las variables locales son destruidas una vez la función termina. Una variable local puede tener el mismo nombre que otra variable local que se encuentra en otra función y no habrá interferencia entre ambas, pues son dos variables completamente independientes pero que casualmente tienen el mismo nombre.

En la invocación de la función, los valores que se le mandan se conocen como argumentos. La cantidad y el tipo de los argumentos deben coincidir con la cantidad y tipos de parámetros declarados en el encabezado de la función. Si como argumento se coloca una expresión, el tipo del valor resultante de evaluar esta expresión debe coincidir con el parámetro que recibirá dicho valor.



Otra cosa que notar es la instrucción *return* que se encuentra al final de la función. Esta instrucción es la que hace que la función pueda devolver el valor calculado al punto donde se realizó la invocación de la función. La instrucción *return* no necesariamente debe ser la última del cuerpo de una función, ni encontrarse solo una vez. Cuando se encuentra una instrucción *return*, la función se abandona y el valor colocado a su derecha es entregado.

Hay una cuestión más que es importante notar: las funciones que no son la función *main*, se declaran arriba del programa. La forma de declararlas es colocar el encabezado de la función, finalizándolo con punto y coma. Dentro de los paréntesis, solo necesitamos poner los tipos, no los nombres de los parámetros. Si la función requiere de más de un parámetro, se colocarían los tipos de cada uno, separados por comas. La declaración de una función no debe extrañarnos ya que con las funciones predefinidas pasa algo similar: debemos declarar su librería en el encabezado del programa. A este tipo de declaración de nuestras funciones se le conoce como *prototipo de la función* o *declaración forward*.

Otro problema para explorar la interacción con funciones

Hagamos otro ejemplo para manejar funciones que devuelven valores. Esta vez le pediremos un valor al usuario y luego le preguntaremos, por medio de un menú, si desea duplicarlo, triplicarlo cuadruplicarlo o quintuplicarlo. El menú será cíclico y residirá en la función *main* . Las operaciones estarán, cada una, en una función por separado.

```
#include "iostream"

using namespace std;
int duplicar(int);
int triplicar(int);
int cuadruplicar(int);
int quintuplicar(int);

int main(void)
{
    int opcion, n;
    bool repetir = true;

    do{
        cout << endl;
        cout << "MULTIPLICAR UN NÚMERO" << endl << endl;
        cout << "Digite un número entero: ";
        cin >> n;
        cout << endl;
        cout << "Las opciones para multiplicar son:"<< endl;
        cout << "1) Duplicar el valor." << endl;
        cout << "2) Triplicar el valor." << endl;
        cout << "3) Cuadruplicar el valor." << endl;
        cout << "4) Quintuplicar el valor." << endl;
        cout << "5) Salir" << endl;
        cout << "Elija su opción: ";
        cin >> opcion;
        cout << endl;
        switch(opcion)
        {
            case 1:
                cout << "El doble es: " << duplicar(n) << endl << endl;
                break;
            case 2:
                cout << "El triple es: " << triplicar(n) << endl << endl;
                break;
            case 3:
                cout << "El cuádruple es: " << cuadruplicar(n) << endl << endl;
                break;
            case 4:
                cout << "El quíntuple es: " << quintuplicar(n) << endl << endl;
                break;
            case 5:
                repetir = false;
                break;
            default:
                cout << "Opción de menú no válida" << endl << endl;;
        }
    }while(repetir);
}

int duplicar(int n)
{
    return 2 * n;
}

int triplicar(int n)
{
    return 3 * n;
```

```
}

int cuadruplicar(int n)
{
    return 4 * n;
}

int quintuplicar(int n)
{
    return 5 * n;
}
```

Notar que la acción del programa se repite por medio de un lazo que revisa el valor de una variable booleana. El menú desplegable y la instrucción de ramificación múltiple están ambos dentro de la instrucción cíclica. Cada una de las ramas del *switch* lleva a la ejecución de una de las funciones que operan al valor dado por el usuario. Cada función recibe un valor y retorna un resultado. Notemos también que cada función ha sido declarada al inicio del programa.

Ejemplos:

1) La raíz de un número se puede calcular por medio de la aplicación repetida de la fórmula:

$$r_{sig} = \frac{\frac{n}{r} + r}{2}$$

Donde, un primer valor de r, puede asumirse de manera arbitraria. Escriba un programa que calcule la raíz de un número con, al menos, cuatro cifras decimales exactas. Solicite el valor del número en la función main y luego invoque a una función que calculará su raíz y devolverá su resultado a la función main para ser mostrado en pantalla.

```
#include <iostream>

using namespace std;

float rcuad(float);

int main(void)
{
    float n;

    cout << endl;
    cout << "RAIZ CUADRADA DE UN NÚMERO" << endl << endl;

    cout << "Digite el valor de n: ";
    cin >> n;

    cout << "La raíz es: " << rcuad(n) << endl;

    cout << endl;
    return 0;
}

float rcuad(float n)
{
    float r = 10, rsig, deltax;

    do{
        rsig = (n/r + r)/2;
```

```
        deltax = abs(rsig - r);
        if(deltax >= 0.0001)
            r = rsig;
    }while(deltax >= 0.0001);

    return rsig;
}
```

2) El seno de un x, dado en radianes, se puede calcular por medio de la serie:

$$sen(x) = \sum_{k=0}^{\infty} \frac{(-1)^k * x^{2*k+1}}{(2*k+1)!}$$

Note que es una sumatoria de infinitos términos, pero, en la práctica, con unos cuantos términos se obtiene una muy buena aproximación. Escriba un programa que solicite el valor de x y la cantidad de términos a incluir en la sumatoria, luego invoque una función que calcule una aproximación de sen(x). Esta función deberá retornar el valor calculado. Ojo: no debe utilizar la función seno de la librería cmath.

```
#include <iostream>
#include <cmath>

using namespace std;

float sen(float, int);
int factorial(int);

int main(void)
{
    float x, senx;
    int n;

    cout << endl;
    cout << "SENO DE X (EN RADIANES)" << endl << endl;

    cout << "Digite el valor de x (en radianes): ";
    cin >> x;
    cout << "¿Cuántos términos de la serie desea incluir?: ";
    cin >> n;

    cout << "sen(" << x << ") = " << sen(x, n) << endl;

    cout << endl;
    return 0;
}

float sen(float x, int n)
{
    int k;
    float senx = 0;
    for(k = 0; k < n; k++)
        senx = senx + pow(-1, k) * pow(x, 2*k+1) / factorial(2*k+1);

    return senx;
}

int factorial(int n)
{
    int fact, i;
```

```
    if(n == 0 || n == 1)
        fact = 1;
    else{
        fact = 1;
        for(i = 1; i <= n; i++)
            fact = fact * i;
    }

    return fact;
}
```

3) Determinar si un número es primo.

Versión 1: sin funciones:

```
#include <iostream>
#include <cmath>

using namespace std;

int main(void)
{
    int n, i, limSup;
    bool esPrimo = true;

    cout << endl;
    cout << "DETERMINAR SI UN NÚMERO ES PRIMO" << endl << endl;

    cout << "Digite un número entero: ";
    cin >> n;

    limSup = sqrt(n);

    i = 2;
    while(i <= limSup && esPrimo == true){
        if(n % i == 0)
            esPrimo = false;
        i = i + 1;
    }
    if(esPrimo == true)
        cout << "El número es primo" << endl;
    else
        cout << "El número NO es primo" << endl;

    cout << endl;
    return 0;
}
```

Versión 2: con función:

```
#include <iostream>
#include <cmath>

using namespace std;

bool primo(int);
```

```
int main(void)
{
    int n;

    cout << endl;
    cout << "DETERMINAR SI UN NÚMERO ES PRIMO" << endl << endl;

    cout << "Digite un número entero: ";
    cin >> n;

    if(primo(n) == true)
        cout << "El número es primo" << endl;
    else
        cout << "El número NO es primo" << endl;

    cout << endl;
    return 0;
}

bool primo(int n)
{
    int i, limSup;
    bool esPrimo = true;
    limSup = sqrt(n);

    i = 2;
    while(i <= limSup && esPrimo == true){
        if(n % i == 0)
            esPrimo = false;
        i = i + 1;
    }

    return esPrimo;
}
```