Subprogramas - Revisión 2022

Los primeros problemas que se modelaron utilizando una computadora eran problemas matemáticos, bastante cortos, donde a lo sumo se debía resolver utilizando un método numérico iterativo.

Este tipo de problemas, podrían encararse utilizando lenguajes de bajo nivel, sea binario o assembler. Sin embargo, a medida que las computadoras y los lenguajes fueron evolucionando, se modelaban problemas cada vez más complejos. Naturalmente, nadie pensaría en hacer un sistema bancario o de un hospital en assembler, pero la historia es distinta al tener lenguajes como C, C++ o Pascal o cualquier otro de alto nivel.

El problema es que a medida que aumenta la envergadura del modelo que se intenta plasmar en un sistema computacional, aparecen ciertas dificultades:

- Pruebas del sistema:
  Las pruebas de un software de mediana envergadura pueden no ser tan sencillas. Aun identificando un error, a veces, puede ser difícil identificar la línea que posee la falla que lo genera.
  Pensemos el siguiente caso. ¿Una fábrica de automóviles verifica cada componente del vehículo al salir de la línea de ensamblado? O, más bien, se realiza un control de calidad de cada componente por separado y luego se ensamblan para producir un automóvil, realizando finalmente una prueba integral de la unidad terminada? De esta forma se debe pensar cuando se fabrica una solución informática, cada parte debe tener un control, debe ser probada antes de ser ensamblada.

- División de trabajo:
  Si se debe confeccionar un modelo, un algoritmo de cientos de miles de líneas de código, ¿cómo hacer para dividir el trabajo? Seguramente no sea una única persona la encargada de escribir el código.
  ¿Cuántas personas trabajan en una fábrica de automóviles? ¿Están todas juntas trabajando sobre la misma parte de un automóvil?

- Mantenimiento:
  Si mi modelo consta de, como hemos dicho, cientos de miles de líneas de código. Ante la necesidad de un cambio, ¿Cómo se puede saber exactamente qué parte se debe actualizar, corregir o reemplazar?

- Reutilización:
  No sería conveniente que si se dedican horas y recursos al desarrollo de un algoritmo se pueda reutilizar ese trabajo en futuras oportunidades?

Para todo esto se ideó el diseño ***descendente o top down.*** El mismo consiste en dividir el problema grande en subproblemas o módulos. Cada módulo puede descomponerse en submódulos. De esta manera un programa podría tener la siguiente estructura:

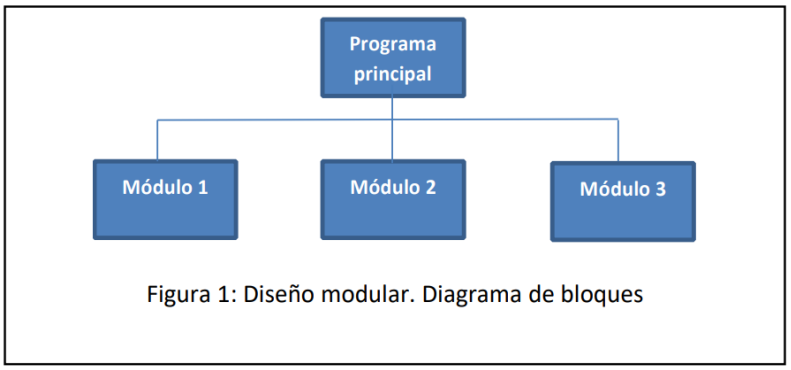

Figura 1: Diseño modular. Diagrama de bloques

Cada módulo resuelve una parte del programa principal. Este modelo está simplificado para fines didácticos. La descomposición del problema debe ser de los niveles que sean requeridos y cada rama puede tener distinta cantidad de subniveles.

Un caso muy común es un programa que consiste en un menú y varios subprogramas que son invocados según la opción elegida por el usuario en el menú. Imaginemos un programa que carga un cierto tipo de dato, supongamos alumnos de la facultad, teniendo la posibilidad de mostrarlos con algún filtro por pantalla. Además este sistema tiene también un módulo para hacer reportes. En este caso un posible diagrama podría ser:

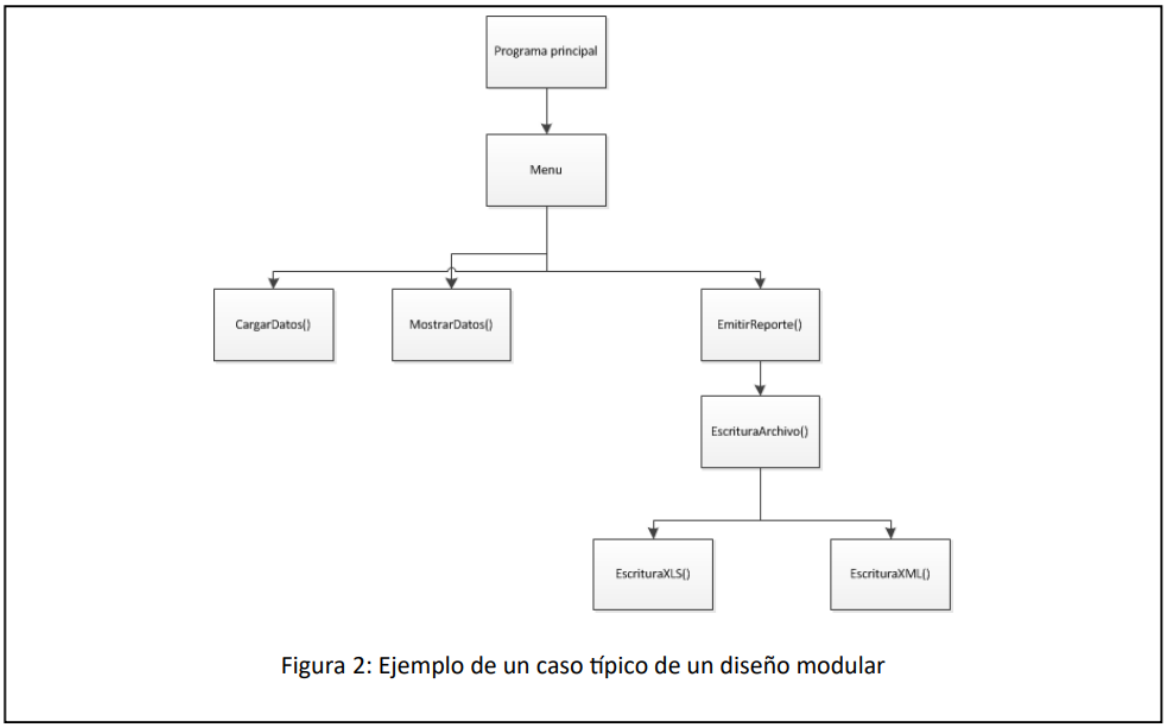

Figura 2: Ejemplo de un caso típico de un diseño modular

Definición de subprogramas en C++

Las funciones son subprogramas que tienen un tipo asignado. Devuelven un valor a través de su invocación.

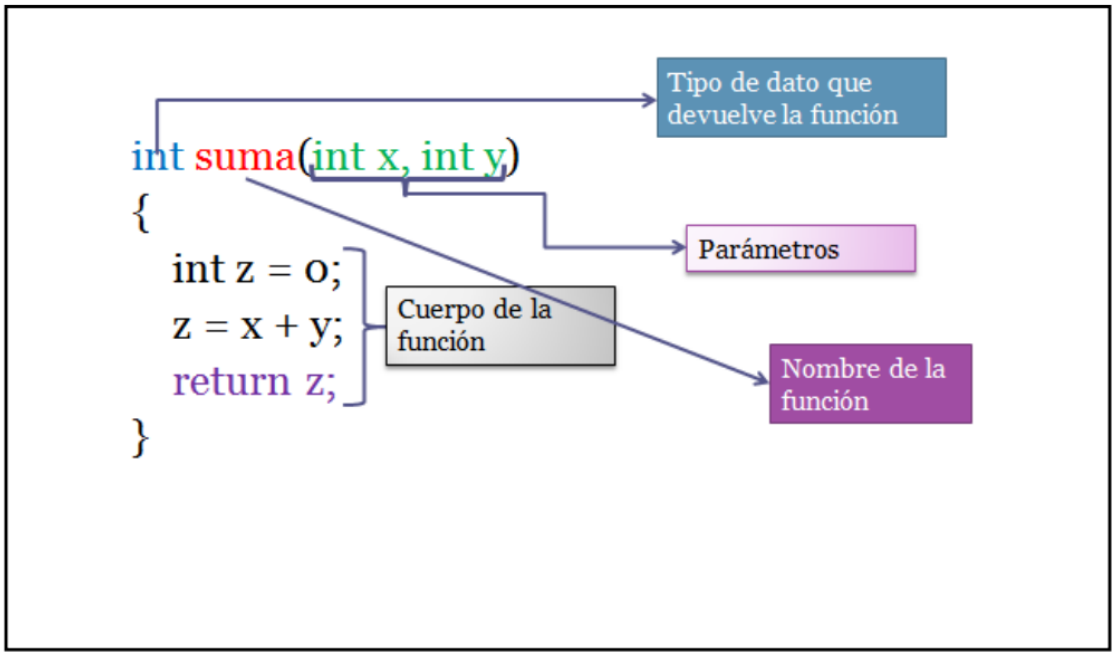

Figura 3: Función suma

Los procedimientos no devuelven ningún valor a través de su invocación, esto se indica con la palabra “void”:

```
void suma(int x, int y)
{
   int z = 0;
   z = x + y;
   cout  << z << endl;
   return;
}
```

Figura 4: Suma implementado como procedimiento

Invocación de subprogramas

Bien, hasta ahora hemos visto la definición de funciones y procedimientos. La cuestión pendiente es cómo hacer que se ejecute, formalmente cómo invocarla. Por ejemplo, desde el main podríamos llamar a la función suma como se muestra en la línea resaltada en rojo del siguiente código:

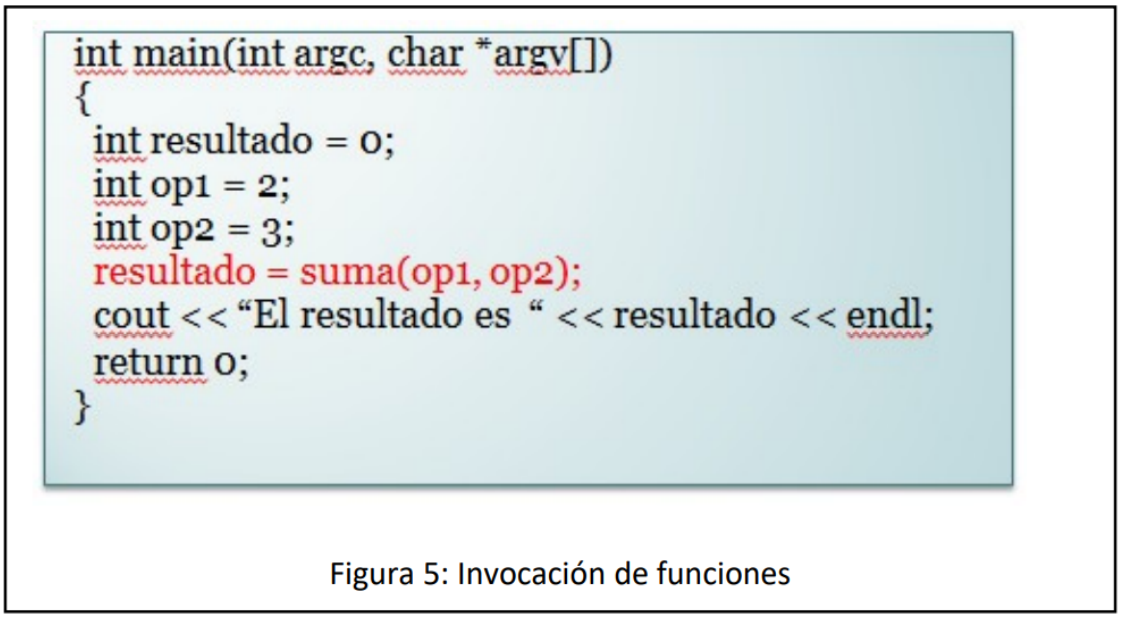

Figura 5: Invocación de funciones

En la invocación se realiza una asignación del valor devuelto por la función a una variable del mismo tipo. Esto es sólo posible con funciones, no con procedimientos ya que estos no devuelven ningún valor. La asignación del valor de una función también puede ser externa de salida, es decir que la sentencia "*cout << nombrefuncion(parámetros);*" es válida.

Op1 y op2 son los parámetros actuales de la función, osea las variables que son pasadas por parámetro desde la invocación. Mirando la definición de la función en la figura 3 vemos que x e y son los parámetros formales.

Ámbito de las variables (scope)

Otra clasificación a tener en cuenta es el ámbito de las variables. Las variables declaradas fuera de cualquier subprograma son variables globales y su uso es universal, pueden ser utilizadas desde cualquier lugar del programa. No obstante, tener en cuenta que en esta materia está prohibido el uso de variables globales dentro de un subprograma. Esto es así porque en realidad, utilizar una variable global en un subprograma rompe su independencia con su contexto y el valor devuelto puede variar según el estado de una variable externa.

Debemos tener en cuenta que cualquier variable global ocupa espacio en memoria durante toda la ejecución del programa.

Las variables declaradas en los subprogramas, ya sean los parámetros formales o variables declaradas internamente, son variables locales del subprograma. Sólo pueden ser utilizadas dentro del mismo y sólo ocupan espacio en memoria durante la ejecución del subprograma.

Pasaje de parámetros por valor vs pasaje de parámetros por referencia

Cuando una variable se pasa **por valor** el parámetro actual tendrá una copia del valor del parámetro formal, por lo tanto los cambios realizados sobre el parámetro formal no van a ser reflejados sobre el parámetro actual, ya que se trabaja en otra dirección de memoria.

```
void suma1valor(int x)
{
    x = x + 1;
    return;
}
```

Figura 6: pasaje por valor

Cuando una variable se pasa por referencia el parámetro formal tendrá una referencia a la dirección de memoria (puntero) donde está alojada la variable del parámetro actual, por lo tanto los cambios realizados sobre el parámetro formal van a ser reflejados sobre el parámetro actual al terminarse el subprograma.

```
void suma1ref(int &x)
{
    x = x + 1;
    return;
}
```

Figura 7: pasaje por referencia

El siguiente código ejemplifica el resultado de distintas invocaciones, una por valor y la siguiente por referencia.

```
int main()
{
    int x=1;
    suma1valor(x);
    cout << x << endl;
    suma1ref(x);
    cout << x << endl;
    return 0;
}
```

Figura 8: prueba de pasaje por valor vs pasaje por referencia

Luego de la primera invocación, es decir, en el primer cout, el programa muestra por pantalla el valor 1, es decir que los cambios realizados al parámetro formal no son reflejados en el parámetro actual luego de la invocación del subproprograma.

En cambio, el segundo cout muestra el valor 2, ya que los cambios realizados sobre el parámetro formal x de suma1ref son reflejados en el parámetro actual.

Ejemplos:

1) Escriba una función que calcule la potencia de un número.

```cpp
#include <iostream>

using namespace std;


int potencia(int base, int exp)
{
    int result, i;
    result=1;
    for (i=1; i<= exp; i++)
    {
        result = result * base;
    }
    return result;
}


int main()
{
    int r, x, y;
    cout << "Ingrese base y exponente" << endl;
    cin >> x >> y;
    r = potencia(x,y);
    cout << "El resultado es " << r << endl;
    return 0;
}
```

2) Escriba un subprograma que calcule la intersección de dos rectas.

Implementación 1:

```
void interseccion(int m1, int b1, int m2, int b2)
{
    float xi, yi;
    if (m1!=m2)
    {
        xi = (b2 - b1) / (m1 - m2);
        yi = m1 * xi + b1;
        cout << "La interseccion es en el punto (" << xi << "; " << yi << ")." << endl;
    }
    else
    {
        if (b1==b2)
            cout << "Las rectas son iguales, la interseccion es toda la recta" << endl;
        else
            cout << "Las rectas son paralelas, no existe interseccion" << endl;
    }
    return;
}
```

La invocación del subprograma sería:

```
int main()
{
    int m1, b1, m2, b2;
    cout << "Ingrese ordenada al origen y pendiente de la recta 1" << endl;
    cin >> m1 >> b1;
    cout << "Ingrese ordenada al origen y pendiente de la recta 2" << endl;
    cin >> m2 >> b2;
    interseccion(m1,b1,m2,b2); //se invoca al procedimiento
}
```

Implementación 2:

```
int interseccion(int m1, int b1, int m2, int b2, float &xi, float &yi)
{
    int codigo;
    if (m1!=m2)
    {
        xi = (float) (b2 - b1) /  (float) (m1 - m2);
        yi = (float) (m1 * xi + b1);
        codigo = 0;

    }
    else
    {
        if (b1==b2)
            codigo = 1;
        else
            codigo = 2;
    }
    return codigo;
}
```

Esta implementación debería ser invocada de la siguiente manera:

```
int main()
{
    int m1, b1, m2, b2;
    float x, y;
    cout << "Ingrese ordenada al origen y pendiente de la recta 1" << endl;
    cin >> m1 >> b1;
    cout << "Ingrese ordenada al origen y pendiente de la recta 2" << endl;
    cin >> m2 >> b2;
    switch (interseccion(m1,b1,m2,b2,x,y))
    {
        case 0:
            cout << "El punto de intersección es: (" << x << "; " << y << ")." << endl;
        break;
        case 1:
            cout << "Las rectas son iguales, la intersección es toda la recta." << endl;
        break;
        case 2:
            cout << "No existe solución, las rectas son paralelas." << endl;
        break;
    }
    return 0;
}
```

Analizando ambas soluciones vemos que el subprograma de la segunda implementación es más independiente que el de la primera, ya que no se encarga de imprimir él mismo, sino que devuelve de alguna forma el resultado, dejando al programa que lo invoca que determine qué hacer con la información devuelta.

3) Escriba un programa que permita al usuario:
   - Leer tres números.
   - Obtener el promedio.
   - Obtener el mayor de ellos.
   - Salir del programa.

Planifiquemos un poco. Este problema podríamos dividirlo en las siguientes partes: necesitamos algo que muestre las opciones, una parte que lea tres números, otra que pueda calcular el promedio de tres números y finalmente otra que calcule el promedio de tres valores también.

Entonces podríamos realizar los siguientes módulos:

```
int mayorvalor(int a, int b, int c)
{
    int mayor;
    if (a > b && a > c)
    {
        mayor = a;
    }
    else
    {
        if (b>c)
            mayor = b;
        else
            mayor = c;

    }
    return mayor;
}
```

```
float promedio(int a, int b, int c)
{
    return (float) (a+b+c)/3;
}
```

```
void ingresar(int &v1, int &v2, int &v3)
{
    cout << "Ingrese el valor 1:" << endl;
    cin >> v1;
    cout << "Ingrese el valor 2:" << endl;
    cin >> v2;
    cout << "Ingrese el valor 3:" << endl;
    cin >> v3;
    return;
}
```

```
void mostrarmenu()
{
    cout << "Ingrese una opción:" << endl;
    cout << "a - Ingresar valores" << endl;
    cout << "b - Calcular promedio" << endl;
    cout << "c - Calcular mayor" << endl;
    cout << "Esc - Salir" << endl;
}
```

Y Finalmente los ensamblamos en un módulo principal:

```
int main()
{
    char opcion;
    int x,y,z;
    do
    {

        mostrarmenu();
        opcion=getch();
        switch (opcion)
        {
            case 'a':
            case 'A':
                    ingresar(x,y,z);
                break;
            case 'b':
            case 'B':
                    cout << "El promedio es: " << promedio(x,y,z) << endl;
                break;
            case 'c':
            case 'C':
                    cout << "El mayor valor es:" << mayorvalor(x,y,z) << endl;
                break;
        }

    } while ((int)opcion != 27); //27 es el código ascii de la tecla Esc. Tengo que convertirlo a int para obtener su código en tabla
    return 0;
}
```