Funciones

La mayoría de los programas que prestan servicio en la vida real son complejas construcciones de cientos y cientos de líneas de código. Como toda tarea compleja, el desarrollo de software requiere de una estrategia para su abordaje: no es posible "pensarlo" ni codificarlo de una sola vez; es necesario dividirlo en partes, definir y desarrollar cada una de sus "piezas" y luego ensamblarlas en el orden que corresponda. Si bien esta estrategia, dada su importancia, será estudiada en otras materias de la carrera, debemos desde ahora tratar de pensar los programas como un conjunto de módulos que interactúan, en vez de pensarlos como líneas de código en secuencia.

La mayoría de los lenguajes de programación permiten la división de un programa en módulos. De acuerdo al lenguaje que se esté utilizando pueden ser denominados subprogramas, subrutinas, procedimientos, funciones, etc.. Si bien la definición formal de la teoría de los lenguajes de programación diferencia entre procedimiento (módulo que devuelve 0 ó más valores a través de parámetros) y función (módulo que devuelve sólo un valor), en C todos los módulos se denomina funciones, y sólo devuelven un valor.

La división en módulos tiene varios objetivos:
1) Facilita el diseño y la construcción de un sistema, al proveer un mecanismo de abstracción.
2) Permite el reuso de los componentes de software.
3) Facilita la lectura de un programa, y la búsqueda y solución de errores.
4) Mejora la eficiencia de los programas, en cuanto al uso de los recursos de hardware.

1) Abstracción

Como ya se ha visto en los cursos anteriores, una de las tareas más complejas en la construcción de un sistema es la comprensión clara del problema que se pretende resolver, esto es, definir cuales son los datos de entrada con que se dispone, cuales son las salidas esperadas, y cuales las relaciones entre los datos de entrada y la salida.

Una de las formas usuales para encarar la resolución de un problema es
a) Comprender la generalidad del problema.
b) Dividir el problema en partes más pequeñas, y analizarlas por separado.
c) Para cada una de las partes, definir los procesos necesarios para resolverlas.
d) Para cada proceso, detallar las tareas necesarias y la secuencia en la cual deben desarrollarse.

Esta es la forma en la cual, muchas veces hasta de manera inconciente, resolvemos los problemas que se nos presentan cotidianamente. Vamos desde lo más abstracto, más general, hacia los detalles. La posibilidad de abstraer lo general de la implementación nos permite abordar cuestiones complejas, analizarlas, y resolverlas.

Por ejemplo, al decidir cual carrera universitaria seguir, no empezamos buscando los temas de cada una de las materias de todas las carreras universitarias existentes (detalles), sino que desde una definición general ("me interesa trabajar con computadoras"), fuimos viendo las universidades, las carreras, la duración de éstas, etc.

De esta misma manera debe encararse la implementación de un sistema. Y las funciones de C, nos permiten hacerlo, ya que podemos pensar en asociar cada parte o proceso con una o más funciones, y cada tarea con una o más líneas de código dentro de la función. Invocar una función dentro de otra función de jerarquía superior, nos permite olvidarnos de los detalles y concentrarnos en la resolución del problema más general: basta llamar a la función pasándole los argumentos necesarios y esperar que ésta nos devuelva el resultado.

2) Reusabilidad

Es común que dentro de un programa se repita de manera idéntica un proceso. Por ejemplo, puede ser que en más de una ocasión haya que buscar el máximo dentro de un vector, o comprobar si un dato pertenece o no a un conjunto de valores almacenados en memoria o en un archivo. Si no se dispusiera de funciones, ¡debería volverse a tipear exactamente el mismo código una y otra vez!.

Es probable también que una función definida para un programa en particular, pueda ser de utilidad en otro.

La reusabilidad es una de las características que se espera de un lenguaje de programación, ya que hace más sencilla la tarea del programador, y reduce los tiempos de desarrollo. Por esta razón, C dispone de un conjunto importante de funciones en su librería, que evitan que cada programador deba escribir sus propias funciones para operaciones elementales tales como ingresar un dato desde el teclado, o imprimir un resultado en la pantalla.

3) Lectura y corrección de errores

Imagínese que deba dársele mantenimiento, o corregir algún detalle de un programa con cientos, o miles de líneas de código escritas una tras otra sin ninguna separación, ni ningún comentario sobre lo que cada línea hace. Sería un trabajo extremadamente dificultoso.

En cambio, si el programa está dividido en funciones, es más sencilla la lectura del archivo, y por lo tanto cualquier tarea que deba realizarse sobre él.
Veamos un ejemplo sencillo:

```
#include<iostream>

using namespace std;

int main()
	{
	int v[5],n[5],s[5], i;
	system(“cls”);
	for(i=0;i<5;i++)
		v[i]=0;
	for(i=0;i<5;i++)
		n[i]=0;
	for(i=0;i<5;i++)
		cout<<v[i];
	system(“pause”);
	system(“cls”);
	for(i=0;i<5;i++)
		{
		cout<<"\nIngrese un número entero: ";
		cin><v[i];
		}
	for(i=0;i<5;i++)
			cout<<v[i]<<endl;
	system(“pause”);
	system(“cls”);
	for(i=0;i<5;i++)
		{
		cout<<”Ingrese un número entero: ";
		cin>>a[i];
		}
	for(i=0;i<5;i++)
```

```
                cout<< n[i]<<endl;
        system(“pause”);
        system(“cls”);
        for(i=0;i<5;i++)
                s[i]=v[i]+n[i];
        for(i=0;i<5;i++)
                cout<<s[i]<<endl;
        system(“pause”);
        return 0;
        }


#include<iostream>

using namespace std;

void ponerCeroVector(int *, int);
void cargarVector(int *, int);
void mostrarVector(int *, int);
void sumarVectores(int *,int *,int *, int);

int main()
        {
        int v[5],n[5],s[5];
        int e=5;
        system(“cls”);
        ponerCeroVector(v,e);
        ponerCerovector(n,e);
        mostrarVector(v,e);
        mostrarVector(n,e);
        cargarVector(v,e);
        mostrarVector(v,e);
        cargarVector(n,e);
        mostrarVector(n, e);
        sumarVectores(v,n,s,5);
        mostrarVectir(s,5);
        system(“pause”);
        return 0;
        }

void ponerCeroVector(int *a, int x)
        {
        int i;
        for(i=0;i<x;i++)
                a[i]=0;
        }

void cargarVector(int *a,int x)
        {
        int i;
        for(i=0;i<x;i++)
                {
                cout<<Ingrese un número entero: ";
                cin>>a[i];
```

```
                }
        }


void mostrarVector(int *a, int x)
        {
        int i;
        for(i=0;i<x;i++)
                cout<<a[i]endl;
        system("pause");
        system("cls");
        }

void sumarVectores(int *a, int *b, int *c, int x)
        {
        int i;
                for(i=0;i<x;i++)
                        c[i]=a[i]+b[i];
        }
```

Ambos programas hacen exactamente lo mismo. Ponen en cero vectores, lo muestran, lo cargan con valores que se ingresan por teclado, lo vuelven a mostrar, suman los dos vectores y muestran el vector resultante.

¿Cuál de los 2 programas es más sencillo de comprender, y eventualmente de modificar o comprobar un error no trivial, es decir un error que no sea detectado por el compilador?.

Como ejercicio, se pide que se modifiquen ambos programas para poder usar vectores de 10 elementos, y se tome nota de la cantidad de correcciones que se debió hacer en uno y otro programa.

4) En relación a este punto, podemos analizar que sucede con la cantidad de variables utilizadas. En el caso de no utilizar funciones, todas las variables deben ser declaradas en main(), y por lo tanto reservarse memoria para cada una de ellas, independientemente si se usan sólo una vez, en una sola línea, o si se utilizan en todo el programa. Al utilizar funciones, y declarar la variables locales en ellas, como las variables sólo van a ocupar espacio mientras la función esté en uso – y nunca se usan todas las funciones a la vez-, el compilador determina un espacio total de memoria menor que la suma de todas las variables involucradas.

En contra del uso de funciones podría decirse que las llamadas que realiza un programa construido a partir de ellas, consume más recursos que uno monolítico (así se suele denominar a los programas largos sin funciones o con escasas llamadas). No obstante esto, las ventajas señaladas justifican su estudio y su uso.

Programa C

Un programa correctamente codificado en C, cuenta con una función principal (main()), generalmente de pocas líneas, y un conjunto de funciones.

Las funciones pueden ser internas, contenidas en el mismo archivo fuente del programa, o externas, es decir, contenidas en otro archivo. También pueden ser funciones de librería generales (de la librería estandar de C u otras librerías existentes), o definidas por el programador. En este último caso están las funciones escritas específicamente para ese programa, y puede haber otras funciones de librería escritas para otros programas, que son de utilidad. Las funciones definidas por

el programador pueden ser internas o externas. En el transcurso de las clases se explicará como hacer una librería propia e incluirla en un programa.

Para poder usar las funciones de la librería de C, debe incluirse al inicio del archivo fuente una referencia al archivo de cabecera que la contiene (ejemplo: #include <stdio.h>)

El programa principal (main()) se relaciona con las funciones llamándolas, y si es necesario pasándoles los parámetros (datos) que éstas necesitan. La llamada es una línea de código con el nombre de la función y entre paréntesis los parámetros. Si la función devuelve un valor, éste debe ser asignado a una variable en la misma línea de la llamada. Al llamar a la función la ejecución del programa continúa dentro del cuerpo (el código) de la función llamada. Cuando esta termina, devuelve el control a main(), es decir, se ejecuta la línea de código inmediata siguiente.

En C, cualquier función puede llamar a otra función; incluso una función puede llamarse a sí misma (esto se llama recursividad y es muy útil en algunos casos). El funcionamiento es igual que el mencionado anteriormente: main() llama a una función f1(), la ejecución continúa dentro de f1(). Si f1() llama a otra, f2(), se ejecuta el código de f2(). Cuando termina f2() el programa vuelve a f1(), y cuando termina f1(), el control vuelve a main().

Nota: se puede terminar la ejecución de cualquier función, y en cualquier parte de ella con la sentencia return.

Prototipo de funciones
Para poder utilizar una función no incluída en ninguna librería, es obligatorio escribir el prototipo de la función. Al leer el prototipo, el compilador sabe que cuando se haga referencia a ese nombre de función, debe ejecutar el código asociado a ese nombre.

Un prototipo de función tiene el siguiente formato:

valor_devuelto nombre_funcion(tipo_argumento1, tipo_argumento_2,....tipo_argumento_n)

El compilador necesita saber:
- el nombre de la función
- el tipo de dato que devuelve. Si no devuelve nada hay que poner void
- el tipo de dato de los argumentos, o sea, los tipos y la cantidad de datos que recibirá la función. Si no tiene argumentos, se pone void

Tomando como ejemplo una función del programa expuesto más arriba:

void ponerCero(int *, int)
- El nombre de la función es ponerCero
- Como la función no devuelve valor, el tipo de dato devuelto es void.
- Los tipos de datos que la función recibirá son un puntero a entero, y un entero.

Nota: si se omite el valor devuelto, por defecto el compilador entenderá que sea función devolverá un valor entero.

Semántica de las funciones
Si volvemos al ejemplo descripto, veremos que las funciones propias se mencionan de manera diferente en 3 lugares diferentes:
- Al inicio del código, después de los archivos de cabecera, se colocan los prototipos.
- Dentro de main(), cuando se las llama.
- Fuera de main().

En los 3 casos la forma de referirse a la función es diferente, y por lo tanto deben escribirse de manera distinta, siguiendo las reglas que el lenguaje impone.

Ya se explicó el por qué, y la forma de escritura de los prototipos.

Para el caso de las llamadas, sólo hay que poner el nombre de la función y los parámetros que recibirá. En el caso que devuelva un valor, éste debe ser asignado a una variable del mismo tipo que el valor devuelto por la función. Por ejemplo, si tenemos una función que se llama suma, que devuelve un valor entero, y acepta como parámetros 2 valores enteros, la llamada en main() sería:

```
int main()
        {
        int a, b, c;
        ----
        ----
        a=suma(b,c);
        -----
        -----
        return 0;
        }
```

También podría ponerse
a=suma(3,5);

ya que la función espera recibir 2 valores de tipo entero, y no le interesa si se trata de dos variables a las que se le ha asignado valores previamente, o directamente 2 números.

En la llamada no hay que poner el tipo de dato devuelto, ni especificar cuales son los tipos de datos de los parámetros, porque se la está llamando, usando, no se la está declarando.

El valor devuelto por una función puede utilizarse directamente, sin necesidad de asignarlo a otra variable. Por ejemplo, si se quisiera mostrar el valor que devuelve la función suma() podría ponerse:

```
cout<<suma(b,c);
```

También podría utilizarse el valor devuelto en una proposición lógica:

```
if(suma(b,c)>15)
```

Otra posibilidad es que el valor devuelto por una función sea utilizado como argumento en una función:

```
cout<<suma(suma(5,6),suma(6,7));
```

En cualquier caso, lo que debe tenerse en cuenta es que la operación que se pretenda hacer sea una operación válida para el tipo de dato que la función devuelve.

Nota: una función puede devolver cualquier tipo de dato, incluyendo punteros.

Por último, fuera de main() se escribe la función completa. Para la función suma sería:

```
int suma(int x, int y)
        {
```

```
    int suma;
    suma=x+y;
    return suma;
    }
```

En la primera línea es obligatorio poner el tipo de dato que devuelve la función, y los tipos y nombres que tendrán las variables que almacenarán los valores enviados desde la función que llamó. Para el caso de suma, el valor que tenía b en main() se copiará en la variable x, y el de c se copiara en la variable local a la función suma y. En el cuerpo de la función estas variables no deben volver a ser declaradas: directamente se las utiliza.

Otra forma, más resumida, de escribir la misma función sería:

```
int suma(int x, int y)
    {
    return x+y;
    }
```

Paso de parámetros por valor y por dirección

De acuerdo a la operación que querramos que la función realice, o de los tipos de datos involucrados, el pasaje de parámetros que hace la función que llama a otra función puede realizarse por valor, o por dirección (también se suele denominar a este método por referencia).

En el primer caso, paso por valor, se hace una copia del valor en una variable local a la función. La función no tiene acceso a la variable original, y por lo tanto no la puede modificar.

Cuando el paso de parámetros se realiza por dirección, lo que se envía es la dirección física de la variable. La función puede a través de la dirección que recibe modificar el contenido de la variable original. Esta forma de pasaje es la única que se puede utilizar para enviar un vector o una matriz a una función.

Como norma, o criterio para seleccionar la forma en la que una función recibe los parámetros podemos decir:

- Si no queremos –ni necesitamos- que la función modifique el valor de la variable original debemos hacer el pasaje de parámetros por valor.
- Si necesitamos que la función modifique el valor de una o más variables que se envían como parámetros, o el parámetro es un vector o una matriz, debemos hacer el pasaje por dirección.

Por último, recordar que se debe respetar el orden en el cual fueron establecidos los parámetros en el prototipo, tanto en la llamada como en la definición de la función.

Consideraciones sobre el diseño de funciones

Si bien desde el punto de vista del compilador cualquier conjunto de líneas de código encerradas entre llaves, con su correspondiente cabecera (valor devuelto, nombre y parámetros) es una función que puede ser llamada por otra función, desde la perspectiva de lo que suele denominarse como "buenas prácticas de programación" no siempre es así.

Al diseñarse una función deben tenerse en cuenta aquellas características que se señalaron al inicio del capítulo: toda función debería favorecer la abstracción, ser potencialmente reutilizable, y facilitar la comprensión del programa del que forman parte así como la búsqueda y corrección de errores.

Sobre lo señalado se insistirá a lo largo de la materia; a medida que los programas que desarrollemos incrementen su complejidad, se verá en la práctica lo beneficioso (a veces imprescindible) que resulta el enfoque propuesto. Para empezar veamos algunas de las características que deberían tener las funciones:

- Unicidad: una función debe hacer una sola cosa, esto es encargarse de una tarea clara y bien definida. Esto favorece la abstracción y hace que la función pueda ser reutilizada como tal o con pocas modificaciones en otro programa. Si se presenta el último caso, debería hacerse un esfuerzo de generalización (hacer que la función sea más genérica, por ejemplo, agregándole un parámetro más), para evitar tener más de una función para la misma tarea.

- Bajo acoplamiento: se denomina acoplamiento al grado de relación que existe entre módulos o funciones, esto es el grado de dependencia entre ellas. Cuanto menor sea esta dependencia, más independientes serán entre ellas, y por lo tanto estarán más cerca de cumplir con la característica anterior (unicidad). En términos de código, esto lo podemos ver en la cantidad de parámetros que una función recibe: no suele ser el mejor criterio de diseño para una función que reciba "muchos" parámetros. Si bien no puede de antemano establecerse cuántos parámetros deben considerarse "muchos", ya que depende de la función y el programa que se está desarrollando, si puede asegurarse que una función sólo debe recibir los parámetros que sean estrictamente necesarios para cumplir con su tarea. Cada uno de esos parámetros deberán contener datos (números, caracteres o direcciones), y en ningún caso enviarles variables sin valor.

- Nombres significativos: a las funciones se les debe asignar nombres significativos, esto es, nombres que indiquen lo más claramente posible la tarea que la función realiza. Tal es el caso de las funciones utilizadas más arriba (ponerCeroVector(), o cargarVector()). Si se nos dificulta encontrar un nombre significativo, puede que tengamos que analizar si la función tal como está cumple con las otras características señaladas.