

---

# Asignatura Programación

## Apuntes de clase

### Funciones

#### Desde el sano ejercicio del diseño

Los ejercicios propuestos y desarrollados en las clases son en realidad pequeños problemas que representan, potencialmente, problemas complejos y extensos. La metodología de trabajo usada a lo largo del curso, es la que se basa en la descomposición de los problemas a resolver, técnica basada en el principio "Divide & Conquer".

Los problemas se dividen en subproblemas y estos a su vez en subproblemas más pequeños y así hasta que cada uno sea más fácil de resolver. Las soluciones de los mismos se plantean en forma independiente y están encapsuladas y constituyen los módulos, cada uno de los cuales tendrá una tarea específica y se comunicarán entre sí en forma adecuada para dar solución al problema inicial.

Un módulo es un conjunto de instrucciones más un conjunto de definiciones de datos que lleva a cabo una tarea organizada lógicamente. El método de diseño descendente (Top Down) permite modularizar, fomenta el trabajo en equipo y favorece la detección, búsqueda y corrección de errores, tareas que se llevarán a cabo en forma independiente según los módulos que resultaren de la subdivisión del problema original.

En una tarea compleja, llevada a cabo por varios programadores, donde cada uno ha escrito sus propios módulos, se presenta la situación donde cada profesional define sus propios datos con sus identificadores. Esto provoca algunos inconvenientes: comienzan a convivir demasiados identificadores, surgen conflictos entre nombres, la integridad de los datos pierde consistencia y se producen los llamados efectos laterales con consecuencias nefastas.

Estos problemas se pueden disminuir trabajando desde la perspectiva del ocultamiento de los datos y el uso de parámetros. Para el análisis de estos conceptos, es oportuno aclarar que los **datos globales** son aquellos que se declaran en forma independiente o externa de cualquier módulo, en tanto los **datos locales** son los que se declaran dentro del contexto de un módulo en particular.

La filosofía de trabajo propuesta es: todo dato que es significativo para un módulo debe ser resguardado de los módulos restantes. La localidad del dato con respecto al módulo conduce al ocultamiento del mismo y al resguardo de su integridad. Respecto de los datos globales, sólo se declaran de este modo en el caso que sean usados por más de un módulo.

¿Cómo se comunican los módulos entre sí? El canal de comunicación entre módulos es el parámetro. La función del parámetro es transferir información entre las partes que componen el algoritmo. Un algoritmo bien organizado definirá la conexión entre sus módulos usando parámetros. Toda la información que viaja desde y hacia un módulo debe hacerse vía parámetro, en vez de usar variables globales.

#### Desde el código C

La programación estructurada alcanza sus mejores resultados en el contexto de la funciones. Los programadores generalmente escriben sus programas usando funciones predefinidas que están en las bibliotecas que el ANSI C proporciona y que están incorporadas al sistema. También usan funciones que ellos mismos han escrito previamente y otras nuevas. O sea que, un programa escrito en código C se compone de muchas funciones.

Existe una función muy especial de la cual no podrá escapar nadie que se proponga escribir un programa en C. Se trata de la función de nombre **main**. Todo programa empieza a ejecutarse al inicio de **main**, lo cual significa que cualquier programa debe tener un **main** en alguna parte. Al decir de los autores del lenguaje: "main llamará a otras funciones que la ayuden a realizar su trabajo, algunas que Ud. ya escribió y otras de biblioteca

previamente escritas". Como comprenderá, el principal objetivo de main es controlar la ejecución de las funciones contenidas en ella.

### Funciones de biblioteca

Ellas no forman parte el lenguaje, pero están disponibles en todas las implementaciones. Cada compilador de C incluye un repertorio de funciones. Las bibliotecas mas conocidas son las que contienen funciones para llevar a cabo cálculos matemáticos comunes, manipulaciones con cadenas, pruebas y conversión de caracteres, funciones de entrada y salida, entre otras.

Generalmente las funciones de biblioteca se usan escribiendo el nombre de la función, seguido del o los argumentos requeridos encerrados entre paréntesis. A modo de ejemplo: si en un programa se quiere escribir el valor de la raíz cuadrada de 900.57, se usaran dos funciones de biblioteca: *sqrt* y *printf* de los archivos de cabecera *math.h* y *stdio.h* respectivamente. Para ello, en el sector de instrucciones para el preprocesador se escribirán las directivas apropiadas, en este caso *#include <math.h>* y *#include <stdio.h>*, luego se invocará a las funciones de manera similar a la siguiente: *printf("%f", sqrt(900.57));*

### Funciones definidas por el usuario

Cuando el programador diseñó la solución del problema, escribió módulos de propósitos específicos. Al codificar esos módulos en código C, está escribiendo sus propias funciones. En código C, los módulos se convierten en funciones.

Las funciones constituyen una parte aislada y autónoma del programa, que pueden estar dentro o fuera del código fuente del programa que las invoca. Las proposiciones o sentencias que definen la función se escriben una sola vez y se guardan de manera apropiada. Con funciones bien diseñadas es posible ignorar "cómo" lleva a cabo su trabajo, es suficiente saber "qué hace". Frecuentemente se ven funciones cortas, definidas y empleadas una sola vez, esto es porque el uso de funciones esclarece alguna parte del código. Las funciones podrán tener o no una lista de parámetros. Las funciones se invocan (o llaman o usan) mediante una **llamada de función**. Una llamada de función puede estar en cualquier parte de un programa.

### Cómo se define una función

La sintaxis del lenguaje establece que la forma general de definir una función es:

```

Tipo nombre (lista de parámetros)           // Encabezado
{
    declaraciones                             // Sector de dec. de v. locales
    Proposiciones;                             // Cuerpo de la función
}

```

Se distinguen los sectores de: Encabezado, Declaraciones y Cuerpo de la función.

#### // Encabezado

**Tipo** : es el tipo asociado a la función y está ligado al valor de retorno de la misma.

**nombre**: es el nombre que identifica a la función. Es un identificador válido.

**lista de parámetros**: listado de los parámetros formales de la función, de la forma:

**tipo<sub>1</sub> param1, tipo<sub>2</sub> param2...** Donde **tipo<sub>i</sub>** es el tipo de dato asociado al parámetro.

#### // Sector de declaración de variables locales

De la forma: **tipo<sub>1</sub> var<sub>1</sub>; tipo<sub>2</sub> var<sub>2</sub>; ... tipo<sub>i</sub> var<sub>i</sub>;** Donde **tipo<sub>i</sub>** es el tipo de dato asociado a la variable.

Es un listado de las variables internas o locales a la función.

## // Cuerpo de la función

Formado por **proposiciones o sentencias** válidas.

A tener en cuenta:

- Respecto de los nombres de las funciones, se sugiere usar nombres activos para las mismas, si es posible seguidos de sustantivos, por ejemplo: *dibujar\_triangulo*. Recuerde que un nombre no sólo identifica, sino también aporta información al lector.
- El tipo del valor de retorno indica si la función habrá de devolver valores o no ( ver tipo de dato void). Si el tipo asociado a la función se omite, el compilador lo considera entero.
- La lista de parámetros permite la transmisión de datos entre funciones. Hay que declarar el tipo de cada parámetro, si se omite, se asume entero. Una función puede no tener su lista de parámetros, en cuyo caso los paréntesis estarán vacíos o la palabra reservada void.
- El cuerpo de la función puede tener declaraciones que estarán al comienzo del mismo. Las variables declaradas en ese ámbito son variables locales a la función, al igual que los parámetros del encabezamiento. Las declaraciones y las proposiciones constituyen el cuerpo de definición de la función y está encerrado entre llaves.
- Varias partes pueden omitirse. Una función mínima: ***nada(){} // no hace ni regresa nada***  
Puede servir para reservar lugar al desarrollar un programa.

Importante: a diferencia de otros lenguajes de programación, el C no admite la definición de funciones dentro de otra función.

En resumen, las funciones:

- se declaran
- se definen
- se asocian a un tipo de datos
- se invocan
- se ejecutan
- se pueden usar como factores de una expresión
- pueden devolver un valor/valores
- pueden ejecutar tareas sin retornar ningún valor
- pueden llamarse desde distintas partes de un mismo programa

Un ejemplo simple: Imagine la siguiente situación: un usuario escribe un programa que realiza ciertas tareas y hay un cálculo que se repite a lo largo del mismo, es el cuadrado de un número. Para simplificar y esclarecer sectores del código, el programador escribe una apropiada. A continuación, se transcriben sectores del código en cuestión.

```

:
:
int calcular_cuadrado(int dato);

void main ()
{ int dat1, dat2,resul1, resul2, i;
:
scanf("%d %d",&dat1,&dat2);
resul1 = 25 + calcular_cuadrado(dat1); /* la función es usada como factor de una expresión */
:
:
for (i=1; i<= 10; i++)
printf("El cuadrado del número %d es : %d \n",i,calcular_cuadrado(i));
:
}

int calcular_cuadrado(int dato)
{ int aux;

```

```

aux = dato*dato;
return(aux); }

```

Una versión alternativa de la función:

```

int calcular_cuadrado(int dato)
{
    return(dato*dato); }

```

### Un tipo particular de funciones

Existe un tipo particular de funciones: el tipo *void*. Se usa en los casos en que la función no devuelve ningún valor. Se dice entonces que la función produce un efecto, por ejemplo, dibujar una figura, saltar líneas de impresión, imprimir una constancia, etc., que lleva a cabo un procedimiento.

La forma de definición de una función de este tipo responde a la forma general indicada previamente. Pueden ser invocadas en cualquier parte del programa, para ello, basta escribir el nombre de la misma, seguida de los argumentos indicados. Invocar una función *void* es similar a escribir una proposición en código C. Por último, no se puede poner la función en una expresión donde se requiera un valor.

Para obtener una figura como la siguiente, un código posible es:

```

1          #include <stdio.h>
12         #include <stdlib.h>
123        void dibujar(int filas);
1234       main()
12345     { int n,fil,i;
1         printf("Ingrese cuantos niveles del árbol quiere\n");
1         scanf("%d",&n);
12        printf("Ingrese cuantos filas quiere en cada nivel\n");
123       scanf("%d",&fil);
1234      for(i=1;i<=n;i++)          /* aquí dibuja el árbol */
12345     dibujar(fil);
1         for(i=1;i<=n-n/2;i++)    /* aquí dibuja la base */
12        dibujar(1);
123       return 0;
1234      }
1234     void dibujar(int filas)
12345    { int exte, num;
1         for(exte=1;exte<=filas;exte++)
1         { for(num=1;num<=exte;num++)
1           printf("%d",num);
1           printf("\n");
1         }
1234     }

```

Se sugiere al alumno una lectura detenida y reflexiva del código. También la modificación de código de manera que main se vea como sigue:

```

:
:
int n,fil,i;
printf("Ingrese cuantos niveles del árbol quiere\n");          //LEER (n);
scanf("%d",&n);
printf("Ingrese cuantos filas quiere en cada nivel\n");        // LEER(fil);
scanf("%d",&fil);
dibujar_todo(n,fil);                                           // dibujar-todo(n,fil);
return 0; }                                                    // PARAR.

```

De la lectura de este sector de código se interpreta rápidamente “que hace” pero no hay información acerca de “cómo lo hace”.

Para responder: ¿Qué valores fueron asignados a las variables  $n$  y  $fil$  para obtener esa figura?

## Acerca de variables locales y globales

### Alcance de un identificador

El alcance de un nombre es la parte del programa dentro de la cual se puede usar el nombre. Para una variable declarada al principio de una función, el alcance es la función dentro de la cual está declarado el nombre.

### Variables Locales o Automáticas

Las variables declaradas en el ámbito de una función son privadas o locales a esa función. Ninguna otra función puede tener acceso directo a ellas, se puede pensar que los otros módulos o funciones “no las pueden ver” y por lo tanto no las pueden modificar.

El ciclo de vida de las variables locales es muy breve: comienzan a “existir” cuando la función se activa como efecto de una llamada o invocación y “desaparecen” cuando la ejecución de la función termina. Por esta razón estas variables son conocidas como variables automáticas. Las variables locales con el mismo nombre que estén en funciones diferentes no tienen relación.

Debido a que estas variables locales “aparecen y desaparecen” con la invocación de funciones, no retienen sus valores entre dos llamadas sucesivas, por lo cual deben ser inicializadas explícitamente en cada entrada, sino, contendrán “basura informática”.

### Variables Globales o Automáticas

En oposición a las variables locales, existen las variables externas o públicas o globales. Estas son variables que “no pertenecen” a ninguna función, ni siquiera a *main* y que pueden ser accedidas por todas las funciones. Algunos autores se refieren a ellas como a variables públicas. Se declaran en forma externa al *main*.

Puesto que ellas son de carácter público (desde el punto de vista del acceso) pueden ser usadas en lugar de la lista de parámetros, para comunicar datos entre funciones, lo cual no garantiza que sus valores no se modifiquen. Para evitar esta situación, se puede trabajar con parámetros, los cuales son copiados por la función, (esto se estudiará luego). A diferencia de las variables locales, ellas tienen un ciclo de vida importante, existen mientras dura la ejecución del programa completo.

Cuadro de situaciones posibles, como ejercicio de reconocimiento de variables locales y globales:

```
#include <....>
void funcion1(int par1);
int función2(float par2, char par22);
:
int glo;          /* variable global , es "vista" por todas las funciones */
:
main()
{   int a,b;      /* variables locales a main */
    .....
    .....}

void funcion1(int par1)
{   unsigned a ;  /* variables locales a funcion1 : a, b, par1 */
    char b;       /* No se puede ver a "int a" ni "int b" */
}
```

```
int función2(float par2, char par22)
{
    float c;      /* variables locales a funcion2 : c, d, b, par2, par22*/
    short d;     /* No se puede ver a "int a" ni "int b" */
    int b;       /* esta "int b" es diferente del "int b" de main */
}

```

Se debe tener en cuenta que los parámetros funcionan como variables locales a la función donde se declaran.

### El valor de retorno

A fines de obtener un valor de retorno, se usará la proposición *return*. Ella tiene dos usos importantes:

- Se usa para devolver un valor
- Provoca la salida inmediata de una función ,concluyendo con la ejecución de la función transfiriendo el control de la ejecución.

Las dos formas de uso de esta proposición son las siguientes:

***return;***

***return (expresión);***

Las funciones que no regresan valor alguno pueden tener una proposición *return* sin expresión asociada. La misma hace que el control regrese al programa, pero no devuelve algún valor de utilidad.

En el caso de las funciones que devuelven valores al medio que la invocó, el *return* irá acompañado de una expresión. La misma se convertirá al tipo de retorno de la función si es necesario.

A modo de conclusión, se puede afirmar que las formas de regresar el control de la ejecución al punto donde se invocó la función pueden ser:

1. Si la función no regresa ningún resultado, el control se devuelve cuando la ejecución llega a la llave derecha que cierra el bloque o cuando se ejecuta el enunciado o proposición *return* ( que no tendrá argumento).
2. Si la función regresa un resultado, la proposición : *return (expresión);* devuelve el valor de expresión al punto donde fue invocada la misma.

### Un análisis de situaciones

Se propone al lector analizar el siguiente cuadro de situaciones y justificar cada una de ellas, si es posible.

<pre>main() {     :     :     return (0); }  void main () {     :     :     [return] }  void main() {     :     :     : } </pre>	<pre>double calcular (....) {     int exp;     :     :     :     exp = ...;     return(exp); }  main() {     :     :     :     return; } </pre>	<pre>main() {     :     : }  void dibujar (...) {     :     :     :     [return;] } </pre>
--	---	--

---

## Prototipos de Funciones

El estándar ANSI C establece que una función debe ser declarada y definida. La declaración de una función se hace a través del prototipo de la misma. Se dice que un prototipo es la declaración expandida de la función.

La forma general del *prototipo* de una función es la siguiente:

Tipo nombre\_funcion(lista de tipos de argumentos);

Esta forma general coincide sustancialmente con la primera línea de la definición de una función –el encabezamiento-, con dos pequeñas diferencias:

1° En vez de la lista de argumentos formales o parámetros, *en el prototipo basta incluir los tipos de dichos argumentos*. Se pueden incluir también identificadores a continuación de los tipos, pero son ignorados por el compilador. Ejemplo: `double pitagoras( double, double); // prototipo`

2° El *prototipo* termina con un carácter (;).

Un prototipo de función le indica al compilador el tipo de dato asociado a la función, el número y tipo de parámetros que tiene la función y el orden en el cual se esperan. A cambio de la función prototipo, el compilador lleva a cabo servicios tales como la: detección de errores y conversión de argumentos entre funciones, esto es, la conversión del tipo del valor de retorno y el control que las llamadas de funciones sean coherentes con lo que está expresado o indicado en el prototipo de la función.

Antes de incorporar esta modalidad de control del compilador respecto de las funciones, los diferentes compiladores del lenguaje C no llevaban a cabo esta verificación, lo cual conducía a errores fatales en tiempo de ejecución (run time errors) con los inconvenientes que ello significa.

Cuando no hay una lista de parámetros, se pone entre los paréntesis la palabra **void**, y se pone también **void** precediendo al nombre de la función cuando no hay valor de retorno.

La declaración de las funciones mediante los prototipos suele hacerse al comienzo del fichero, después de los **#define** e **#include**.

En muchos casos –particularmente en programas grandes, con muchos ficheros y muchas funciones–, se puede crear un fichero (con la extensión **.h**) con todos los prototipos de las funciones utilizadas en un programa, e incluirlo con un **#include** en todos los ficheros en que se utilicen dichas funciones.

### Un paréntesis...

**Acerca de parámetros y argumentos:** los parámetros son aquellas variables declaradas en la definición de una función. Parámetro designa un objeto de entrada descrito dentro de la definición de una función. En tanto, se habla de argumento, cuando se hace referencia a la variable o expresión usada en la invocación de la función. También se usan los nombres parámetros formales y parámetros actuales para hacer la misma distinción. Finalmente, el término argumento actual se usa para designar a las variables involucradas en la invocación o llamada de la función.

El compilador controla y exige coherencia en cuanto a tipo, orden y cantidad entre los parámetros formales y los parámetros actuales, los argumentos pasados a una función deben coincidir en tipo, orden y cantidad con los parámetros de la definición de la función. Los argumentos actuales pueden ser no sólo variables y/o constantes, sino también expresiones.

El prototipo para la función que calcula los cuadrados de los números enteros del 1 al 10 puede ser:

*int cuadrado(int );*

Este prototipo le indica al compilador que la función de nombre *cuadrado* tiene declarado un parámetro entero y regresa un resultado de tipo `int`.

La definición de la misma función tendrá el siguiente encabezado: `int cuadrado(int zz);`

Una invocación posible es: `xx = cuadrado(xx+12);` // suponiendo que xx fue declarada int

Otra invocación posible es : `cuadrado (xx +12);`

En esta última invocación, se ejecutará el código correspondiente a la función y la misma retornará un valor entero que, al no haber sido asignado a ninguna variable, no tendrá ningún sentido en el código. El compilador no indica este tipo de errores., podemos decir que un valor de retorno en estas circunstancias es inservible.

Una llamada o invocación de función que no coincida con el prototipo de la misma, causará un error de sintaxis, por ejemplo, si tenemos una función declarada como `void figura();` y en el contexto de `main` la invocamos como `figura(5);` el compilador nos informa de un error en la invocación de la misma. Esto ocurre porque no hay coherencia entre los parámetros y los argumentos. Lo correcto sería invocarla así: `figura();` donde no hay ningún argumento actual en la llamada, tal cual está predefinido en el prototipo de la función.

### Mecanismo de Pasaje de Parámetros: Llamada por Valor y Llamada por referencia

El lenguaje C ha escogido como mecanismo de pasaje de parámetros entre funciones la llamada por valor, conocida también como paso por valor. Cuando el programa que llama encuentra el nombre de la función, evalúa los argumentos actuales contenidos en la llamada, los convierte si es necesario al tipo de los argumentos formales, y pasa copias de dichos valores a la función junto con el control de la ejecución. La función que se invoca recibe los valores de sus argumentos en variables temporales y no en las originales. Una explicación sencilla al respecto sería que cuando se trabaja con este mecanismo de pasaje de parámetros, por valor, la función invocada trabaja sobre "una fotocopia" o "borrador" de las variables originales.

De este modo, las modificaciones que la función haga a la copia, no alteran el valor de la variable original. La función que se está ejecutando sólo puede alterar su copia privada y temporal.

La llamada por valor debe usarse siempre que la función invocada no necesite modificar el valor de la variable original. Esto evita efectos colaterales y permite preservar los valores de las variables.

Entre las ventajas de pasar un parámetro por valor, está el hecho que permite como argumento o parámetro actual una expresión en lugar de ser necesariamente una variable. Si es una simple variable se protege su valor de posibles alteraciones por arte de la función.

El inconveniente es que impide que se transfiera información desde la función hasta el punto de llamada mediante los argumentos, se dice que este mecanismo es unidireccional, la información a través de los argumentos viaja en una sola dirección, hacia adentro de la función, son parámetros de entrada.

Un ejemplo, una función de nombre `pot` que eleva la base a la n-ésima potencia,  $n \geq 0$ :

```
#include <stdio.h>
/*..borlandc\prog02\func12.cpp */
#include <conio.h>
/* 16/oct/02 */
int pot1(int , int );
int pot2(int , int );
main()
{
    short num, bs;

    printf(" Base y potencia \n");
    scanf("%d %d",&bs, &num);
    printf("%d\n",pot1(bs,num));
    printf("%d",pot2(bs,num));
    return 0;}

int pot1(int base,int n)
{
    int l, p; // 1º version

    p = 1;
    for(i=1; i<=n; i++)
        p = p*base;
    return p;
}
int pot2(int base,int n)
{
    // 2º version
    int p;

    for(p=1; n>0; n--)
        p = p*base;
    return p;}
```

En la 2ª versión de la función *pot*, el parámetro *n* se usa como variable temporal y se decrementa hasta que llega a cero, ya no es necesaria la variable *i*. Cualquier cosa que se le haga a *n* dentro de *pot2* no tiene efecto sobre el argumento con que se llamo a *pot2*. Queda a cargo del alumno hacer las pruebas necesarias para verificar esto.

Cuando sea necesario, es posible hacer que una función modifique una variable dentro de una función. En ese caso se simulará el mecanismo de pasaje de parámetro por referencia, esto es, se hará una llamada por referencia, para lo cual es necesario estudiar primero conceptos de direcciones y punteros, que son las herramientas con las cuales se podrá acceder a la dirección de la variable para poder luego modificar su valor.

## La recursividad

La recursividad es una herramienta que permite expresar la resolución de problemas evolutivos, donde es posible que un módulo se invoque a sí mismo en la solución del problema. (Garland, 1986), (Aho, 1988). Las funciones escritas en código C pueden ser recursivas, esto es, pueden llamarse a sí mismas. El código recursivo es compacto y más fácil de escribir y de entender que su equivalente no recursivo. La recursividad es un recurso del lenguaje muy apto para trabajar con estructuras de datos complejas definidas en forma recursiva.

Un ejemplo típico es el cálculo del factorial de un número, definido en la forma:

$$N! = N * (N-1)! = N * (N-1) (N-2)! = N * (N-1)*(N-2)*... *2*1$$

La función factorial, escrita de forma recursiva, sería como sigue:

```
unsigned long factorial(unsigned long numero)
{
if ( numero == 1 || numero == 0 )
    return 1;
else
    return numero*factorial(numero-1);
}
```

Por lo general la recursividad no ahorra memoria. Tampoco es más rápida, sino más bien todo lo contrario, pero el código recursivo es más compacto y a menudo más sencillo de escribir y comprender. Este tema está fuera de los alcances de este curso.

## Comentarios finales

Con buena definición, identificación y elaboración de funciones, los programas pueden ser creados partiendo de funciones estandarizadas, que lleven a cabo tareas específicas, en vez de ser escritos usando código personalizado. Esta técnica se conoce como abstracción.

Entre las razones que impulsan la "funcionalización" de un programa, se debe considerar el principio de Divide & Conquer, que hace que el desarrollo del programa sea mas manipulable. Otra razón, la reutilización del software: el uso de funciones existentes como módulos constructivos para crear nuevos programas. La reutilización del software es uno de los principios básicos de la programación orientada a objetos

## Bibliografía

- De Giusti A. E. 2001 *Algoritmos, datos y programas*
- Deitel H. M. y P. J. Deitel 2000 *Como programar en C/C++*
- Gottfried B. 1997 *Programación en C*
- Kernighan B. W. y D. M. Ritchie 2001 *El lenguaje de programación C*
- Kernighan B. W. y R. Pike 2000 *La práctica de la programación*
- Pappas Ch. y W. Murray 1995 *Manual de Borland C++*

Apuntes revisados y corregidos en Septiembre de 2009.