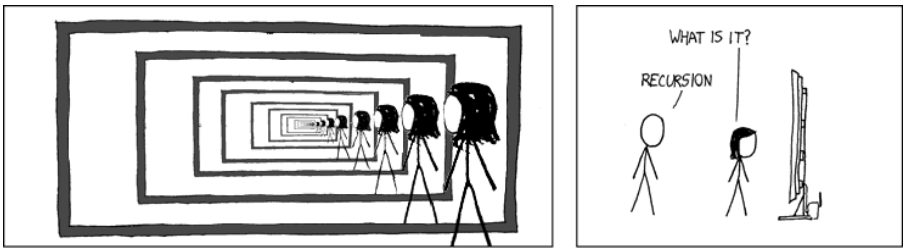
75.40 Algortimos y Programación I Curso 4
Recursividad en C

Dr. Mariano Méndez

Facultad De Ingeniería. Universidad de Buenos Aires

28 de agosto de 2020



1. Recursividad

La **recursividad** es un método de solucionar problemas en términos de versiones más pequeñas del mismo problema, hasta llegar a una versión denominada caso base, que permite solucionarse directamente. Para saber si el proceso de solución recursivo está bien encaminado se debe tener en cuenta:

- El problema se vuelve cada vez más pequeño.
- La solución incluye el caso base.
- Cada caso está tratado correctamente.

Cuando se indica que el problema se vuelve cada vez mas pequeño, se está refiriendo a que en cada paso uno se vá acercando a la solución intentando llegar al caso base. Es decir, calcular $!n$ es más complejo que calcular $!(n-1)$, y a su vez calcular $!(n-2)$ es aun más simple que los dos problemas anteriores, si se sigue restando se llegará al problema más sencillo (**el caso base**) y directamente calculable que es $!1 = 1$

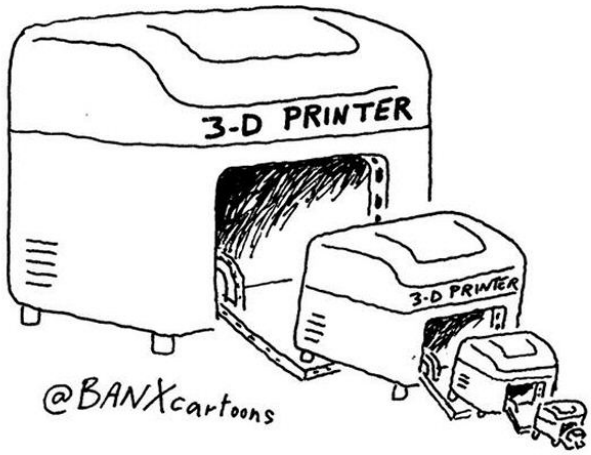


Figura 1

En informática la recursividad está fuertemente asociada a las funciones y a los tipos de datos. En este apunte se estudia su relación con la **implementación de las funciones recursivas**. Se dice que *una función es recursiva si en el cuerpo de la función hay una llamada a sí misma*. Para poder ver cual sería este caso, es conveniente mirar algunos

problemas matemáticos cuya naturaleza es recursiva, por ejemplo, los números factoriales. Un número factorial se define de la siguiente forma:

$$n\,! = \begin{cases} 1 & si\ n = 0 \\ (n-1)\ n! \times n & si\ n > 0 \end{cases}$$

En este caso puede verse como en la definición de factorial se invoca a factorial, este es uno de los mejores ejemplos del concepto de recursivo. Por el momento la versión conocida, para determinar el factorial de un número, debe estar construida en forma iterativa:

```
long factorial (int un_numero){
    long producto;
    int  contador;

    producto=1;
    contador=un_numero;
    while ( contador > 0 ){
        producto *= contador;
        contador--:
    }
    return producto;
}
```

1.1. El Ámbito de una Función (Scope)

Para entender más profundamente el concepto y el funcionamiento de la recursividad en el lenguaje de programación C, es necesario saber que pasa cuando una función es **llamada** o **invocada**. La acción de llamar o invocar a una función *desencadena una serie de mecanismos (complejos) que crean para cada función un espacio de memoria en una sección del programa* llamada **stack de ejecución** o **pila de ejecución**. La pila de ejecución se denomina de esta forma, ya que funciona como una pila de cosas (papeles, libros,platos, etc.). Este tipo de estructura se denomina **IIFO** (**Last in First Out**), esto quiere decir que cuando algo se apila se lo pone en el tope de la misma , y cuando algo se desapila se lo saca del tope de la pila.

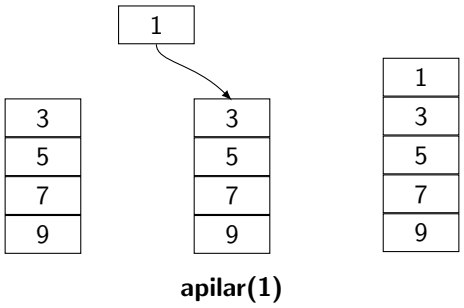


Figura 2: Apilar un elemento en una pila

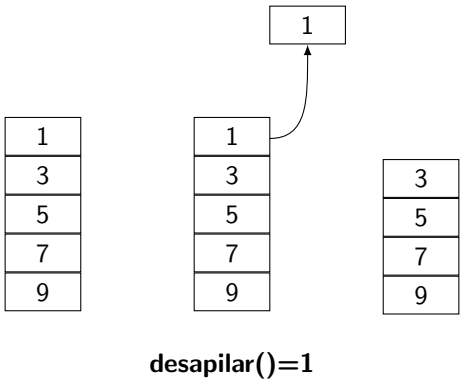


Figura 3: Desapilar un elemento en una pila

Qué se apila en el stack de ejecución, el ámbito de la última función llamada o invocada. Si mientras esta función se está ejecutando se llama a otra función, se crea un nuevo ámbito y se apila en el stack de ejecución. Así sucesivamente, hasta que la última función llamada termina de ejecutarse. En ese momento lo que sucede cuando termina la ejecución de la misma el ámbito de ejecución de la función se destruye automáticamente (el programador no debe ocuparse de nada, esto lo hace automáticamente mientras el programa se ejecuta).

1.1.1. Qué se Guarda en el Ámbito de una Función

El ámbito de una función es una sección de memoria del programa encargada de almacenar:

1. todas las variables locales definidas en la función.
2. los parámetros de la función.

Esto quiere decir que cada vez que una función se llama, sucede lo siguiente:

1. se crea el ámbito de la función como una sección de memoria del programa,
2. se crean los parámetros de la función y se copian los valores de los parámetros actuales,
3. se reserva memoria para cada una de las variables locales que se declaran en la función.

tomando esta función como ejemplo:

```c
long factorial (int un_numero){
    long producto;
    int  contador;

    producto=1;
    contador=un_numero;
    while ( contador > 0 ){
        producto *= contador;
        contador--:
    }
    return producto;
}
```

Ambito de factorial:

Parámetros:

(int) un_numero

Variables locales:

(int) producto

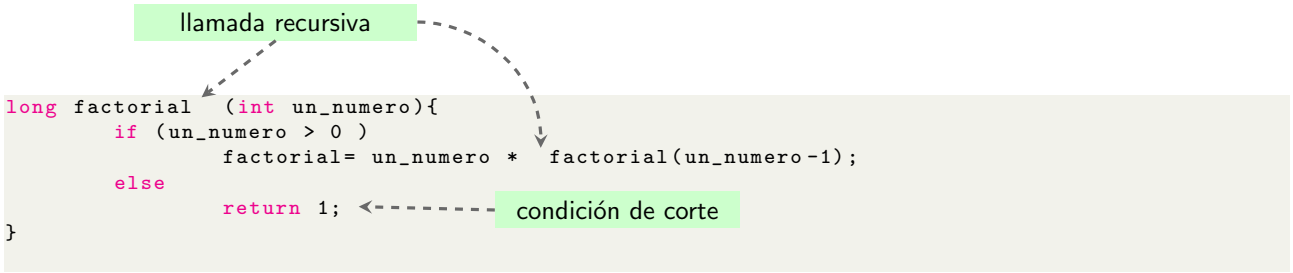
(int) contador

1.2. Función Recursiva

¿Será posible escribir una versión recursiva del algoritmo para calcular el factorial de un numero?

En el lenguaje de programación C es posible construir funciones recursivas de la misma forma que en matemática. Para ello lo único que hay que hacer es llamar otra vez a la función dentro de su ámbito. Para lograr realizar esto se utiliza el **stack o pila de ejecución**, este no es más que la parte del programa donde se mantiene la información sobre cual función es la que está siendo ejecutada, sus variables y parámetros. Todos los programas tienen su propio stack, que es creada cuando comienza a ejecutarse, ver Figura 4. Si se observa detalladamente una función recursiva debe cumplir con ciertas reglas para que funcione correctamente, estas son:

- debe poseer una condición de corte
- debe poseer una llamada recursiva, es decir así misma dentro de la función.



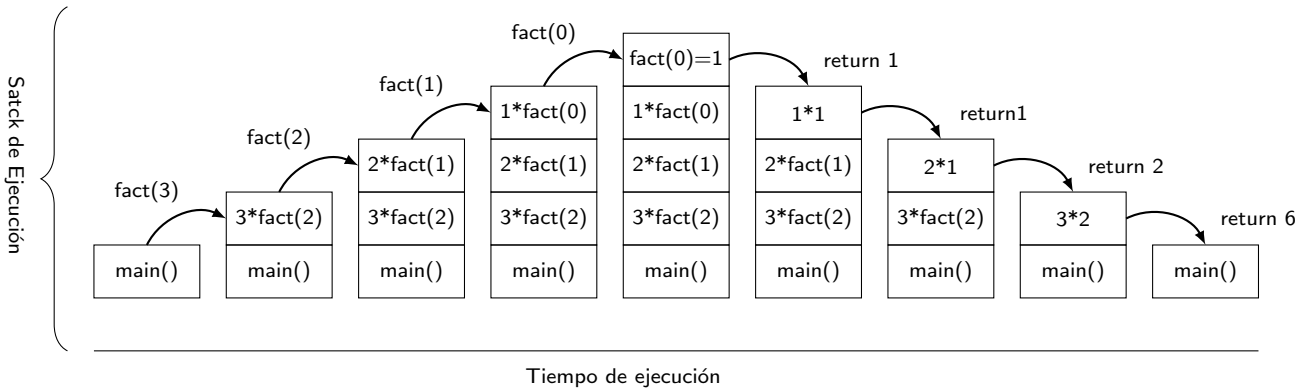


Figura 4: stack de ejecución

1.2.1. Nota

Es importante observar que el proceso de ejecución de la función recursiva tiene tres etapas como se puede observar en la Figura 4:

1. Las llamadas recursivas
2. La llegada a la condición base o punto de corte
3. El retorno de la llamada recursiva.

Estas etapas normalmente se dan en ese orden, y depende de la solución que se quiera implementar se aprovecha la etapa 1 o la etapa 3.

1.2.2. Ejemplo

Se le solicita al usuario que ingrese un número por teclado, luego se calcula e imprime por pantalla el resultado de la Serie de Fibonacci para dicho número.

```
long int fibonacci(numero){
        if (numero < 2)   // Condicion de corte
                return numero;

        return fibonacci(numero-1)+fibonacci(numero-2); //Llamado recursivo para numero-1 y luego para numero-2
}

int main (){

        int numero;

        scanf("%i",&numero);

        long int resultado = fibonacci(numero);

        printf("El Fibonacci de %i es %lu",numero,resultado);
        return 0;

}
```

2. Algunos Ejemplos Importantes

A continuación se propondrán ciertos ejemplos que permitan ver y entender el concepto de recursividad aplicado al lenguaje de programación C

2.0.1. De Iteración (while) a Función Recursiva

Sea una función cualquiera que realice su cometido utilizando una iteración, por ejemplo la suma de los n primeros números. Se mostrará a continuación como se puede obtener una versión recursiva de la misma:

```
long sumar (int numero){
    long suma;
    int  contador;

    suma=0;
    contador=un_numero;
    while ( contador > 0 ){
        suma += contador;
        contador--:
    }
    return suma;
}
```

La idea está en que el problema se puede solucionar de la siguiente forma por ejemplo (no es la única) $sumar(x) = x + x - 1 + x - 2 + x - 3.. + 2 + 1 + 0$. las preguntas que uno se debería hacer es ¿Cuando termino de sumar? Cuando x=0. ¿Si x es distinto de cero? acumulo el valor de x y vuelvo a intentar solucionar el problema con el numero predecesor de x (es decir x-1). Aquí se puede ver que naturalmente salen los dos elementos que se necesitan para crear la misma solución pero recursiva, el punto de corte $-> x = 0$, la llamada recursiva es la función con el predecesor de x. Una posible solución en base a esta estrategia es:

```
long sumar (int numero, long suma){
    if (x==0)<-------- condición de corte
        return 0;
    else
        return suma + sumar(x-1) ;   llamada recursiva
}
```

Es importante observar que la llamada recursiva corresponde a llamar a la misma función pero con un problema mas pequeño que solucionar. La llamada a la función seria `sumar(10,0);`

2.0.2. De Iteración (for) a Función Recursiva

El mismo ejemplo del caso anterior puede aplicarse a una función que utilice una estructura de control iterativa distinta a un while como por ejemplo un for. En este caso la función devuelve cuantos número pares hay hasta el número dado:

```
long cuantos_pares (int numero){
    long cantidad;

    cantidad=0;
    for(int i =1; i<=numero; i++ ){
        if (i mod 2) cantidad ++;
    }
    return cantidad;
}
```

La idea de la función es ir incrementando en la variable cantidad cuantos número son divisibles por dos, por ende, son pares. Para transformar esta función en recursiva hay que buscar los dos elementos claves de una función recursiva:

- condición de corte
- llamada recursiva

La condición de corte debe responder a la pregunta ¿Cuando paro? cuando i es igual al numero que se ingresa. La llamada recursiva se da cuando se vuelve a llamar al problema con una solución mas pequeña del problema, en este caso el siguiente numero par.

```
long cuantos_pares (int numero, int i){
    if (i==numero)<-------- condición de corte
        if ( (i mod 2) ==0)
            return 1;
        else
            return 0;
    else
        if ( (i mod 2)==0) )
            return 1 + cuantos_pares(numero,i+1) ;
        else
            return 0 + cuantos_pares(numero,i+1) ;   llamada recursiva
}
```

2.0.3. Impresión en Funciones Recursivas

Es interesante saber como imprimir cuando se está utilizado soluciones recursivas, el lugar en el cual se pone la acción de impresión es fundamental.

2.0.4. Imprimir Ascendentemente

Imprimir los primeros n número en orden ascendente. A continuación se implementara la versión iterativa del mismo :

```
void imprimir_n_numeros (int numero){
    int i ;

    i=1;
    for( i ; i<=numero; i++ )
        printf("%i\n");
}
```

La idea de la función es ir incrementando en la variable i, utilizando un for e imprimir por cada iteración el valor correspondiente. Para transformar esta función en recursiva hay que buscar los dos elementos claves de una función recursiva:

- condición de corte
- llamada recursiva

La condición de corte debe responder a la pregunta ¿Cuándo paro? Se para cuando i es igual al numero que se ingresa. La llamada recursiva se da cuando se vuelve a llamar al problema con una solución mas pequeña del problema, en este caso el siguiente numero par.

```
void imprimir_n_numeros (int numero, int i){
    if (i<=numero){
        printf("%i",i);
        imprimir_n_numeros(numero,i+=1);
    }
}
```

2.0.5. Imprimir Descendentemente

Imprimir los primeros n número en orden ascendente. A continuación se implementara la versión iterativa del mismo :

```
void imprimir_n_numeros (int numero){
    int i ;

    i=numero;
    for(i ; i<=0; i-- )
        printf("%i\n");
}
```

La idea de la función es igual a la del ejercicio anterior pero ente caso ir decrementando en la variable i, utilizando un for e imprimir por cada iteración el valor correspondiente. Para transformar esta función en recursiva hay que buscar los dos elementos claves de una función recursiva:

- condición de corte
- llamada recursiva

La condición de corte debe responder a la pregunta ¿Cuándo paro? Se para cuando i es igual al 0. La llamada recursiva se da cuando se vuelve a llamar al problema con una solución mas pequeña del problema, en este caso el siguiente numero par. En este caso existen dos posibles soluciones, se analizarán ambas:

Primera Opción esta opción puede ser la solución más intuitiva, y sigue el esquema planteado arriba:

```
void imprimir_n_numeros (int numero){
    if ( numero >= 0 ) {
        printf("%i",i);
        imprimir_n_numeros(numero-=1);
    }
}
```

Segunda Opción esta opción puede ser la solución mucho menos intuitiva, en la cual se aprovecha la estructura del stack de ejecución para alcanzar la solución:

```
void imprimir_n_numeros (int numero, int i){
    if (i<=numero){
        imprimir_n_numeros(numero,i+=1);
        printf("%i",i);
    }
}
```

2.1. Recursividad Cruzada

Se dice que una funcion usa recursividad directa, si la misma se llama a sí misma. Por otro lado existe otra forma de recursividad, la **recursividad indirecta**, que se dá cuando una función f llama a una función g, la función g llama a la función g y así sucesivamente... En la Figura 6 se puede ver el esquema de funcionamiento de la recursividad indirecta:

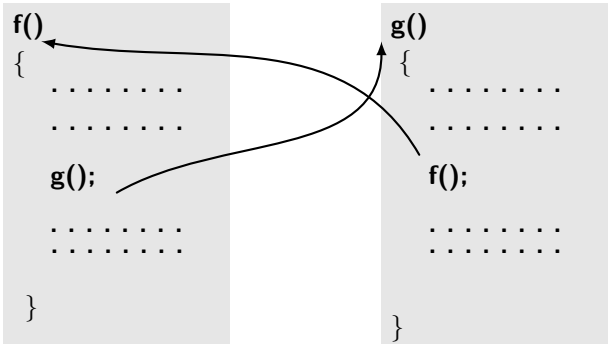


Figura 5: Simulacion de recursividad mutua o indirecta

Al igual que en la recursividad directa ambas funciones deben tener un punto de corte.

Ejercicio 1

La Granja de los Conejos Inmortales Tomás el granjero le compró al brujo del pueblo un par de conejos inmortales recién nacidos con la intención que se reproduzcan y tener su propio ejercito de conejos inmortales super adorables y poder así conquistar el mundo. Tomás le preguntó al brujo que es lo que tenía que tener en cuenta a la hora de tener su propia granja de conejos inmortales y el brujo le comentó lo siguiente:

1. Un par de conejos bebe tardan un mes en crecer.
2. Un par de conejos adulto tardan un mes en producir otro par de conejos bebe.
3. A partir de que un par de conejos adulto tienen un par de bebés, producen uno nuevo cada mes indefinidamente.

1. Tomás inmediatamente se da cuenta que va a ser difícil llevar la cuenta de cuántos conejos tendrá en el futuro así que nos pidió que lo ayudemos y hagamos una serie de algoritmos en C que determinen cuántos conejos bebe, adultos y totales tendrá el n-ésimo mes.

Solución del ejercicio 1

Análisis: en primer lugar se debería poder escribir un ejemplo a mano del problema planteado.

Mes	Bebes	Adultos
1	1	0
2	0	1
3	1	1
4	1	2
5	2	3
6	3	5
7	5	8

Figura 6: Simulacion de recursividad mutua o indirecta

Para saber la cantidad de conejos adultos basta con mirar la tabla, y se puede observar que:

$$Adultos(n) = \begin{cases} 0 & si\ n = 0 \\ \\ Bebes(n-1) + Adultos(n-1) & si\ n > 0 \end{cases}$$

para saber la cantidad conejos bebes en un determinado tiempo n, basta ver en la tabla que :

$$Bebes(n) = \begin{cases} 1 & si\ n = 0 \\ \\ Adultos(n-1) & si\ n > 0 \end{cases}$$

Con la definición de estas funciones ya se puede implementar el problema en forma computacional. **Implementación:**

```c
#include <stdio.h>

//Linea de compilacion: gcc resuelto.c -o resuelto -Wall -Werror -std=c99 -g

/*
 * Todas estas funciones devuelven la cantidad de parejas
 * de conejos totales, bebes y adultas respectivamente.
 * Pre: El numero del mes es mayor o igual a 0
 */
unsigned int cantidad_total(unsigned int mes);
unsigned int cantidad_bebes(unsigned int mes);
unsigned int cantidad_adultos(unsigned int mes);

int main(){
  unsigned int mes;
  printf("El codigo funciona con cualquier nÃºmero, pero es recomendable no
  poner numeros mayores a 40 porque puede llegar a tardar mucho\n", );
  printf("Mes nÃºmero:");
  scanf("%d", &mes);
  printf("La cantidad total de parejas es:%d\n", cantidad_total(mes));
  printf("La cantidad de parejas bebes es:%d\n", cantidad_bebes(mes));
  printf("La cantidad de parejas adultas es:%d\n", cantidad_adultos(mes));
  return 0;
}

unsigned int cantidad_total(unsigned int mes){
  return cantidad_bebes(mes)+cantidad_adultos(mes);
}

unsigned int cantidad_bebes(unsigned int mes){
  /*
   * La cantidad de bebes en un mes es igual a la cantidad
   * de adultos del mes anterior. Si el mes es el primero,
   * la cantidad de bebes es 1.
   */

  if(mes == 1)
    return 1;

  return cantidad_adultos(mes-1);
}

unsigned int cantidad_adultos(unsigned int mes){
  /*
   * La cantidad de adultos en un mes es igual a la cantidad
   * de adultos y bebes del mes anterior. Si el mes es el primero,
   * la cantidad de adultos es 0.
   */

  if(mes == 1)
    return 0;

  return cantidad_adultos(mes-1)+cantidad_bebes(mes-1);
}
```

3. Recursividad Avanzada

En esta sección se verán ejercicios más complejos de recursividad que utilizan estructuras de datos complejas, como los vectores.

3.1. Longitud de un String

Ciertos ejercicios clásicos de string pueden ser realizados en forma recursiva. Un ejemplo clásico es la implementación de una función que calcule el la longitud de un string.

Versión Iterativa lo más común es iniciar desde el primer caracter e ir chequeando que el mismo sea distinto del caracter fin de string, en ese caso se incrementa el índice del arreglo, hasta encontrarlo:

```
int longitud (const char * string){
    int i;
    int longitud;

    longitud=0;
    i=0;
    while ( string[i]!='\0' ){
        longitud+=1;
        i+=1;
    }
    return longitud;
}
```

Versión Recursiva la idea es la misma que en el ejercicio anterior pero con una visión de solución recursiva. La idea es ir avanzando hasta encontrar caracter fin de string, cuando se llega allí se regresa 0 y se cortan las llamadas recursivas:

```
int longitud_rec (const char * string, int i){
    if (string[i]=='\0')
        return 0;
    else{
        return ( 1 + longitud_rec (string,i+1) ) ;
    }
}
```

3.2. Imprimir un Vector

Este ejercicio está relacionado con uno parecido a la sección de ejercicios básicos de recursividad. la idea es recorrer el vector e imprimir caracter por caracter hasta llegar al fin del string:

Versión Iterativa lo más común es iniciar desde el primer caracter e ir chequeando que el mismo sea distinto del caracter fin de string, en ese caso se imprime y se incrementa el índice del arreglo, hasta encontrarlo:

```
void imprimir_vector (const char * string){
    int i;

    i=0;
    while ( string[i]!='\0' ){
        printf("%c",string[i]);
        i+=1;
    }
}
```

Versión Recursiva la idea es la misma que en el ejercicio anterior. La idea es ir avanzando e imprimiendo hasta encontrar caracter fin de string, cuando se llega allí se regresa 0 y se cortan las llamadas recursivas:

```
void imprimir_vector (const char * string, int i){
    if ( string[i]=='\0' )
        return;
    else{
        printf("%c",string[i]);
        imprimir_vector(string,i+=1);
    }
}
```

3.3. Imprimir un Vector al Revés

Este es otro ejercicio clásico que es interesante resolverlo en forma recursiva, ya que la versión iterativa lo que utiliza es la función strlen() e iterando desde allí hasta 0, decrementando un índice. habiendo visto varios ejemplos solo se explicará la solución recursiva. la solución ataca el problema llamando recursivamente a sí misma, mientras no se encuentre el marcador de fin de string. Se aprovechará el funcionamiento del **stack de ejecución** para imprimir una vez que las llamadas se cortan:

```
void imprimir_vector (const char * string, int i){
    if ( string[i]=='\0' )
        return;
    else{
        imprimir_vector(string,i+=1);
        printf("%c",string[i]);
    }
}
```

3.4. Calcular el factorial de un número (Recursivo)

```
#include <stdio.h>

int calcular_factorial(int numero){

        if(numero == 1){           //condición de corte
                return 1;
        }

        return ( numero * calcular_factorial(numero-1));  //llamada recursiva
}


int main(){

        int valor = 4;
        int factorial;

        factorial = calcular_factorial(valor);
        printf("El factorial de 4 es: %i\n", factorial);

        return 0;
}

/*En este ejemplo calcularemos el factorial de 4. Veamos qué sucede:
Con la primer llamada a la función "calcular_factorial" se crea un ámbito (ámbito 1)
en el cual la variable numero vale 4.
Con la segunda llamada se crea un segundo ámbito (ámbito 2), donde numero vale 3.
En la tercer llamada, ámbito 3, numero vale 2.
Cuarta llamada, ámbito 4, numero vale 1. Por la condición de corte, el ámbito 4
retorna 1.

La ejecución vuelve al ámbito 3 donde el valor de la variable numero se multiplica
por el retorno del ámbito 4:
2*1 -> el ámbito 3 retorna 2.

De nuevo en el ámbito 2, se multiplica el valor de numero de este (3) por el retorno
del ámbito 3:
3*2 -> el ámbito 2 retorna 6.

Por último, en el ámbito 1, se multiplica el valor de numero (4) por el retorno del
ámbito 2:
4*6 -> ámbito 1 retorna 24.

Ahora el valor de factorial es 24. Finalmente, se imprime por pantalla:
"El factorial de 4 es: 24"*/
```