Funciones

Las funciones son bloques de código com un nombre asociado que realizan una tarea en particular. En C todo se construye con funciones, (main es la funcion principal del programa).

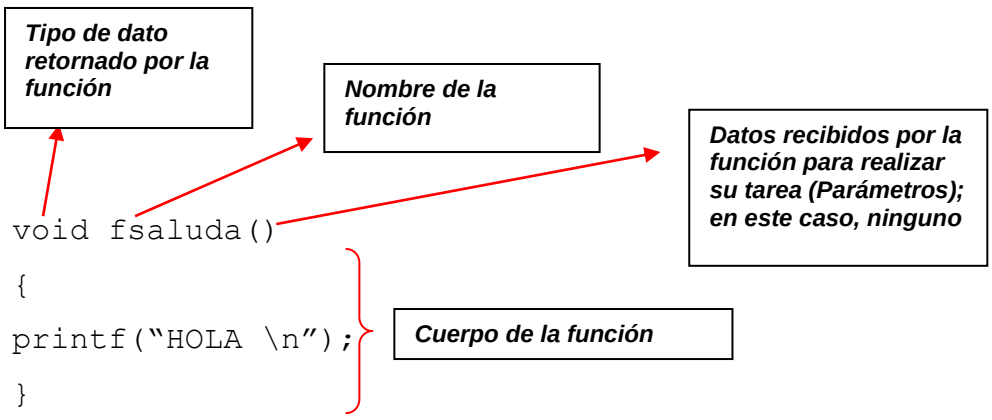
Una función se escribe una vez y luego se invoca, es decir, se ordena que se ejecute, todas las veces que sea necesario.

Algunas funciones requieren que se les pase uno o más datos para poder llevar a cabo su tarea; otras, no. Algunas retornan (es decir, devuelven) un dato, y otras no. En general, trataremos de que una función cumpla un único objetivo. Si al enunciar lo que debe hacer una función nos encontramos con que enumeramos tareas simples que son desarrolladas completamente por ella, lo más probable es que la función este mal planteada.

Por ejemplo, si una función tuviera que ingresar una secuencia de N números, calcular el mayor y emitirlo, no estaría bien diseñada, porque es poco probable que muchas veces se necesite realizar esa tarea compleja. Sin embargo, es cierto que muchas veces, se requiere ingresar una serie de números y almacenarlos, y que muchas veces se necesita obtener el valor mayor de un grupo de números ingresados previamente.

Entonces, lo correcto es escribir una función que permita ingresar y almacenar una secuencia de N números, escribir otra función que reciba una secuencia de N números y obtenga el mayor, y escribir una tercera función, que llame a las otras dos en el orden correcto.

Ejemplo: definición de función que emite un saludo por pantalla.



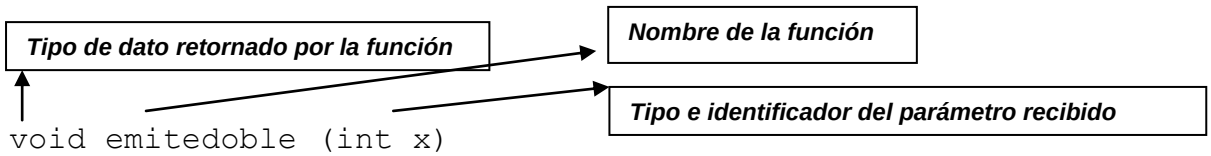
Ejemplo de invocacion a fsaluda:

```
int main()
{
fsaluda();
fsaluda();
```

Módulo 3

```
                    printf("invocamos una vez mas\n");
                    fsaluda();
                    system("pause");
                    return 0;
                    }
```

Ahora veremos una función que recibe un dato entero y emite el doble del mismo por pantalla:




```
void emitedoble (int x)
{
     int aux;
     aux=2*x;
     printf ("El doble de %d es %", x, aux);
}
```

Variable local auxiliar

El parámetro x es una variable entera local, que se inicializa con el valor del argumento de llamada, aux es otra variable local pero no inicializada. Las variables locales solo existen mientras se esta ejecutando la función. Se generan en la pila o stack.

Ejemplo de uso:

```
int main()
{
int a,=5, b=8;
//linea 1
emitedoble(5);
//línea 2
emitedoble(8);
//línea 3
emitedoble(10);
//linea 4
printf("invocamos una vez
mas");  //linea 5
emitedoble(a+b+1);
//linea 6
system ("pause");
return 0;
}
```

Cuando se ejecuta la línea 2 y se invoca a emitedoble, el control del programa "sale" de la función main y pasa a la función emitedoble. Lo mismo ocurre en las líneas 3, 4 y 6. En cada caso, al terminar la ejecución de la función invocada, el control regresa a sentencia siguiente de la invocación. Esto es posible porque se almacena, también en la pila o stack, la dirección correspondiente a la siguiente instrucción, para que pueda continuarse la ejecución de main.

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de

Módulo 3

		emitedoble(a))
main	a= 5 b=8	----------

Cuando se ejecuta la linea 2, el control pasa a emitedoble, y la situación es:

control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de emitedoble(a))
emitedoble	a= 5 b=8	x=5 aux

Al finalizar la ejecución de emitedoble(a), el estado de las variables es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(a))
emitedoble	a= 5 b=8	x=5 aux =10

Cuando el control vuelve a main, antes de la llamada emitedoble(b), la situación es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de emitedoble(b))
main	a= 5 b=8	----------

Cuando se invoca a emitedoble(b), sucede esto:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de emitedoble(b))
emitedoble	a= 5 b=8	x=8 aux

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(b))
emitedoble	a= 5 b=8	x=8 aux =16

Cuando el control vuelve a main, antes de la llamada emitedoble(10), la situación es:

control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de emitedoble(10))
main	a= 5 b=8	----------

Luego se invoca a emitedoble(10), y es:

Cuando se invoca a emitedoble(b), sucede esto:

Módulo 3

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de emitedoble(10))
emitedoble	a= 5 b=8	x=10 aux

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(10))
emitedoble	a= 5 b=8	x=10 aux =20

Cuando el control vuelve a main, antes de la llamada emitedoble(10), la situación es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de emitedoble(a+b+1))
main	a= 5 b=8	----------

Finalmente se invoca a emitedoble(a+b+1), resultando:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de emitedoble(a+b+1))
emitedoble	a= 5 b=8	x=10 aux

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(a+b+1))
emitedoble	a= 5 b=8	x=10 aux =20

Cuando el control vuelve a main, antes de la llamada emitedoble(10), la situación es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila
main	a= 5 b=8	----------

Como se observa, en ningún caso se altera el contenido de las variables definidas en main, llamadas variables estáticas. Este modo de pasar los argumentos a una función se denomina pasaje por valor. La variable local parámetro, que corresponde al argumento de la función, se inicializa con una copia del valor del argumento correspondiente.

Otro ejemplo: Esta es una función que recibe dos enteros llamados m y n y emite por pantalla m%n si m>=n, o n%m en caso contrario (recordar que el operador % devuelve el resto entero de división entera).

```
void resto (int m, int n)
{
//se ha elegido comparar a m con n para intercambiarlos en caso de
que m<n

//aux es la variable local auxiliar para el intercambio

     int aux;
     if(m<n)
       {
         aux=m;
         m=n;
         n=aux;
  }
printf ("El resto de la division entre %d y %d es %d\n", m, n,
m%n);
}
```

Ejemplo de invocaciones a resto:

```
main()
{
      int a, b, c;
      a=100, b=27, c=39;
      resto(a,b);   //m corresponde a a y n a b
      resto (b,a);  //m corresponde a b y n a a
     resto(23,c);   //m corresponde a 23 y n a c
     resto (200, 49); //m corresponde a 200 y n a 49
     resto(209+b-c, 25+a); //m corresponde a 209+b-c, y n a 25+a
     system ("pause");
     return 0;
}
```

Observar que la función resto puede recibir como argumento tanto variables, (de las cuáles hará copia de sus contenidos en sus variables locales) como valores *inmediatos,* (como por ejemplo 23, el cual se almacenará en el parámetro correspondiente). Cómo lograr que las funciones alteren el contenido de las variables correspondientes a los argumentos de llamada?.

Supongamos que queremos una función que altere el contenido de una variable entera, duplicándolo. Para resolver esta cuestión, se pasará por valor la copia de la dirección de la variable. Entonces, la función en cuestión (llameémosla duplicar), recibirá un puntero a la variable, conteniendo la dirección de la variable. Es decir, que main podría ser, por ejemplo:

```
int main()
{
int s=4, h=18;
printf(" s vale %d\n", s)
//Le pasaremos a duplicar
la dirección de s para que
duplicar modifique su
contenido
duplicar(&s);
printf("ahora s vale
%d\n", s)
printf(" h vale %d\n", h)
//Ahora le pasaremos a
duplicar la direccion de h
para que duplicar
modifique su contenido
duplicar(&h);
printf("ahora h vale
%d\n", h)
system("pause");
return 0;
}
```

El código de duplicar puede ser, por ejemplo, (observar en la declaración que duplicar recibe un puntero):

```
void duplicar (int *p)
{
  *p=(*p) *2;
}
```

p es &s cuando se invoca duplicar (&s)

p es &h cuando se invoca duplicar (&h)

Considerando, para ejemplificar, que &s sea 1000 y que &h sea 1004, es

control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de duplicar(s))
main	s= 4 h=18	----------

Cuando el control pasa a duplicar(s), la situación es:

control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de duplicar(s))
duplicar	s= 4 h=18	p=1000 (p contiene l adireccion de s, y además *p vale **4**)

Al finalizar la ejecución de duplicar(s), el estado de las variables es:

control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(a))
duplicar	s= **8** h=18	p=1000 (ahora *p vale **8**)

Cuando el control vuelve a main, antes de la llamadaduplicar(h), la situación es:

Módulo 3

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (antes de la ejecución de duplicar(h))
main	s= 4 h=18	----------

Cuando el control pasa a duplicar(s), la situación es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (al comenzar la ejecución de duplicar(s))
duplicar	s= 4 h=18	p=1004 (p contiene la dirección de h y además *p vale 18)

Al finalizar la ejecución de duplicar(s), el estado de las variables es:

Control	Estado de las variables de la Memoria estática	Estado de las variables de la Pila (un instante antes de finalizar la ejecución de emitedoble(a))
duplicar	s= 8 h=**36**	p=1000 (ahora *p vale **36**)

Es decir, como en el caso anterior, los parámetros de la función son variables locales, que solo existen mientras se ejecuta la función, pero como se trata de un puntero, mediante ese puntero se puede alterar el valor de la variable de llamada.

Observar que en este caso, carece de sentido invocar a la función pasándole un valor *inmediato* (como duplicar(100), por ejemplo), ya que la función exige por su diseño que se le pase la dirección de una variable para modificar su contenido.

Sintetizando

Memoria estática	**Pila o Stack**
Aquí se ubican las variables definidas en main, que existen hasta la finalización de la ejecución de la misma.	Aquí se generan las variables de cada función que se invoca. Sólo existen mientras se ejecute esa función. Los parámetros son variables locales inicializadas - con el contenido de una variable o bien -con la dirección de una variable

Otro ejemplo: una función que permite ingresar un valor por teclado, se espera que la función permita leer un float y pueda ser invocada así:

```
int main()
{
  float f;
  leer(&f);
```

```
      printf ("La variable f contiene %f\n");
      system("pause");
      return 0;
    }
```

La función puede ser:

```
void  leer(float *q)
{
     printf("Ingrese un valor real\n");
     scanf("%f", q);  //observar que se puede usar q para
     ingresar el real, ya que es la dirección en donde debe
     almacenarse el valor entrante
     system ("pause");
     return 0;
}
```

Funciones que retornan un valor: Ahora se diseñará una función que reciba tres valores enteros y retorne el promedio de los mismos. El valor devuelto por la función es float. La función (llamémosla prom) podrá invocarse de estos modos:

```
main ()
{
int a=4,b=10,c=8;
float r;
r = prom(a,b,c);

printf("El promedio entre %d,
%d y %d es %f \n",a,b,c,r);
r= prom(100, 34, a);

printf("El promedio entre %d,
%d y %d es %f \n",100,34,a,r);
printf (("El promedio entre
%d, %d y %d es %f \n",70,30,b,
prom(70,30,b));
```

```
if(prom(3,a,4)>5) printf ("El
promedio obtenido es mayor que
5\n");
     else printf ("El promedio
obtenido es menor que 5\n");

r=prom (34,56,a) + prom
(a,b,c) - 100;

printf("El resultado de
efectuar prom (34,56,a) + prom
(a,b,c) - 100 es %f\n",r)
system ("pause");
return 0;
}
```

Observar que la invocación a prom se utiliza como si fuera una expresión float, y debe, o bien ser almacenada en una variable, o ser usada en una expresión aritmética, o para construir una expresión lógica, o directamente en un printf. Nada de esto era posible con las funciones que devolvían void. La función prom puede diseñarse así:

Tipo de dato retornado por prom

```
float prom(int x, int y , int z)
{
float aux=(x+y+z)/3;
return aux;
}
```

Toda función que retorne un tipo de dato diferente de void debe tener al menos una línea return.

Otro ejemplo: otra forma de ingresar un valor desde teclado, si diseñamos una función del siguiente modo:

```
int lee()
{
int aux;
printf("Ingrese un valor entero");
scanf("%d", &aux);
return aux;
}
```

La variable aux es local; sólo existe mientras se ejecuta lee(). La dirección de aux corresponde a alguna posición dentro de la zona de pila. El valor almacenado en aux se retorna. La invocación puede ser :

```
int main()
{
     int a,b;
     a=lee();
     printf("a vale %d\n", a);
     b=lee();
     printf("b vale %d\n", b);
     a=lee();
     printf("Ahora a vale %d\n", a);
     return 0;
}
```

Funciones recursivas

Una función se dice recursiva si durante su ejecución se invoca directa o indirectamente a sí mismo. Esta invocación depende al menos de una condición que actúa como condición de corte que provoca la finalización de la recursión. Un algoritmo recursivo consta de:

- ✓ Al menos un caso trivial o base, es decir, que no vuelva a invocarse y que se ejecuta cuando se cumple cierta condición, y
- ✓ El caso general que es el que vuelve a invocar al algoritmo con un caso más pequeño del mismo.

Los lenguajes que soportan recursividad, dan al programador una herramienta poderosa para resolver ciertos tipos de problemas reduciendo la complejidad u ocultando los detalles del problema. La recursión es un medio particularmente poderoso en las definiciones matemáticas. Para apreciar mejor cómo es una llamada recursiva, estudiemos la descripción matemática de factorial de un número entero no negativo:

$$n! = \begin{cases} 1, & \text{si } n = 0 \\ n * (n - 1) * (n - 2) * .. * 1 = n * (n - 1)! & \text{si } n > 0 \end{cases}$$

Por supuesto, no podemos hacer la multiplicación aún, puesto que no sabemos el valor de 3!

3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1

Podemos ahora completar los cálculos

1! = 1 * 1 = 1
2! = 2 * 1 = 2
3! = 3 * 2 = 6
4! = 4 * 6 = 2

Ejemplo

Cálculo del factorial para enteros no negativos

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

Diagrama de Llaves:

Factorial(n)

(*Pre: n >= 0*)
n = 0 | Factorial ← 1 (*Caso base*)
⊕
n = 0 | Factorial ← n * Factorial (n-1) (*Caso Gral*)
(*Pos: Factorial = n!*)

```
// código en C
long factorial(int p)
{
 if (p == 0 )
   return 1; //(base)
 else
   return p * factorial(p - 1); //(general)
}
```

Diseño de Algoritmos Recursivos

Para que una función o procedimiento recursivo funcione se debe cumplir que:

- Existe una salida no recursiva del procedimiento o función y funciona correctamente en ese caso.

- Cada llamada al procedimiento o función se refiere a un caso más pequeño del mismo.
- Funciona correctamente todo el procedimiento o función.

Para poder construir cualquier rutina recursiva teniendo en cuenta lo anterior, podemos usar el siguiente método:

1. Primero, obtener una definición exacta del problema.
2. A continuación, determinar el tamaño del problema completo a resolver. Así se determinarán los valores de los parámetros en la llamada inicial al procedimiento o función.
3. Tercero, resolver el caso base en el que problema puede expresarse no recursivamente. Esto asegurará que se cumple el punto 1 del test anterior.
4. Por último, resolver correctamente el caso general, en términos de un caso más pequeño del mismo problema (una llamada recursiva).

Cuando el problema tiene una definición formal, posiblemente matemática, como el ejemplo del cálculo del factorial, el algoritmo deriva directamente y es fácilmente implementable en otros casos debemos encontrar la solución más conveniente.

Cómo funcionan los Algoritmos Recursivos

Para entender cómo funciona la recursividad es necesario que tengamos presente las reglas y los tipos de pasaje de parámetros provistos por C.

Si un procedimiento o función p invoca a otro q, durante la ejecución de q se reservarán locaciones de memoria para todas las variables locales de q y para los parámetros pasados por valor. Al terminar la ejecución de q este espacio es desocupado. Ahora bien, si durante la ejecución de q se produce una llamada a sí mismo, tendremos una segunda “instancia” de q en ejecución, la primera instancia se suspende hasta que la instancia recursiva termine. Antes de iniciarse la ejecución recursiva de q, se ocupan nuevas locaciones de memoria para las variables locales y parámetros por valor de q. Cualquier referencia a estas variables accederá a estas locaciones. Las locaciones reservadas durante la ejecución inicial de q son inaccesibles para la 2da. instancia de q.

Cuando la 2da. instancia de q termine, el control vuelve a la primera instancia de q, exactamente al lugar siguiente a la llamada recursiva. Cualquier referencia a las variables locales o parámetros por valor de q accederá a las locaciones reservadas inicialmente, inaccesibles para la segunda instancia de q.

Como vemos, no se conoce la cantidad de memoria que va a utilizarse al ejecutar un procedimiento o función recursivo sino que se produce una asignación dinámica de memoria, es decir, a medida que se producen instancias nuevas, se van “apilando” las que quedan pendientes: se ponen al menos tres elementos en la pila:
una para la dirección de vuelta, otra para el/los parámetro/s formal/es y otra para el identificador de la función que en esencia es un parámetro pasado por variable.
Cuando la última instancia de la recursión - elige el caso base- se cumple, se “desapila” esa instancia y el control vuelve a la instancia anterior; así hasta “desapilar” todas las instancias. Esta “pila” que se va generando en memoria es importante tenerla en cuenta por lo que debemos asegurarnos que el algoritmo recursivo no sea divergente. Vamos a ver cómo funciona la recursión en algunos problemas vistos a través de una traza dónde iremos guardando los valores de los parámetros formales.

```
long factorial(int p)
{
  if (p == 0)
    return 1; //(caso base)
  else
    return p * factorial(p - 1); //(caso general)
}
```

Si la llamada inicial es factorial(5). La primera instancia se coloca en la pila, luego a medida que se va invocando la función, vamos apilando el valor del parámetro formal que se envía:

Como n=0, el algoritmo va por la rama de la condición verdadera llegando al caso base obteniendo factorial(0) = 1	0	Instancia 6
	1	Instancia 5
	2	Instancia 4
	3	Instancia 3

4	Instancia 2
5	Instancia 1
N	

Ahora desapilamos y vamos reemplazando el resultado obtenido en la instancia posterior:

1- Desapilo Instancia 6	~~0~~	~~factorial(0)~~	1
2- Desapilo Instancia 5	~~1~~	~~1 * factorial(0)~~	1 * 1 = 1
3- Desapilo Instancia 4	~~2~~	~~2 * factorial(1)~~	2 * 1 = 2
4- Desapilo Instancia 3	~~3~~	~~3 * factorial(2)~~	3 * 2 = 6
5- Desapilo Instancia 2	~~4~~	~~4 * factorial(3)~~	4 * 6 = 24
6- Desapilo Instancia 1	~~5~~	~~5 * factorial(4)~~	5 * 24 = **120**
	N	n* factorial(n-1)	**factorial**