Funciones y punteros

Utilidad de una función

La función es un recurso que permite la modularización de un programa. A través de ella se puede descomponer un programa complejo en un conjunto de subprogramas (funciones) más simples, lo que facilita la construcción del programa.

Otras ventajas:

- Disminuir la cantidad de líneas de código, ya que la función se define una sola vez y se la puede invocar cuantas veces se necesite.
- Disminuir la cantidad de errores, ya que evita la duplicación de código. Una buena pista para detectar la necesidad de crear una nueva función, es justamente, encontrar duplicación de líneas de código en distintas partes del programa. La solución es extraer este código y "encerrarlo" dentro de una función. Luego, en reemplazo de este código se invoca a la función para que realice la tarea que antes realizaba el código extraído.
- Reducir la complejidad del programa ("divide y vencerás").
- Eliminar código duplicado.
- Limitar los efectos de los cambios (aislar aspectos concretos).
- Ocultar detalles de implementación (p.ej. algoritmos complejos).
- Promover la reutilización de código (p.ej. familias de productos).
- Mejorar la legibilidad del código.
- Facilitar la portabilidad del código.

Definición

Conjunto de instrucciones que realizan una tarea específica. En general toman valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno; aunque tanto unos como el otro pueden no existir.

Al igual que con las variables, las funciones pueden declararse y definirse. Una declaración es simplemente una presentación, una definición contiene las instrucciones con las que realizará su trabajo al función.

Una vez definida, una función puede invocarse. En general, la definición de una función se compone de las siguientes secciones:

```
<tipo de retorno> <nombre_función> ( <lista de parámetros formales> )
{
     <intrucciones>
     return <valor>
}
```

En general, la invocación de una función se compone de las siguientes secciones:

```
<variable receptora> = <nombre_función> (<lista de parámetros actuales>);
```

Si la función tiene valor de retorno, o :

```
<nombre_función> ( < lista de parámetros actuales > );
```

Si la función no tiene valor de retorno.

Una vez definida una función, la misma puede *invocarse* (usarse) en diferentes partes del programa, para que realice una determinada tarea. En cada invocación, es de esperar que la función arroje diferentes resultados, de acuerdo con los valores de los parámetros (actuales) usados. Por ejemplo:

`int suma(int a, int b)`
`{`
`  int respuesta;`
`  respuesta = a + b;`
`  return respuesta;`
`}`
	tipo de retorno: **int**
	
	nombre de función: **suma**
	
	lista de parámetros formales:
	int a
	int b
		instrucciones:
		int respuesta;
		respuesta = a + b;
		return respuesta;
		variables locales:
		int respuesta;
		valor de retorno: **respuesta**

Se ve la definición de la función suma, que justamente, suma el valor de dos números enteros.

`int main()`
`{`
`  int p, q, r;`
`  p = 5;`
`  q = 10;`
`  r = suma(p, q);`
`  ...`
`  ...`
`}`
	variable receptora: **r**
	
	nombre de función: **suma**
	
	lista de parámetros actuales:
	p, q
		La función suma recibe los datos 5 y 10 a través de las variables p y q, que además son los parámetros actuales.
		
		Luego de realizar los cálculos necesarios retorna el valor (15) que es almacenado en la variable r.

1. El **tipo del valor de retorno**, que puede ser "void", si no necesitamos valor de retorno. Si no se establece, por defecto será "int". Aunque en general se considera de mal gusto omitir el tipo de valor de retorno.
2. El **nombre de la función**. Es costumbre, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo

leerlo. Cuando se precisen varias palabras para conseguir este efecto existen varias reglas aplicables de uso común. Una consiste en separar cada palabra con un "_", la otra, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo, si se hace una función que busque el número de teléfono de una persona en una base de datos, se puede llamar "busca_telefono" o "BuscaTelefono".

3. Una **lista de declaraciones de parámetros** entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable. Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía.

    a. **Parámetros formales:** están puestos en la definición de la función. Se declaran dentro de la lista de parámetros, estableciendo nombre y tipo para cada uno. En el momento de declararlos no contienen datos, están vacíos, y sirven a los fines de posibilitar la codificación del algoritmo de la función. Son la “puerta de entrada y a veces de salida” de los datos que necesita procesar la función.

    b. **Parámetros actuales:** se usan en la invocación de la función. Son variables del programa invocante (el que llama a la función) y se ubican reemplazando a los parámetros formales. Los parámetros actuales contienen datos (valores) que son usados por el cuerpo de la función para realizar los cálculos propios y generar (si hubiere) el valor de retorno.

    c. **Pasaje de parámetros:** existen dos maneras de usar los parámetros actuales, lo que determina el tipo de pasaje: pasaje por valor (o copia) y pasaje por referencia. En el primer caso (valor o copia) el valor original del parámetro actual “olvida” todo tipo de modificación que sufriera dentro de la función (a través del parámetro formal). En el primer caso, el valor original del parámetro actual puede “recordar” cualquier modificación realizada a través del parámetro formal. Ver el próximo apartado **Pasaje de parámetros.**

4. Un **cuerpo de función** que representa el código que será ejecutado cuando se llame a la función. El cuerpo de la función se encierra entre llaves "{}" Una función muy especial es la función "main". Se trata de la función de entrada, y debe existir siempre, será la que tome el control cuando se ejecute un programa en C.

Variables locales a una función

Una variable que se declara dentro de una función solamente es reconocida dentro de la misma, no existe fuera de los límites de la definición de la función.

Variables globales

Son aquellas que se declaran fuera de toda función. Son accesibles desde cualquier parte del programa. **NO DEBEN USARSE**. Las variables globales son una fuente errores difíciles de encontrar y corregir.

Punteros

Dirección de memoria

Si bien utilizamos nombres para identificar una variable y manipular datos, en realidad, a un nivel más cercano al hardware los datos se almacenan en direcciones de memoria. Las variables reemplazan a las direcciones de memoria y hacen el trabajo de la programación mucho más simple, dejando al compilador la traducción de “variables” a “direcciones de memoria”.
Toda variable tiene una dirección de memoria, y si bien es un valor que podemos conocer, no podemos asignarlo en forma arbitraria, sino a través de la invocación de funciones y operadores especiales.

Operador &

Este operador, aplicado a una variable retorna la dirección de memoria de la misma, siguiendo la siguiente sintaxis: `&idVariable` .

El valor retornado puede asignarse a otra variable (ver definición de puntero) o imprimirse por pantalla usando `%p` **.**

Definición de Puntero

Se denomina variable de tipo puntero a aquella que almacena como dato la dirección de memoria de otra variable, siguiendo la siguiente sintaxis: `tipo_dato * id_variable;`

La variable id_variable podrá almacenar la dirección de memoria de otra variable de tipo tipo_dato.
ejemplo:

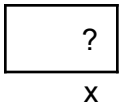
```
int * p;
```

Para poder acceder a la variable apuntada (variable cuya dirección de memoria almacena la variable de tipo puntero) se utiliza el mismo operador `*` .

Siguiendo el ejemplo anterior, `p` contiene la dirección de memoria de la variable `*p` (se dice “p apuntada”). Como se aclaró previamente, no se puede asignar a una variable puntero un valor arbitrario, o sea, las direcciones de memoria no pueden ser ingresadas por el programador en forma directa, no se pueden inventar. Cómo se hace entonces para obtener una dirección de memoria y así asignarla a una variable de tipo puntero?. Existen dos recursos:
1. usar el operador & (que retorna justamente la dirección de memoria de una variable existente).
2. usar la función malloc(), que busca una zona disponible de memoria, la reserva y retorna su dirección.

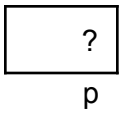
Utilizando el primer recurso, podemos realizar la siguiente composición:

int x;



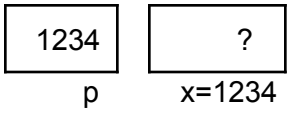
//se crea la variable entera **x**, cuyo primer valor no ha sido asignado.

int * p;



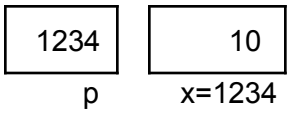
//se crea la variable de tipo “puntero a entero”, cuyo primer valor no ha sido asignado.

p = &x;



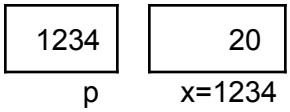
//se asigna a la variable puntero **p** la dirección de memoria de la variable **x.**

x = 10;



//se asigna a x el valor 10. Este valor puede ser accedido a través de la variable **p**, usando el operador ***p**.

***p = 20;**



//se asigna a la variable apuntada por **p** (en este caso **x**) el valor 20.

Pasaje de parámetros

Se realiza en el momento de la invocación de la función. Establece la forma en que el parámetro actual se vincula con el parámetro formal. Existen dos tipos:

Pasaje de parámetros por valor o copia (parámetros de entrada)

El parámetro formal copia el contenido del parámetro actual (pf = pa), por lo tanto cualquier modificación efectuada sobre el parámetro formal no se verá reflejada en el real. O sea, al terminar la función, las variables pasadas como parámetro recuperan sus valores originales.

Pasaje de parámetros por referencia (parámetros de entrada/salida)

Se utiliza una variable puntero como parámetro formal y el parámetro actual es una dirección de memoria. Lo que se copia ahora, es la dirección de memoria de la variable que oficia de parámetro actual. Por lo tanto, dentro de la función se puede acceder directamente al contenido de la variable a través del operador *. Se puede modificar el contenido de la dirección de memoria apuntada por el parámetro, pero no se puede modificar el parámetro.

Ejemplo de pasaje de parámetros por referencia:

```c
void intercambio(int *a, int *b)
{
     int aux;
     aux = *a;
     *a = *b;
     *b = aux;
}
```

```c
int main()
{
     int p = 10;
     int q = 20;
     intercambio(&p, &q);
     printf("p=%d, q=%d", p, q);
         // se muestra p=20, q=10;
}
```

De esta forma, se logra modificar el contenido de p y q, a través de *a y *b que son los contenidos de las memorias apuntadas por a y b.

No se modifican los valores de a y b, sino que modifican los contenidos de las variables apuntadas por a y b.

Cohesión

Medida del grado de identificación de un módulo con una función concreta.

Cohesión aceptable (fuerte)

Cohesión funcional (un módulo realiza una única acción).
Cohesión secuencial (un módulo contiene acciones que han de realizarse en un orden particular sobre unos datos concretos).
Cohesión de comunicación (un módulo contiene un conjunto de operaciones que se realizan sobre los mismos datos).
Cohesión temporal (las operaciones se incluyen en un módulo porque han de realizarse al mismo tiempo; p.ej. inicialización).

Cohesión inaceptable (débil)

Cohesión procedural (un módulo contiene operaciones que se realizan en un orden concreto aunque no tengan nada que ver entre sí).
Cohesión lógica (cuando un módulo contiene operaciones cuya ejecución depende de un parámetro: el flujo de control o “lógica” de la rutina es lo único que une a las operaciones que la forman).
Cohesión coincidental (cuando las operaciones de una rutina no guardan ninguna relación observable entre ellas).

Acoplamiento

Medida de la interacción de los módulos que constituyen un programa.

Niveles de acoplamiento (de mejor a peor):

Acoplamiento de datos (acoplamiento normal): Todo lo que comparten dos rutinas se especifica en la lista de parámetros de la rutina llamada.

Acoplamiento de control: Una rutina pasa datos que le indican a la otra rutina que hacer (la primera rutina tiene que conocer detalles internos de la segunda).

Acoplamiento externo: Cuando dos rutinas utilizan los mismos datos globales o dispositivos de E/S. Si los datos son de sólo lectura, el acoplamiento se puede considerar aceptable. En general, este tipo de acoplamiento no es deseable porque la conexión existente entre los módulos no es visible.

Acoplamiento patológico: Cuando una rutina utiliza el código de otra o altera sus datos locales ("acoplamiento de contenido").
La mayor parte de los lenguajes estructurados incluyen reglas para el ámbito de las variables que impiden este tipo de acoplamiento.

Objetivo

Reducir al máximo el acoplamiento y aumentar la cohesión de los módulos.

Pasos para escribir un subprograma

1. Definir el problema que el subprograma ha de resolver.
2. Darle un nombre no ambiguo al subprograma.
3. Decidir cómo se puede probar el funcionamiento del subprograma.
4. Escribir la declaración del subprograma (cabecera de la función).
5. Buscar el algoritmo más adecuado para resolver el problema.
6. Escribir los pasos principales del algoritmo como comentarios.
7. Rellenar el código correspondiente a cada comentario.
8. Revisar mentalmente cada fragmento de código.
9. Repetir los pasos anteriores hasta quedar completamente satisfecho.

El nombre de un subprograma
Procedimiento: Verbo seguido de un objeto.
Función: Descripción del valor devuelto por la función.
El nombre debe describir todo lo que hace el subprograma.
Se deben evitar nombres genéricos que no dicen nada (p.ej. calcular).

Se debe ser consistente en el uso de convenciones.

Los parámetros de un subprograma
Orden: (por valor, por referencia) == (entrada, entrada/salida, salida)

Si varias rutinas utilizan los mismos parámetros, éstos han de ponerse en el mismo orden (algo que la biblioteca estándar de C no hace).
No es aconsejable utilizar los parámetros de una rutina como si fuesen variables locales de la rutina.
Se han de documentar las suposiciones que se hagan acerca de los posibles valores de los parámetros.
Sólo se deben incluir los parámetros que realmente necesite la rutina para efectuar su labor.
Las dependencias existentes entre distintos módulos han de hacerse explícitas mediante el uso de parámetros.