Laboratorio de Arquitectura de Redes

Funciones en lenguaje C

Funciones en lenguaje C

- Definición de funciones
- Declaración de funciones
- Relación entre variables y funciones
  - Parámetros formales
  - Argumentos
- Llamadas a funciones
- Salida de una función
- Argumentos de la función `main()`
- Funciones recursivas
- Punteros a funciones
- Declaraciones complejas

Introducción (I)

- Las **funciones** son los bloques de sentencias que constituyen los programas en lenguaje C. En ellas se desarrolla toda la actividad de los programas
- Cada función constituye un bloque de código y datos discreto, privado, independiente e indivisible.
  - Una función sólo tiene acceso a las variables globales y a sus propias variables
  - Desde el exterior, a una función sólo se puede acceder por su entrada
  - Es el equivalente de las subrutiras o los procedimientos en otros lenguajes de programación

Introducción (II)

- Todos los programas en lenguaje C constan, al menos, de una función: la función `main()`
  - El programa comienza su ejecución por el comienzo de la función `main`, independientemente de dónde se localice en el archivo fuente
- Para aumentar la portabilidad de los programas, las funciones deben
  - Ser genéricas
  - Recibir la información a través de sus parámetros
  - No utilizar variables globales

Introducción (III)

- Ejemplo: Programa que lee un conjunto de números y obtiene el máximo, el mínimo y la media aritmética:

```
#include <stdio.h>
#define N 10
main()
{
   int max, min, med, listnum[N];
   Leedatos(listnum, N);
   max = Maximo(listnum, N);
   min = Minimo(listnum, N);
   med = Media(listnum, N);
   printf("Máximo: %d, Mínimo: %d, Media: %d",
          max, min, med);
   return 0;
}
```

Definición de funciones (I)

- No es posible definir una función dentro de otra función
- La **definición** de una función, según el estándar ANSI, incluye la cabecera y el cuerpo de la función

```
tipodev nombrefuncion(Lista de parámetros)
{       /* Cuerpo de la función */
        Declaraciones de datos
        Código ejecutable (sentencias);
        Expresiones opcionales de retorno;
}
```

  - `tipodev` indica el tipo de dato válido que la función devuelve al punto de llamada (por omisión es `int`)
  - `nombrefuncion` es el identificador o nombre de la función

Definición de funciones (II)

- La *lista de parámetros* representa una relación (opcional) de identificadores de variables precedidas por el tipo de dato y separadas por comas, denominadas **parámetros formales**
  - Operan como variables locales dentro de la función
  - Recogen los datos que se le pasan a la función
  - La lista de parámetros tiene el siguiente formato:

```
tipo1 iden1, tipo2 ident2, … tipoN identN
```

    - `tipoX` representa cualquier tipo de dato válido
    - `identX` representa a los identificadores de las variables

Definición de funciones (III)

- Ventajas de la utilización de funciones
  - El código se agrupa y se organiza en “compartimentos estancos”
  - Los datos quedan aislados
  - Es más fácil la localización de errores
  - Es posible probar partes del código (funciones)
  - Se ahorra trabajo: Las funciones bien diseñadas pueden ser útiles en diferentes aplicaciones
- Desventajas
  - El código fuente puede parecer más largo
  - En ejecución, la llamada y el retorno de una función requiere un tiempo adicional

Definición de funciones (IV)

- Ejemplo: Definición de una función que recibe una lista de números y devuelve el mayor

```
int Maximo(int *lista, int numdat)
{
   int i, maximo;
   maximo = lista[0];
   for (i=0 ; i<numdat ; i++)
      if (maximo<lista[i]) maximo=lista[i];
   return maximo;
}
```

Declaración de funciones (I)

- La **declaración** de una función *describe la función*
- Declarar una función consiste en escribir una sentencia de código que
  - Tipo de dato que devuelve
  - Identificador
  - Parámetros que puede recibir
  - Finaliza con el símbolo punto y coma «;»
- La línea escrita en la declaración de la función también se llama **prototipo de la función**
- Formato:

```
tipdev nombrefunc (listaparámetros);
```

Declaración de funciones (II)

- `tipodev` representa el tipo de dato que al función devuelve
- `nombrefunc` es el identificador asignado a la función
- `listaparámetros` es la declaración de cada uno de los parámetros formales de la función precedido por su tipo de dato
  - Van separados por comas
  - El identificador es opcional. Es aconsejable porque facilita la verificación y localización de errores
  - Si se trata de una función que no recibe argumentos, no tendrá parámetros y se declarará expresamente de tipo `void`

Declaración de funciones (III)

- Los prototipos de las funciones utilizadas en un programa
  - Se deben colocar antes de la primera llamada a la función
  - Se aconseja ponerlos al comienzo del programa, precediendo a la función `main`.
  - Sirven para “avisar” al compilador de la presencia de la función y de sus características (identificador, tipo devuelto, parámetros que recibe)
  - Impiden los errores en el envío de datos en las llamadas a funciones
    - Por los tipos de datos
    - Por el número de parámetros
- Si no se declara el prototipo de una función, en las llamadas a la misma se promociona, por omisión, de `char` a `int` y de `float` a `double`. Si existe prototipo, se respetan los tipos

Declaración de funciones (IV)

- Es posible declarar funciones con un número indeterminado de parámetros, lo cual se indica mediante tres puntos (puntos suspensivos «...») en la lista de parámetros declarados.
  - Para ello debe haber, al menos, un parámetro definido antes de los puntos suspensivos
- Ejemplo: Declaraciones válidas de la función que recibe una lista de números y devuelve el mayor de todos ellos:

```
int Maximo();
int Maximo(int *, int);
int Maximo(int *, ...);
int Maximo(int *lista, int numdat);
      /* Es recomendable la
         utilización de la última */
```

Relación entre variables y funciones (I)

- **Variables locales** a una función
  - Se declaran en la propia función (opcionalmente con el modificador `auto`)
    - Son desconocidas fuera de la función. Sólo existen mientras se estén ejecutando instrucciones de la función
    - Se guardan en una zona de memoria temporal (la pila o stack)
    - No conservan su valor entre llamadas a la función, salvo si se declaran expresamente como `static`

Relación entre variables y funciones (II)

- **Parámetros formales** de una función
  - Son variables locales a la función que, además, reciben los argumentos que se envían a la función en cada llamada.
  - Se declaran en la definición de la propia función
    - Tienen las mismas características que las variables locales
    - Sus tipos deben coincidir con los tipos de los datos que se envían a la función cuando esta es llamada
  - Son **argumentos de una función** los valores que inicializan los parámetros formales en la llamada a la función (Son los valores que se "pasan" a la función en su llamada)

Relación entre variables y funciones (III)

- **Variables globales** al programa
  - Son las variables que se declaran fuera de todas las funciones
    - Se recomienda que se declaren justo antes de la función `main` (deben declararse antes de su utilización)
    - Son accesibles desde cualquier punto del programa
    - Se almacenan en una memoria que "pertenece" al programa durante todo el tiempo que dure su ejecución
    - Inicialmente toman valores nulos
  - Debe evitarse su uso porque
    - Las funciones que las usan son menos portables y genéricas
    - Pueden alterarse desde cualquier punto del programa, lo que puede dar lugar a "interferencias"
    - Suponen una ocupación permanente de memoria y un mayor tamaño de los programas

Llamadas a funciones (I)

- Una **llamada a una función** se realiza escribiendo en el código fuente el nombre de la función, incluyendo entre paréntesis los argumentos necesarios
- Los argumentos se pueden pasar a la función de dos modos
  - **Por valor**
    - Los argumentos se copian en los correspondientes parámetros formales
    - Los cambios que se realicen dentro de la función no afectarán a las variables usadas en la llamada

      (Si a una función se le pasa el valor contenido en una variable, la función no podrá alterar el contenido de esa variable)
  - **Por referencia**
    - Los argumentos que se pasan a la función son direcciones de variables (punteros)
    - En la función podrán cambiarse los valores contenidos en las variables apuntadas por los argumentos

      (Si a una función se la pasa la localización en memoria de una variable, la función podrá alterar el contenido de esa memoria y, por lo tanto, el contenido de la variable)

Llamadas a funciones (II)

- Ejemplo: La función Maximo() de prototipo `int Maximo(int *lista, int numdat)`
  - Recibe
    - En `lista`, la dirección de un array de números enteros
    - En `numdat` el número de datos en el array
  - Devuelve un número entero: el mayor del array
  - Tras la llamada

    ```
    max=Maximo(array, ndatos);
    ```

    - No habrá cambiado en valor de `ndatos`
    - No habrá cambiado el valor de `array`
    - Podría haber cambiado el valor de los elementos apuntados por `array` (`array[0]`, `array[1]`, `...`)
    - Cambiará el valor de la variable `max`

Llamadas a funciones (III)

- Se dice que una **función recibe un array** cuando recibe la dirección del array (un puntero al array)
  - No recibe una copia de todos los elementos del array, sólo una copia de la dirección
  - Podrá manipular sin restricciones todos los elementos
  - Deberá conocer las dimensiones del array
    - Si se trata de un array unidimensional,deberá conocer los límites del array
      - Si es una cadena, por el carácter nulo
      - Mediante una variable que contenga el número de elementos
    - Si se trata de un array multidimensional
      - Deberá conocer el número total de elementos o deberá conocer las dimensiones
      - Podrá omitirse el tamaño de la primera dimensión si conoce el número total de elementos

Llamadas a funciones (IV)

- Cuando una **función recibe un puntero** se dice que recibe un dato *por referencia*
  - Recibe el puntero por valor
  - Recibe el dato apuntado por referencia
- Ejemplo

```
int dat1, dat2, resul, *punt;
dat1 = 5;   /* Inicializo los datos */
dat2 = 10;
punt = &dat1;     /*Inicializo el puntero */
resul=Func1(punt, dat2);
```

  - La función `Func1()` podrá cambiar el valor contenido en `dat1` pero no podrá cambiar ni el valor de `punt` ni el valor contenido en `dat2`

Llamadas a funciones (V)

- Las estructuras y uniones pueden pasarse a las funciones como cualquier otro tipo de dato y de variable:
  - Cuando se pasan por valor, se pasa una copia
    - Si se trata de estructuras grandes y complejas, ralentiza los programas y aumenta el tamaño de la memoria necesaria
  - Cuando se pasan por referencia
    - Al pasar sólo la dirección, la llamada a la función es un proceso rápido
    - La función puede alterar los valores contenidos en la variable original

Salida de una función (I)

- La sentencia `return` permite salir de una función y regresar al punto en el que fue llamada

  `return` *`expresion;`*

  - `expresion` representa el valor que devolverá al punto en el que fue llamada
    - Debe corresponder en tipo con el tipo devuelto por la función
    - Si la función es de tipo `void`, no debe existir
  - Puede aparecer en cualquier punto y más de una vez
- La llave de cierre «}» de la función es también el punto de finalización de la función y retorno al lugar de la llamada
- Por omisión en su definición, el tipo de vuelto por una función es `int`. Para que una función devuelva otro tipo debe indicarse expresamente en la definición y en el prototipo

Salida de una función (II)

- El tipo devuelto por una función puede ser un *puntero* a cualquier tipo de dato válido
- Declaración de una función que devuelve un puntero

  `tipodato * nombrefuncion(listaparametros);`
  - `tipodato` es cualquier tipo de dato válido
  - El dato apuntado debe seguir existiendo al finalizar la función
  - Cuando se trata de estructuras y uniones, es recomendable la devolución de punteros
  - Mediante un puntero se puede devolver también un array
- La función `exit()` fuerza la finalización del programa,
  - Independientemente del punto de ejecución en el que se encuentre
  - Devuelve el control al sistema operativo
  - Está definida en el archivo `STDLIB.H`

Argumentos de la función `main()`(I)

- La función `main()` puede recibir argumentos y devolver un valor
  - Puede intercambiar información con el sistema operativo
    - Recibe argumentos de la línea de órdenes
    - Devuelve un valor al sistema operativo
- Prototipo

  ```
  int main(int argc, char *argv[]);
  ```

  - `int` indica que devuelve un entero (por defecto)

Argumentos de la función `main()`(II)

- `argc` y `argv[]` son los parámetros opcionales definidos por el estándar ANSI para recibir los argumentos
  - Esos nombres son los utilizados normalmente, pero pueden utilizarse otros
  - `argc` es un entero que indica el número de argumentos presentes en la línea de órdenes, considerando al nombre del programa como primer argumento
  - `argv` es el identificador de un vector de punteros a carácter (vector y de punteros a cadenas)
    - Cada elemento de ese vector apunta a uno de los argumentos de la línea de órdenes (`argv[0]` apunta al nombre del programa, `argv[1]` al siguiente, `...`)
    - El separador de argumentos en la línea de órdenes es el espacio (salvo si están entre dobles comillas)

Argumentos de la función `main()`(III)

- A la función `main()` le llegan tantas cadenas de caracteres como conjuntos de caracteres separados por espacios haya en la línea de órdenes
- Ejemplo: Si se teclea el comando cp fuese un programa escrito en lenguaje C, al escribir en la línea de comandos

  `cp –a` *`archivo_origen archivo_destino`*

  en la función `main()`del programa tendríamos:
  - `argc=4`
  - `argv[0]=“cp”`
  - `argv[1]=“-a:”`
  - `argv[2]=“`*`archivo_origen`*`”`
  - `argv[3]=“`*`archivo_destino`*`”`

Funciones recursivas (I)

- Se llama **recursividad** a la posibilidad de que una función pueda llamarse a sí misma
  - Las *funciones recursivas* suelen llevar alguna instrucción condicional que las finalice
  - Los niveles de recursividad deben limitarse expresamente o estar limitados por definición en el algoritmo, a un número finito y pequeño
  - Cuando una función se llama a sí misma
    - La ejecución previa queda en suspenso
    - Todos los parámetros de la ejecución previa se almacenan en memoria
    - Debe producirse un retorno sucesivo
- Al programar funciones recursivas debe tenerse presente que
  - Las variables `auto` y `register` se inicializan en cada llamada
  - Las variables `static` sólo se inicializan la primera vez que se ejecuta la función

Funciones recursivas (II)

- Ventajas de las funciones recursivas
  - Permiten, en algunos casos, resolver complejos problemas de inteligencia artificial
  - Pueden crear, a veces, versiones más claras y sencillas de algunos algoritmos
  - Se adaptan mejor a la forma de pensar de algunos programadores en determinadas situaciones
- Desventajas de las funciones recursivas
  - No reducen código ni la memoria utilizada
  - Con frecuencia el programa resultante es más lento
  - Sobrecargan de datos la pila corriendo el riesgo de llegar a rebosarla
  - Suelen resultar difíciles de entender

Funciones recursivas (III)

- Ejemplo: programa que muestra los números naturales hasta el número introducido por teclado (I)

```
#include <stdio.h>
void Presenta (int num);
                    /* Prototipo de la función */
main()
{
   int n;
   printf("Introduce el número: ");
   scanf("%d", &n);
   fflush(stdin);
   Presenta(n);    /* Llamada a función con el número
                      introducido */
   return 0;
}
```

Funciones recursivas (IV)

- Ejemplo: programa que muestra los números naturales hasta el número introducido por teclado (II)

```
void Presenta(int num)   /* Función recursiva */
{
   if (num==1) printf ("%d\t", num);
       /* Si num == 1 se imprime y termina */
   else
   {
      Presenta(num-1);  /* Si num!=1 decrementa
                           num y se llama a sí misma */
      printf("%d\t", num);
   }
}                    /* Al volver de las llamadas se
                        imprimen los números */
```

Declaraciones complejas (I)

- La combinación de
  - El operador *puntero a* «*»
  - Los corchetes indicadores de *array* «[]»
  - Los *paréntesis* «()» que agrupan operaciones o nos indicadores de función

  dan lugar a declaraciones complejas y difíciles de descifrar
- Para interpretar correctamente las declaraciones
  1. Se debe comenzar con el identificador, mirando a su derecha
     - Los paréntesis indicarán que es una función
     - Los corchetes indicarán que es un array
  2. Mirar si a la izquierda hay un asterisco lo que indicará que es un puntero
  3. Aplicar las reglas anteriores a cada nivel de paréntesis y de dentro hacia fuera

Declaraciones complejas (II)

- Ejemplos

```
int (*lista)[20];     /* lista es puntero a un
                         array de 20 enteros */
char *datos[20];      /* datos es un array de 20
                         punteros a carácter */
void (*busc)();       /* busc es un puntero a una
                         función que no devuelve nada*/
char (*(*Fun())[])(); /* Func es una función que devuelve
                         un puntero a un array de
                         punteros a funciones que
                         devuelven cada una un carácter*/
int (*(*tim[5])())[3];  /* tim es un array de 5 punteros
                           a funciones que devuelven cada
                           una un puntero a un array de
                           3 enteros */
```