Funciones

Introducción

Las funciones son un conjunto de instrucciones que realizan una tarea específica. En general, toman ciertos valores de entrada, llamados parámetros y proporcionan un valor de retorno. En C, ambos son opcionales y pueden no existir.
Las funciones sirven para optimizar el código y modifican el paradigma de la programación, es un paso imprescindible para luego introducirse a la programación orientada a objetos.

Declaración y definición de funciones

Como ya es bien sabido, se tiene que declarar o definir una variable antes de poder utilizarla. Esto también se aplica a las funciones. En C, debe declarar una función antes de que pueda ser invocada.

Declaración en comparación con definición

Según el estándar ANSI, la declaración de una variable o función especifica la interpretación y atributos de un conjunto de identificadores. Por otra parte, la definición requiere que el compilador de C reserve espacio de almacenamiento para una variable o función nombrada por un identificador.

Una declaración de variable es una definición, pero una declaración de función no lo es

Una declaración de función hace referencia a una función que está definida en otra parte y especifica qué tipo de valor devuelve la función y qué tipos de datos recibe. Una definición de función define lo que hace la función, además de dar el número y tipo de argumentos que recibe la función.

Una declaración de función no es una definición de función

Si se coloca una definición de función en su archivo fuente antes de llamarla por primera vez, no necesita declarar la función. En caso contrario, se deberá declarar la función antes de invocarla.

Ejemplo:

```
#include <iostream>
using namespace std;
void HolaMundo(void){
   cout << "Hola mundo !";
}
int main(){
   HolaMundo();
   return 0;
}
```

T06CF01.cpp

Salida:

```
Hola mundo!
```

En el ejemplo TC06CF01.cpp se puede observar que la definición de la función HolaMundo() está por encima al primer llamado de la misma, por lo tanto, no fue necesaria la declaración.

Ejemplo:

```
#include <iostream>
using namespace std;
void HolaMundo(void);
int main(){
   HolaMundo();
   return 0;
}
void HolaMundo(void){
   cout << "Hola mundo !";
}
```

T06CF02.cpp

Salida:

```
Hola mundo!
```

En el ejemplo TC06CF02.cpp se puede observar que a diferencia de TC06CF01.cpp la definición de la función se encuentra por debajo de la llamada a la misma (en este caso en main). Para hacer que funcione de ésta manera, es necesaria la declaración de la función como se puede observar en la línea 3.

Formas erróneas más comunes:

```
#include <iostream>
using namespace std;
int main(){
   HolaMundo();
   return 0;
}
void HolaMundo(void){
   cout << "Hola mundo !";
}
```

Aquí se invoca a la función antes de la definición y no se la ha declarado previamente.

```
#include <iostream>
using namespace std;
void HolaMundo(void);
int main(){
   HolaMundo();
   return 0;
}
```

Aquí se declara la función pero no se la define.

Forma general de una declaración de función

```
tipo_de_dato_retorno nombre_funcion(tipo_de_dato_parámetro_1,
tipo_de_dato_parámetro_N);
```

Ejemplos:

```
int login(char *, char *);
float suma(float, float);
int esPrimo(int);
void borrarPantalla(void);
```

Forma general de una definición de función

```
tipo_de_dato_retorno nombre_funcion (tipo_de_dato_parámetro_1 nombre_parámetro_1,
tipo_de_dato_parámetro_N nombre_parámetro_N){
        .
        .
        .
}
```

Ejemplos:

```
void borrarPantalla(void){
  system("cls");
}
```

```
float suma(float n1, float n2){
  return n1+n2;
}
```

En las *formas generales de*... que se acaban de mencionar, cabe destacar que tanto en el valor devuelto como la lista de argumentos pueden ser vacías. Esto quiere decir, que no devuelve ningún valor o que no recibe ningún valor. En éstos casos se lo especifica con la palabra reservada void.

Cómo hacer llamadas a funciones

Como se muestra en T06CF03.cpp, cuando se hace una llamada a una función, la ejecución del programa salta a la función y termina la tarea asignada a ésta. Luego, la ejecución del programa continúa en la instrucción siguiente después de que regresa de la función que se llamó.
Una llamada a función es una expresión que se puede utilizar como una sola instrucción o dentro de otras instrucciones.

Ejemplo:

```
#include<stdio.h>
int funcion_1(int, int);
double funcion_2(double x, double y){
   printf("Dentro de funcion_2\n");
   return x - y;
}
int main(void){
```

```
   int x1 = 80;
   int y1 = 10;
   double x2 = 100.123456;
   double y2 = 10.123456;

   printf("Se pasa a la funcion_1 %d y %d.\n", x1, y1);
   printf("funcion_1 devuelve %d.\n", funcion_1(x1, y1));
   printf("Se pasa a la funcion_2 %f y %f.\n", x2, y2);
   printf("funcion_2 devuelve %f.\n", funcion_2(x2, y2));
   return 0;
}
/* definicion de funcion_1() */
int funcion_1(int x, int y){
   printf("Dentro de funcion_1\n");
   return x + y;
}
```

T06CF03.cpp

Salida:

```
Se pasa a la funcion_1 80 y 10.
Dentro de funcion_1
funcion_1 devuelve 90.
Se pasa a la funcion_2 100.123456 y 10.123456.
Dentro de funcion_2
funcion_2 devuelve 90.000000.
```

La finalidad del programa T06CF03.cpp es mostrar cómo declarar y definir funciones. La instrucción de la línea 2 es la declaración de funcion_1 y en la línea 3 se encuentra la definición de funcion_2 que finaliza en la línea 6. Luego en la línea 14, se hace el llamado a funcion_1. Vale la aclaración de que tanto funcion_1 y funcion_2 son llamados dentro de la función printf, lo que significa que primero se ejecutarán las funciones de más adentro hasta llegar a la primera, funcion_1 es ejecutada primero lo cual devolverá como resultado un entero que será utilizado por printf para mostrarlo por pantalla. Lo mismo ocurre con funcion_2 en la línea 16. Por último, en la línea 20 se encuentra la definición de funcion_1.

Pasar la dirección de memoria de una variable a una función

Es posible enviarle a una función la dirección de memoria de una variable, con esto permitimos que la función pueda modificar el contenido de la variable enviada en la dirección donde fue creada.

Ejemplo:

```
#include<iostream>
using namespace std;
void ejemplo(int *, int);

int main(void){
   int cambia=10, no_cambia=1;
   cout << "Valores de las variables antes de enviarlas a ejemplo(): " << endl;
   cout << "cambia: " << cambia << endl;
   cout << "no_cambia: " << no_cambia << endl;
   ejemplo(&cambia, no_cambia);
```

```
   cout << "Valores de las variables luego de enviarlas a ejemplo(): " << endl;
   cout << "cambia: " << cambia << endl;
   cout << "no_cambia: " << no_cambia << endl;
   return 0;
}

void ejemplo(int *x1, int x2){
   *x1 = -256;
    x2 = -128;
    cout << "Valores de las variables en ejemplo(): " << endl;
    cout << "cambia (x1): " << *x1 << endl;
    cout << "no_cambia (x2): " << x2 << endl;
}
```

T06CF04.cpp

Salida:

```
Valores de las variables antes de enviarlas a ejemplo():
cambia: 10
no_cambia: 1
Valores de las variables en ejemplo():
cambia (x1): -256
no_cambia (x2): -128
Valores de las variables luego de enviarlas a ejemplo():
cambia: -256
no_cambia: 1
```

Lo que se puede observar en T06CF05.cpp, es que en la línea 3 la declaración de la función ejemplo() recibe dos enteros, pero el primero de ellos tiene la diferencia que tiene un asterisco. Esto es así porque es un puntero. De modo que, como vimos anteriormente, ésta variable se caracteriza por almacenar una dirección de memoria. En la línea 10 podemos observar como se envía la dirección de memoria de una variable, se hace anteponiendo un & (ampersand) a la variable. Y en la línea 17, efectivamente se declaran dos variables: x1 (puntero a entero) y x2 (entero) y se copian los dos valores desde main: (una dirección de memoria, supongamos 0x22ff74 y el valor entero 1).
De x2 como habíamos visto anteriormente podemos obtener el valor simplemente a partir del nombre de la variable osea *x2*, y su dirección de memoria a partir de *&x2.*
Sin embargo, de x1 podemos obtener tres valores.
Con x1 accedemos a la dirección de memoria que aloja la variable (la dirección de memoria en main de la variable *cambia*) supongamos 0x22ff74.
Con *x1 accedemos al contenido de la dirección de memoria que está alojando x2 (0x22ff74) en este caso el valor 10.
Con &x1 obtenemos la dirección de memoria en la función ejemplo() del puntero x2, ya que al ser una variable tiene su propia dirección de memoria, supongamos 0x22ff50.

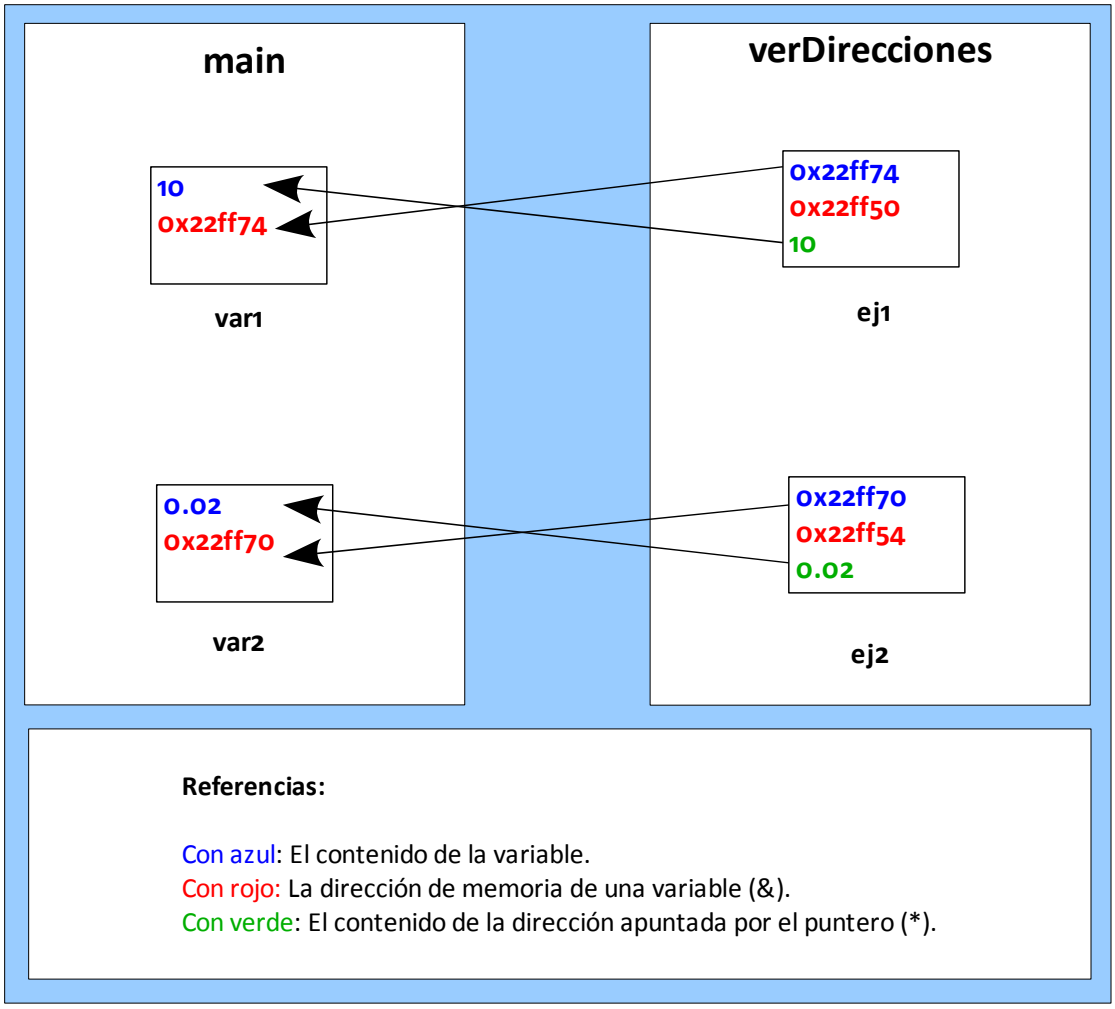
En el ejemplo T06CF06.cpp se hace principal hincapié en las direcciones de memoria para ejemplificarlo mejor:

```
#include<iostream>
using namespace std;
void verDirecciones(int *, float *);
int main(void){
   int var1 = 10;
```

```
   float var2 = 0.02;
   cout << "*** Direccion de memoria y valor en main: (antes de llamar a verDirecciones) ***" <<
endl << endl;
   cout << "DIR var1: " << &var1 << endl << "CONTENIDO var1: " << var1 << endl <<endl;
   cout << "DIR var2: " << &var2 << endl << "CONTENIDO var2: " << var2 << endl << endl;
   verDirecciones(&var1, &var2);
   cout << "*** Direccion de memoria y valor en main:(luego de llamar a verDirecciones) ***" <<
endl << endl;
   cout << "DIR var1: " << &var1 << endl << "CONTENIDO var1: " << var1 << endl << endl;
   cout << "DIR var2: " << &var2 << endl << "CONTENIDO var2: " << var2 << endl << endl;
   return 0;
}

void verDirecciones(int *ej1, float *ej2){
   cout << "*** Direccion de memoria y valor en verDirecicones: ***" << endl << endl;
   cout << "DIR ej1: " << &ej1 << endl <<"CONTENIDO ej1 (dir de var1 en main): " << ej1 << endl <<
"CONTENIDO DE DONDE APUNTA ej1 (valor de var1 en main): " << *ej1 <<endl << endl;
   cout << "DIR ej2: " << &ej2 << endl << "CONTENIDO ej2 (dir de var2 en main): " << ej2 << endl
<< "CONTENIDO DE DONDE APUNTA ej2 (valor de var2 en main): " << *ej2 << endl << endl;
   /* Cambiamos los valores de donde apuntan ej1 y ej2 */
   *ej1 = -40;
   *ej2 = 1.22;
}
```

To6CF05.cpp

```
*** Direccion de memoria y valor en main: (antes de llamar a verDirecciones) ***


DIR var1: 0x22ff74
CONTENIDO var1: 10

DIR var2: 0x22ff70
CONTENIDO var2: 0.02

*** Direccion de memoria y valor en verDirecicones: ***

DIR ej1: 0x22ff50
CONTENIDO ej1 (dir de var1 en main): 0x22ff74
CONTENIDO DE DONDE APUNTA ej1 (valor de var1 en main): 10

DIR ej2: 0x22ff54
CONTENIDO ej2 (dir de var2 en main): 0x22ff70
CONTENIDO DE DONDE APUNTA ej2 (valor de var2 en main): 0.02

*** Direccion de memoria y valor en main:(luego de llamar a verDirecciones) ***

DIR var1: 0x22ff74
CONTENIDO var1: -40

DIR var2: 0x22ff70
CONTENIDO var2: 1.22
```



Dibujo 1 – Representación gráfica del ejercicio anterior.

Ejercicio práctico:

Supongamos que desde main queremos pasar a una rutina dos variables y que el valor de éstas se intercambien entre sí y que además, los valores de las variables permanezcan intercambiados en main cuando retorne la ejecución de la función.
En principio, no sería posible ya que una rutina puede devolver sólo un valor. Para ello sería necesario enviar la dirección de memoria de las variables.
Enviando la dirección de memoria de una variable a una función, le proveemos a la misma la posibilidad de modificar directamente el valor de la variable. Para ello necesitamos utilizar un puntero.

Primero vamos a resolverlo si no utilizaríamos funciones:

```
#include<iostream>
using namespace std;
int main(void){
   int x1, x2, aux;
   x1 = 10;
   x2 = 1000;
   cout << "X1: " << x1;
```

```
    cout << endl << "X2: " << x2;
    cout << endl << endl;
    aux = x1;
    x1 = x2;
    x2 = aux;
    cout << "X1: " << x1;
    cout << endl << "X2: " << x2;
    return 0;
}
```

T06CF06.cpp

Ejercicio sugerido:

Realizar una función que permita intercambiar el valor de dos variables de tipo de dato entero y que sus valores permanezcan intercambiados una vez terminada la ejecución de la función.

Bibliografía:

- Zhang, Tony, Aprendiendo C en 24 horas, Pearson Educación, México, 2001, ISBN: 968-444-495-8.