Repaso de punteros y funciones

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar
drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Informática II - Clase número 1 - Ciclo lectivo 2023

Agenda

Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

Prototipo de funciones

Consiste en una presentación de la función. En el se define que tipo de dato que la función retorna y si lo hace, el nombre de la misma y el tipo de dato de lo/los parámetros que recibe. En caso de ser mas de un parámetro, se los separa por comas.

Ejemplos de prototipos:

```
1  float   promedio (int , int);
2  int     menu (void);
3  void    saludo (void);
4  void    imprimir (int, float, char);
```

De forma general:
tipo_dato_a_devolver nombre_funcion (tipo_datos_de_argumentos)

Invocación o llamada a funciones

Como ya se ha visto con las funciones para entrada y salida de datos (printf y scanf), una función se invoca dando el nombre de la función y transmitiéndole datos como argumentos, en el paréntesis que sigue al nombre de la función.

```
1  printf                    ("String %d,%d",n1,n2)
2  nombre_de_la_funcion      (datos_transmitidos)
3
4  print_max(n1,n2);
```

Es importante aclarar que para este último ejemplo, la función print_max recibe **una copia** del valor almacenado en las variables n1 y n2.

Definición de una función I

Una función se define cuando se escribe. Cada función es definida sólo una vez en un programa y puede ser usada por cualquier otra función.

```
1  encabezado_funcion
2  {
3      cuerpo;
4      declaracion_de_variables;
5      instrucciones_de_c;
6  }
```

Definición de una función II

```
1  void print_max(int n1, int n2)
2  {
3      if(n1>n2)
4      {
5          printf("%d es mayor que %d",n1,n2)
6      }
7      else
8      {
9          printf("%d es mayor que %d",n2,n1)
10     }
11
12 }
```

Los nombres de los argumentos en el encabezado se conocen como **parámetros formales.**

Ejemplo I

Diseñar un programa que solicite al usuario ingresar dos números enteros desde el teclado y luego llame a la función print_max. Dentro de esta función, se imprimirá el valor más grande de los dos números sin realizar ningún retorno explícito. Luego de la ejecución de la función, la función main debe imprimir un mensaje indicando que el programa ha terminado.
Luego de la escritura del programa se debe:

- Analizar el prototipo de la función y cada una de sus partes
- Analizar la llamada a la función
- Reconocer e indicar el encabezado y cuerpo de la función
- Identificar el comportamiento del programa luego de la ejecución de una función

▸ Ver en git (function_full_example)

Ejemplo: llamada a una función sin argumentos I

Modificar el programa anterior para que el mensaje final sea impreso mediante una función que no devuelve ni recibe parámetros. Luego de la escritura del programa se debe:

- Analizar el prototipo de la nueva función y cada una de sus partes
- Analizar la llamada a la nueva función
- Reconocer e indicar el encabezado y cuerpo de la nueva función

▸ Ver en git (function_full_example)

La sentencia return I

Una función en C puede ser capaz de recibir múltiples argumentos, pero sólo es capaz de devolver un único valor a la función que llama.

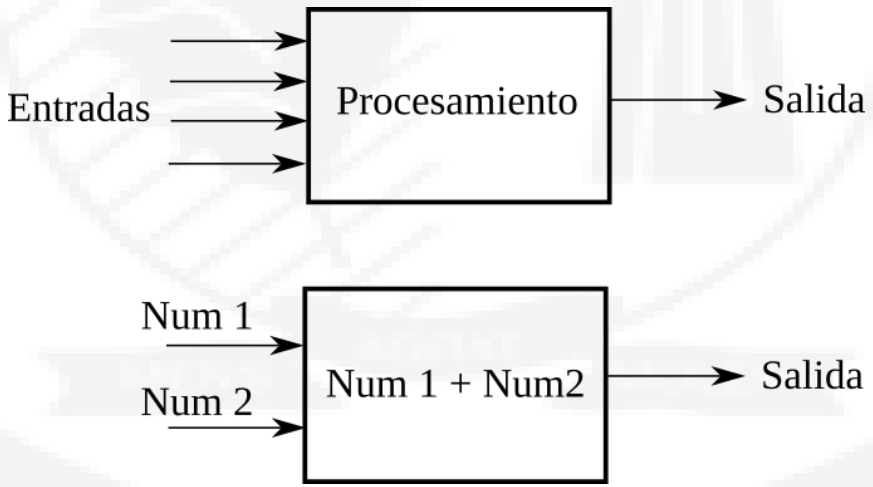

Figure: Diagrama en bloques de una función

La sentencia return II

```
1  float promedio (int n1, int n2)
2  {
3     float resultado=0;
4     resultado=(n1+n2)/2.0;
5     return(resultado)
6  }
```

Es importante mencionar que para que el retorno de datos de una función funcione correctamente, debe respetarse la interfaz entre la función llamada y la que llama se maneje de forma correcta.
Para ello:

```
1  float promedio (int, int);
```

La sentencia return: ejemplo I

Diseñar un programa que solicite al usuario ingresar dos números enteros desde el teclado y luego llame a la función promedio. Dentro de esta función, se realizará el cálculo del promedio de ambos números. El valor del cálculo debe ser impreso en main. Luego de la escritura del programa se debe:

- Analizar el prototipo de la función y cada una de sus partes
- Analizar la llamada a la función
- Reconocer e indicar el encabezado y cuerpo de la función

▸ Ver en git (function_return_full_example)

Reglas de ámbito I

Alcance

Se define al término alcance como la sección del programa donde un identificador, como el nombre de una variable es conocido.

Variables locales (alcance local)

Las variables creadas dentro de la función llevan este nombre puesto a que están disponibles y son accesibles sólo para la función en cuestión. Es decir, se puede repetir el nombre de una variable entre diferentes funciones puesto a que no hay relación entre ellas.

Reglas de ámbito II

Variables globales (alcance global)

Una variable con alcance global, en general denominada variable global, se declara fuera de cualquier función. Estas variables pueden ser utilizadas por todas las funciones que se colocan físicamente después de la declaración de la variable global. **Su uso NO es considerado una buena práctica de programación.**

Reglas de ámbito III

```c
/* Variables Locales*/
#include <stdio.h>
void func2(void);

int main ()
{
  /* Variables locales a Main */
  int a=10, b=20;
  printf ("Impresion en main\n");
  printf ("Valores de a = %d, b = %d\n", a, b);
  func2();
  printf ("Impresion en main\n");
  printf ("Valores de a = %d, b = %d\n", a, b);
  return 0;
}
```

Reglas de ámbito IV

```
16
17  void func2 ()
18  {
19    /* Variables locales a func2 */
20    int a=70, b=89;
21    a++;
22    b++;
23    printf ("Impresion en Func2\n");
24    printf ("Valores de a = %d, b = %d\n", a, b);
25  }
```

Reglas de ámbito V

```c
/* Variables globales*/
#include <stdio.h>
void func2(void);
int a=1, b=2;
int main ()
{
  /* Variables locales a Main */
  a=10, b=20;
  printf ("Impresion en main\n");
  printf ("Valores de a = %d, b = %d\n", a, b);
  func2();
  printf ("Impresion en main\n");
  printf ("Valores de a = %d, b = %d\n", a, b);
  return 0;
}
```

Reglas de ámbito VI

```
16
17  void func2 ()
18  {
19    a++;
20    b++;
21    printf ("Impresion en Func2\n");
22    printf ("Valores de a = %d, b = %d\n", a, b);
23  }
```

Mecanismo de paso de argumentos a funciones I

Paso por valor

El valor del argumento es **copiado** en el parámetro de la subrutina, por lo cual si se realizan cambios en el mismo dentro de la función, el valor original no es modificado.

Diseñar y codificar un programa que reciba como parámetro un número entero, al cual se le sumará tres y luego se retornará el valor a la función llamante: Luego de la codificación, se debe::

- ¿Qué recibe efectivamente la función?
- ¿Puede modificar el contenido original?
- ▸ Ver en git (pointer_and_functions_value)

Mecanismo de paso de argumentos a funciones II

Paso referencia

Se copia la *dirección de memoria* del argumento como parámetro de la función. En este caso, al realizar cambios en parámetro formal este si se ve afectado.

Modificar el ejercicio anterior para que la función reciba al número entero por referencia.

- ¿Qué recibe efectivamente la función?
- ¿Puede modificar el contenido original?
- ▸ Ver en git (pointer_and_functions_reference)

Introducción a los punteros

¿Por qué es importante estudiar punteros?

- Proporcionan los medios por los cuales las funciones pueden modificar sus argumentos de llamada
- Son el soporte para el manejo dinámico de memoria y estructura de datos dinámicas, tales como:
  - Pilas
  - Listas
  - Árboles binarios
- El buen uso de los punteros puede mejorar la eficiencia de ciertas rutinas

Los punteros son una herramienta muy poderosa de C, pero también una de las mas peligrosas. El manejo incorrecto de un puntero, podría causar serios problemas en el sistema.

Definición

¿Qué es un puntero?

Es una variable que contiene una dirección de memoria. En general, esa dirección es la posición de memoria donde otro objeto está almacenado. Si una variable contiene la dirección de otra variable, se dice que la primera *apunta* a la segunda.

Declaración de variables punteros:

```
1 /*Puntero a entero*/
2 int *dato=NULL;
3 /*Puntero a float*/
4 float *dato=NULL;
5 /*Puntero a char*/
6 char *dato=NULL;
7 /*Puntero a puntero*/
8 int **dato=NULL;
```

Operadores de punteros I

Operador de dirección &

Para desplegar la dirección donde una variable está almacenada, C nos provee el operador de dirección (&), el cual significa "la dirección de".

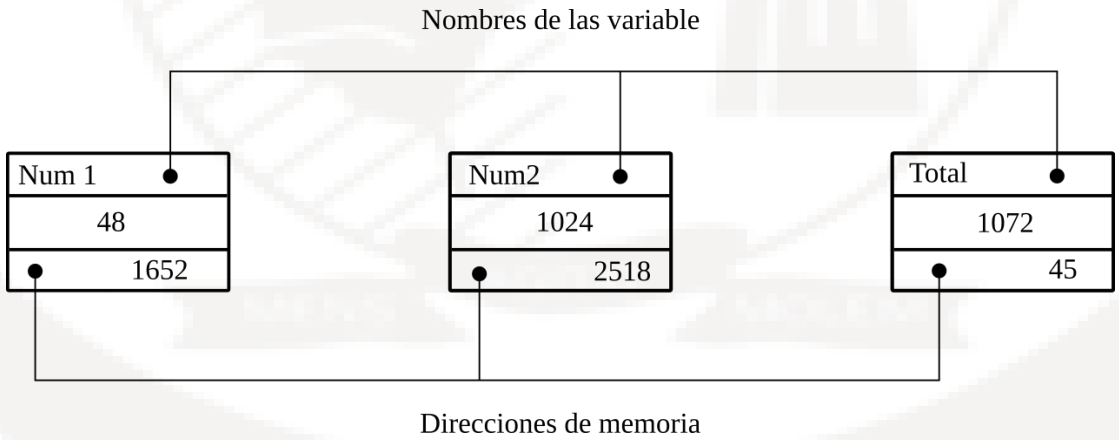

Figure: Declaración de variables

Operadores de punteros II

Diseñar un programa que declare y asigne dos variables entera. Luego, se debe declarar una tercer variable que contenga la suma de las dos anteriores.
Finalmente se debe imprimir la dirección de memoria de cada una de las variables involucradas. Luego de la escritura del programa se debe:

- Analizar los operadores involucrados
- Distinguir las direcciónes de memorias impresas
- ▸ Ver en git (variable_direction)

Operadores de punteros III

Operador de indirección *

Cuando el símbolo * es seguido de una variable puntero, significa: "la variable cuya dirección de memoria está almacenada en". En otras palabras, devuelve el valor del objeto al que apunta su operando.

Diseñar un programa que declare y asigne una variable entera. Luego declare otra variable de tipo **puntero a un entero** y sea asignada con la dirección de memoria de la primera. Luego de la escritura del programa se debe:

- Analizar los operadores involucrados
- Distinguir las direcciónes de memorias impresas
- ▸ Ver en git (variable_indirection_concept)

Operadores de punteros IV

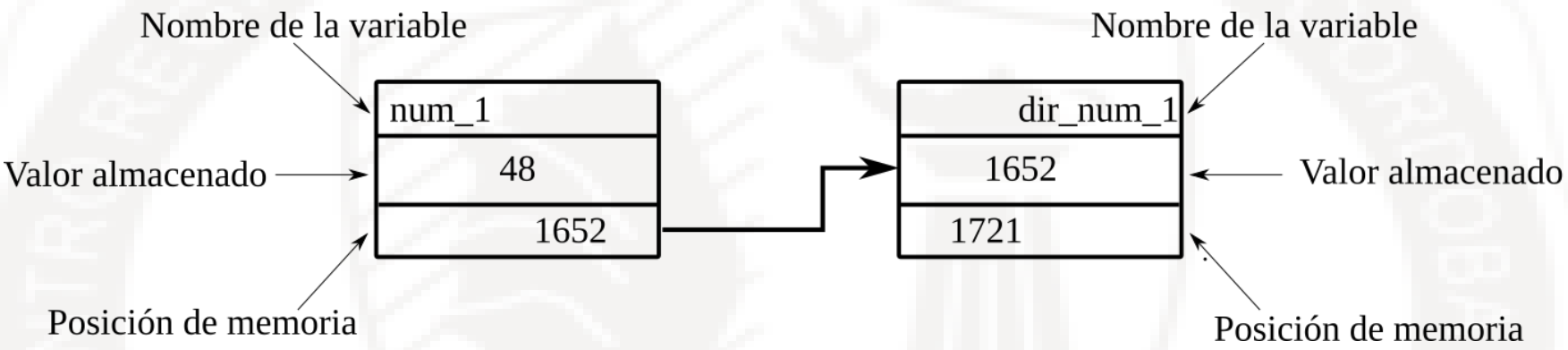

Figure: Uso de punteros

Operadores de punteros V

Diseñar un programa que declare y asigne una variable entera. A continuación, se deberá declarar otra variable de tipo puntero a un entero y asignarle la dirección de memoria de la primera variable. Posteriormente, utilizando la variable de tipo puntero, se modificará el valor de la primera variable.

- Analizar los operadores involucrados
- Distinguir las direcciones de memorias impresas
- ▸ Ver en git (variable_indirection)

Operadores de punteros VI

```
1  #include <stdio.h>
2  int main()
3  {   int num1=48;
4      int *dir_num_1;
5      dir_num_1=&num1;
6      printf("Valor de num1 %d\n",num1);
7      printf("Posicion de memoria de num1 %X\n",&num1);
8      printf("Valor de dir_num_1 %X\n",dir_num_1);
9      printf("Posicion de memoria de num1 %X\n",&dir_num_1);
10     *dir_num_1=10;
11     printf("Valor de num1 %d\n",num1);
12     return (0);
13 }
```

Apuntando doblemente a una variable

```
1  #include <stdio.h>
2  int main()
3  {    int num1=48;
4       int *p1=NULL;
5       int *p2=NULL;
6       p1=&num1;
7       p2=&num1;
8       printf("num1 %d\n",num1);
9       printf("& num1 %p\n",&num1);
10      printf("p1 %p\n",p1);
11      printf("p2 %p\n",p2);
12      *p2=10;
13      printf("num1 %d\n",num1);
14      *p1=90;
15      printf("num1 %d\n",num1);}
16      return(0);
17 }
```

¡Muchas gracias!

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar