Repaso de funciones
Casteo o conversión de tipos
Punteros parte I

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 2022

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 parámetros, 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

```c
#include<stdio.h>
void print_max(int,int); /*Prototipo*/

int main(void)
{
    int x,y;
    scanf("%d",&x);
    scanf("%d",&y);
    print_max(x,y); /*Llamada o Invocacion */
    printf("Gracias!\n");
    return(0);
}
```

Ejemplo II

```
16  void print_max(int n1, int n2) /*Encabezado*/
17  {
18      /*Cuerpo*/
19      if(n1>n2)
20      {
21          printf("%d es mayor que %d\n",n1,n2);
22      }
23      else
24      {
25          printf("%d es mayor que %d\n",n2,n1);
26      }
27
28  }
```

Ejemplo: llamada a una función sin argumentos I

```
1  #include<stdio.h>
2  void print_max(int,int);
3  void bye_bye  (void);
4
5  int main(void)
6  {
7      int x,y;
8      scanf("%d",&x);
9      scanf("%d",&y);
10     print_max(x,y);
11     bye_bye();
12     printf("Gracias!\n");
13     return(0);
14 }
15
```

Ejemplo: llamada a una función sin argumentos II

```
16
17
18
19  void print_max(int n1, int n2)
20  {
21      if(n1>n2)
22      {
23          printf("%d es mayor que %d\n",n1,n2);
24      }
25      else
26      {
27          printf("%d es mayor que %d\n",n2,n1);
28      }
29
30  }
```

Ejemplo: llamada a una función sin argumentos III

```
31
32
33
34  void bye_bye(void)
35  {
36      printf("Chau!\n");
37  }
```

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.

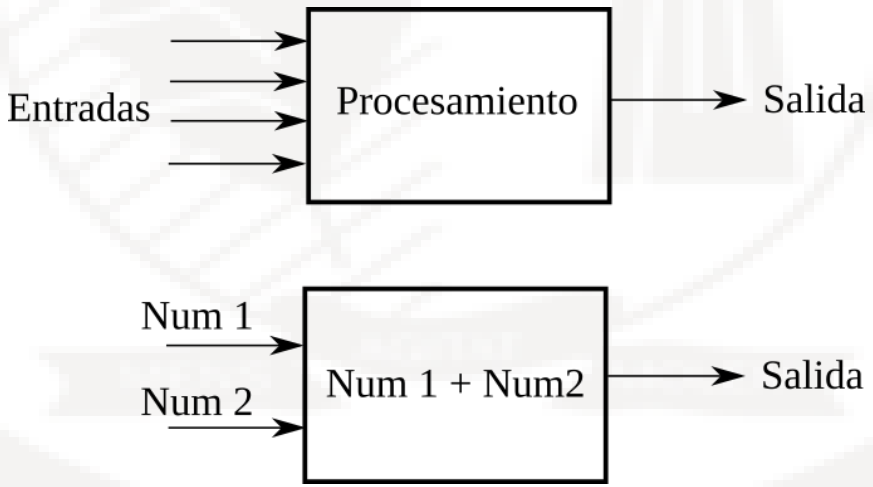

Figure: Diagrama en bloques de una función

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 III

```
1  #include<stdio.h>
2  float promedio(int,int);
3
4  int main(void)
5  {
6      int x,y;
7      float prom;
8      scanf("%d",&x);
9      scanf("%d",&y);
10     prom=promedio(x,y);
11     printf("Promedio vale %f",prom);
12     return(0);
13 }
14
15
```

La sentencia return IV

```
16
17
18  float promedio (int n1, int n2)
19  {
20    float resultado=0;
21    resultado=(n1+n2)/2.0;
22    return(resultado);
23  }
```

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.

```
1  #include<stdio.h>
2
3  /*PROTOTIPO*/
4  int suma_tres(int);
5
6  int main (void)
7  {
8     int n1=0,rdo;
9     printf("Ingrese un numero \n");
```

Mecanismo de paso de argumentos a funciones II

```
10      scanf("%d",&n1);
11      printf("N1 antes de llamar a suma_tres %d \n",n1);
12      rdo=suma_tres(n1);
13      printf("N1 despues de llamar a suma_tres %d \n",n1);
14      printf("Valor de resultado %d \n",rdo);
15        return(0);
16  }
17
18  int suma_tres (int num)
19  {
20     num=(num+3);
21     return(num);
22  }
```

Mecanismo de paso de argumentos a funciones III

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.

```
1  #include<stdio.h>
2
3  /*PROTOTIPO*/
4  int suma_tres(int *);
5
6  int main (void)
7  {
8     int n1=0;
9     printf("Ingrese un numero \n");
10    scanf("%d",&n1);
```

Mecanismo de paso de argumentos a funciones IV

```
11    int rdo;
12
13    printf("N1 antes de llamar a suma_tres %d \n",n1);
14    rdo=suma_tres(&n1);
15    printf("N1 despues de llamar a suma_tres %d \n",n1);
16    printf("Valor de resultado %d \n",rdo);
17    return(0);
18  }
19
20  int suma_tres (int *num)
21  {
22    int resultado;
23    resultado = *num + 3;
24    return(resultado);
25  }
```

Casteo: promoción de tipo

Cuando en una expresión se mezclan constantes y variables de distintos tipos, todo se convierte a un tipo único. El compilador convierte todos los operandos la tipo del mayor operando, esto se lo conoce como **promoción de tipo.**. Esta operación se realiza en etapas, teniendo en cuenta la presedencia de operadores y la aritmética involucrada.

```
1  char c;
2  int i;
3  float f;
4  double d;
5  resultado = (c/i) + (f*d) - (f+i);
6                (int) + (doble) - (float)
7                     (double)   - (float)
8                          (double)
```

Casteo o utilización de moldes I

Se puede forzar a que una expresión sea de un tipo determinado utilizando una construcción denominada **casteo**.

La estructura en C/C++ es:

```
1 (tipo) expresion;
```

donde tipo es un tipo de datos válido.
Es importante aclarar que C++ añade cuatro operadores más de casteo. Todos ellos se estudiarán más adelante.

Casteo o utilización de moldes II

```c
#include <stdio.h>
int main()
{
    float f=0;
    for(int ii=0;ii<10;ii++)
    {
        f=(float) ii/2;
        printf("%f\n",f);
    }
    return(0);
}
```

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".

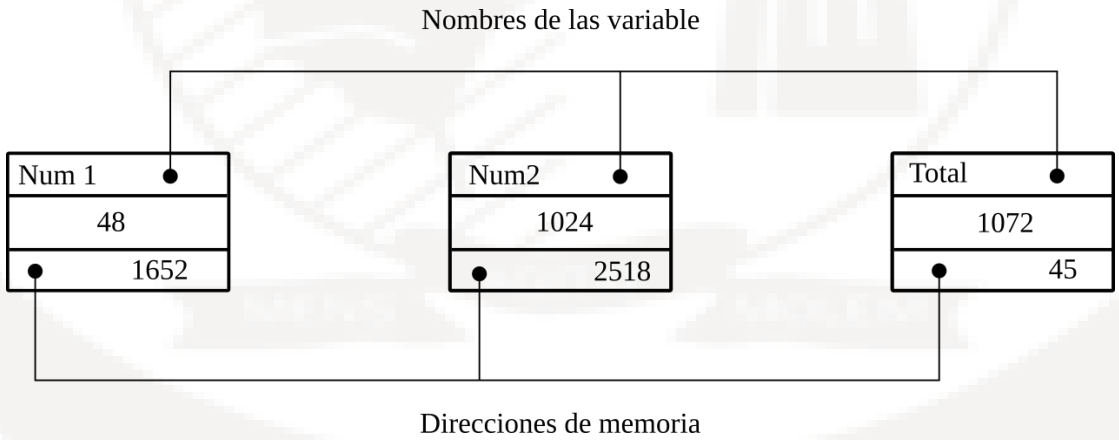

Figure: Declaración de variables

Operadores de punteros II

```
1  #include <stdio.h>
2  int main()
3  {   int num1=48;
4      int num2=1024;
5      int total=num1+num2;
6      printf("Valor de num1 %d\n",num1);
7      printf("Posicion de memoria de num1 %X\n",&num1);
8      printf("Valor de num2 %d\n",num2);
9      printf("Posicion de memoria de num2 %X\n",&num2);
10     printf("Valor de total %d\n",total);
11     printf("Posicion de memoria de total %X\n",&total);
12     return (0);
13 }
```

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.

```
1 #include <stdio.h>
2 int main()
3 {   int num1=48;
4     int *dir_num_1;
5     dir_num_1=&num1;
6     return(0);
7 }
```

Operadores de punteros IV

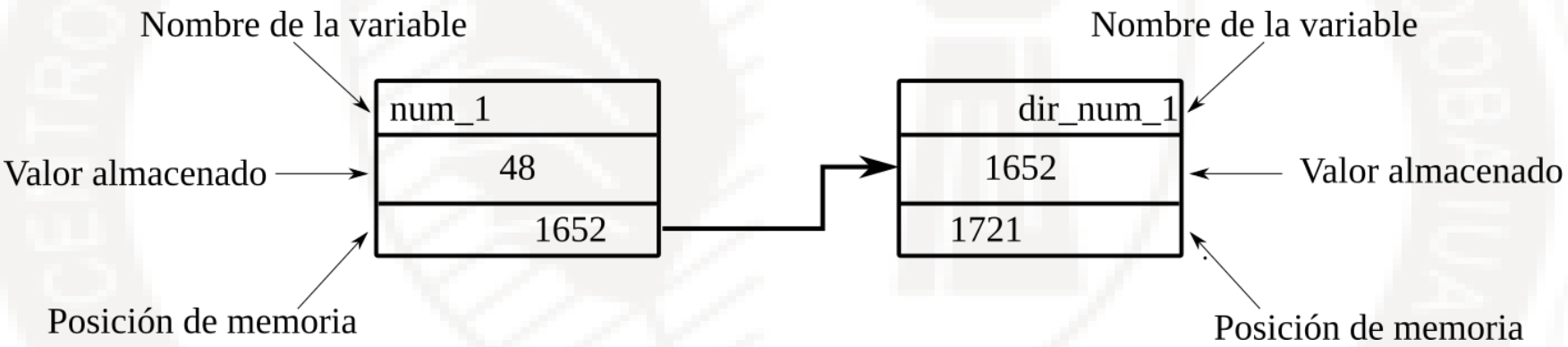

Figure: Uso de punteros

Operadores de punteros V

```
1 #include <stdio.h>
2 int main()
3 {   int num1=48;
4     int *dir_num_1;
5     dir_num_1=&num1;
6     *dir_num_1=10;
7     *dir_num_1++;
8     return(0);
9 }
```

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 }
```

Declaración y almacenamiento de variables I

Declaración de variables

En general, un programa necesita guardar y leer datos desde la memoria de la computadora. De forma conceptual y simplificada, se pueden pensar a las posiciones de memoria donde esto ocurre se como habitaciones de un hotel con un número de indentificación único e irrepetible.

Si graficamos cada declaración de variables:

Declaración y almacenamiento de variables II

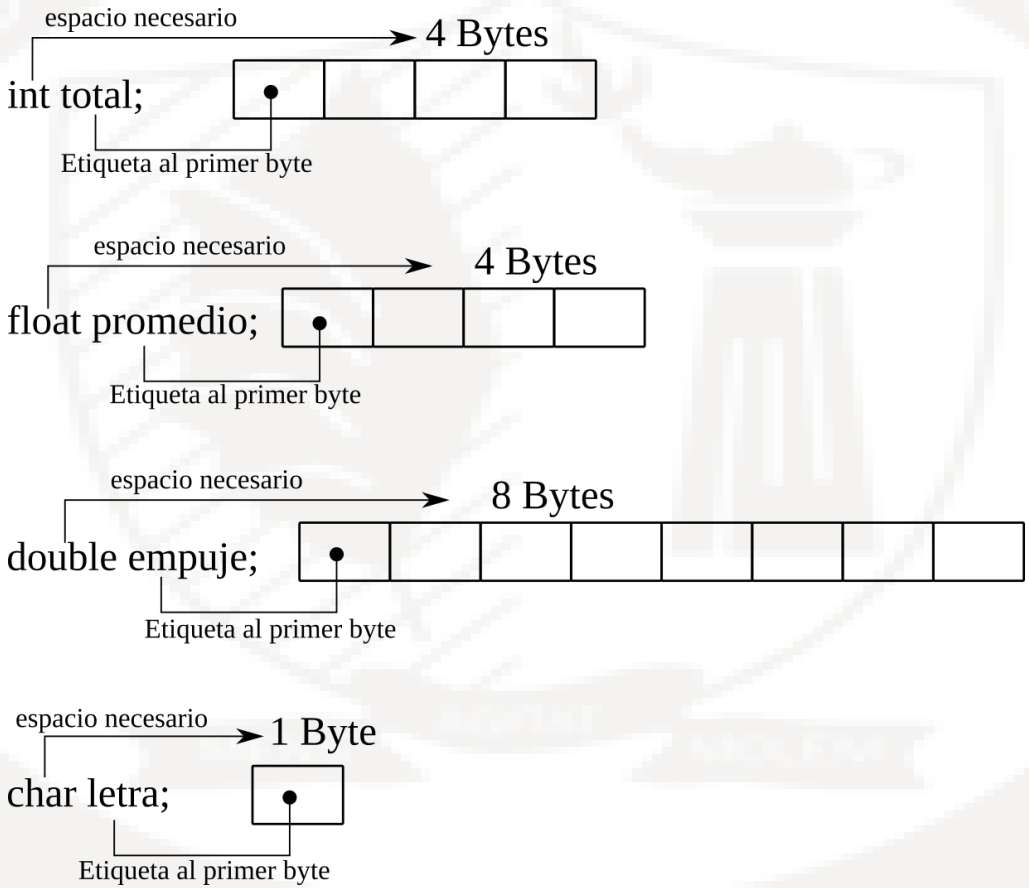

Figure: Diagrama en bloques declaración de variables.

Aritmética de punteros I

Existen sólo dos operaciones aritméticas que se pueden usar con punteros **la suma y la resta.**

Supongamos que la variable var1 está almacenada en la posición de memoria 1938:

```
1  int var1=10; //almacenado en 1938
2  int *p=&var1;
3  p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (4bytes) = 1942$$

y no 1939 como es pensando.

Aritmética de punteros II

Veamos otro ejemplo con double:

```
1 double var1=10.23; //almacenado en 1938
2 double *p=&var1;
3 p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (8bytes) = 1946$$

Aritmética de punteros III

Veamos otro ejemplo con char:

```
1 char var1="a"; //almacenado en 1938
2 char *p=&var1;
3 p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (1byte) = 1939$$

Cada vez que se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cuando se aplica a punteros de tipo "char", la aritmética es normal, ya que los caracteres ocupan un byte, **el resto de los punteros aumentan o decrecen en la longitud del tipo de dato a los que apuntan.**

¡Muchas gracias!

Consultas:

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