Punteros parte II
Casteo o conversión de tipos
Manejo de memoria dinámica 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 3 - 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

Arreglos multidimencionales y punteros I

Puede tenerse acceso a arreglos multidimencionales usando notación de punteros. La notación se vuelve mas compleja conforme aumentan las dimensiones del arreglo.

```
1  int nums[2][3]={{16,18,20},{25,26,27}};
```

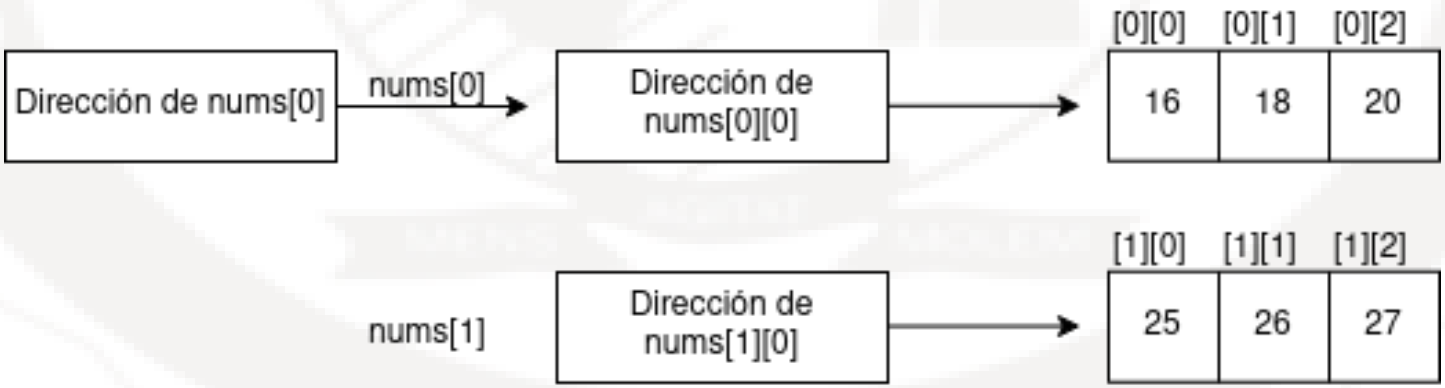

Figure: Arreglo de dos dimensiones y punteros.

Arreglos multidimencionales y punteros II

Notación de puntero	Notación de subíndice	Valor almacenado
*(*nums)	nums[0][0]	16
*(*nums +1)	nums[0][1]	18
*(*nums +2)	nums[0][2]	20
((nums+1))	nums[1][0]	25
((nums+1) +1)	nums[1][1]	26
((nums+1) +2)	nums[1][2]	27

La utilidad de esto se verá cuando se estudie el tema de creación de arreglos dinámicamente.

Arreglos multidimencionales y punteros III

Diseñar y codificar un programa que cargue e imprima un arreglo de enteros de NxM, utilizando aritmética de punteros. También imprimir las direcciones de memoria donde estan almacenados cada uno de los elementos. (En esta instancia sólo utilizar un nivel de indirección).
Luego de la escritura del programa se debe:

- Analizar los operadores involucrados
- Distinguir las direcciónes de memorias impresas
- ¿Cuantos bytes se incrementan al moverse en filas?
- ¿Cuantos bytes se incrementan al moverse en columnas?
- ¿Que sucede si la matriz es de tipo de dato char? ¿Que se debe cambiar?
- ▸ Ver en git (matrix_and_pointers)

Punteros a funciones I

Definición

Una función tiene una ubicación física en memoria que puede asignarse a un puntero, por lo tanto, un puntero a función es una variable que almacena la dirección de memoria de una función. Al igual que los punteros regulares, los punteros a función se usan para acceder y trabajar con el objeto (en este caso, una función) al que apuntan.

Declaración:

```
1  returnType (*pointerName)(paramType1, paramType2, ...);
```

Donde returnType es el tipo de dato que la función devuelve, y paramType1, paramType2, etc. son los tipos de parámetros que acepta la función.

Punteros a funciones II

Ejemplo:

```
1  void cubo(float x); //prototipo
2
3  void (*ptrAFunc) (float);   //puntero a funcion
4  ptrAFunc = cubo;            //Asignacion
5  ptrAFunc(x);                //Llamada
```

Punteros a funciones III

Los punteros a función son útiles en situaciones como:

- Selección dinámica de función: usar un puntero a función para seleccionar y llamar a una función específica en tiempo de ejecución, en lugar de usar una declaración condicional larga.
- Callbacks: se puede pasar un puntero a función como argumento a otra función, permitiendo que esa función llame de vuelta a la función apuntada.
- Implementación de tablas de funciones: es decir, se tiene un arreglo de punteros a función y se asigna un elemento de este arreglo basado a fin de ejecutar la función requerida.

Punteros a funciones IV

```c
#include<stdio.h>

void hola_mundo(void);
void chau_mundo(void);

int main(void)
{
    void (*ptr_to_func)(); //Puntero a funcion
    ptr_to_func = hola_mundo; //Puntero a hola mundo
    (*ptr_to_func)();    //Llamando
    ptr_to_func = chau_mundo;
    (*ptr_to_func)();
    return(0);
}

```

Punteros a funciones V

```
16
17  void hola_mundo(void)
18  {
19      printf("Hello IUA\n");
20  }
21
22  void chau_mundo(void)
23  {
24      printf("ByeBye IUA\n");
25  }
```

Punteros a funciones VI

Diseñar y codificar un programa que mediante 4 funciones diferentes pueda sumar, restar, multiplicar y dividir dos números enteros. Todas las funciones deben recibir los mismos parámetros formales y no devolver ningún valor.
Solicitar al usuario seleccionar la opción deseada e implementar la llamada utilizando punteros a función.
Luego analice:

- Operandos involucrados
- Sintaxis de los punteros a función
- ¿Cómo afecta la sintaxis el cambio de un parámetro formal?
- ▸ Ver en git (variable_direction)

Punteros a funciones VII

Callbacks

Pasar punteros a funciones como argumentos a otras funciones es una técnica poderosa que se utiliza para implementar callbacks y permitir que una función llame a otra función de manera dinámica. Aquí tienes un ejemplo simple en C que demuestra cómo puedes pasar punteros a funciones a otras funciones.

Punteros a funciones VIII

```c
#include <stdio.h>
// Funciones de operaciones matematicas
int suma(int a, int b) {
    return a + b;
}

int resta(int a, int b) {
    return a - b;
}

// Funcion que recibe un puntero a funcion como argument
void operacion(int (*pop)(int, int), int x, int y) {
    int resultado = pop(x, y);
    printf("Resultado: %d\n", resultado);
}
```

Punteros a funciones IX

```
16
17  int main() {
18      int a = 10, b = 5;
19
20      printf("Suma:\n");
21      operacion(suma, a, b);
22
23      printf("Resta:\n");
24      operacion(resta, a, b);
25
26      return 0;
27  }
```

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

```
1  #include <stdio.h>
2  int main()
3  {
4      int ii;
5      float f;
6      for(int ii=0;ii<10;ii++)
7      {
8          f=(float) ii/2;
9          printf("%f\n",f);
10     }
11     return(0);
12 }
```

Gestión dinámica de la memoria: introducción

La asignación dinámica de memoria significa que un programa puede solicitar o devolver memoria al sistema operativo en **tiempo de ejecución**. Es decir, mientras se está ejecutando.

Para realizar esto en C[1], se utilizan las funciones malloc y free [2], junto al operador sizeof.

[1] C++ usa otras alternativas mejoradas que se estudiarán luego.
[2] Muchos compiladores soportan otras, pero estas son las más importantes

Las funciones malloc & free I

Cada vez que la función malloc() es llamada, una porción de la memoria libre del sistema es asignada. Por el contrario, cada llamada a la función free() libera memoria para ser utilizada por otro programa o el sistema operativo.

El prototipo de la función malloc() es:

```
1  void * malloc(number_of_bytes);
```

Tras una llamada fructífera devuelve un puntero al primer byte de la región de memoria dispuesta en el montón.

```
1  char *p=NULL;
2  p = (char *) malloc(1000); /*obtener 1000 bytes*/
```

Las funciones malloc & free II

Como el segmento del montón no es infinito, luego de intentar asignar memoria se debe verificar que el puntero sea distinto de NULL. Eso significa que la asignación fue exitosa.

El prototipo de la función free() es:

```
1 void free(void *p);
```

En este ejemplo, se asume que el puntero *p fue asignado previamente por una llamada a malloc. Llamar a free con un argumento inválido, dañaría el sistema de asignación.

El operador size_of I

Es un operador unario que retorna la longitud en **bytes** de la variables precedida por el. A los fines prácticos, se puede tomar el valor retornado por este operador como equivalente a un entero sin signo).

El operador size_of II

```c
#include <stdio.h>
int main()
{
    int     a=10;
    float   b=10.30;
    char    c='a';
    double  d=10.189;

    printf("Espacio en bytes de a: %ld\n",sizeof(a));
    printf("Espacio en bytes de b: %ld\n",sizeof(b));
    printf("Espacio en bytes de c: %ld\n",sizeof(c));
    printf("Espacio en bytes de d: %ld\n",sizeof(d));
    return(0);
}
```

Gestionando memoria dinámica con malloc() I

```
1  #include <stdio.h>
2  #include <stdlib.h>
3   int main()
4   {
5       int      *p=NULL;
6       int cantidad=0,ii=0,valor=0;
7       printf("Ingrese la cantidad de elementos\n");
8       scanf("%d",&cantidad);
9      p=(int *) malloc(cantidad*sizeof(int));
10      if(p==NULL)
11      {
12          printf("No hay memoria disponible\n");
13          exit(1);
14      }
15      else
```

Gestionando memoria dinámica con malloc() II

```
16      {
17          for(ii=0;ii<cantidad;ii++)
18          {
19              printf("Ingrese el elemento %d\n",ii);
20              scanf("%d",&valor);
21              *(p+ii)=valor;
22          }
23
24          for(ii=0;ii<cantidad;ii++)
25          {
26              printf("%d\n",*(p+ii));
27          }
28      }
29      free(p);
30      return(0);}
```

La función realloc() I

La función realloc() cambia el tamaño de la memoria asignado previamente por una llamada a malloc().
El nuevo tamaño reservado de memoria puede ser mayor o menor al asignado previamente.

Protitotipo de realloc();

```
1  void * realloc(void *ptr, size_t size);
```

La función realloc() II

```c
#include <stdio.h>

#include <stdlib.h>

int main() {
   int * p = NULL;
   int ii=0;
   /*Solicitando espacio para 15 valores enteros*/
   p = (int *) malloc(15 * sizeof(int));
   /*Carga de datos*/
   for (ii = 0; ii < 15; ii++)
   {
      *(p + ii) = ii;
   }

```

La función realloc() III

```
16      printf("Start address %p\n", p);
17      /*Impresion de datos*/
18      for (ii = 0; ii < 15; ii++)
19      {
20          printf("%d\n", *(p + ii));
21      }
22      /*Redimension de la memoria*/
23      p = (int *) realloc(p, 25 * sizeof(int));
24      /*Carga de nuevos datos*/
25      for (ii = 0; ii < 10; ii++)
26      {
27          *(p + ii + 14) = ii;
28      }
29
30
```

La función realloc() IV

```
31      printf("Start address %p\n", p);
32    /*Impresion total*/
33      for (ii = 0; ii < 25; ii++)
34      {
35          printf("%d\n", *(p + ii));
36      }
37    /*Impresion de posiciones de memoria */
38      for (ii = 0; ii < 25; ii++)
39      {
40          printf("%p\n", (p + ii));
41      }
42      free(p);
43      return (0);
44  }
```

¡Muchas gracias!

Consultas:

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