Punteros

Pablo R. Ramis

Universidad Nacional de Rosario, Instituto Politécnico, Dto. de Informática,
prramis@ips.edu.ar,
WWW home page: http://informatica.ips.edu.ar

Resumen Cuando uno declara una variable, tipo de dato primitivo o estructura esta se crea en un espacio estático de memoria, no es modificable luego y depende del sistema operativo su liberación. De allí es la importancia del uso de forma dinámica de la memoria que permitiría incrementarse en la medida que se usa o necesita.
Para que podamos entender, la memoria esta organizada de la siguiente forma:

Figura 1. Memoria

En el *segmento de código* encontramos el código de los programas, en la *memoria estática* las variables globales o estáticas, y en la *pila o stack* las varibles locales y en el Heap las variables dinámicas.

1. Repaso de conceptos

1.1. Definición

Recordamos que la definición del puntero es una variable que contiene una dirección de memoria como dato.

$$< tipo > * < nombre_variable >$$

Es importante hacer notar que existe una dirección de memoria especial que es la *NULL*, para usarla hay que incluir la lib *stdlib.h*.

Otro dato importante es que cuando se declara un puntero de un tipo determinado de dato, se reserva la memoria para guardar una dirección de memoria, pero NO PARA EL DATO AL QUE SE APUNTA. La reserva para almacenar un puntero es siempre la misma independientemente al dato al que apunta...

```
/*...*/
char a = 'c';

char *ptrc;
char *ptri;
/*...*/
```

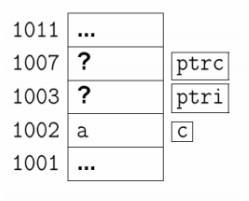

Figura 2. Representación de las declaraciones

1.2. Operadores

Los operadores & y * son fundamentales, el primero da la dirección de memoria de una variable, el segundo el contenido de un puntero.

```
/*...*/
int i;
int *ptr;
/* ...*/
 ptr = &i;
/*...*/
char c;
char *ptr;
/*...*/
ptr = &c;
*ptr = ''A; // Equivale a escribir: c = ''A
```

1.3. Errores comunes

- Asignar punteros de distintos tipos.
- Utilizar punteros no inicializados.
- Asignar valores al puntero y no a la variable a la que apunta.
- Intentar asignarle un valor al dato apuntado por un puntero cuando éste es NULL

2. Punteros a punteros

Intuitivamente podemos entender que es un puntero que contiene la dirección de memoria de otro puntero.

```
int a = 5;
int *p; // Puntero a entero
int **q; // Puntero a puntero
p = &a;
q = &p;

/* los accesos son como se explicaron antes
teniendo en cuenta que:
*q es un puntero a entero
**q es el entero
*/
```

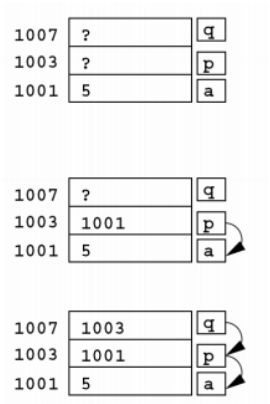

Figura 3. Puntero a puntero

3. Arrays Bidimensionales

$$< tipo > mat[DimF][DimC]$$

Donde *mat* es el nombre del array.

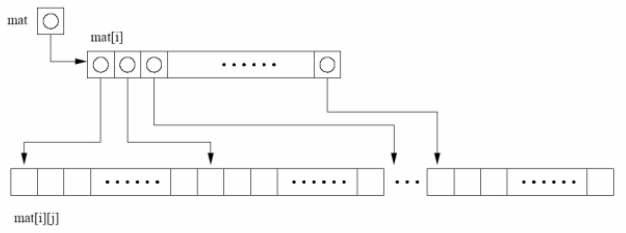

Figura 4. Array bidimensional

Al fijar un valor para en DimF puedo recorrer los datos guardados en DimC. Un ejemplo:

```c
/*bidimensional.c*/

#include<stdio.h>
#include<stdlib.h>

int main( )
{
    int matriz[10][10], x, y;

    for(x = 0; x < 10; x++) {
        for(y = 0; y < 10; y++) {
            matriz[x][y] = 1 + rand()%9;
            printf("%d ", matriz[x][y]);
        }
        printf("\n");
    }
    return 0;
}
```

```
$ ./bidimensional
2 8 1 8 6 8 2 4 7 2
6 5 6 8 6 5 7 1 8 2
9 9 7 7 9 9 9 5 2 2
6 1 1 4 6 4 2 8 5 8
7 1 1 3 6 5 6 3 3 4
3 2 2 9 9 1 6 6 5 5
7 1 6 7 3 9 8 4 5 3
```

```
1 1 1 1 3 7 3 6 7 6
8 7 7 9 6 4 7 3 9 2
7 7 9 1 2 2 8 1 4 3
$
```

En caso de estos arreglos, el concepto general es igual a los unidimensionales. El nombre ya es el puntero al inicio de la estructura generada en la memoria.

3.1. Pasaje por parámetros

Existe un problema importante al intentar pasar por parametro un array bidimensional, la segunda dimensión debe estar definida en el parametro de entrada:

void ImprimeMatriz(int m[][3], int filas)

Esto trae algunos complicaciones ya que se necesita conocer por anticipado el tamaño de esa dimensión. Eso puede llevar a errores o a sobredimensionar la estructura con riesgo igualmente a que sea insuficiente.

Veremos un poco mas adelante, cuando trabajemos dinámicamente la memoria que podremos reemplazar esta definición por un puntero a puntero. O sea, usaremos ***int * m*** en vez de ***int m[][]***

3.2. Punteros a punteros de punteros de punteros...

De la misma forma que el puntero a puntero, podemos seguir jugando con la misma situacion y generar un puntero a puntero de puntero, y asi sucesivamente... El proceso a llevar a cabo es el mismo, se deberan desreferenciar segun los niveles deseado para ir asignando segun se pretenda.

4. Punteros genéricos

Son de la siguiente forma:

void * vptr

Es un puntero capaz de recibir el valor de cualquier otro tipo de puntero.
Ejemplo:

```
1 int x = 1;
2 float r = 1.0;
3 void* vptr = &x;            // puntero-a-void
4                             // actualmente apuntando a int x
5 int main () {               // =============
6    *(int *) vptr = 2;       // M.1: modifica el valor de x
7    vptr = &r;               // actualmente apuntando a float r
8    *(float *)vptr = 1.1;    // M.3: modifica el valor de r
```

```
9
10      return 0;
11  }
```

Igual hay que tener en cuenta un par de restricciones:

- No se pueden asignar las direcciones de una constante. Ejemplo:

```
1   int const x;
2   void* p = &x;       // Error!
```

- Los punteros a void tampoco pueden ser asignados a punteros a constantes

```
1   int const x;
2   int const *p;
3   void *pv;
4   p = &x;
5   pv = p; //       Error!
```

Si bien para el primer caso existe una alternativa

```
1   int const x;
2   void* p = (void*) &x;    // Ok: usando cast...
3   void const* p = &x;      // Ok:
4   int const* p = &x;       // Ok: Esta es el modo formal
```

5. Gestión de memoria dinámica

Una de las formas mas flexibles para el uso de cadenas de caracteres o números es haciéndolo de forma dinámica.

Para esto, una vez definido el puntero, se reserva la memoria que se usará, esto el sistema lo hace en el *heap*.

La función básica que realiza la reserva de memoria se llama *malloc*.

void* malloc(size_t numero_de_bytes)

Como se ve, el retorno es un puntero a void por lo tanto, en el momento de usar dicha función es común que se realice el *cast* que sea necesario.

Ejemplo:

(char*) ptr = (char*)malloc(1000);

Como nos decía el prototipo, el tamaño de la reserva se debe hacer en bytes, asi que si vamos a reservar el espacio para 10 caracteres, se multiplicará por 10 al tamaño del *chat*. Si el tipo es una estructura en la cual no se sabe exactamente

que tamaño tiene se utiliza *sizeof*. Ejemplo tenemos un tipo T, usaremos *sizeof(T)* que retornará el tamaño

Si se recuerda, la reserva de memoria se realiza en el *heap*, esto implica que será nuestra responsabilidad el liberarla ya que ni el sistema ni el programa lo hará. Para esto se utiliza la función *free*.

void free (void* puntero);

Ejemplo:

free(ptr)

5.1. Ejemplos

Lectura de cantidad de números dinámicamente

```
#include<stdio.h>
#include<stdlib.h>

int leerCantidad() {
    int cantidad;
    printf("Cuantos numeros va a ingresas ?: ");
    scanf("%d", &cantidad);
    return cantidad;
}

void leer(int cantidad, int *numeros) {
    int i;
    for (i  = 0 ;i < cantidad; i++) {
        printf("Ingrese el numero %d: ", i+1);
        scanf("%d", numeros+i);
        printf("\n");
    }
}

void imprimir(int cantidad, int *numeros) {
    int i;
    printf("Los numeros ingresados son:\n");

    for (i = 0;i < cantidad; i++) {
       printf("%d ", *(numeros+i));
    }

    printf("\n");
}

int main(void) {
    int cantidad = leerCantidad();
    int* numeros = (int*) malloc(cantidad * sizeof(int));

```

```
35      if (numeros == NULL)
36          return -1;
37
38      leer(cantidad, numeros);
39      imprimir(cantidad, numeros);
40
41      free(numeros);
42      return 0;
43  }
```

Si bien el programa es muy simple en lo que realiza, nos muestra un par de detalles que pueden resultarnos muy útiles. No estoy necesitando generar un array de tamaño arbitrariamente grande por no saber la cantidad de datos a guardar, además, como hemos visto, si el malloc se hiciera en la fuunción que lee la cantidad, podria retornarse el puntero sin miedo a perder el dato, cosa que si ocurriria si fuera un arreglo (ver el tema del scope y el funcionamiento del segmento de stack y heap de la memoria).

Lectura de cadena de caraceteres

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include <string.h>
5
6   char* ingreso(){
7
8       char frase[256];
9       printf("Ingrese una frase: ");
10
11      scanf("%[^,'\n']", frase);
12
13      char* frase1 = (char*) malloc (strlen(frase));
14
15      strcpy (frase1, frase);
16
17      return frase1;
18  }
19
20
21  int main() {
22
23      char * frase = ingreso();
24
25      printf("Frase ingresada: %s\n", frase);
26
27      printf("Frase modificada: %s  - largo %d\n", frase, (int)
            strlen(frase));
28
```

```
29      free(frase);
30
31      return 0;
32  }
```

En este caso en particular, no muy diferente al ejemplo anterior, la lectura de la frase ingesada se realiza en una función, en ella, se guarda en un arreglo con capacidad de 256 caracteres (linea 8), esto no es un problema ya que esta variable será borrada al finalizar la función.

Notese que *scanf* (línea 11) posee una sintaxis particular, se le está dando la expresión regular *[^ ' n']* para que lea todo lo ingresado por teclado hasta el enter, si no se pusiera esto y solo usaramos el comúnmente %s solo tendriamos la primer palabra de la frase.

El *strlen* de la línea 13 donde se invoca el *malloc* es una función de librería que me retorna el largo de una cadena de caracteres y *strcpy* copia una cadena en otra.

Bidimensión con puntero Repetiremos un ejemplo similar al del *array bidimensional*, definiremos, valorizaremos y mostraremos una matriz, pero generada dinamicamente.

```
1
2   #include <stdio.h>
3   #include <stdlib.h>
4
5
6   int main(){
7       int f, c, i, w;  // f = numero de filas
8                        // c = numero de columnas
9
10      int **m = NULL;
11
12
13      printf("Elija el numero de filas que tiene su matriz: ");
14      scanf("%d", &f);
15
16      printf("Elija el numero de comunas que tiene su matriz: "
            );
17      scanf("%d", &c);
18
19      // Se crea la matriz de forma dinamica
20      m = (int **)malloc(sizeof(int*)*f);
21
22      for(i=0; i<f; i++)
23          m[i]=(int *)malloc(sizeof(int)*c);
24
```

```
25      // Pide el valor de cada elemento de la matriz
26      printf("Se completa la matriz con RAMDOM");
27
28      for(i=0; i < f; i++)
29          for(w=0; w < c; w++){
30              m[i][w] = 1 + rand()%9;
31          }
32
33      printf("\n La matriz es:\n");
34
35      for (i=0; i < f; i++ ){
36          for (w = 0; w < c; w++){
37              printf("%d\t", m[i][w]);
38          }
39          printf("\n");
40      }
41      // Libero la memoria utilizada por la matriz
42      for(i=0; i < f; i++)
43          free(m[i]);
44
45      free(m);
46
47      return 0;
48  }
```

6. Estructuras

Si tenemos en cuenta a la estructura como un tipo de dato, el puntero no significa mayor dificultad. Se declararía de la misma manera. Pero al tener en cuenta que la estructura es un tipo complejo de dato, compuestos por campos que a su vez son tipos de datos, variables, debemos hacer uso de un operador en particular para acceder a estos: $->$.

En forma esquemática podemos ver diferentes modos de definirlas:

```
1
2   struct tipo_estructura{
3           tipo nombre_var;
4           ...
5           tipo nombre_var;
6   } variable_estructura;
7
8   /* Se define el nombre de la estructura    fecha y la
        ódescripcin, pero no se ha
9   definido ninguna  variable. */
10
11  struct fecha {...};
12
```

```
/* No se le ha dado nombre a la estructura, pero si se ha
    definido una variable j la
cual tiene una ó descripcin que áest  definida entre { y }.
    */

struct {....} j;

/* Se declara una variable llamada s, cuya ódefinicin del
    contenido es X. (El que fue
o debe ser previamente definido).*/

struct X s;

/* Se declaran 3 variables, todas con la estructura X */

struct X a,b,c;
```

Teniendo en cuenta esto, usaremos la primera definición.

Ejemplo

```

struct fecha {
        int dia; /* Elementos de la estructura */
        int mes;
        int a;
} d; /* Nombre de la variable */

/* fecha es el nombre de la estructura */
```

Para referenciar a un elemento de la estructura se utiliza el operador (.)

variable_estructura.nombre_elemento

6.1. Acceso a la estructura

```

#include <string.h>

void main()
{
        struct direccion{
                char calle[25];
                int numero;
                char nombre[30];
        } d;

        strcpy(d.calle,"Avd. Alemania");
```

```
13          d.numero = 2010;
14          strcpy(d.nombre,"Fulano");
15  }
```

En el caso de ser requerido no habría inconveniente en poder declarar un arreglo de estructuras y por lo tanto la forma de acceder a estas sería bajo la misma sintaxis de un *array*

struct direccion dir[100];
dir[0].numero = 4524;

Y cuando una estructura está compuesta por campos que son datos primitivos, la copia de dicha estructura en otra del mismo tipo

6.2. Definición de tipo

Se puede definir explícitamente nuevos nombre para los tipos de datos usando la palabra clave ***typedef***. Realmente no se crea un nuevo tipo de datos, sino que se define un nuevo nombre para un tipo ya existente.

```
1
2   void main()
3   {
4           typedef struct{
5                   char nombre[20];
6                   int edad;
7           } persona;
8
9           persona p[100];  /* óDeclaracin de un arreglo de 100
                personas */
10  }
```

Como se ve en el ejemplo, deja de ser necesario anteponer la palabra ***struct*** a *persona* cada vez que usamos la estructura.

6.3. Punteros a estructuras

El uso de los punteros y las estructuras no varia a las formas en general del uso de la memoria. El operador ***-¿*** será usado para acceder a las variables que la compongan.

```
1
2   #include<string.h>
3
4   typedef struct {
5           char calle[25];
6           int numero;
```

```
7           char nombre[30];
8   } direccion;
9
10  void main()
11  {
12          direccion d, *p; /* se declara  d */
13
14          p = &d; /* p contiene la direccion del primer byte de
                d */
15
16          strcpy(p->calle,"Avd. Alemania");
17
18          p->numero = 2010;
19
20          strcpy(p->nombre,"Fulano");
21
22          /* ... */
23  }
```

6.4. Asignación de memoria dinámica

También usaremos el comando *malloc*.

```
1
2   #include <string.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   typedef struct {
7       char calle[25];
8       int numero;
9       char nombre[30];
10  } direccion;
11
12  void main()
13  {
14      direccion *pdir;
15
16      pdir = (direccion *)malloc (sizeof(direccion));
17      strcpy(pdir->calle,"Avd. Alemania");
18      pdir->numero=2010;
19      strcpy(pdir->nombre,"Fulano");
20
21      printf("La ódireccin de %s es: %s %d\n", pdir->nombre,
            pdir->calle, pdir->numero);
22
23  }
```

```
$ gcc -o direccion direccion.c
$ ./direccion
La ódireccin de Fulano es: Avd. Alemania 2010
```

Si quisieramos reemplazar la sintaxis del *array* de estructuras que vimos en la subsección anterior por la de una definición con el uso del puntero, podemos modificar el ejemplo previo por este:

```
1
2  #include <string.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  typedef struct {
7      char calle[25];
8      int numero;
9      char nombre[30];
10 } direccion;
11
12 void main()
13 {
14     direccion *pdir, *pdiri;
15
16     pdir = pdiri = (direccion *)malloc (1000 * sizeof(
           direccion));
17     strcpy(pdir->calle,"Avd. Alemania");
18     pdir->numero=2010;
19     strcpy(pdir->nombre,"Fulano");
20
21     printf("La ódireccin de %s es: %s %d\n", pdir->nombre,
           pdir->calle, pdir->numero);
22
23     pdiri++; /*pasa a la siguiente estructura*/
24
25     strcpy(pdiri->calle,"Av. Pellegrini");
26     pdiri->numero=250;
27     strcpy(pdiri->nombre,"Ingenieria");
28     printf("La ódireccin de %s es: %s %d\n", pdiri->nombre,
           pdiri->calle, pdiri->numero);
29
30 }
```

En la línea *16* vemos que se definen dos punteros a un conjunto de estructuras allocada dinámicamente, esto va a ser imprescindible ya que siempre vamos a tener que disponer de uno apuntando al inicio y no perderlo, y otro para poder recorrer las diferentes estructuras como se ve en la línea *23*.

Al ejecutarlo vemos:

```
$ gcc -o direcciones direcciones.c
$ ./direcciones
La ódireccin de Fulano es: Avd. Alemania 2010
La ódireccin de Ingenieria es: Av. Pellegrini 250
```

7. Otros tipos de datos

Veremos breves ejemplos de otros dos tipos de datos que posee C ademas de las *struct*. La idea no es profundizar en ellos en este momento pero si mencionarlos para que puedan investigarse y probarse.

7.1. Uniones

Es un tipo de dato especial que puede almacenar diferentes tipos de datos en la misma locación de memoria. La *union* puede ser definida con muchos miembros pero solo una de ellas podrá estar valuada a la vez.

La definición es similar a la de la estructura:

```

union [union tag] {
   tipo definicion;
   tipo definicion;
   ...
   tipo definicion;
} [una o mas variables union];
```

El *union tag* puede ser opcional. C reservara la memoria de la variable mas grande.

```

#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data
       ));

```

```
    return 0;
}
```

```
$ gcc -o union union.c
$ ./union
Memory size occupied by data : 20
```

Accediendo a las uniones

```

#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Si vemos la salida en pantalla del programa vemos que las dos primeras variables poseen valores corruptos:

```
$ gcc -o accUniones accUniones.c
$ ./accUniones
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Esto ocurrió por lo dicho en un principio, solo queda guardado el valor de la ultima asignación ya que todas comparten la misma posición de la memoria.

Si lo hacemos en forma ordenada, el código sería:

```

```

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
$ gcc -o accCorrecto accCorrecto.c
$ ./accCorrecto
data.i : 10
data.f : 220.500000
data.str : C Programming
```

7.2. Enumeraciones

En grandes rasgos, ***enum*** es un conjunto de constantes enteras.
Un ejemplo simple de su usu podría ser:

```c

#include <stdio.h>

typedef enum TiposFormas {
  circulo,
  cuadrado,
  rectangulo
}TiposFormas;  // Must end with a semicolon like a struct

int main() {
  TiposFormas forma = circulo;
  // Activities here....
```

```
13    // Now do something based on what the shape is:
14    switch(forma) {
15      case circulo:
16          printf("Se eligio el circulo ->valor: %d\n", forma);
17          printf("Si fuera otro como el cuadrado el valor seria
                : %d\n", cuadrado);
18          break;
19      case cuadrado:  /* codigo... */ break;
20      case rectangulo:  /* mas codigo */ break;
21    }
22    return 0;
```

La salida en pantalla es la siguiente:

```
$ gcc -o enumeracion1 enumeracion1.c
$ ./enumeracion1
Se eligio el circulo ->valor: 0
Si fuera otro como el cuadrado el valor seria: 1
```

Como se ve, la declaración de la enumeración tiene similitud con la estructura y la unión. Aqui se asume que las variables que la componen son constantes enteras y en este caso, fueron autonumeradas de 0 a 2 en orden.

En otras oportunidades se vio que hay otras formas de definir constantes, pueden hacerce con ***#define*** o anteponiendo ***const***, en cada caso se deberá tener que evaluar cual es mas útil o práctica para la necesidad del momento.

Otro posible uso seria:

```
1  enum T_HTML_ERROR
2  {
3      E_PAGE_NOT_FOUND = 404
4  };
```

Como vemos, podemos asignar arbitrariamente un valor.

Si tenemos en cuenta el programa anterior, el de las formas geométricas y en la última línea agregamos por ejemplo:

circulo = 7;

al compilar tendremos el siguiente mensaje:

```
$ gcc -o enumeracion1 enumeracion1.c
enumeracion1.c: In function ‘’main:
enumeracion1.c:22:11: error: lvalue required as left operand of assignment
   circulo = 7;
           ^
$
```

No es posible asignar un valor a una constante.

8. Pasando estructuras por argumento

La estructura no deja de ser una tipo de dato mas, aunque tenga una complejidad mayor, y por lo tanto el pasaje a funciones de las mismas corren con las mismas características.

Hay que recordar que C no pasa parametros por referencia, sino por valor, o sea, la función genera una copia de la variable pasada. Para no tener dualidad de variables y perder los cambios, lo correcto es que el prototipo de la función reciba un puntero a la estructura.