Propedéutico Programación

Victor Muñiz

(victor_m@cimat.mx)

Maestría en Cómputo Estadístico CIMAT

Temario

- Apuntadores y arreglos
- Entrada/salida archivos
- Asignación dinámica de memoria

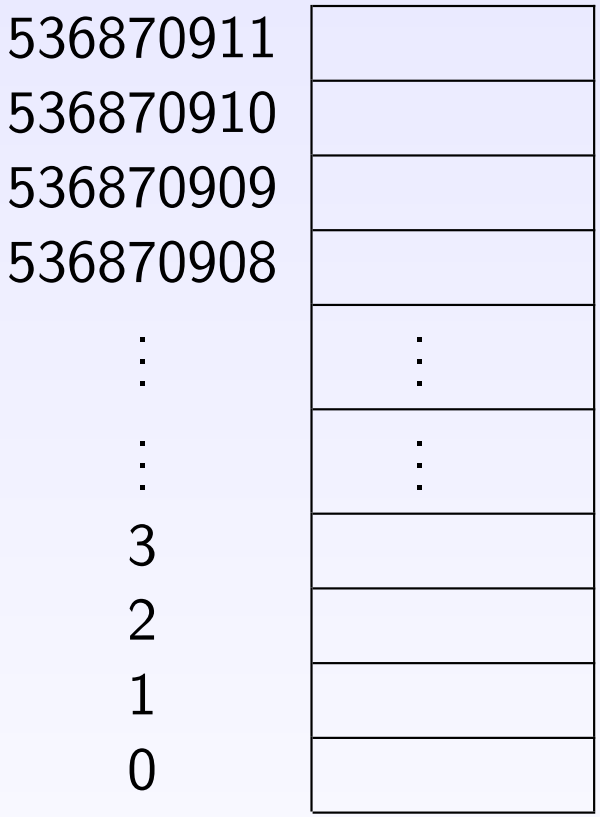
Apuntadores y direcciones en memoria

Un apuntador es una variable que contiene la **dirección** de la **localización en memoria** donde se almacena alguna otra variable (incluso otro apuntador).

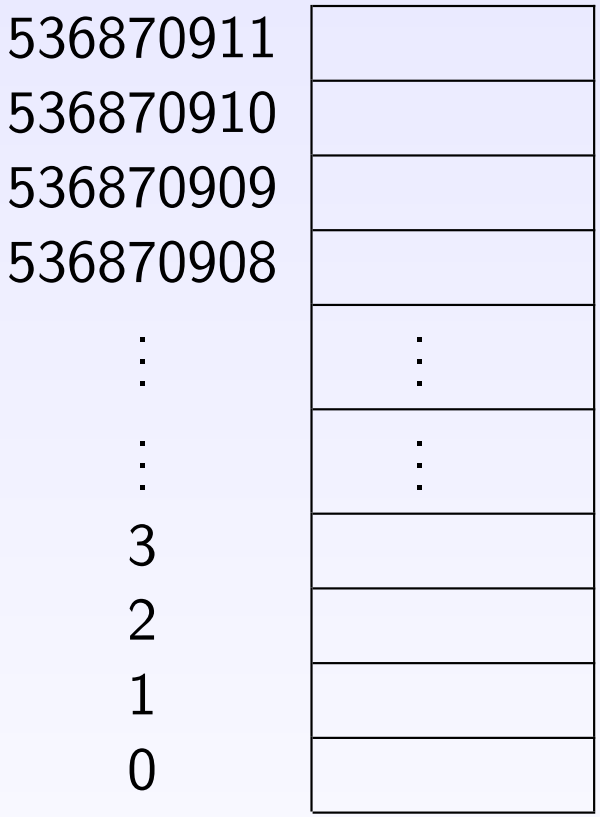
Un apuntador es una variable que contiene la **dirección** de la **localización en memoria** donde se almacena alguna otra variable (incluso otro apuntador).

La definición anterior implica conocer los conceptos básicos del tipo de variables y cómo se almacenan en memoria.

Ejemplo: memoria RAM de 512 Mb (536870912 byte)



Ejemplo: memoria RAM de 512 Mb (536870912 byte)



- Observa que, Byte $\neq$ Bit
- En general, 1 byte = 1 octeto = 8 bits

Puedes checarlo en limits.h. Por ejemplo, en linux (victor@vlinux2 ~ $ less /usr/include/limits.h)

```
#ifndef _LIBC_LIMITS_H_
#define _LIBC_LIMITS_H_ 1

#include <features.h>

/* Maximum length of any multibyte character in any locale.
   We define this value here since the gcc header does not define
   the correct value.  */
#define MB_LEN_MAX      16

/* If we are not using GNU CC we have to define all the symbols ourself.
   Otherwise use gcc's definitions (see below).  */
#if !defined __GNUC__ || __GNUC__ < 2

/* We only protect from multiple inclusion here, because all the other
   #include's protect themselves, and in GCC 2 we may #include_next through
   multiple copies of this file before we get to GCC's.  */
# ifndef _LIMITS_H
#  define _LIMITS_H     1

#include <bits/wordsize.h>

/* We don't have #include_next.
   Define ANSI <limits.h> for standard 32-bit words.  */

/* These assume 8-bit `char's, 16-bit `short int's,
   and 32-bit `int's and `long int's.  */

/* Number of bits in a `char'.  */
#  define CHAR_BIT      8
```

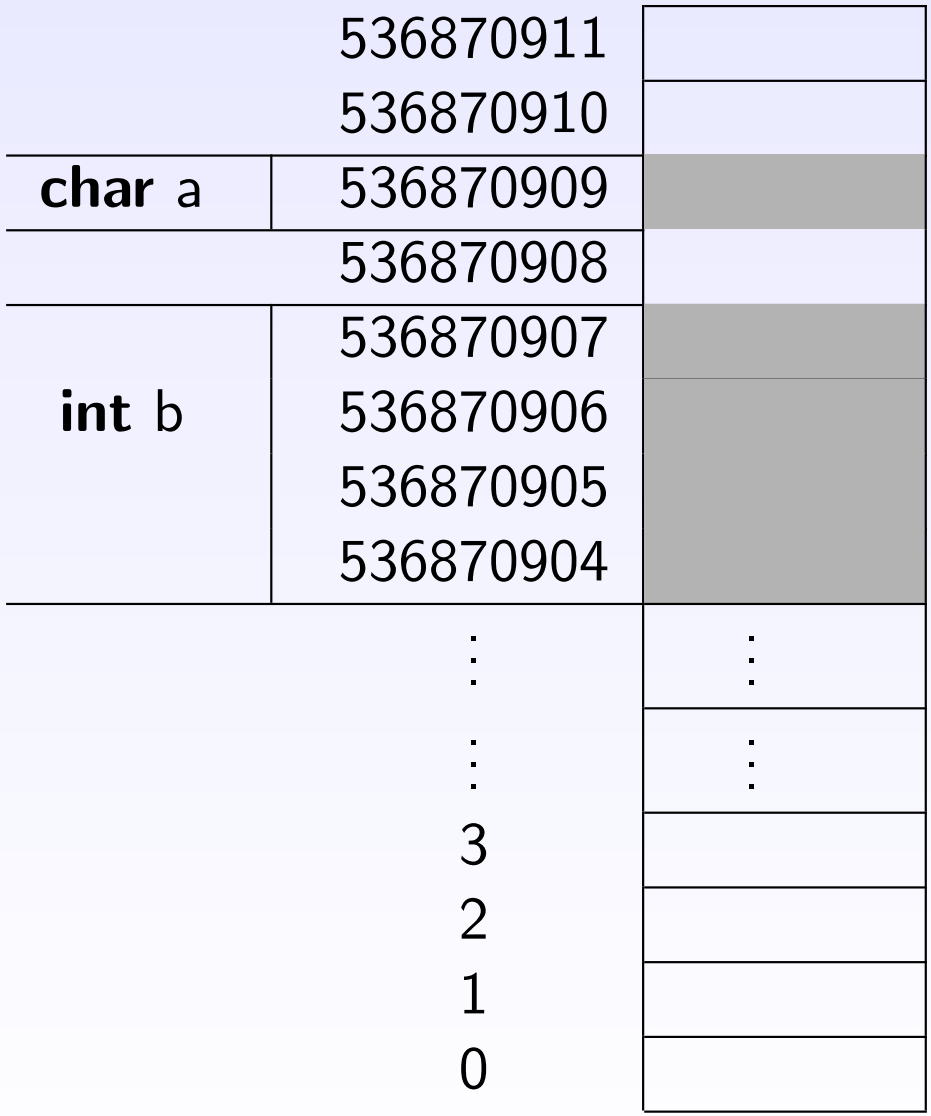
Recuerda que, en C y C++, el **tipo** de las variables representa el **número de bytes (u octetos)** y la forma en que se usará cada byte.

```
#include <stdio.h>

int main()
{
  printf("Size char:  %d bytes\n", sizeof(char));
  printf("Size int:   %d bytes\n", sizeof(int));
  printf("Size float: %d bytes\n", sizeof(float));
  printf("Size double: %d bytes\n", sizeof(double));
}
```

```
Size char:  1 bytes
Size int:   4 bytes
Size float: 4 bytes
Size double: 8 bytes
```

Entonces,


536870911
536870910
char a
536870909
536870908
536870907
int b
536870906
536870905
536870904
3
2
1
0

¿Cómo obtener la dirección en memoria de una variable?

¿Cómo obtener la dirección en memoria de una variable?

En C, el operador & nos da la dirección de un objeto

```
#include <stdio.h>
int j, k, z[3];
char a[2];

int main (void){
  j = 1;
  k = 2;
  printf("\n");
  printf("j tiene el valor: %d y esta alojado en: %p\n", j, (void *)&j);
  printf("k tiene el valor: %d y esta alojado en: %p\n", k, (void *)&k);
  printf("a[0] esta alojado en: %p\n", (void *)&a[0]);
  printf("a[1] esta alojado en: %p\n", (void *)&a[1]);
  printf("z[0] esta alojado en: %p\n", (void *)&z[0]);
  printf("z[1] esta alojado en: %p\n", (void *)&z[1]);
  printf("z[2] esta alojado en: %p\n", (void *)&z[2]);
  return 0;
}
```

Salida:

```
j tiene el valor: 1 y esta alojado en: 0x601060
k tiene el valor: 2 y esta alojado en: 0x60108c
a[0] esta alojado en: 0x601088
a[1] esta alojado en: 0x601089
z[0] esta alojado en: 0x601070
z[1] esta alojado en: 0x601074
z[2] esta alojado en: 0x601078
```

Salida:

```
j tiene el valor: 1 y esta alojado en: 0x601060
k tiene el valor: 2 y esta alojado en: 0x60108c
a[0] esta alojado en: 0x601088
a[1] esta alojado en: 0x601089
z[0] esta alojado en: 0x601070
z[1] esta alojado en: 0x601074
z[2] esta alojado en: 0x601078
```

Más importante:

¿Para qué nos sirve obtener la dirección en memoria de una variable?

Salida:

```
j tiene el valor: 1 y esta alojado en: 0x601060
k tiene el valor: 2 y esta alojado en: 0x60108c
a[0] esta alojado en: 0x601088
a[1] esta alojado en: 0x601089
z[0] esta alojado en: 0x601070
z[1] esta alojado en: 0x601074
z[2] esta alojado en: 0x601078
```

Más importante:

¿Para qué nos sirve obtener la dirección en memoria de una variable?

La utilidad la veremos a continuación...

Declaración de apuntadores:

```
tipo *nom;
```

donde

- `tipo` es cualquier tipo de variabe válida en C (incluso otro apuntador)
- `nom` es el nombre que se le da al apuntador
- `*` es el símbolo mediante el cual el compilador identifica que `nom` es un apuntador

Operadores de referenciación y desreferenciación.

&	Referenciación (la dirección de...)
*	Desreferenciación (el contenido de la dirección donde apunta...)

Operadores de referenciación y desreferenciación.

- & Referenciación (la dirección de...)
- * Desreferenciación (el contenido de la dirección donde apunta...)

```
int a;
int *ptr_to_a;

somevar = 5;
ptr_to_somevar = &(somevar); //la direccion de a
printf (" %d",*ptr_to_somevar); //el contenido de la direccion del apuntador
*ptr_to_somevar = 56; //igual a somevar=56
```

Otro ejemplo:

```
int a = 1;
int b = 2;
int c = 3;
int *p;
int *q;
p = &a;
q = &b;
c = *p;
p = q;
*p = 13;
```

Otro ejemplo:

```
int a = 1;
int b = 2;
int c = 3;
int *p;
int *q;
p = &a;
q = &b;
c = *p;
p = q;
*p = 13;
```

¿Qué valores tienen al final `a`, `b`, `c`, `p` y `q`?

Aritmética de apuntadores

Con los apuntadores, no todas las operaciones aritméticas están definidas, y **no todas tienen sentido de usar**.

Aritmética de apuntadores

Con los apuntadores, no todas las operaciones aritméticas están definidas, y **no todas tienen sentido de usar**.

Supón que `i,y,*ip` son int. ¿Qué obtendriamos en cada paso?

- `i=4; ip=&i;`
- `y=*ip+1;`
- `*ip+=1;`
- `++*ip;`
- `*ip++;`
- `ip++;`
- `ip--;`

Aritmética de apuntadores

```
int main(){
  int y,i, *ip;
  i=5;
  ip=&i;

  y=*ip+1;
  printf("y= %d, i= %d \n",y,i);
  *ip+=1;
  printf("Luego de *ip+=1. i= %d \n",i);
  ++*ip;
  printf("luego de ++*ip. i= %d \n",i);
  (*ip)++;
  printf("luego de (*ip)++. i= %d \n",i);
  *ip++;
  printf("luego de *ip++. i= %d, *ip= %d \n",i,*ip);

  return 0;
}
```

Aritmética de apuntadores

```
victor@vlinux2 ~/cursos/prope2016/programs $ ./varios
y=6, i=5
Luego de *ip+=1. i=6
Luego de ++*ip. i=7
Luego de (*ip)++. i=8
Luego de *ip++. i=8, *ip=6 <-------
```

Paréntesis: precedencia de operador en C/C++

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	++ --	pre-increment and decrement	
	()	Function call	
	[]	Array subscripting	
	.	Object member	
	->	Element selection through pointer	
	()	Function-style type cast	
3	++ --	post-increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	C-style type cast	
	*	Pointer dereference	
	&	Address-of	
	sizeof	The number of atomic units of memory used	
	new	dynamic memory allocation	
	delete	dynamic memory deallocation	
4	.* ->*	Pointer to member	Left-to-right
5	* / \%	Multiplication, division, and remainder	
6	+ -	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	< <=	Relational operators >	
	> >=	Relational operators \greaterequal $\geq$	
9	== !=	Relational equality and inequality	
10	&	Bitwise AND	

Paréntesis: precedencia de operador en C/C++

Precedence	Operator	Description	Associativity
11	^	Bitwise XOR	
12	|	Bitwise OR	
13	&&	Logical AND	
14	||	Logical OR	
15	?:	ternary	Right-to-left
	throw	throw an exception	
	=	assignment	
	+= -=	additive assignment	
	*= /= \%=	Multiplicative assignment	
	<<= >>=	Bitwise left shift and right shift assignment	
	\&= ^= |=	Bitwise AND, XOR, and OR assignment	
16	,	Comma	Left-to-right

Aritmética de apuntadores

```
int main(){
  int i=5;
  int *ip;

  ip=&i;
  ip++;
  printf("Luego de ip++. i= %d, *ip= %d \n",i,*ip);
  ip--;
  printf("Luego de ip--. i= %d, *ip= %d \n",i,*ip);
  ip--;
  printf("Luego de ip--. i= %d, *ip= %d \n",i,*ip);
  return 0;
}
```

Aritmética de apuntadores

```
int main(){
  int i=5;
  int *ip;

  ip=&i;
  ip++;
  printf("Luego de ip++. i= %d, *ip= %d \n",i,*ip);
  ip--;
  printf("Luego de ip--. i= %d, *ip= %d \n",i,*ip);
  ip--;
  printf("Luego de ip--. i= %d, *ip= %d \n",i,*ip);
  return 0;
}
```

```
victor@vlinux2 ~/cursos/prope2016/programs $ ./varios
Luego de ip++. i= 5, *ip=-2042994808
Luego de ip--. i= 5, *ip=5
Luego de ip--. i= 5, *ip=4196096
```

Aritmética de apuntadores

Algunos operadores aritméticos con apuntadores (que pueden tener sentido)

- Sumar entero: apuntador del mismo tipo en algún lugar trasladado de memoria
- Restar un entero: igual que el anterior
- Restar dos apuntadores: regresa un entero, que es el espacio de memoria entre los dos apuntadores

Las operaciones son en unidades que corresponden al tipo de apuntador, no en bytes.
Se pueden usar operadores de comparación siempre y cuando sean apuntadores del mismo tipo.

Aritmética de apuntadores

- Excepto en casos muy especiales (por ejemplo, en arreglos), las operaciones suma y multiplicación, no tienen sentido con apuntadores.

Aritmética de apuntadores

- Excepto en casos muy especiales (por ejemplo, en arreglos), las operaciones suma y multiplicación, no tienen sentido con apuntadores.
- Debe tenerse mucho cuidado con los apuntadores, ya que se puede acceder por error a sectores de memoria que pueden estar ocupados por otros programas o datos.

Apuntadores y funciones

Una de los usos principales de los apuntadores es en las funciones.

Apuntadores y funciones

Una de los usos principales de los apuntadores es en las funciones.

- Por default, los argumentos de las funciones se pasan **por valor**. Internamente, se crean **copias** de los argumentos.

Apuntadores y funciones

Una de los usos principales de los apuntadores es en las funciones.

- Por default, los argumentos de las funciones se pasan **por valor**. Internamente, se crean **copias** de los argumentos.
- Otra forma de pasar los argumentos es **por referencia**. En este caso, los argumentos contienen **las direcciones** de las variables.

Apuntadores y funciones

Una de los usos principales de los apuntadores es en las funciones.

- Por default, los argumentos de las funciones se pasan **por valor**. Internamente, se crean **copias** de los argumentos.
- Otra forma de pasar los argumentos es **por referencia**. En este caso, los argumentos contienen **las direcciones** de las variables.
- Las razones para usar apuntadores en funciones son principalmente:
  - Eficiencia. Piensa en un programa que use matrices float de 10000 x 10000.
  - Lógica. Hay algoritmos donde surge de forma natural el uso de apuntadores

Apuntadores y funciones

Considera la función `swap(a,b)`, que intercambia los valores de sus argumentos.

```
void swap(int x, int y){
  int temp;

  temp=x;
  x=y;
  y=temp;
}
```

Apuntadores y funciones

Considera la función `swap(a,b)`, que intercambia los valores de sus argumentos.

```
void swap(int x, int y){
  int temp;

  temp=x;
  x=y;
  y=temp;
}
```

Como los argumentos se pasan por valor, la función `swap` no puede cambiar los argumentos `a` y `b` **fuera del alcance** de la función.

Lo correcto sería entonces

Apuntadores y funciones

```
void swap(int *x, int *y);

int main (){
  int a, b;

  a=2;
  b=10;
  swap(&a, &b); //pasa la direccion de las variables
  return 0;
}

void swap(int *x, int *y){
  int temp;

  temp=*x;
  *x=*y;
  *y=temp;
}
```

Apuntadores y funciones

```
void swap(int *x, int *y);

int main (){
  int a, b;

  a=2;
  b=10;
  swap(&a, &b); //pasa la direccion de las variables
  return 0;
}

void swap(int *x, int *y){
  int temp;

  temp=*x;
  *x=*y;
  *y=temp;
}
```

Los apuntadores como argumentos permite a las funciones accesar y modificar objetos en la función que la llama.

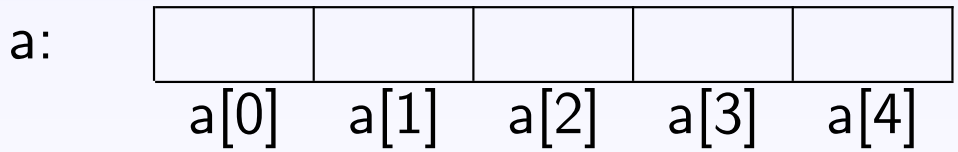
Arreglos y apuntadores

En C, los apuntadores y arreglos están muy relacionados.

Recordemos la declaración de un arreglo, por ejemplo, de enteros:

```
int a[5];
```

Internamente, el compilador reserva memoria para un bloque de 5 objetos de forma consecutiva:



Arreglos y apuntadores

Usar un apuntador en un arreglo es muy natural:

```
int a[5], *pa;
pa = &a[0];
```

La instrucción anterior declara un arreglo de enteros y un apuntador, luego, asigna el apuntador a la dirección de `a[0]`.

Arreglos y apuntadores

Usar un apuntador en un arreglo es muy natural:

```
int a[5], *pa;
pa = &a[0];
```

La instrucción anterior declara un arreglo de enteros y un apuntador, luego, asigna el apuntador a la dirección de `a[0]`.

La aritmética de apuntadores tiene mucho sentido al trabajar con arreglos. En el código anterior,

- `pa+i` apunta `i` elementos depués de `pa`
- `pa-i` apunta `i` elementos antes de `pa`
- `*(pa+i)` tiene el contenido de `a[i]`

Arreglos y apuntadores

Ejemplo:

```
int main(){
  int n=5;
  int nums[n], *ptr, i;

  //llena el arreglo
  for(i=0; i<n; i++)
  nums[i]=i-5;

  //muestra su contenido
  for(ptr= &nums[0]; ptr< &nums[n]; ptr++)
  printf("%d \n",*ptr);
  for(ptr= &nums[n-1]; ptr>= &nums[0]; ptr--)
  printf("%d \n",*ptr);
  return 0;
}
```

Arreglos y apuntadores

Cuando se pasa un arreglo como argumento a una función, se pasa la **dirección** del primer elemento. Dentro de la función es una variable local, y por lo tanto, un apuntador.

Arreglos y apuntadores

Cuando se pasa un arreglo como argumento a una función, se pasa la **dirección** del primer elemento. Dentro de la función es una variable local, y por lo tanto, un apuntador.
Considera por ejemplo la siguiente función:

```
int strlen(char *s){
  int n;
  for(n=0; *s!='\0'; s++)
  n++;

  return n;
}
```

Arreglos y apuntadores

Que pude usarse mediante:

```
int len;
len=strlen("hola, como estas");
```

```
int len;
char *s="prope cimat Monterrey";
len=strlen(s);
```

```
int len;
char s[10]={'h','o','l','a'};
len=strlen(s);
```

Apuntadores de caracteres

Una cadena de texto en C (por ejemplo ‘‘hola’’) se representa como un arreglo de caracteres:

h	o	l	a	\0
[0]	[1]	[2]	[3]	[4]

donde \0 es el caracter nulo que indica el fin de la cadena.

Apuntadores de caracteres

Una cadena de texto en C (por ejemplo ‘‘hola’’) se representa como un arreglo de caracteres:

h	o	l	a	\0
[0]	[1]	[2]	[3]	[4]

donde \0 es el caracter nulo que indica el fin de la cadena.
Podemos declarar variables de texto mediante

- Un arreglo: `char texto[]="hola";`
- Un apuntador: `char *texto="hola";`

Sin embargo, la forma en que se procesan es mediante apuntadores. Por ejemplo `printf("hola \n");` es una función que recibe un apuntador al primer elemento del argumento *string constant* ‘‘hola’’.

Apuntadores de caracteres

Una cadena de texto en C (por ejemplo ‘‘hola’’) se representa como un arreglo de caracteres:

h	o	l	a	\0
[0]	[1]	[2]	[3]	[4]

donde \0 es el caracter nulo que indica el fin de la cadena.
Podemos declarar variables de texto mediante

- Un arreglo: `char texto[]="hola";`
- Un apuntador: `char *texto="hola";`

Sin embargo, la forma en que se procesan es mediante apuntadores. Por ejemplo `printf("hola \n");` es una función que recibe un apuntador al primer elemento del argumento *string constant* ‘‘hola’’.
C **no tiene operadores para procesar cadenas de caracteres como una sola unidad**.

Apuntadores de caracteres

Ejemplo

Escribe un programa que copie el texto `b="dedo"` *en* `a="hola"`*:*

1. *usando arreglos*
2. *usando apuntadores*

Apuntadores de caracteres

Ejemplo

Escribe un programa que copie el texto `b="dedo"` *en* `a="hola"`*:*

1. *usando arreglos*
2. *usando apuntadores*

- ¿A alguno de ustedes se le ocurrió hacer: `a=b`?
  ¿Qué pasaría en ese caso?

Apuntadores de caracteres

Ejemplo

Escribe un programa que copie el texto `b="dedo"` *en* `a="hola"`*:*

1. *usando arreglos*
2. *usando apuntadores*

- ¿A alguno de ustedes se le ocurrió hacer: `a=b`?
  ¿Qué pasaría en ese caso?
- Ahora prueba tu programa con `a="Monterrey"`
  ¿Qué resultados obtienes usando arreglos y apuntadores?

Apuntadores a apuntadores

Ya vimos que un apuntador es una variable, por lo tanto, podemos realizar operaciones como con cualquier otra variable, incluido:

- Usar un apuntador a otro apuntador:

```
int a = 5;
int *ptr1 = &a;
int **ptr2 = &ptr1;
```

Apuntadores a apuntadores

Ya vimos que un apuntador es una variable, por lo tanto, podemos realizar operaciones como con cualquier otra variable, incluido:

- Usar un apuntador a otro apuntador:

```
int a = 5;
int *ptr1 = &a;
int **ptr2 = &ptr1;
```

- Crear arreglos de apuntadores

```
char *txt[3];
```

donde

  - `txt[i]` es un **apuntador char**
  - `*txt[i]` es el **texto i** al que apunta. Como es un arreglo de char, será **el primer caracter** del texto i.

Apuntadores a apuntadores

Ya vimos que un apuntador es una variable, por lo tanto, podemos realizar operaciones como con cualquier otra variable, incluido:

- Usar un apuntador a otro apuntador:

```
int a = 5;
int *ptr1 = &a;
int **ptr2 = &ptr1;
```

- Crear arreglos de apuntadores

```
char *txt[3];
```

donde

  - `txt[i]` es un **apuntador char**
  - `*txt[i]` es el **texto i** al que apunta. Como es un arreglo de char, será **el primer caracter** del texto i.

Ejemplo: ve la tarea 1.

Arreglos multidimensionales

Hasta ahora, hemos manejado arreglos unidimensionales (vectores), pero podemos usar más dimensiones. Generalmente, arreglos bidimensionales son suficientes para la mayoria de las aplicaciones.

Arreglos multidimensionales

Hasta ahora, hemos manejado arreglos unidimensionales (vectores), pero podemos usar más dimensiones. Generalmente, arreglos bidimensionales son suficientes para la mayoria de las aplicaciones.
La instrucción para declarar arreglos bidimensionales es:

```
tipo var[m][n];
```

donde `tipo` es cualquier tipo de variable válida (incluyendo apuntadores), `var` es el nombre de la variable y `m,n` es la dimensión del arreglo.

Arreglos multidimensionales

Inicialización de arreglos.

Hay varias formas de inicializar arreglos bidimensionales:

```
/* Correcto */
int a[2][2] = {1, 2, 3 ,4 };
/* Correcto */
int a[2][4] = {{10, 11, 12, 13},{14, 15, 16, 17}};
/* Correcto */
int a[][2] = {1, 2, 3 ,4 };
/* Incorrecto: debes especificar la segunda dimension*/
int a[][] = {1, 2, 3 ,4 };
/* Incorrecto */
int a[2][] = {1, 2, 3 ,4 }
```

Arreglos multidimensionales

Ejemplo

Sumar dos matrices.

Arreglos multidimensionales

Ejemplo

Sumar dos matrices.

Arreglos como argumentos de funciones.
Cuando se pasa un arreglo bidimensional a una función debe especificarse al menos, el número de columnas:

```
void fun(int a[2][2]) {  }
void fun(int a[][2]) {  }
void fun(int (*a)[2]) {  }
```

Arreglos multidimensionales y apuntadores

Considera las siguientes declaraciones:

```
int a[10][20];
int *b[10];
int **c;
```

Arreglos multidimensionales y apuntadores

Considera las siguientes declaraciones:

```
int a[10][20];
int *b[10];
int **c;
```

Las tres son declaraciones correctas para referirse a arreglos bidimensionales, sin embargo, la cantidad de memoria asignada a cada variable es diferente:

```
Size a: 800 bytes
Size b: 80 bytes
Size c: 8 bytes
```

Arreglos multidimensionales y apuntadores

Considera las siguientes declaraciones:

```
int a[10][20];
int *b[10];
int **c;
```

Las tres son declaraciones correctas para referirse a arreglos bidimensionales, sin embargo, la cantidad de memoria asignada a cada variable es diferente:

```
Size a: 800 bytes
Size b: 80 bytes
Size c: 8 bytes
```

De aquí puedes imaginar la ventaja de trabajar con dobles (o triples, etc...) apuntadores.

Entrada y salida en archivos

Entrada de información

C ofrece varias formas para introducir información a través de la libreria `stdio.h`.
Mediante la entrada estándar (teclado), las funciones más usadas son
`scanf(char *format,...)` y
`sscanf(char *string,char *format,arg1,arg2,...)`

Entrada de información

`C` ofrece varias formas para introducir información a través de la libreria `stdio.h`.
Mediante la entrada estándar (teclado), las funciones más usadas son
`scanf(char *format,...)` y
`sscanf(char *string,char *format,arg1,arg2,...)`

Ejemplo

La suma de matrices

Entrada de información

Sin embargo, la forma más común de introducir datos es a través de archivos.
C permite abrir, leer y escribir archivos a través de funciones como `fopen, fread, fclose` y con una estructura `FILE` que están contenidas en `stdio.h`.

Entrada de información

Sin embargo, la forma más común de introducir datos es a través de archivos.
C permite abrir, leer y escribir archivos a través de funciones como `fopen, fread, fclose` y con una estructura `FILE` que están contenidas en `stdio.h`.
Los datos del archivo se guardan en una zona de memoria dedicada llamada **buffer**. Esta gestión de memoria y colocación de la información es algo transparente para nosotros, solo hace falta conocer:

- dónde está el buffer
- dónde es el inicio de lectura
- qué modo es (lectura o escritura)

Entrada de información

La lectura y escritura de archivos se realiza a través de las declaraciones:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

`fp` es un apuntador a la estructura `FILE`, y `fopen` regresa un apuntador a `FILE`.
`name`, `mode` son el nombre del archivo y el modo, respectívamente.
Usualmente, los modos son `r` de lectura y `w` de escritura, pero hay otros.
investiga cuáles otros modos hay.

Entrada de información

La lectura y escritura de archivos se realiza a través de las declaraciones:

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

`fp` es un apuntador a la estructura `FILE`, y `fopen` regresa un apuntador a `FILE`.
`name`, `mode` son el nombre del archivo y el modo, respectívamente.
Usualmente, los modos son `r` de lectura y `w` de escritura, pero hay otros.
investiga cuáles otros modos hay.
Finalmente, el buffer o flujo de datos se cierra con la función

```
int *fclose(FILE *fp);
```

Entrada de información

Una vez que tenemos un archivo de datos abierto, podemos manipularlo para extraer o escribir información mediante

```
int fscanf(FILE *fp, char *format,...);
int *fprintf(FILE *fp, char *format,...);
```

que son el equivalente a `scanf` y `printf`.

Entrada de información

Una vez que tenemos un archivo de datos abierto, podemos manipularlo para extraer o escribir información mediante

```
int fscanf(FILE *fp, char *format,...);
int *fprintf(FILE *fp, char *format,...);
```

que son el equivalente a `scanf` y `printf`.

Ejemplo

Suma de matrices usando archivos

Entrada de información

Una vez que tenemos un archivo de datos abierto, podemos manipularlo para extraer o escribir información mediante

```
int fscanf(FILE *fp, char *format,...);
int *fprintf(FILE *fp, char *format,...);
```

que son el equivalente a `scanf` y `printf`.

Ejemplo

Suma de matrices usando archivos

¿Qué problemas pueden surgir al manejar archivos de datos?

Entrada de información

Una vez que tenemos un archivo de datos abierto, podemos manipularlo para extraer o escribir información mediante

```
int fscanf(FILE *fp, char *format,...);
int *fprintf(FILE *fp, char *format,...);
```

que son el equivalente a `scanf` y `printf`.

Ejemplo

Suma de matrices usando archivos

¿Qué problemas pueden surgir al manejar archivos de datos?
C no tiene un mecanismo "automático" de manejo de errores, **tu debes detectar los posibles errores y programar un manejo adecuado de ellos**.

Asignación dinámica de memoria

Asignación dinámica

- Hasta ahora, hemos creado objetos (arreglos, pero también pueden crearse otros) de forma estática, donde fijamos de antemano, el tamaño de tal objeto.

Asignación dinámica

- Hasta ahora, hemos creado objetos (arreglos, pero también pueden crearse otros) de forma estática, donde fijamos de antemano, el tamaño de tal objeto.
- Sin embargo, muchas veces el tamaño del objeto puede cambiar a lo largo del programa según las necesidades del usuario, lo que nos lleva a una asignación dinámica de la memoria que solicitamos.

Asignación dinámica

C proveé funciones para esta asignación dinámica a través de las funciones

- `void *malloc(size_t n)`: regresa un apuntador a *n* bytes de memoria sin inicializar

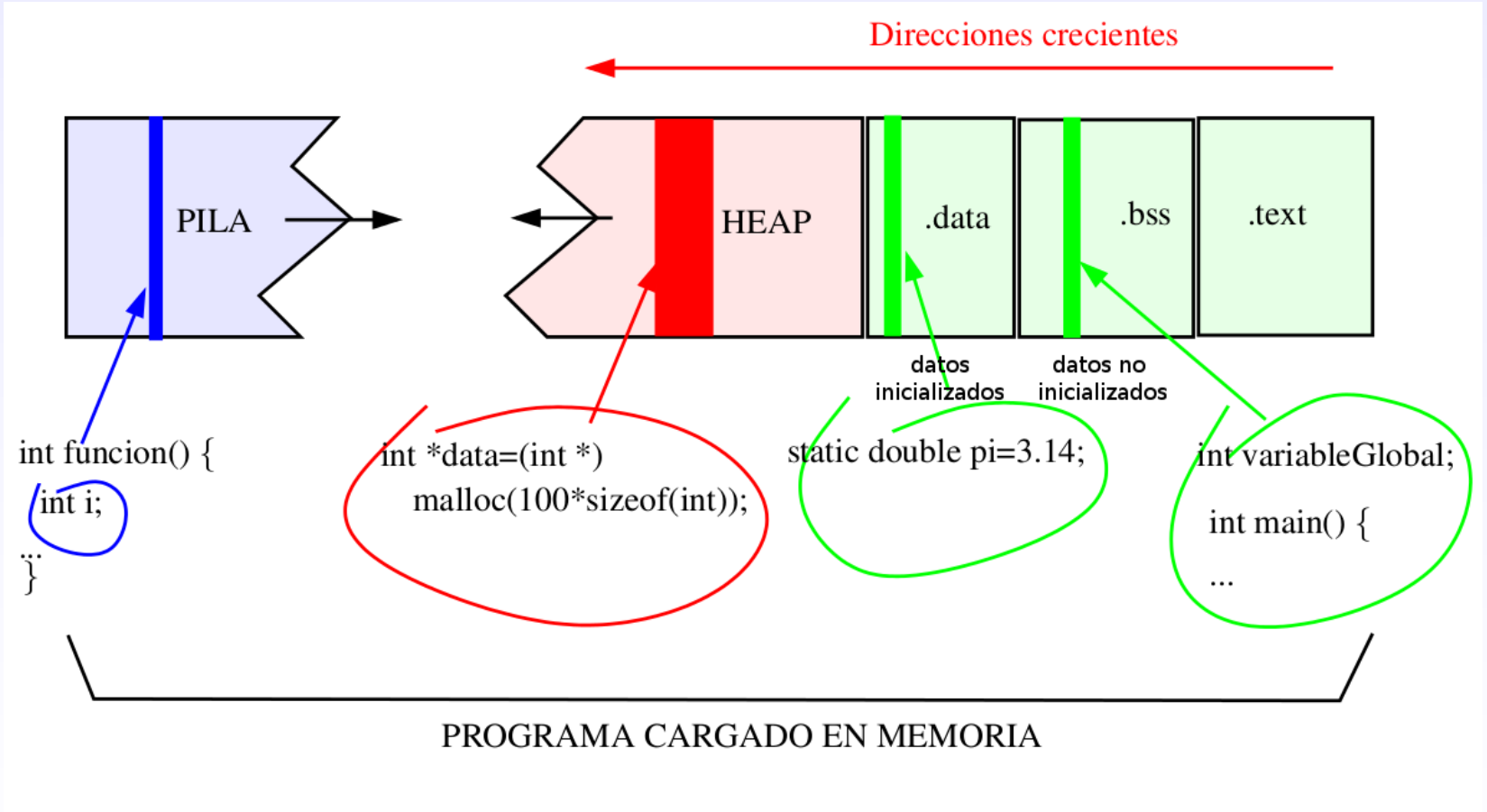
```
int *arregloDinamico =
(int *) malloc(tamanoArreglo * sizeof(int));
```

- `void *calloc(size_t n, size_t size)`: regresa un apuntador para un espacio en memoria suficiente de un arreglo de *n* objetos del tamaño especificado, o `NULL` si no es posible reservar tal cantidad de memoria. El espacio es inicializado a 0.

```
int *arregloDinamico =
(int *) calloc(tamanoArreglo,sizeof(int));
```

Asignación dinámica

La memoria reservada dinámicamente se aloja en el heap (memoria persistente):



Asignación dinámica

La memoria que se solicita es **persistente**, es decir, sobrevive fuera de la función donde se crea, por lo tanto, **debemos liberar** tal memoria cuando ya no la utilizamos. Para esto, usamos la función

`free(p)`

que libera el espacio de memoria apuntado por `p`.

```
int *arregloDinamico =
(int *) malloc(tamanoArreglo * sizeof(int));
free(arregloDinamico);
```

Asignación dinámica

Siempre debemos verificar que la solicitud de memoria `void *malloc` y `void *calloc` regresan un apuntador nulo si no pueden reservar la memoria solicitada

```
int *arregloDinamico =
(int *) malloc(tamanoArreglo * sizeof(int));
if(arregloDinamico==NULL){
  ...
}
```

Asignación dinámica

Siempre debemos verificar que la solicitud de memoria `void *malloc` y `void *calloc` regresan un apuntador nulo si no pueden reservar la memoria solicitada

```
int *arregloDinamico =
(int *) malloc(tamanoArreglo * sizeof(int));
if(arregloDinamico==NULL){
  ...
}
```

Ejemplo

Creación y lectura de una matriz

Para finalizar...

- Recuerda: solo se aprende a programar, programando

Para finalizar...

- Recuerda: solo se aprende a programar, programando
- Tienes una gran cantidad de fuentes a las cuáles recurrir: tutoriales en línea, blogs especializados en la red, etc...

Para finalizar...

- Recuerda: solo se aprende a programar, programando
- Tienes una gran cantidad de fuentes a las cuáles recurrir: tutoriales en línea, blogs especializados en la red, etc...
- Nunca te quedes solamente con lo que ves en clase. Investiga, busca y aprende por tu cuenta...