ICF 122 - Fundamentos de Programación
Ingeniería Civil Informática

Unidad 7: Punteros

Facultad de Ingeniería

Universidad Andrés Bello

- Un **puntero** es una variable que contiene la **dirección de memoria** de un dato u otra variable.
- El **puntero** apunta al espacio físico donde está el dato o la variable.
- Los punteros pueden apuntar a cualquier tipo, como por ejemplo, una estructura o una función.
- Los punteros se utilizan principalmente para referenciar y manipular estructuras de datos, bloques de memoria asignados dinámicamente y proveer el paso de argumentos por referencia.
- Declaración:

```
tipo *nombrePuntero;
```

```c
#include <stdio.h>

void main()
{
  int a = 0;      //variable entera de tipo entero
  int *ptr;       //variable puntero de tipo entero
  ptr = &a;  //direccion de memoria de a

  printf("El valor de a es: %d. \n",a);
  printf("El valor de *ptr es: %d. \n", *ptr);
  printf("La direccion de a es: %p. ", ptr);
}
```

En cada ejecución, se nota que la dirección de memoria es distinta, esto es debido a que el sistema operativo es quien esta encargado de administrar la memoria y es este quien dice que espacios podrá tomar el programa.

Veamos el siguiente código

```
int x[10], b, *pa, *pb;

x[5] = 10;  //asignamos el valor de 10, al 6to elemento.

pa = &x[5]; //asignamos al puntero pa, la direccion de x[5]

b = *pa + 1; //al valor que tiene x[5] se le suma 1.

b = *(pa + 1); //pasa a la siguiente direccion y luego lo referencia

pb = &x[3]; //al puntero pb se le asigna la direccion de x[3]

*pb = 0; //al valor que tiene el puntero se le asigna 0

*pb += 2; //el valor del puntero se incrementa en dos

(*pb)--; //el valor del puntero se decrementa en uno.
```

- ¿Cómo podríamos construir una función `swap(a, b)` que intercambie los valores entre `a` y `b`?
- Una idea

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

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

- ¿Se obtiene el intercambio en los valores?

- El intercambio no se logra, ya que las funciones solamente reciben valores como argumentos, por lo tanto, no podemos alterar la variable propiamente tal.
- Debemos recurrir a los punteros. ¿Por qué?

```
void swap(int *px , int *py){
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

- Ahora debemos llamar a la función como `swap(&a, &b)` ¿Se obtiene el intercambio en los valores?

Punteros

Operaciones

```c
#include <stdio.h>

void f1(int u, int v){
  u = v = 0;
  printf("En la funcion f1: u = %d, v=%d\n",u,v);
}

void f2(int *pu, int *pv){
  *pu = *pv = 0;
  printf("En la funcion f2: *pu = %d, *pv=%d\n",*pu,*pv);
}

void main(){
  int u = 1, v = 3;

  printf("Antes de llamar a f1: u = %d, v=%d\n",u,v);
  f1(u,v);
  printf("Despues de llamar a f1: u = %d, v=%d\n",u,v);

  printf("Antes de llamar a f2: u = %d, v=%d\n",u,v);
  f2(&u,&v);
  printf("Despues de llamar a f2: u = %d, v=%d\n",u,v);
}
```

Punteros

Operaciones

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

void scanLinea(char linea[],int *pv,int *pc,int *pd,int *pw,int *po){
  char c;
  int cont = 0;

  while ((c = toupper(linea[cont])) != '\0'){
    if (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U')
      ++ *pv;
    else if (c >= 'A' && c <= 'Z')
      ++ *pc;
    else if (c >= '0' && c <= '9')
      ++ *pd;
    else if (c == ' ' || c == '\t')
      ++ *pw;
    else
      ++ *po;
    ++cont;
  }
}
```

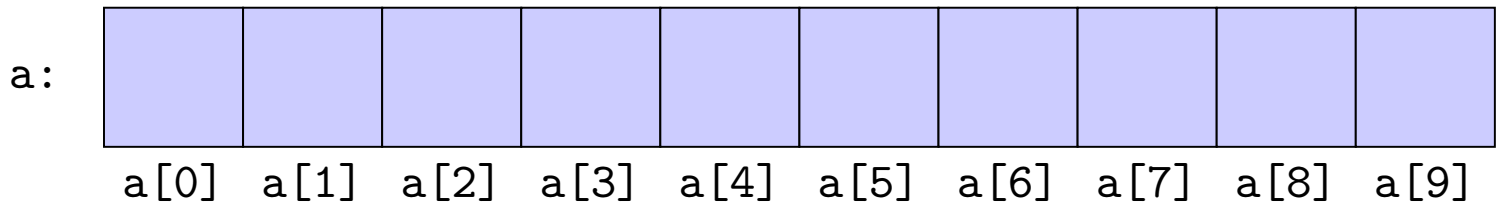
```c
void main(){
  char linea[80];
  int vocales = 0;
  int consonantes = 0;
  int digitos = 0;
  int espacios = 0;
  int otro = 0;

  printf("Ingrese una linea de texto:\n");
  scanf("%[^\n]",linea);
  scanLinea(linea,&vocales,&consonantes,&digitos,&espacios,&otro);

  printf("\nVocales: %d", vocales);
  printf("\nConsonates: %d", consonantes);
  printf("\nDigitos: %d", digitos);
  printf("\nEspacios: %d", espacios);
  printf("\nOtros: %d\n", otro);
}
```

Punteros

- En C, hay una poderosa relación entre los punteros y los arreglos.
- Cualquier operación que se realiza con los índices de un arreglo, es posible realizarlo con los punteros.
- La declaración

```
int a[10];
```

define un arreglo a de tamaño 10, el cual es, un bloque de 10 objetos consecutivos denominados `a[0]`, `a[1]`, `...`, `a[9]`



Punteros

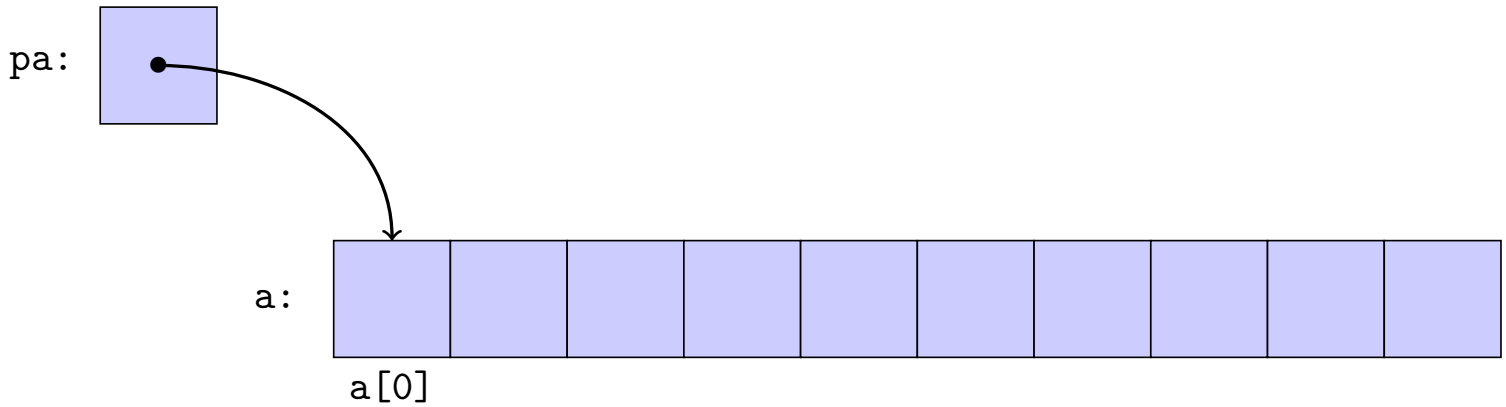
- La notación `a[i]` se refiere al i-ésimo elemento del arreglo.
- Si `pa` es un puntero a un entero declarado como

```
int *pa;
```

entonces la asignación

```
pa = &a[0];
```

permite que `pa` apunte al primer elemento de `a`, esto es, `pa` contiene la dirección de `a[0]`

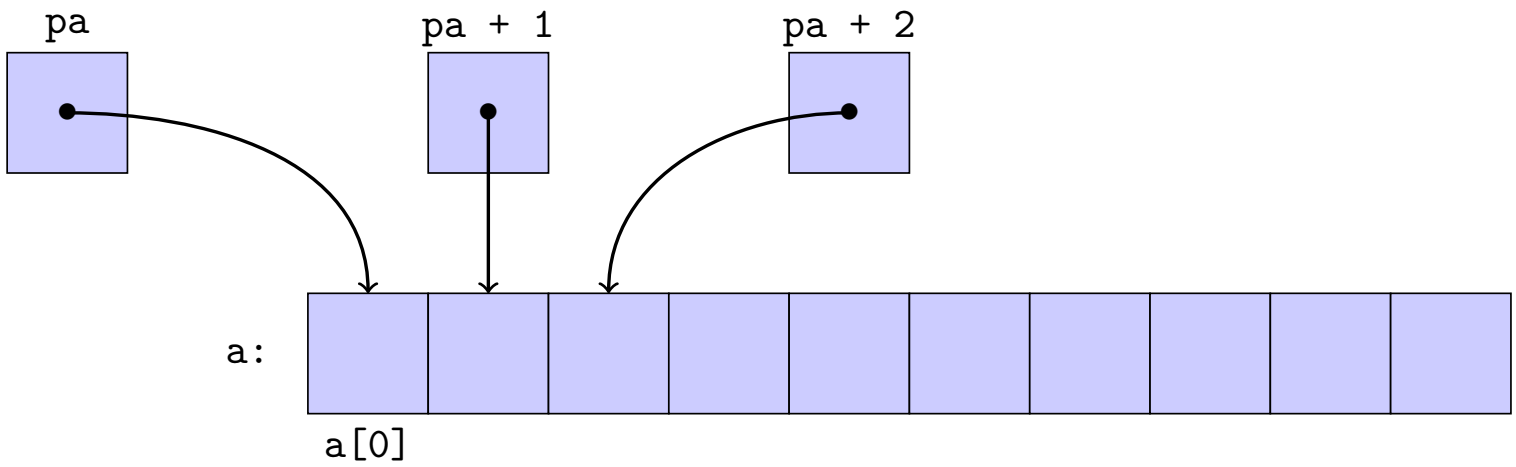


- Luego la asignación

```
x = *pa;
```

copiará el contenido de `a[0]` en `x`.

- Si `pa` apunta a un elemento en particular de un arreglo, entonces por definición `pa + 1` apunta al próximo elemento, luego `pa + i` apunta al i-ésimo elemento después de `pa` y `pa - i` apunta al i-ésimo elemento antes.



Punteros

- Dado que el nombre del arreglo es sinónimo de la ubicación del elemento inicial, la asignación `pa = &a[0]` también se puede escribir como

```
pa = a;
```

- Una referencia a `a[i]` también se puede escribir como `*(a + i)`, al igual que `pa[i]` es idéntico a `*(pa + i)`.
- **Importante:** Un puntero es una variable, luego `pa = a` y `pa++` es válido, pero un arreglo no es una variable, por lo tanto `a = pa` y `a++` no son expresiones válidas.

Función strlen

```
/*strlen: retorna el largo del string s*/
int strlen(char *s){
 int n;

 for(n = 0; *s != '\0'; s++)
   n++;
 return n;
}
```

Todas estas llamadas funcionan

```
strlen("hola, mundo");  /*string constante    */
strlen(arreglo);        /*char arreglo[100];  */
strlen(ptr);            /*char *prt           */
```

Punteros

- Un **string constante**, como ''Soy un string'' es un arreglo de caracteres. Internamente, un arreglo es finalizado con el caracter nulo '\0'.
- Es importante conocer la diferencia entre estas definiciones

```
char mensaje[] = "ahora es tiempo"; /*un arreglo*/
char *pmensaje = "ahora es tiempo"; /*un puntero*/
```

mensaje es un arreglo, lo suficientemente grande para almacenar la secuencia de caracteres más el caracter nulo '\0'. Los elementos de mensaje pueden cambiar, pero la variable siempre hará referencia al mismo espacio. En cambio, pmensaje es un puntero a un string constante, el puntero puede apuntar a cualquier otro lugar, pero el resultado quedará indefinido si se trata de modificar el contenido del string.

Punteros

- Función cpy

```
/*cpy : copia t a s*/
void cpy(char *s, char *t){
 int i;

 i = 0;
 while((*s = *t) != '\0'){
   s++;
   t++;
 }
}
```

- Función cmp

```
/*cmp : retorna < 0 si s < t, 0 si s = t
y > 0 si s > t*/
int cmp(char *s, char *t){
 for ( ; *s == *t; s++, t++)
   if (*s == '\0') return 0;
 return *s - *t;
}
```

Punteros

- Escriba la función `strEnd(s,t)` la cual retorna valor 1 si el string `t` se encuentra al final del string `s`, y 0 en caso contrario.
- Escriba la función `strCopiarN(s,t,n)` la cual copia los primeros `n` caracteres del string `t` al string `s`.

Punteros

```c
#include <stdio.h>

void main(){
  int x[6] = {10,11,12,13,14,15};
  int i;

  for (i = 0; i < 6; i++){
    printf("\ni = %d x[i] = %d *(x + i) = %d", i, x[i], *(x + i));
    printf("&x[i] = %p x + i = %p", &x[i], (x + i));
  }
  printf("\n");
}
```

```
i = 0   x[i] = 10   *(x + i) = 10   &x[i] = 0x7ffe4095db70   x + i = 0x7ffe4095db70
i = 1   x[i] = 11   *(x + i) = 11   &x[i] = 0x7ffe4095db74   x + i = 0x7ffe4095db74
i = 2   x[i] = 12   *(x + i) = 12   &x[i] = 0x7ffe4095db78   x + i = 0x7ffe4095db78
i = 3   x[i] = 13   *(x + i) = 13   &x[i] = 0x7ffe4095db7c   x + i = 0x7ffe4095db7c
i = 4   x[i] = 14   *(x + i) = 14   &x[i] = 0x7ffe4095db80   x + i = 0x7ffe4095db80
i = 5   x[i] = 15   *(x + i) = 15   &x[i] = 0x7ffe4095db84   x + i = 0x7ffe4095db84
```

Punteros

```c
#include <stdio.h>

void main(){
  int *px;
  int i = 1;
  float f = 0.3;
  double d = 0.002;
  char c = '*';

  px = &i;
  printf("Valores\n");
  printf("i = %d \nf = %f \nd = %f \nc = %c\n\n",i,f,d,c);

  printf("Direcciones\n");
  printf("&i = %p \n&f = %p \n&d = %p \n&c = %p\n\n",&i,&f,&d,&c);

  printf("Punteros\n");
  printf("px = %p \npx + 1 = %p\n",px,px+1);
  printf("px + 2 = %p \npx + 3 = %p\n",px+2,px+3);
}
```

Punteros

```
Valores
i = 1
f = 0.300000
d = 0.002000
c = *

Direcciones
&i = 0x7fff119a9498
&f = 0x7fff119a949c
&d = 0x7fff119a94a0
&c = 0x7fff119a9497

Punteros
px = 0x7fff119a9498
px + 1 = 0x7fff119a949c
px + 2 = 0x7fff119a94a0
px + 3 = 0x7fff119a94a4
```

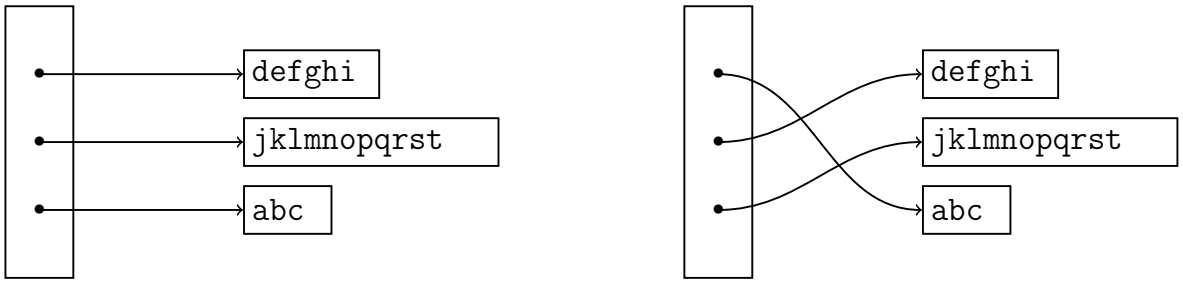
Punteros

```c
#include <stdio.h>

void main(){
  int i,*px,*py;
  static int a[6] = {1,2,3,4,5,6};

  px = &a[0];
  py = &a[5];

  printf("px = %p        py = %p\n\n",px,py);
  printf("py - px = %ld\n",py - px);
}
```

```
px = 0x601040 py = 0x601054

py - px = 5
```

- Dado que los punteros son variables, pueden ser almacenados en arreglos.
- Por ejemplo, podríamos ordenar un conjunto de líneas de texto en orden alfabético.
- ¿Cómo entonces almacenamos, en primer lugar las líneas de texto si son de distinto tamaño?
- Necesitamos una representación de datos que nos permita almacenar eficientemente cada línea de texto.
- Solución: **arreglo de punteros**.

- Cada línea será accedida mediante un puntero al primer elemento, por lo tanto, los punteros será almacenados en un arreglo, y podríamos llamar a la función `strcmp` para comparar.
- ¿Qué sucede si debemos cambiar las líneas?, simplemente cambiamos los punteros sin alterar las líneas de texto



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

#define MAXLINES 500

char *lineptr[MAXLINES];

void main(){
  int i;
  lineptr[0] = "hola mundo";
  lineptr[1] = "chao jefe";
  lineptr[2] = "ingenieria es cool";

  for(i = 0; i < 3; i++)
    printf("%s\n",lineptr[i]);
}
```

Asignación dinámica de memoria

malloc

- La función `malloc` está definida en la biblioteca `stdlib.h`

```
void *malloc(size_t size);
```

  donde `size_t` es el tipo de dato a considerar.
- La función reserva tantos **bytes consecutivos** como indique el valor de la variable `size` y devuelve la **dirección del primero de esos bytes**.
- Esa dirección debe ser tomada por una variable puntero: si no se toma, tendremos la memoria reservada pero no podremos acceder a ella porque no sabremos dónde ha quedado hecha esa reserva.

Asignación dinámica de memoria

malloc

- Podemos notar que la función `malloc` es de tipo `void*`, pero no tiene sentido trabajar con direcciones de tipo `void`. Por eso, en la asignación al puntero que debe tomar la dirección del array, se debe indicar, mediante el **operador forzar tipo**, el tipo de la dirección.
- Veamos un ejemplo

```
int dim;
float *vector:

printf("Dimension del vector: ");
scanf("%d",&dim);

vector = (float*)malloc(sizeof(float) * dim);
```

- Cuando la función `malloc` no logra satisfacer la demanda de memoria, devuelve el puntero nulo.

Asignación dinámica de memoria

calloc

- La función `calloc` también está definida en la biblioteca `stdlib.h`

```
void *calloc(size_t nItems, size_t size);
```

  donde `size_t` es el tipo de dato a considerar.

- La función reserva tantos **elementos consecutivos** como indique el valor de la variable `nItems`, donde cada elemento es de tamaño `size`. Al igual que la función anterior, devuelve la **dirección del primero de los bytes** reservados.

- La instrucción anterior

```
vector = (float*)malloc(sizeof(float) * dim);
```

  ahora, con la función `calloc` quedaría:

```
vector = (float*)calloc(dim, sizeof(float));
```

Asignación dinámica de memoria

free

- La función `free` también está definida en la biblioteca `stdlib.h`

```
void *free(void *block);
```

  donde `block` es un puntero que tiene asignada la dirección de cualquier bloque de memoria que haya sido reservada previamente mediante una función de memoria dinámica.

- La función libera la memoria que ha sido reservada, ya sea por la función `malloc` o por `calloc`.
- Para liberar la memoria asignada a la variable `vector`, la instrucción es simplemente

```
free(vector);
```