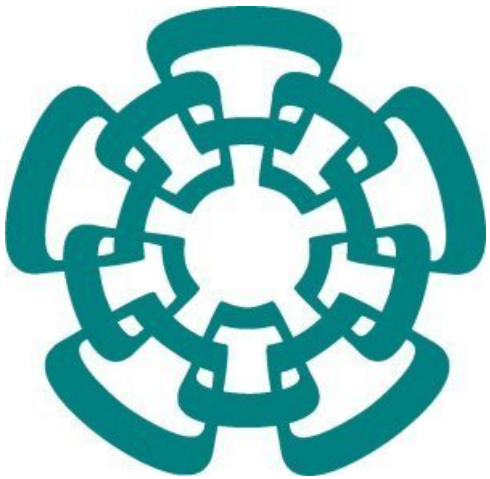
Maestría en Ciencias en Ingeniería y Tecnologías Computacionales

Proceso de Admisión 2022



Memoria y apuntadores en C

Dr.Miguel Morales Sandoval

Tópicos:

1. Memoria: conceptos básicos
2. Manejo de memoria en C
3. Apuntadores
4. Artimética de apuntadores
5. Ejemplos
6. Ejercicios

Diapositivas y código fuente:

`https://www.tamps.cinvestav.mx/~mmorales/courses.html`

Memoria

La memoria es un componente escencial en un dispositivo computacional

- Almacena el programa que el hardware (CPU) ejecutará

Memoria

La memoria es un componente escencial en un dispositivo computacional

- Almacena el programa que el hardware (CPU) ejecutará
- Almacena los valores de los datos de entrada, valores intermedios y resultados que se generarán durante la ejecución del programa

Memoria

La memoria es un componente escencial en un dispositivo computacional

- Almacena el programa que el hardware (CPU) ejecutará
- Almacena los valores de los datos de entrada, valores intermedios y resultados que se generarán durante la ejecución del programa
- Una **variable** en el programa, un **valor** o **cálculo**, se almacenará siempre en la memoria principal del sistema (RAM)

Memoria

Organización de la memoria para un programa en C

- Conjunto de localidades contiguas, de un tamaño fijo (1 byte)

Memoria

Organización de la memoria para un programa en C

- Conjunto de localidades contiguas, de un tamaño fijo (1 byte)
- Cada localidad de memoria tiene una dirección, que la identifica de manera única

Memoria

Organización de la memoria para un programa en C

- Conjunto de localidades contiguas, de un tamaño fijo (1 byte)
- Cada localidad de memoria tiene una dirección, que la identifica de manera única
- Sobre ella, se pueden realizar dos operaciones: **leer** o **escribir** un valor

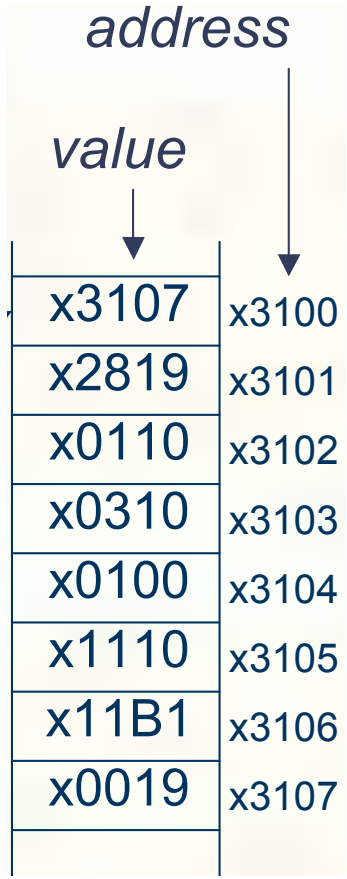
Memoria

Organización de la memoria para un programa en C

- Conjunto de localidades contiguas, de un tamaño fijo (1 byte)
- Cada localidad de memoria tiene una dirección, que la identifica de manera única
- Sobre ella, se pueden realizar dos operaciones: **leer** o **escribir** un valor
  - `ESCRIBIR(dirección, valor)`
  - `valor` $\leftarrow$ `LEER(dirección)`

Memoria

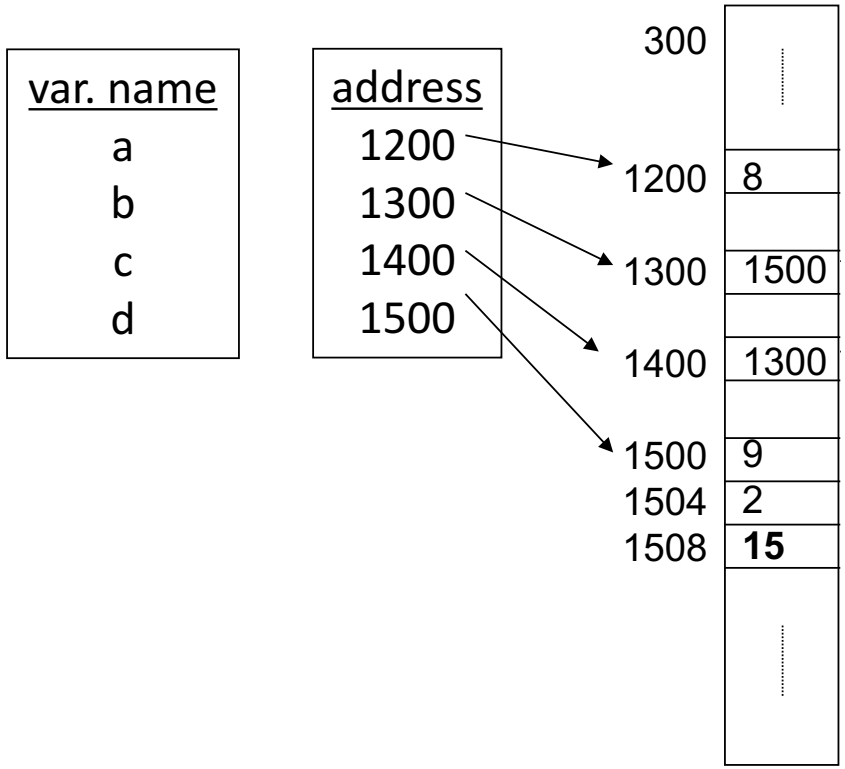
Modelo lógico



La memoria como un conjunto de localidades contiguas, identificadas por una dirección, cada una de longitud fija (bytes).

Variables

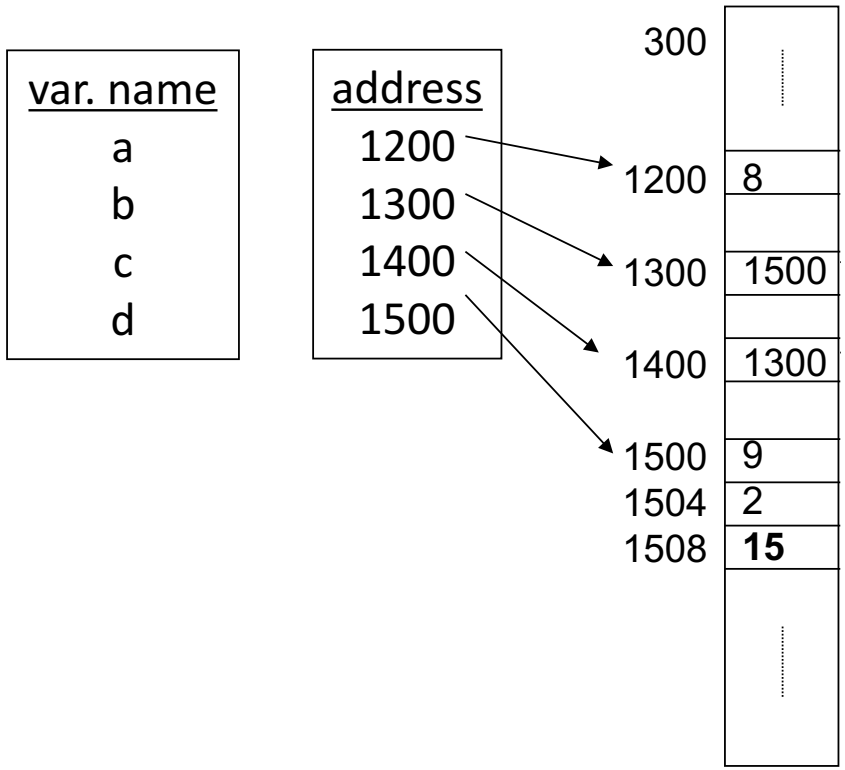
Una **variable** en el programa, que representa un **valor** o **cálculo**, ocupará siempre **algunas** localidades de la memoria del sistema (RAM)



Una variable en el programa siempre tiene asociada una dirección y un valor. Ocupa una o varias localidades en memoria.

Variables

Una **variable** en el programa, que representa un **valor** o **cálculo**, ocupará siempre localidades de la memoria del sistema (RAM)



La dirección está dada por el nombre de la variable, generalmente no nos preocupamos por ello. Con apuntadores, si nos preocupamos por ello.

Variables

```c
#include <stdio.h>

int main()
{
    int x = 13; // Variable de tipo entero


//La direccion de variable nunca cambia, su valor si.

    printf("\nValor de variable: \t%d", x);
    printf("\nDireccion de variable: \t%X", &x);

    x = 17;

    printf("\nValor de variable: \t%d", x);
    printf("\nDireccion de variable: \t%X", &x);

    return 0;
}
```

Programa 1: valor y dirección de una variable (var.c)

Apuntadores o punteros

Apuntador o puntero: Es como cualquier *variable*, pero su valor es una dirección de memoria. Sintáxis para declarar un puntero:

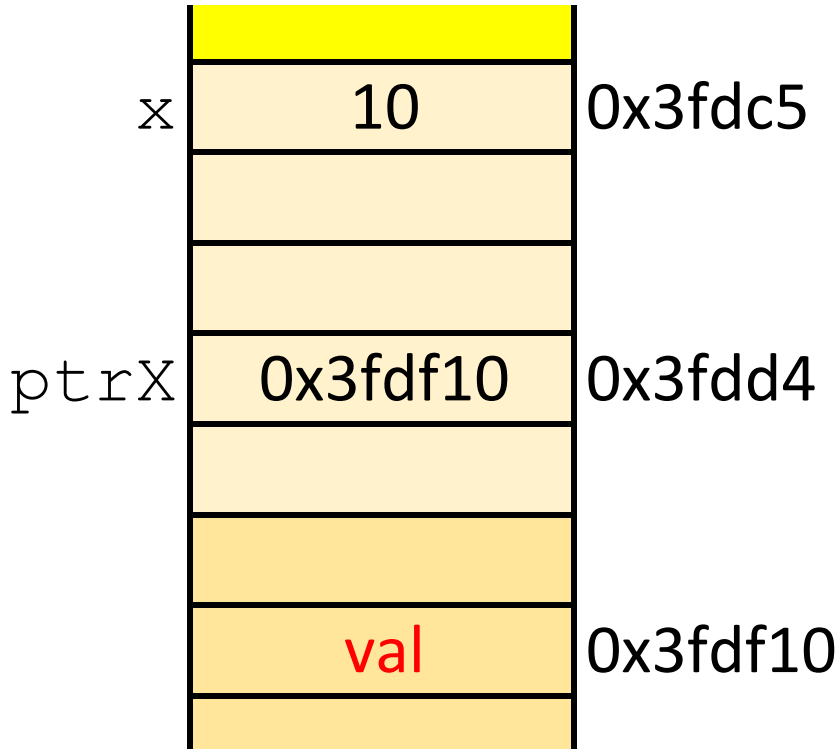
`tipo *nom_var;`

- Ejemplo: `int *x = 10`
  - `x` es el nombre de la variable. '*' indica que esa variable es un apuntador.
  - Al ser `x` un apuntador, su valor se considera una dirección de memoria, en donde se encuentra *otro* valor de interés.

Un apuntador debe incializarse con una dirección válida.

Apuntadores o punteros

```
int x = 10;
int *ptr = 0x3fdf10;
```

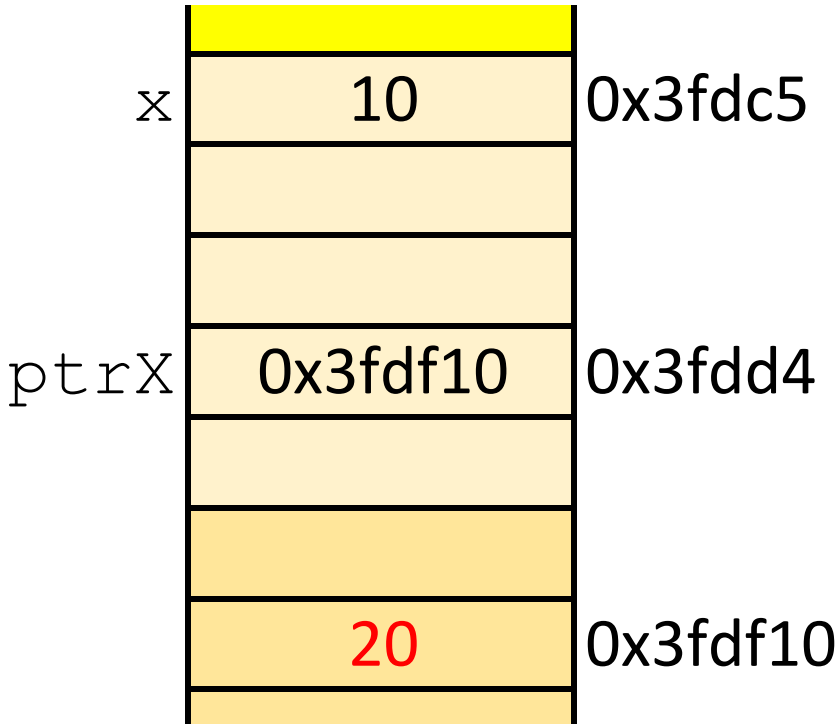


El valor del apuntador es una dirección, en donde está un valor de interés.

¿De qué tipo es el valor en la dirección que almacena el apuntador?

Apuntadores o punteros

```
int x = 10;
int *ptr = 0x3fdf10;
```




```
int y = *ptr;  // y = 20
```

`ptr` es una variable de tipo apuntador. Ocupa una localidad de memoria, tiene una dirección y un valor.

La dirección de `ptr` es `0x3fdd4`. El valor de `ptr` es `0x3fdf10`. El valor a donde apunta `ptr` es 20.

Apuntadores o punteros

```
tipo *nom_var;
```

- Un apuntador, '*apunta*' a la dirección de otra variable, ¿para qué?

Apuntadores o punteros

```
tipo *nom_var;
```

- Un apuntador, '*apunta*' a la dirección de otra variable, ¿para qué? $\rightarrow$ manipular a esa variable de manera indirecta mediante el apuntador.

Apuntadores o punteros

```
tipo *nom_var;
```

- Un apuntador, '*apunta*' a la dirección de otra variable, ¿para qué? $\rightarrow$ manipular a esa variable de manera indirecta mediante el apuntador.
- El `tipo` es muy importante, indica a que tipo de variables puede ese apuntador apuntar.

Apuntadores o punteros

`tipo *nom_var;`

- Un apuntador, '*apunta*' a la dirección de otra variable, ¿para qué? $\rightarrow$ manipular a esa variable de manera indirecta mediante el apuntador.
- El `tipo` es muy importante, indica a que tipo de variables puede ese apuntador apuntar.

  `int *x;`, x solo puede apuntar a variables del tipo entero.

  `float *x;`, x solo puede apuntar a variables del tipo float.

Apuntadores o punteros

`tipo *nom_var;`

- Un apuntador, '*apunta*' a la dirección de otra variable, ¿para qué? $\rightarrow$ manipular a esa variable de manera indirecta mediante el apuntador.
- El `tipo` es muy importante, indica a que tipo de variables puede ese apuntador apuntar.

  `int *ptrX;`, $ptrX$ solo puede apuntar a variables del tipo entero.

  `float *ptrY;`, $ptrY$ solo puede apuntar a variables del tipo float.

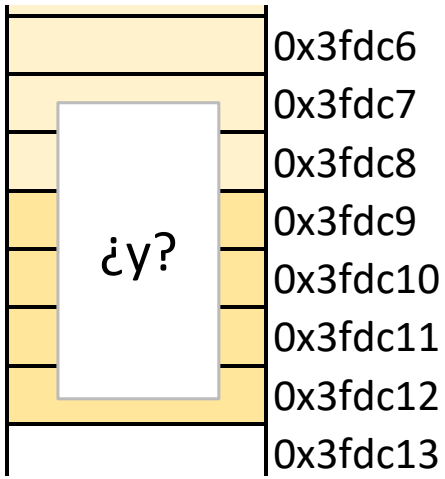
```
int x = 3; ptrX = &x;
```

```
float y = 0.7; ptrY = &y;
```

Apuntadores o punteros

`tipo *nom_var;`

- Recordar que el modelo lógico de la memoria es un conjunto consecutivo de localidades de memoria, de **1 byte de longitud** cada una.
- Si tenemos `int y = 10`, ¿cuántas localidades de memoria ocupa la variable `y`? (se puede saber usando `sizeof(y)`).



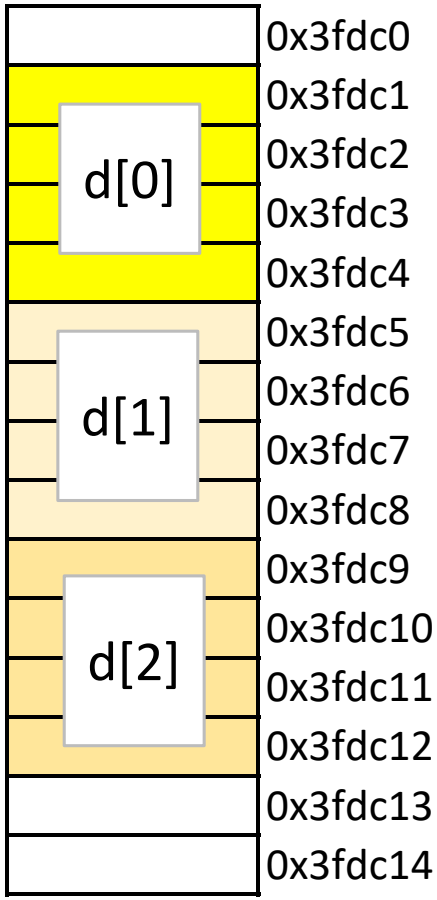
¿Cual es la dirección de la variable `int y;`?

Apuntadores o punteros

```c
#include <stdio.h>

int main(){
    int d[3] = {1,2,3}; // Tres variables contiguas
//La direcciones de las variables son contiguas

    printf("\nValor de variable: \t%d", d[0]);
    printf("\nDireccion de variable: \t%X", &d[0]);

    printf("\nValor de variable: \t%d", d[1]);
    printf("\nDireccion de variable: \t%X", &d[1]);

    printf("\nValor de variable: \t%d", d[2]);
    printf("\nDireccion de variable: \t%X", &d[2]);

    printf("\nValor de variable: \t%X", d);
    printf("\nDireccion de variable: \t%X", &d);

    return 0;
}
```

Programa 2: Variable de tipo arreglo en localidades contiguas (`varsArray.c`)

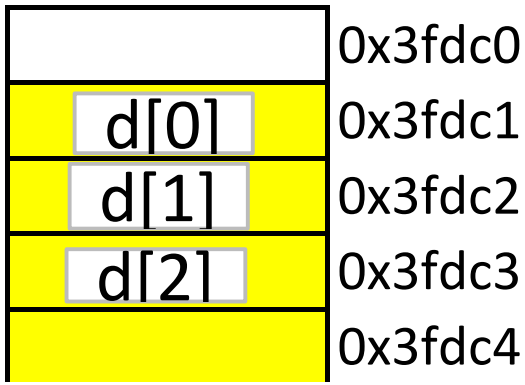
Apuntadores o punteros



Organización de variables consecutivas en memoria, de acuerdo con el tipo de dato que representan, en este caso `int d[3];`

Apuntadores o punteros

Tipo	Bytes
`char`	1
`int`	4
`float`	4
`double`	8



Organización de variables consecutivas en memoria, de acuerdo con el tipo de dato que representan, en este caso `char d[3];`

Apuntadores o punteros

```
int x = 10;

int y = 15;

int z = 20;

printf("\nDireccion de x: \t%X", &x);
printf("\nDireccion de y: \t%X", &y);
printf("\nDireccion de z: \t%X", &z);
```

Apuntadores o punteros

```
char x = 10;
int y = 15;
float z = 20.0;
double w = 25.0;
char c = 65;

printf("\nDireccion de x: \t%X", &x);
printf("\nDireccion de y: \t%X", &y);
printf("\nDireccion de z: \t%X", &z);
printf("\nDireccion de w: \t%X", &w);
printf("\nDireccion de c: \t%X", &c);
```

El tamaño en bytes de las variables puede variar.

Apuntadores o punteros

```
float z = 20.0;

printf("\nValor de z: \t%f", z);
printf("\nDireccion de z: \t%X", &z);

float *ptrFloat = &z;
printf("\nValor de ptrFloat: \t%X", prtFloat);
```

Apuntadores o punteros

```
float z = 20.0;

printf("\nValor de z: \t%f", z);
printf("\nDireccion de z: \t%X", &z);

float *ptrFloat = &z;
printf("\nValor de ptrFloat: \t%X", prtFloat);

printf("\nValor en la dirección a la que \
                apunta ptrFloat: \t%f", *prtFloat);
```

Apuntadores o punteros

```c
#include <stdio.h>

int main(){
   float z = 20.0;

   printf("\nValor de z: \t%f", z);
   printf("\nDireccion de z: \t%X", &z);

   float *ptrFloat = &z;       //accede a direccion
   printf("\nValor de ptrFloat: \t%X", ptrFloat);

   printf("\nValor en la direccion a la que \
        apunta ptrFloat: \t%f", *ptrFloat); //lectura de direccion

   *ptrFloat = 0.5;  //escritura en direccion
   printf("\nValor de z: \t%f", z);

   return 0;
}
```

Programa 3: Uso de apuntadores (pointers1.c)

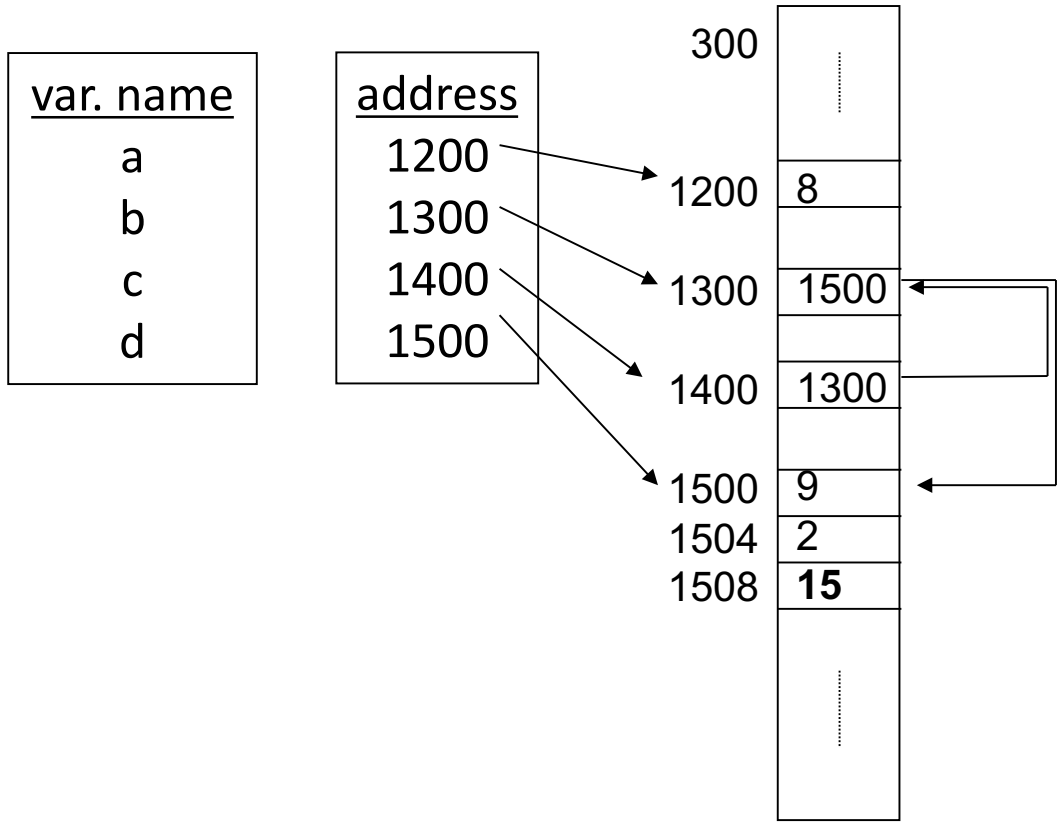
Apuntadores o punteros

Pasos para usar apuntadores:

- Declarar el apuntador, debe coincidir con el tipo a donde se va a apuntar.
- Inicializar el apuntador, con una dirección válida (usar el operador &)
- El valor del apuntador es una dirección (se accede solo con el nombre de la variable)
- El valor que se encuentra en la dirección a donde apunta el apuntador se accede o lee (o se modifica) con el operador '*' y el nombre de la variable apuntador.

Apuntadores o punteros

```
int a=10;
int *b;
int **c;
int d[3]={1,2,3};

b=&a;
*b=5;
c=&b;
*(*c)=8;
b=d;
*(*c)=9;
*(*c+2)=15;
```



Variables y apuntadores a variables.

Apuntadores o punteros

¿Porqué usar punteros?:

- El código es más eficiente, al acceder directamente a la memoria y manipularla. Debe realizarse con mucha precaución.
- Permiten crear memoria de manera dinámica (que no se sabe si se requerirá o no en tiempo de compilación. Se usa la que realmente se requiere y cuándo se requiere)
- Simplifican el paso de parámetros y recolección de resultados, evitando la duplicidad del espacio en memoria (en vars simples, estucturas u objetos).

Operaciones con punteros

Asignación '=': Un puntero puede asignarse a otro puntero, por ejemplo:

```
int a = 10,  b = 15;
int *p, *q;

p = &a;
q = p;
printf("En la direccion %X está el dato %d\n", q, *q);
p = &b;
printf("En la direccion %X está el dato %d\n", q, *q);
```

¿Qué se obtiene como salida?

Operaciones con punteros

Operaciones aritméticas:

```
int a = 10;
int *ptrInt = &a;

printf("\nDireccion de x: %X", ptrInt);
ptrInt++;

printf("\nValor de ptrInt: %X\n", ptrInt);
```

¿Qué se obtiene como salida?

Operaciones con punteros

Operaciones aritméticas:

```
int a = 10;
int *ptrInt = &a;

printf("\nDireccion de x: %X", ptrInt);
ptrInt++;

printf("\nValor de ptrInt: %X\n", ptrInt);
```

¿Qué se obtiene como salida? ¿Hay un valor válido en la nueva dirección del apuntador?

Operaciones con punteros

Operaciones aritméticas:

- Podemos realizar cualquier operación +, -, con apuntadores, lo que estamos haciendo es avanzar o retroceder en las localidades de memoria.
- +1 avanza a la siguiente dirección, +2 a las siguientes dos direcciones, +3....El **offset** está dado por **el tipo de dato** del apuntador.
- Ocurre lo mismo con -1, -2, ...
- Se podrían usar *2, *3, ..., pero es poco común

Operaciones con punteros

```
int a = 10;
int *ptrInt = &a;

printf("\nDireccion de x: %X", ptrInt);
ptrInt += 1;
ptrInt += 2;
ptrInt += 3;
printf("\nValor de ptrInt: %X\n", ptrInt);
```

Si suponemos que la primera impresión en pantalla es `0xA14F0`, ¿qué resultará en la segunda impresión? ¿Hay un valor válido en la nueva dirección del apuntador?

Operaciones con punteros

```
int a = 10;
int *ptrInt = &a;
```

- `ptrInt` es una dirección válida ($\&a$).
- `*ptrInt` es un valor válido (10, que es el valor de a a donde el apuntador está apuntando)
- `(ptrInt + 2)` es una dirección nueva (no necesariamente válida).
- `*(ptrInt + 2)` es el valor en la dirección indicada, no necesariamente es válido.
- La aritmética con punteros debe ser consistente, y asegurar que las nuevas direcciones son válidas, con valores válidos.

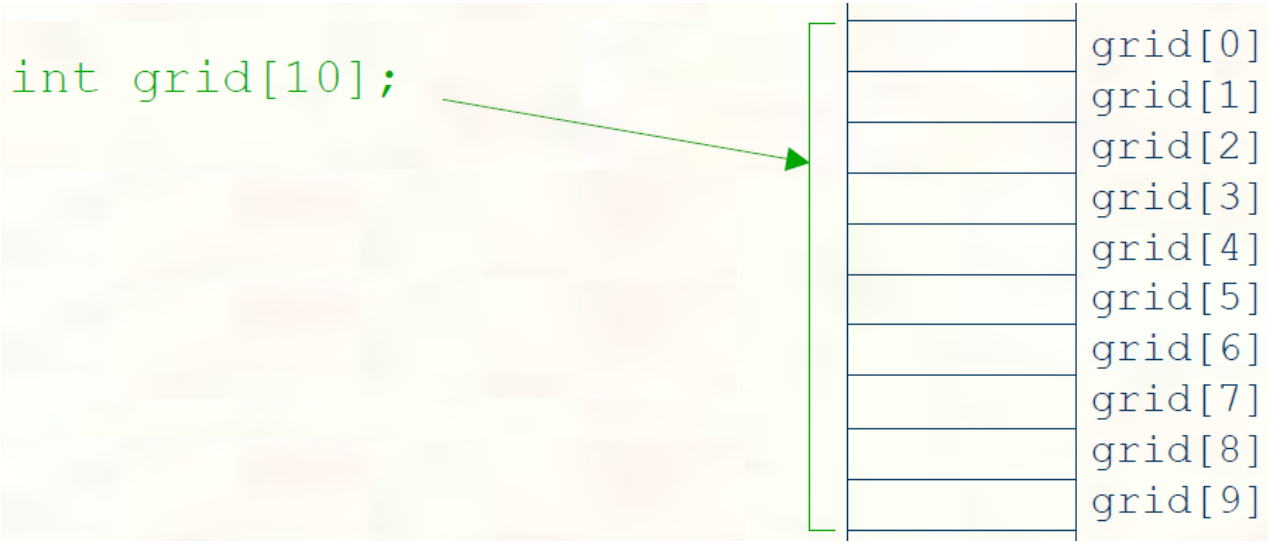
Operaciones con punteros

Buena práctica:

- En la declaración `int *ptrInt;`, el apuntador no está inicializado, puede tener cualquier valor basura (mucho ojo!). Si no está inicializado, no puede usar.
- Al declarar una variable de tipo apuntador, si no se sabe aún a que debe apuntar, se debe inicializar con `NULL` (dirección no válida).
- Antes de usar un apuntador, ya sea para leer su valor o para escribir sobre el, se debe verificar que el aputnador es distinto de `NULL`.

```
int * ptrInt = NULL;

if ( ptrInt != NULL){...}
```

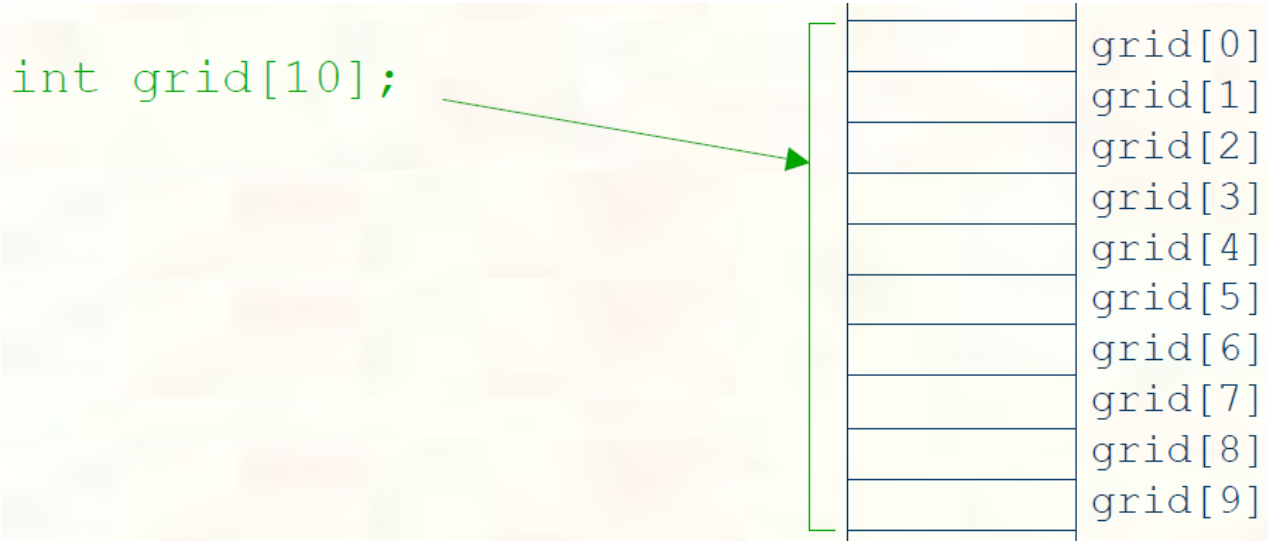
Operaciones con punteros



Variables y apuntadores a variables.

- `int *ptrInt = grid;` apunta a $grid[0]$.
- `int *ptrInt = &grid[0];` igual que el anterior.

Operaciones con punteros

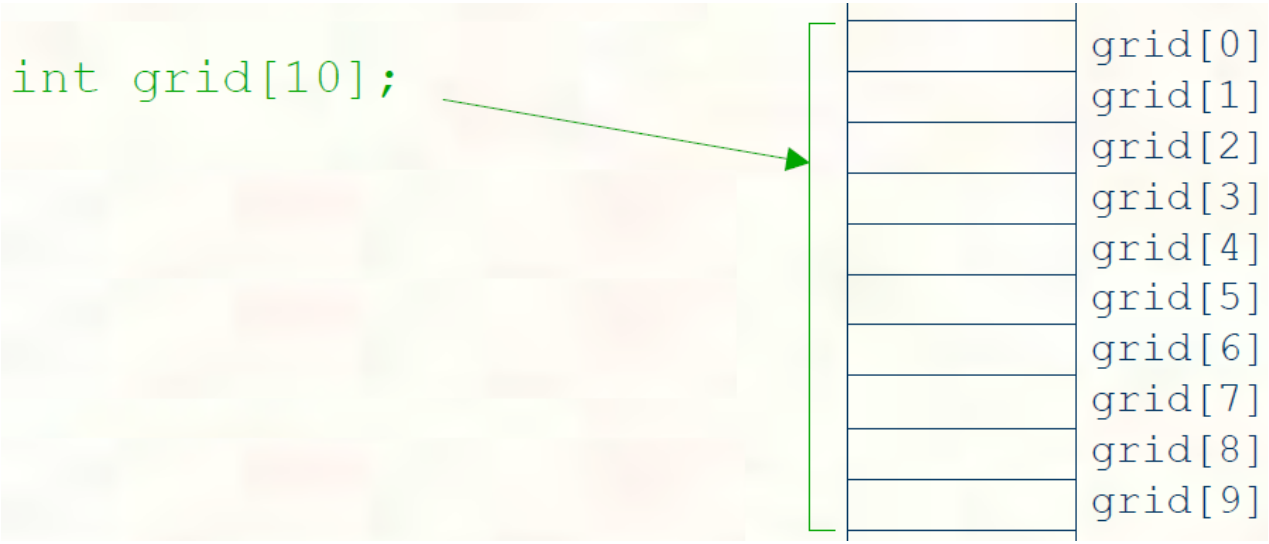


Variables y apuntadores a variables.

```
*ptrInt = 2; //grid[0] = 2.
ptrInt ++;
ptrInt += 2;
*ptrInt = 8;
```

¿Qué posición del arreglo tiene el valor 8?

Operaciones con punteros



Variables y apuntadores a variables.

```
ptrInt = grid;
for (int i = 0; i < 10; i++)
      printf("\ngrid[%d]: %d",i, *(ptrInt++));

ptrInt = grid;
for (int i = 0; i < 10; i++)
     *(ptrInt++) = i*2;
```

Punteros y cadenas de caracteres

Una cadena de caracteres es un arreglo de variables del tipo `char`. Como un arreglo, se puede manipular por medio de apuntadores.

```
char *nombre = "Javier Hernandez";
printf("%s", nombre);
```

El compilador generalmente añade al final de la cadena el carácter nulo, para detectar la longitud del arreglo (o el final de las localidades que esa cadena ocupa).

Muchas funciones del lenguaje que operan sobre cadenas de caracteres requieren manipular variables del tipo `*char`

Punteros y funciones

Cuerpo general de una función:

```
tipo nombre_funcion(tipo arg1, tipo arg2, ...){

    tipo local1;
    tipo local2;
    ...
    //logica que usa variables locales y
    //varibles de los argumentos
    ...
return variable_tipo;
}
```

Todas las variables son locales, se destruyen cuando la función termina. En return, lo que se regresa es un valor.

Punteros y funciones

Cuerpo general de una función:

```
tipo nombre_funcion(tipo arg1, tipo *arg2, ...){

    tipo local1;
    tipo *local2;
    ...
    //logica que usa variables locales y
    //varibles de los argumentos
    ...
return variable_tipo;
}
```

Se pueden pasar PUNTEROS a las funciones. Se recibe una dirección, se opera sobre esa dirección. El cambio realizado sobre esa dirección, al interior de la función, perdurará después de que termine la función.

Apuntadores en Funciones

```c
#include <stdio.h>

//intercambia los valores
void swap(int x, int y);

int main(void){
    int x = 1;
    int y = 0;

    swap(x,y);

    printf("Valor de x: %d\n", x);
    printf("Valor de y: %d\n", y);
    return 0;
}

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

    return;
}
```

Programa 4: Intercambio de valores dentro de una función (`swap.c`)

¿Funciona?

Apuntadores en Funciones

```c
#include <stdio.h>

//intercambia los valores
void swap(int x, int y);

int main(void){
    int x = 1;
    int y = 0;

    swap(x,y);

    printf("Valor de x: %d\n", x);
    printf("Valor de y: %d\n", y);
    return 0;
}

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

    return;
}
```

Programa 5: Intercambio de valores dentro de una función (`swap.c`)

Cuando se llama a `swap(x,y)`, se pasa una copia de las variables x e y, es como si fueran otras variables. Las manipulaciones al interior de `swap` se hace con otras variables **locales**, no a las definidas en la función `main`.

Apuntadores en Funciones

```c
#include <stdio.h>

//intercambia los valores
void swap(int *x, int *y);

int main(void){
    int x = 1;
    int y = 0;

    swap(&x,&y);

    printf("Valor de x: %d\n", x);
    printf("Valor de y: %d\n", y);
    return 0;
}

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

    return;
}
```

Programa 6: Intercambio de valores dentro de una función (`swapPtr.c`)

Pasamos las direcciones, para manipular a las variables directamente.

Apuntadores en Funciones

```
tipo funcion(tipo *var,.....){
   ...
   //manipular var (direccion), en operaciones de escritura
   ...
 return (tipo);
}
....
tipo var1 = val;
...
funcion(&var1,....);

//cuaquier escritura en la función modificara el valor de var1
...
```

Apuntadores en Funciones

Paso de parámetros por referencia: los parámetros transferidos a la función no son valores sino las direcciones de las variables que contienen esos valores, de manera que los parámetros actuales pueden sufrir cambios al regresar el control a la función que la llamó. Pueden ser transferidos variables de cualquier tipo de datos.

```
tipo identificador(tipo *id1, tipo *id2, . . . , tipo *idN)
{
   /* cuerpo de la función */
   //usar id1, id2,... para acceder a las direcciones
   //usar *id1, *id2, ... para acceder a los valores en esas direcciones
   //una operación de escritura modificará el valor en la dirección a la que ap
}
```

Apuntadores en Funciones

```c
#include <stdio.h>

int myStrLen(char *);

int main(void){
   char *cadena = "Una cadena de caracteres relativamente larga";

   int longitud = myStrLen(cadena);

   printf("La cadena:\n\'%s\'\nTiene %d caracteres.", cadena, longitud);

   return 0;
}

/* Funcion que devuelve el largo de la cadena */
int myStrLen(char *str){
   if(str == NULL)
      return 0;

   char *p = str;
   int len = 0;

   while(*p != '\0'){
      len++;
      p++;
   }
   return len;
}
```

Programa 7: Cadena de caracteres y punteros (`pointchar.c`)

Apuntadores en Funciones

- Al llamar a la función, se pasan las direcciones de las variables, usando el operador &, excepto si la variable es un arreglo.

```
funcion(&variable1, &variable2, arreglo1)
```

Apuntadores en Funciones

```
#include <stdio.h>

int main()
{
    void fillArray(int d[]);
    int d[3];

    fillArray(d);


    printf("%d %d %d", d[0], d[1], d[2]);

    return 0;
}

void fillArray(int d[]){
    d[0] = 5;
    d[1] = 10;
    d[2] = 15;
}
```

Programa 8: Pasando un arreglo como parámetro en una función (`ejer01.c`)

¿Qué imprime el código anterior?

Apuntadores en Funciones

```c
#include <stdio.h>

int main()
{
    void fillArray(int *d);
    int d[3];

    fillArray(d);


    printf("%d %d %d", d[0], d[1], d[2]);

    return 0;
}

void fillArray(int *d){
    *d = 5;
    *(d + 1) = 10;
    *(d + 2) = 15;
}
```

Programa 9: Pasando un arreglo como parámetro en una función (`ejer02.c`)

¿Qué imprime el código anterior?

Apuntadores en Funciones

```
#include <stdio.h>

int main()
{
    void fillArray(int *d);
    int d[5];

    fillArray(d);

    for(int i = 0; i < 5; i++)
       printf("%d\t", d[i]);

    return 0;
}

void fillArray(int *d){
    printf("Ingresa 5 numeros:\n");
    for(int i = 0; i < 5; i++)
       scanf("%d", &d[i]);
}
```

Programa 10: Pasando un arreglo como parámetro en una función (`ejer03.c`)

¿Qué imprime el código anterior?

Apuntadores en Funciones

```
#include <stdio.h>

int main()
{
    void fillArray(int *d);
    int d[5];

    fillArray(d);

    for(int i = 0; i < 5; i++)
       printf("%d\t", *(d + i));

    return 0;
}

void fillArray(int *d){
    printf("Ingresa 5 numeros:\n");
    for(int i = 0; i < 5; i++)
       scanf("%d", (d + i));
}
```

Programa 11: Pasando un arreglo como parámetro en una función (`ejer04.c`)

¿Qué imprime el código anterior?

Memoria dinámica

```
int d[10];
```

- la memoria es estática (fija, se han reservado 10 localidades de memoria de 4 bytes cada una). ¿Qué pasa si no se cuantas localidades voy a necesitar sino hasta el momento de la ejecución del programa, o si al momento necesito más memoria?.
- En este caso, se requiere memoria dinámica (creada no en tiempo de compilación, sino en tiempo de ejecución).
- Esta memoria se toma de un lugar distinto de donde se almacenan las variables globales y locales del programa, es un espacio llamado *heap*.
- La memoria se *pide* (en bytes) y se debe *regresar*.

Memoria dinámica

- La memoria se pide con la función

```
void* malloc (unsigned numero_de_bytes );
```

- Generalmente se hace un cast para asignar el tipo correcto. Se usa junto con `sizeof`

```
int *ptrInts = (int*) malloc
  (sizeof(int)*num_ints);
```

- En el ejemplo previo, `num_ints` es el número de enteros que se requieren, y que se sabe en tiempo de ejecución.

Memoria dinámica

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

int main()
{
    void fillArray(int *d, int n);

    int n;

    printf("Cuantos datos deseas ingresar:");
    scanf("%d", &n);

    int *d = (int*) malloc(3*sizeof(int));

    fillArray(d, n);

    for(int i = 0; i < n; i++)
       printf("%d\t", *(d + i));

    free(d);

    return 0;
}

void fillArray(int *d, int n){
    printf("Ingresa %d numeros:\n", n);
    for(int i = 0; i < n; i++)
       scanf("%d", (d + i));
}
```

Programa 12: Memoria dinámica (`ejer05.c`)

Memoria dinámica

- Realizar un programa que reciba como entrada desde el teclado una cantidad variable de numeros reales y los almacene en un arreglo.
- Desplegar el contenido del arreglo para verificar que los datos se leyeron correctamente.

Memoria dinámica

- Realizar un programa que reciba como entrada desde el teclado una cantidad variable de numeros reales y los almacene en un arreglo.
- Después, realizar una función que reciba como parámteros el arreglo de números reales, y calcule la sumatoria de todos esos números.
- Imprimir el valor de la sumatoria.

Variables de tipo `struct`

```c
typedef struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
} Person;

Person person;

Person *ptrPerson;
ptrPerson = (Person*) malloc(sizeof(Person));
```

Variables de tipo `struct` y apuntadores.

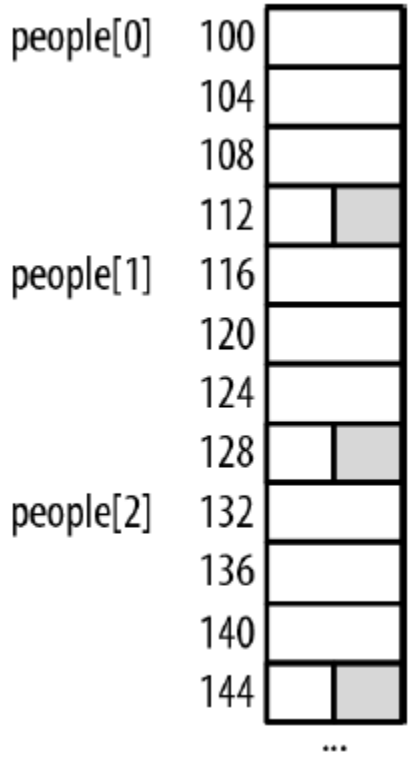
Variables de tipo `struct`

```c
typedef struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
} Person;

Person person;
person.firstName = (char*)malloc(strlen("Emily")+1);
strcpy(person.firstName,"Emily");
person.age = 23;

Person *ptrPerson;
ptrPerson = (Person*)malloc(sizeof(Person));
ptrPerson->firstName = (char*)malloc(strlen("Emily")+1);
strcpy(ptrPerson->firstName,"Emily");
ptrPerson->age = 23;
```

Variables de tipo `struct` y apuntadores.

Variables de tipo `struct`

```c
typedef struct _alternatePerson {
    char* firstName;
    char* lastName;
    char* title;
    short age;
} AlternatePerson;


AlternatePerson people[30];
```



Variables de tipo `struct` y apuntadores.

Ejercicios

Ejercicio 4.a: Dado un arreglo con n números, $[x_0, x_1, \ldots, x_{n-1}]$, se pueden calcular los siguientes estadísticos:

- **Media aritmética:** $\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} x_i$
- **Desviación estándar:** $s = \sqrt{\frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2}$
- **Máximo y mínimo:** valores más grande y más pequeño en el arreglo.

Escriba un programa que calcule éstos estadísticos, introduciendo por teclado el tamaño `N` del arreglo y sus respectivos elementos, pueden ser enteros o reales. Se deben llamar funciones para calcular cada estadístico.

- **Entrada:**

```
Ingresa el tamaño del arreglo: 5
Ingresa 5 números:
1.1 1.2 0.9 1.0 1.23
```

- **Salida:**

```
Media = 1.086
Desv. Est. = 0.138
Máximo = 1.230
Mínimo = 0.900
```

Ejercicios

Ejercicio 4.b: Escriba un programa que a partir de dos número ingresados por teclado realice las operaciones aritméticas de suma, resta, multiplicación, y división usando solamente punteros.

- **Entrada:**

```
Ingresa dos numeros: 2.38 9.78
```

- **Salida:**

```
2.38+9.78=12.16
2.38-9.78=-7.40
2.38x9.78=23.28
2.38/9.78=0.24
```

Ejercicios

Ejercicio 4.c: Escriba un programa en el que se ingrese por teclado una cadena de caracteres y todas las letras minúsculas las convierta a mayúsculas usando apuntadores. Además, implemente la conversión de minúsculas a mayúsculas en una función. En C, cada carácter se puede representar como un número entero, como se muestra en la siguiente tabla. Por ejemplo, a$= 97$ y a$-32 =$A, de modo que A$=$65.

- **Entrada:**

  ```
  Ingrese un texto:
  hola, mundo
  ```

- **Salida:**

  ```
  Texto en mayusculas:
  HOLA, MUNDO
  ```

Decimal	Carácter	Decimal	Carácter
65	A	97	a
66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
74	J	106	j
75	K	107	k
76	L	108	l
77	M	109	m
78	N	110	n
79	O	111	o
80	P	112	p
81	Q	113	q
82	R	114	r
83	S	115	s
84	T	116	t
85	U	117	u
86	V	118	v
87	W	119	w
88	X	120	x
89	Y	121	y
90	Z	122	z

Ejercicios

Ejercicio 4.d: Escriba un programa para eliminar elementos duplicados en un arreglo que tiene n valores enteros que se introducen por teclado. Después de realizar la operación de eliminación, el arreglo solo debe contener un valor entero único.

- **Entrada:**

```
Ingrese el tamano del arreglo: 10
Ingrese los elementos del arreglo:
1 2 3 1 2 3 1 2 3 4
```

- **Salida:**

```
Arreglo sin elementos duplicados:
1       2       3       4
```

Bibliografía

- B. W. Kernighan & D. M. Ritchie. *The C Programming Language*. 2nd Edition. Prentice Hall, 1988.
- K. N. King. *C Programming: A Modern Approach*. 2nd Edition. W. W. Norton & Company, 2008.
- Greg Perry. *Absolute Beginner's Guide To C*. 2nd Edition. Sams Publishing, 1994.
- Stephen Prata. *C Primer Plus*. 5th Edition. Sams, 2004.
- Peter V. Linden. *Expert C Programming: Deep C Secrets*. Prentice Hall, 1994.
- Francisco Javier Ceballos. *Enciclopedia del Lenguaje C*. Alfaomega, 1997.