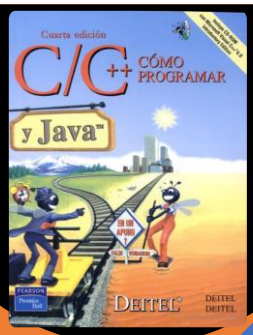
Algunos ejemplos de esta clase han sido tomados desde:

- **http://www.it.uc3m.es/pbasanta/asng/course_notes**



Punteros en C

Estructuras de Datos, 620505

1

¿Por qué usar punteros?

La razón de ser principal de los punteros reside en manejar datos alojados en la zona de memoria dinámica.

Se usan principalmente porque optimizan el código a ejecutar.

2

¿Qué es un puntero?

- Todo dato está almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada también como un dato más.

- Un puntero es una **variable** que contiene una **dirección de memoria,** normalmente, esa dirección es la posición de otra variable en la memoria.

Dos conceptos son fundamentales para comprender el funcionamiento de los punteros: el **tamaño de las variables** y su **posición en memoria**.

- Así, cuando una variable se declara, se asocian tres atributos fundamentales con la misma: su ***nombre****, su* ***tipo*** y su ***dirección*** *en memoria.*

3

Declaración de punteros

- Una declaración de un puntero consiste en un tipo base, un * y el nombre de la variable.
- La forma general es: **tipo * var_nombre;**

Donde tipo es cualquier tipo válido y var_nombre es el nombre de la variable que se ha declarado como puntero.

- Declaración de un puntero:

```
char *p1;          //puntero a un char .
int *p2;           //puntero a un entero.
float *p3;         //puntero a un float
int *ptr1, *ptr2;  //punteros a enteros
```

4

Declaración de punteros

- El tipo base del puntero define el tipo de variables a las que puede apuntar.
- Técnicamente, cualquier tipo de puntero puede apuntar a cualquier dirección de la memoria.
- Sin embargo, toda la aritmética de punteros está hecha en relación a sus tipos base, por lo que es importante declarar correctamente el puntero.

5

Punteros en C:

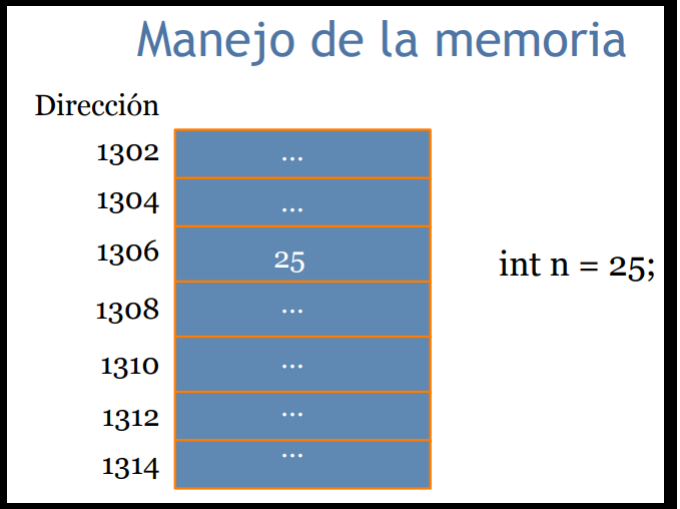
Los punteros sólo almacenan una dirección, sin importar a que tipo apunten, por lo tanto usan 4 Bytes.

Tipo T	Tamaño (bytes) [a]	Puntero a T	Tamaño (bytes)	Ejemplo de uso
int	4	int *	4	int *a, *b, *c;
unsigned int	4	unsigned int *	4	unsigned int *d, *e, *f;
short int	2	short int *	4	short int *g, *h, *i;
unsigned short int	2	unsigned short int *	4	unsigned short int *j, *k, *l;
long int	4	long int *	4	long int *m, *n, *o;
unsigned long int	4	unsigned long int *	4	unsigned long int *p, *q, *r;
char	1	char *	4	char *s, *t;
unsigned char	1	unsigned char *	4	unsigned char *u, *v;
float	4	float *	4	float *w, *x;
double	8	double *	4	double *y, *z;
long double	8	long double *	4	long double *a1, *a2;

6

Manejo de la memoria

- Las direcciones de memoria dependen de la arquitectura del ordenador y de la gestión que el sistema operativo haga de ella.
- En C no se debe indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable (direcciones de memoria). Lo que interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.



7

El valor NULL en los punteros

- Uno de los principales errores que se pueden cometer cuando se programa en C es utilizar punteros que no apuntan realmente a nada.
- Para evitar esto, es importante que un puntero siempre apunte a una variable.
- Cuando no es posible saber a qué variable tiene que apuntar el puntero, debe utilizarse un valor especial, NULL, como valor inicial de los punteros.

8

PUNTERO NULL

- Un *puntero nulo (NULL) apunta a ninguna parte en particular,* no direcciona ningún dato válido en memoria.
- Se utiliza para proporcionar a un programa un medio de conocer cuando una variable puntero no direcciona a un dato válido.
- Una forma de inicializar una variable puntero a nulo es:

```
char *p = NULL;
```

- Algunas funciones C también devuelven el valor NULL si se encuentra un error o no existe respuesta.

9

9

Generalidades

- Un puntero inicializado adecuadamente apunta a alguna posición específica de la memoria.
- Un puntero no inicializado, como cualquier variable, tiene un valor aleatorio hasta que se inicializa el puntero.
- Se requiere que las variables puntero utilicen direcciones de memoria válida.

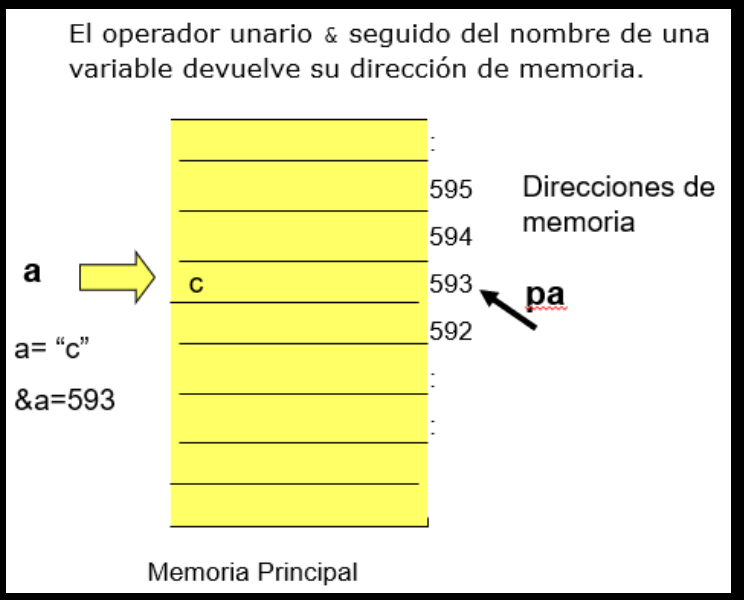
10

10

Operador de dirección &

- Operador unario que se aplica como prefijo a una variable.
- Devuelve la dirección de memoria en la que se encuentra almacenada la variable.

- Ejemplo:
```
char a="c";
char *pa=&a;
float n = 3.9;
float *ptr = &n;
```



11

Operador de contenido (*)

- Se aplica como prefijo a un puntero.
- Devuelve el valor contenido en la dirección de memoria almacenada en el puntero.

```
float n = 3.9;
float *ptr = &n;
n = 3.8;
printf("El valor de ptr es %f\n", *ptr);
```

¿Qué valor se imprime?:

12

```c
#include <stdio.h>
int main(int argc, char** argv)
{
    int num1, num2;
    int *ptr1, *ptr2;

    ptr1 = &num1;
    ptr2 = &num2;

    num1 = 10;
    num2 = 20;

    *ptr1 = 30;
    *ptr2 = 40;

    *ptr2 = *ptr1;

    return 0;
}
```

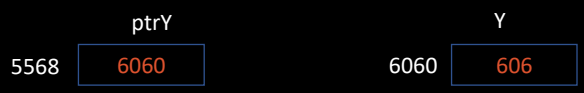
Operador *

El **operador *** también puede entregar el contenido de aquello que está almacenado en la dirección de memoria apuntada por una variable tipo puntero.

¿Cuánto vale *ptr2 finalmente?
¿qué imprime la siguiente instrucción?
printf("*ptr2:%d", *ptr2);

13

13



El operador * llamado operador de *indirección* u operador *de desreferencia*, devuelve el valor de la variable a la que apunta.

Por ejemplo, la instrucción printf("%d", *ptrY);
imprime el valor 606.

Un **error frecuente** se produce al intentar acceder al contenido de una variable que no ha sido inicializada aún; este error puede alterar valores o provocar el fallo de la aplicación, por lo que debe se debe tener cuidado al programar.

14

```c
/* Figura 7.4: fig07_04.c
   Uso de los operadores & y * */
#include <stdio.h>

int main()
{
   int a;         /* a es un entero */
   int *ptrA;     /* ptrA es un apuntador a un entero */

   a = 7;
   ptrA = &a;     /* ptrA toma la dirección de a */

   printf( "La direccion de a es %p"
           "\nEl valor de ptrA es %p", &a, ptrA );

   printf( "\n\nEl valor de a es %d"
           "\nEl valor de *ptrA es %d", a, *ptrA );

   printf( "\n\nMuestra de que * y & son complementos "
           "uno del otro\n&*ptrA = %p"
           "\n*&ptrA = %p\n", &*ptrA, *&ptrA );

   return 0; /* indica terminación exitosa */

} /* fin de main */
```

15

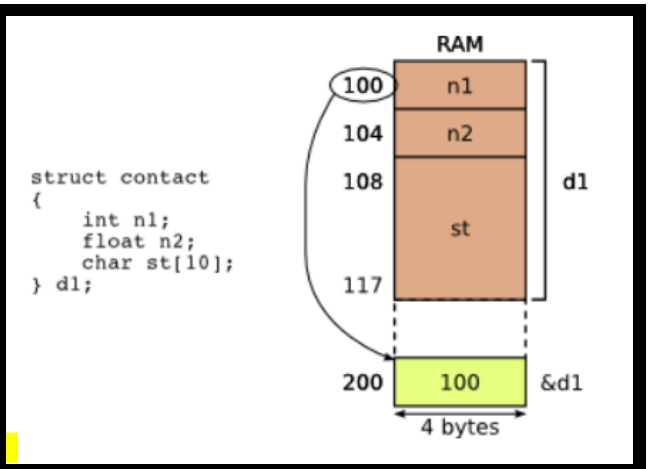
```
La direccion de a es 0012FF7C
El valor de ptrA es 0012FF7C

El valor de a es 7
El valor de *ptrA es 7

Muestra de que * y & son complementos uno del otro
&*ptrA = 0012FF7C
*&ptrA = 0012FF7C
```

Figura 7.4 Operadores & y * con apuntadores. (Parte 2 de 2.)

16

Indirección

- En C se obtiene la dirección de memoria de cualquier variable mediante el operando `&` seguido del nombre de la variable.
- En la figura, la expresión `&d1` devuelve el valor 100.
- El valor que devuelve el operador `&` depende de la posición de memoria de su operando.



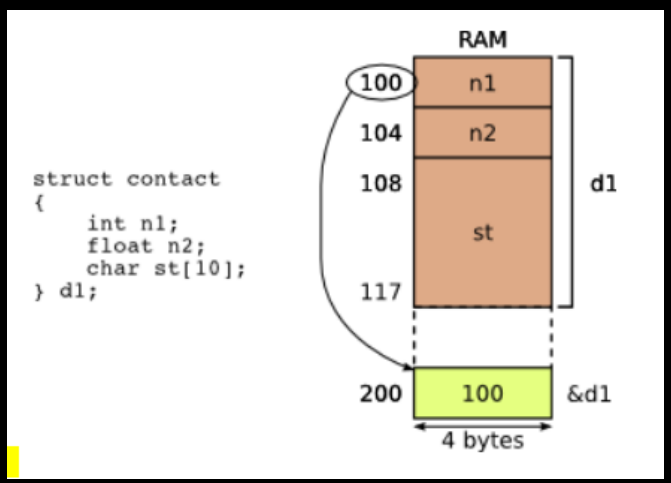
A la dirección de memoria almacenada como dato en la posición 200 se le denomina "puntero" pues su valor "apunta" a donde se encuentra la variable `d1`.

Otra forma de enunciarlo: **en la posición 200 hay un puntero a `d1`.**

17

Indirección

- Si en la posición 200 está almacenado un puntero a la variable `d1` se puede acceder a los datos mediante una "indirección".
- Para ello se toma el dato almacenado en la posición 200 y su valor (el número 100) se interpreta como una dirección.



- Se accede a esa dirección y desde allí a los campos de `d1`. Esta es una forma de acceder a `d1` de forma indirecta, a través de una "**indirección**".

18

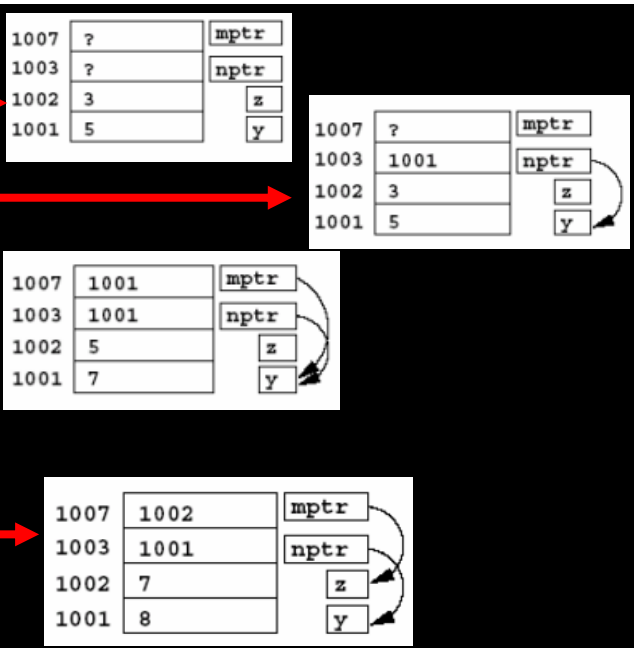
```
1   #include <stdio.h>
2   int main()
3   {
4       int num1, num2;
5       int *ptr1, *ptr2;
6
7       ptr1 = &num1;
8       ptr2 = &num2;
9
10      num1 = 10;
11      num2 = 20;
12
13      ptr1 = ptr2;
14      ptr2 = NULL;
15
16      return 0;
17  }
```

- Las líneas 4 y 5 definen dos enteros y dos punteros a enteros respectivamente.
- Los punteros no tienen valor inicial al comienzo del programa.
- Las líneas 7 y 8 asignan las direcciones de las dos variables. La dirección de una variable existe desde el principio de un programa, y por tanto esta asignación es correcta a pesar de que todavía no hemos almacenado nada en las variables `num1` y `num2`.
- Con esto, *ptr1 apunta a num2 (valor 20) y *ptr2 queda en nulo)

19

Ejemplo:

```
int main () {
    int y = 5;
    int z = 3;
    int *nptr;
    int *mptr;
    nptr = &y;
    z = *nptr;
    *nptr = 7;
    mptr = nptr;
    mptr = *z;
    *mptr = *nptr;
    y = (*nptr) + 1;
    return 0;
}
```



20

Ejercicio: ¿Cuáles valores se imprimen?

```
int main() {
int a, b, c, *p1, *p2;
p1 = &a;
 *p1 = 1;
 p2 = &b;
 *p2 = 2;
 p1 = p2;
 *p1 = 0;
 p2 = &c;
 *p2 = 3;
 printf("%d %d %d\n", a, b, c); }
```

21

```
#include int main() {
 int a,b, c, *p1, *p2;
p1 = &a; // Paso 1. La dirección de a es asignada a p1
*p1 = 1; // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;
 p2 = &b; // Paso 3. La dirección de b es asignada a p2
*p2 = 2; // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;
p1 = p2; // Paso 5. El valor del p1 = p2
*p1 = 0; // Paso 6. b = 0
p2 = &c; // Paso 7. La dirección de c es asignada a p2
*p2 = 3; // Paso 8. c = 3
printf("%d %d %d\n", a, b, c); // Paso 9. ¿Qué se imprime? }
```

22

Ejercicios

1. Dadas los siguientes segmentos de código, dibuja cómo quedaría la memoria después de ejecutarlas.

```
int x, *p1, *p2; *p1 = NULL;
*p2 = NULL;
 x = 15;
 p1 = &x;
 p2 = p1;
```

23

2. Encuentra el error en el siguiente código en C:

```
 #include <stdio.h>
main(){
float x = 55.4;
int *p;
p = &x;
printf("El valor correcto es: %f\n", x);
printf("Valor apuntado: %f \n", *p);
}
```

El Puntero p es de distinto tipo que variable a la que apunta

24

3. Encuentra el error en el siguiente código en C:

```
#include
main(){
int i, *p;
i = 50;
 *p = i;
 printf(“El valor de i es %i \n”, i);
printf(“El valor de *p es %i \n”, *p);
}
```

25

3. Encuentra el error en el siguiente código en C:

```
#include
main(){
int i, *p;
i = 50;
 *p = i;
 printf(“El valor de i es %i \n”, i);
printf(“El valor de *p es %i \n”, *p);
}
```

Falta & antes de la i

26

Comparación

- Comparar (==, !=) dos punteros significa evaluar las direcciones de memoria a las que apuntan.

```
int main() {
int uno, dos, *p_uno, *p_dos;
p_uno = &uno;
p_dos = &dos;
if(p_uno == p_dos) printf("Apuntan a la misma dirección\n");
else printf("NO apuntan a la misma dirección\n");
system("PAUSE"); }
```

27

Ejercicio:

Dadas las siguientes definiciones y asignaciones, ilustra cómo quedaría la memoria después de ejecutarlas:

```
int A = 1, B = 2, C = 3, *P1, *P2;
P1 = &A;
P2 = &C;
*P1 = (*P2)++;
P1 = P2;
P2 = &B;
*P1 - = *P2;
```

28

Acceso indirecto a campos de una estructura

En el siguiente fragmento de código se define y declara una estructura, y luego se declara un puntero que se inicializa a su dirección.

Se puede acceder a los campos de la estructura en dos formas diferentes:

Ambas formas son equivalentes, depende del programador cuál utilizar.

```
1   struct s
2   {
3       int x;
4       char c;
5   }
6
7   struct s element;
8   struct s *pointer;
9
10  pointer = &element;
```

```
(*pointer).x = 10;
(*pointer).c = 'l';
```

```
pointer->x = 10;
pointer->c = 'l';
```

29

Ejemplo de utilización en C:

```
1.  /* Definición de la estructura */
2.  struct coordenadas {
3.  float x;
4.  float y;
5.  float z;
6.  };

7.  int main() {
8.  /* Declaración de dos estructuras */
9.  struct coordenadas location1, location2;

10. /* Declaración de dos punteros */
11. struct coordenadas *ptr1, *ptr2;

12. /* Asignación de direcciones a los punteros */
13. ptr1 = &location1;
14.  ptr2 = &location2;

15.  /* Asignación de valores a la primera estructura */ ptr1->x = 3.5;
16. ptr1->y = 5.5;
17. ptr1->z = 10.5;

18.  /* Copia de valores a la segunda estructura */
19. ptr2->x = ptr1->x;
20.  ptr2->y = ptr1->y;
21.  ptr2->z = ptr1->z;
22.  return 0;
23.  }
```

30

https://www.it.uc3m.es/pbasanta/asng/course_notes/ch05s10.html

Suponga que se definen las siguientes estructuras de datos para guardar la información sobre las celdas con las que tiene posibilidad de conexión un teléfono móvil:

```
1  #define SIZE 100
2  /* Información sobre la celda */
3  struct informacion_celda
4  {
5      char nombre[SIZE];                /* Nombre de la celda */
6      unsigned int identificador;      /* Número identificador */
7      float calidad_senal;              /* Calidad de la señal (entre 0 y 100) */
8      struct informacion_operador *ptr_operador; /* Puntero a una segunda estructura */
9  };
10
11 /* Información sobre el operador */
12 struct informacion_operador
13 {
14     char nombre[SIZE];           /* Cadena de texto con el nombre */
15     unsigned int prioridad;     /* Prioridad de conexión */
16     unsigned int ultima_comprob; /* Fecha de la última comprobación */
17 };
```

1.¿Qué tamaño en bytes ocupa una variable de tipo `struct informacion_celda` en memoria?
2.Las siguientes dos líneas declaran dos variables. ¿Cuál de ellas ocupa más espacio en memoria?

```
struct informacion_celda a;
struct informacion_celda *b;
```

31

https://www.it.uc3m.es/pbasanta/asng/course_notes/ch05s10.html

```
1  struct pack3
2  {
3      int a;
4  };
5  struct pack2
6  {
7      int b;
8      struct pack3 *next;
9  };
10 struct pack1
11 {
12     int c;
13     struct pack2 *next;
14 };
15
16 struct pack1 data1, *data_ptr;
17 struct pack2 data2;
18 struct pack3 data3;
19
20 data1.c = 30;
21 data2.b = 20;
22 data3.a = 10;
23 dataPtr = &data1;
24 data1.next = &data2;
25 data2.next = &data3;
```

Decide si las siguientes expresiones son correctas y en caso de que lo sean escribe a que datos se acceden.

Expresión	Correcta	Valor
data1.c		
data_ptr->c		
data_ptr.c		
data1.next->b		
data_ptr->next->b		
data_ptr.next.b		
data_ptr->next.b		
(*(data_ptr->next)).b		
data1.next->next->a		
data_ptr->next->next.a		
data_ptr->next->next->a		
data_ptr->next->a		
data_ptr->next->next->b		

32

https://www.it.uc3m.es/pbasanta/asng/course_notes/ch05s10.html

Considera las dos versiones del siguiente programa:

Versión 1

```
#include <stdio.h>
struct package
{
int q;
};

void set_value(struct package data, int value)
{
data.q = value;
}

int main()
{
struct package p;
p.q = 10;
set_value(p, 20);
printf("Value = %d\n", p.q);
return 0;
}
```

Versión 2

```
#include <stdio.h>
struct package
{
int q;
};

void set_value(struct package *d_ptr, int value)
{
d_ptr->q = value;
}

int main()
{
struct package p;
p.q = 10;
set_value(&p, 20);
printf("Value = %d\n", p.q);
return 0;
}
```

La versión 1 del programa imprime el valor 10 por pantalla, y la versión 2 imprime el valor 20. Explica por qué.

33

¿Qué cantidad de memoria ocupan estas dos estructuras? ¿Cuál es su diferencia?

```
info_celda t[SIZE];
cell_info *t[SIZE];
```

Fuente: https://www.it.uc3m.es/pbasanta/asng/course_notes/ch05s10.html

34