

Metodología de la Programación

Tema 2. Punteros y memoria dinámica

Departamento de Ciencias de la Computación e I.A.



DECSAI
Universidad de Granada



ugr

Universidad
de Granada

ETSIIIT Universidad de Granada

Curso 2015-16

Parte I: Tipo de Dato Puntero

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros

Parte II: Gestión Dinámica de Memoria

- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Motivación

- En muchos problemas es difícil saber en tiempo de compilación la cantidad de memoria que se va a necesitar para almacenar los datos que se requieren para dicho problema.
- Este problema tendría solución si pudiéramos definir variables cuyo espacio se reserva en tiempo de ejecución.
- La memoria dinámica permite justamente eso, crear variables en tiempo de ejecución.
- La gestión de esta memoria es **responsabilidad del programador**.
- Para poder realizar la gestión es necesario el uso de variables **tipo puntero**.

Parte I

Tipo de Dato Puntero

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Definición de una variable tipo puntero

Tipo de dato puntero

Tipo de dato que contiene la dirección de memoria de otro dato.

- Incluye una dirección especial llamada *dirección nula* que es el valor 0.
- En C esta dirección nula se suele representar por la constante NULL (definida en `stdlib.h` en C o en `cstdlib` en C++).

Sintaxis

`<tipo> *<identificador>;`

- `<tipo>` es el tipo de dato cuya dirección de memoria contiene `<identificador>`
- `<identificador>` es el nombre de la variable puntero.

Ejemplo: Declaración de punteros

```
1
2  .....
3
4  // Se declara variable de tipo entero
5  int i=5;
6
7  // Se declara variable de tipo char
8  char c='a';
9
10 // Se declara puntero a entero
11 int * ptri;
12
13 // Se declara puntero a char
14 char * ptrc;
15
16 .....
17
```


Ejemplo: Declaración de punteros

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

// Se declara la variable de tipo entero

```
int i=5;
```

// Se declara la variable de tipo char

```
char c='a';
```

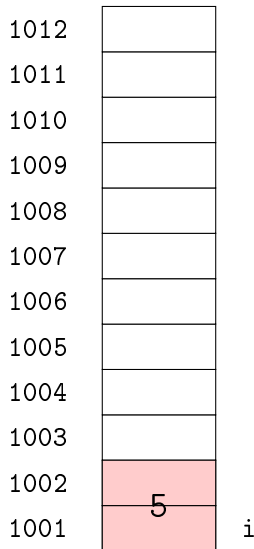
// Se declara puntero a entero

```
int * ptri;
```

// Se declara el puntero a char

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

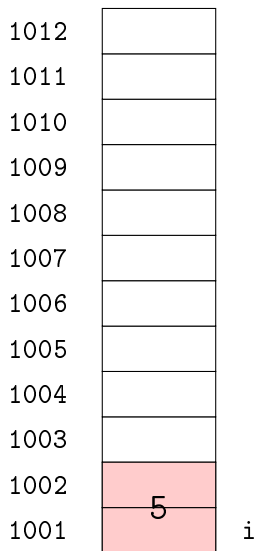
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

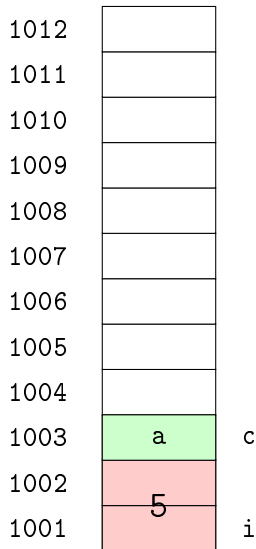
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

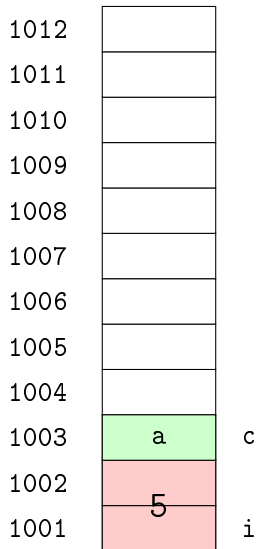
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

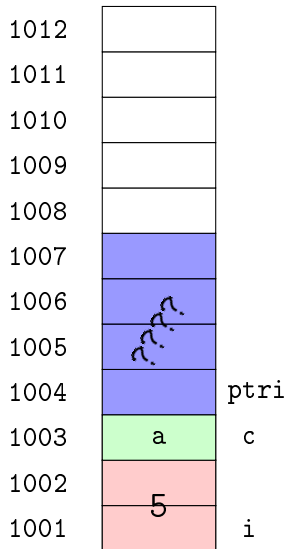
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

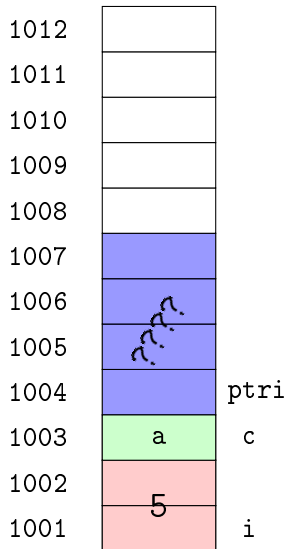
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

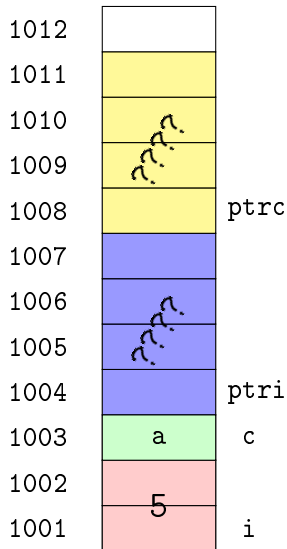
```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```


Ejemplo: Declaración de punteros



```
// Se declara la variable de tipo entero
int i=5;

// Se declara la variable de tipo char
char c='a';

// Se declara puntero a entero
int * ptri;

// Se declara el puntero a char
char * ptrc;
```

Se dice que

- `ptri` es un *puntero a enteros*
- `ptrc` es un *puntero a caracteres*.

¡Nota!

Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en sí.

¡Nota!

El tamaño de memoria reservado para albergar un puntero es el mismo independientemente del tipo de dato al que 'apunte' (será el espacio necesario para albergar una dirección de memoria, 32 ó 64 bits, dependiendo del tipo de procesador usado).

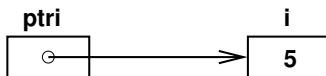
Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros**
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Operador de dirección &

- `&<var>` devuelve la dirección de la variable `<var>` (o sea, un puntero).
- El operador `&` se utiliza habitualmente para asignar valores a datos de tipo puntero.

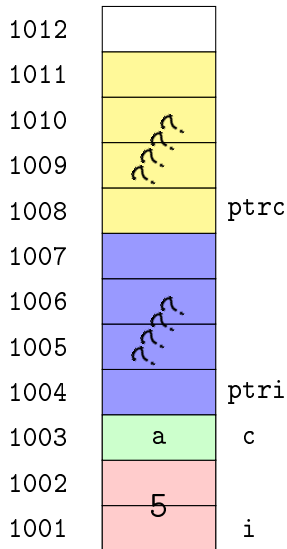
```
int i = 5, *ptri;  
ptri = &i;
```



- `i` es una variable de tipo entero, por lo que la expresión `&i` es la dirección de memoria donde comienza un entero y, por tanto, puede ser asignada al puntero `ptri`.

Se dice que `ptri` *apunta* o *referencia* a `i`.

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

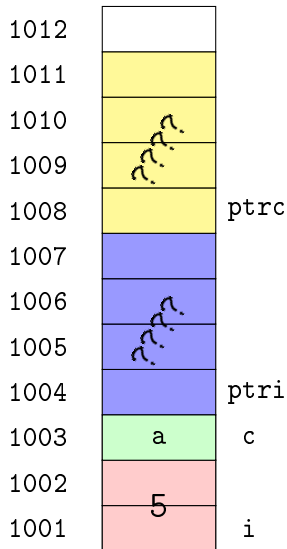
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

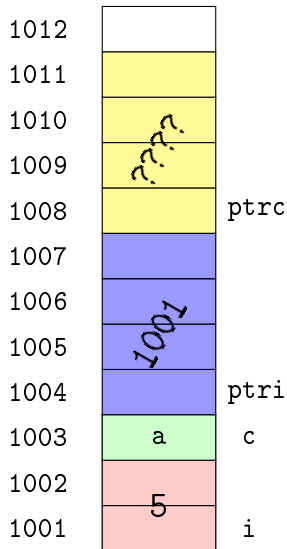
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

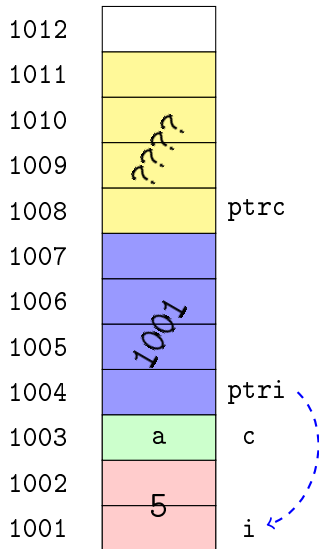
```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```

Operador de dirección &



```
// Se declara la variable de tipo entero
```

```
int i=5;
```

```
// Se declara la variable de tipo char
```

```
char c='a';
```

```
// Se declara puntero a entero
```

```
int * ptri;
```

```
// Se declara el puntero a char
```

```
char * ptrc;
```

```
// ptri apunta a la variable i
```

```
ptri=&i;
```


Operador de indirección *

- *<puntero> devuelve el valor del objeto apuntado por <puntero>.

```
char c, *ptrc;
```

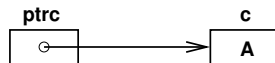
```
.....
```

```
// Hacemos que el puntero apunte a c
```

```
ptrc = &c;
```

```
// Cambiamos contenido de c mediante ptrc
```

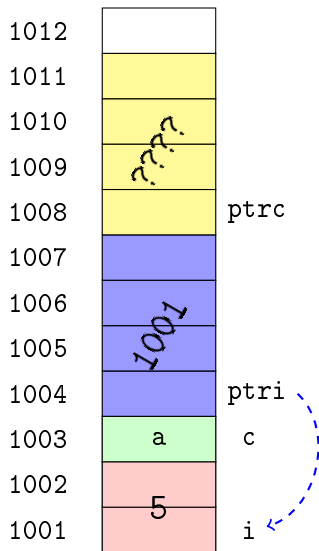
```
*ptrc = 'A'; // equivale a c = 'A'
```



- ptrc es un puntero a caracter que contiene la dirección de c, por tanto, la expresión *ptrc es el objeto apuntado por el puntero, es decir, c.

Un puntero contiene una dirección de memoria y se puede interpretar como un número entero aunque un puntero no es un número entero. Existen un conjunto de operadores que se pueden aplicar sobre punteros (como veremos más adelante): +, -, ++, --, !, ==

Operador de indirección *



```
// Se declara la variable de tipo entero
int i=5;

// Se declara la variable de tipo char
char c='a';

// Se declara puntero a entero
int * ptri;

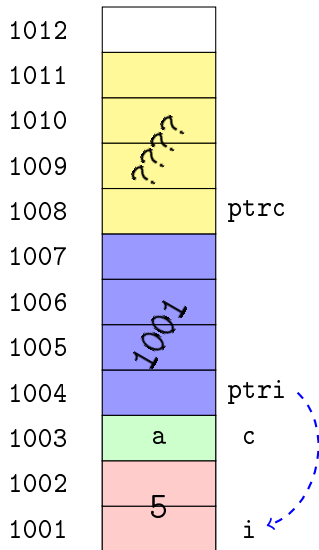
// Se declara el puntero a char
char * ptrc;

// ptri apunta a la variable i
ptri=&i;

// ptrc apunta a c
ptrc=&c;

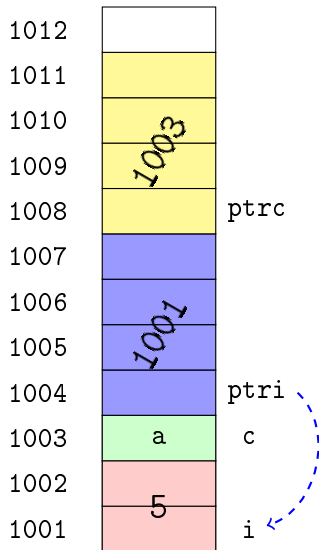
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



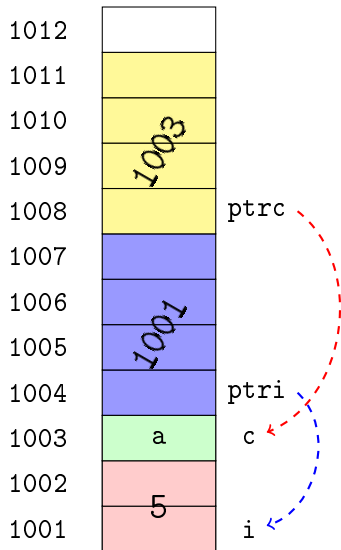
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



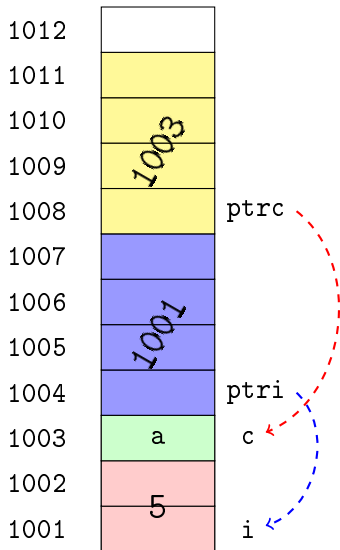
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



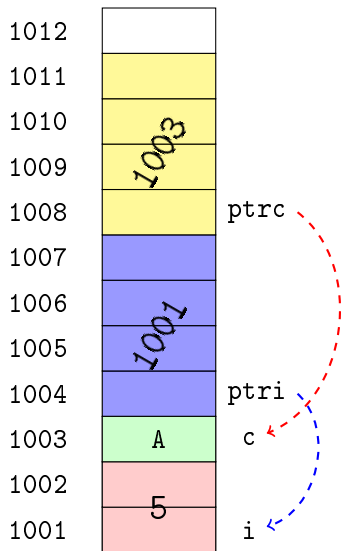
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Operador de indirección *



```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

Asignación e inicialización de punteros

- Un puntero se puede inicializar con la dirección de una variable:

```
int a;  
int *ptri = &a;
```

- A un puntero se le puede asignar una dirección de memoria. La única dirección de memoria que se puede asignar directamente a un puntero es la dirección nula:

```
int *ptri = 0;
```


Asignación e inicialización de punteros

- La asignación sólo está permitida entre punteros de igual tipo.

```
int a=7;  
int *p1=&a;  
char *p2=&a; //ERROR: char *p2 = reinterpret_cast<char*>(&a);  
int *p3=p1;
```

```
asignacionPunteros.cpp: En la función 'int main()':  
asignacionPunteros.cpp:8:14: error: no se puede convertir 'int*' a 'char*' en la inicialización
```



Asignación e inicialización de punteros

- Un puntero debe estar correctamente inicializado antes de usarse

```
int a=7;  
int *p1=&a, *p2;  
*p1 = 20;  
*p2 = 30; // Error
```

Violación de segmento ('core' generado)



- Es conveniente inicializar los punteros en la declaración, con el puntero nulo: 0

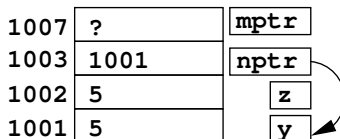
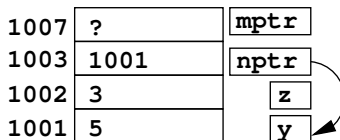
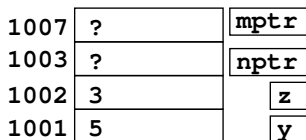
```
int *p2=0;
```

Ejemplo

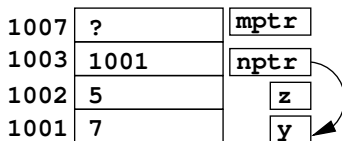
```
int main() {
    char y = 5, z = 3;
    char *nptr;
    char *mptr;
```

```
nptr = &y;
```

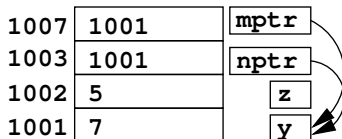
```
z = *nptr;
```



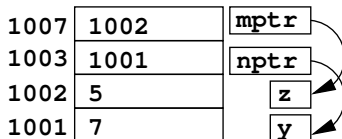
```
*nptr = 7;
```



```
mptr = nptr;
```

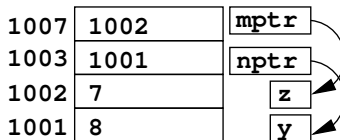
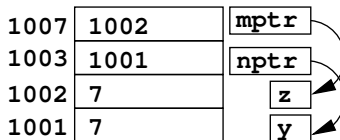


```
mptr = &z;
```



```
*mptr = *nptr;
```

```
y = (*mptr) + 1;
}
```



Ejemplo anterior animado

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		
1002	3	z
1001	5	y

```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```


Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		
1002	3	z
1001	5	y

```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

Ejemplo anterior animado

1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		nptr
1002	3	z
1001	5	y

```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

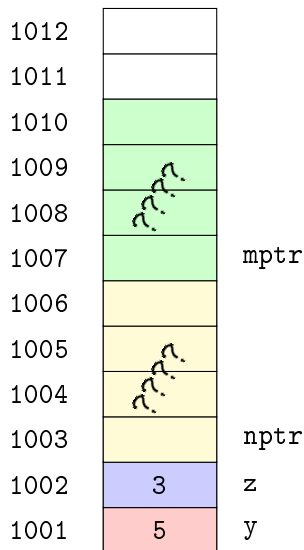
1012		
1011		
1010		
1009		
1008		
1007		
1006		
1005		
1004		
1003		nptr
1002	3	z
1001	5	y

```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

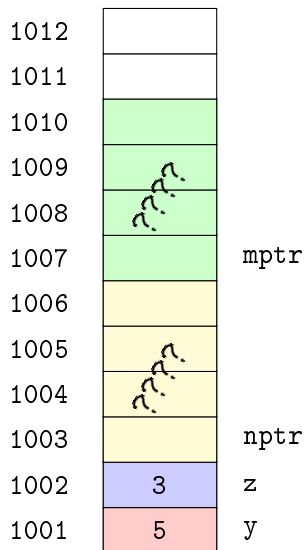


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

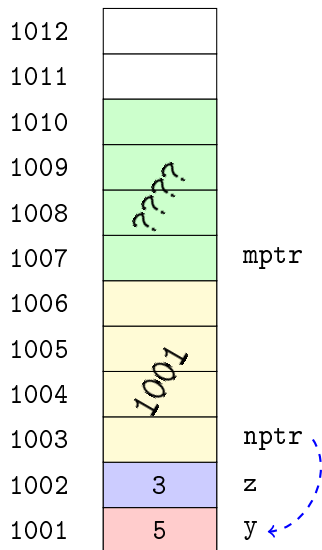


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

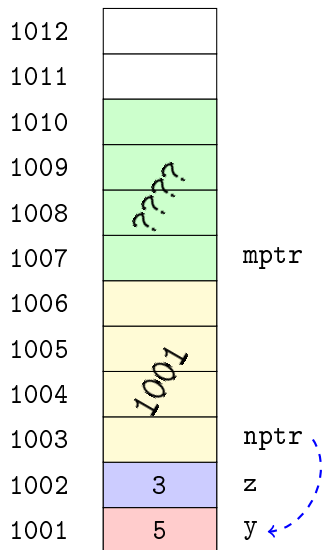


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

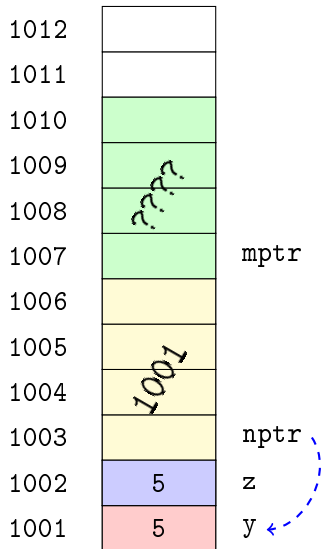


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

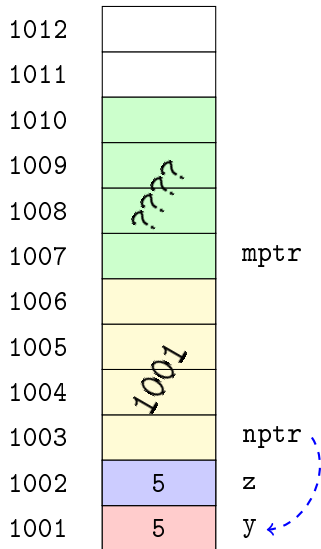


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```


Ejemplo anterior animado

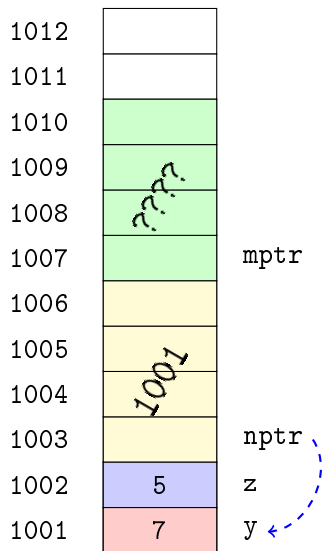


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

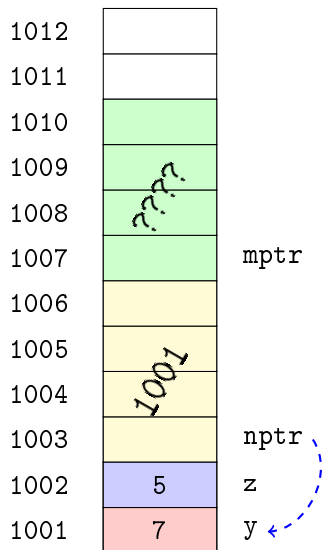


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

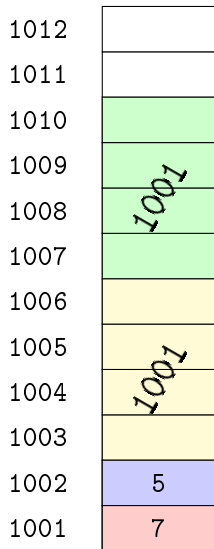


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado



mptr

nptr

z

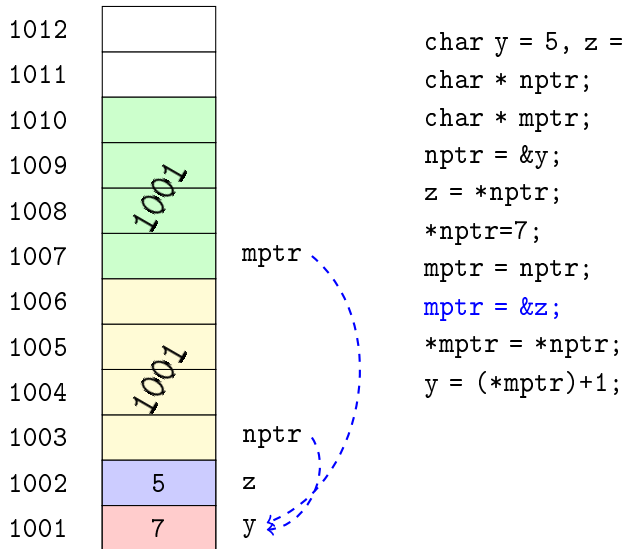
y

```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

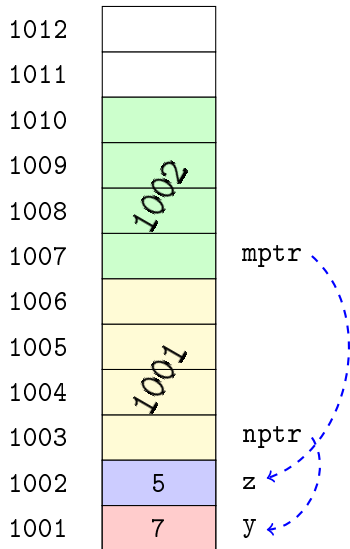


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

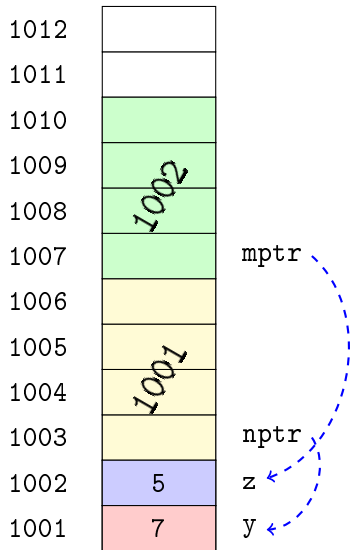


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

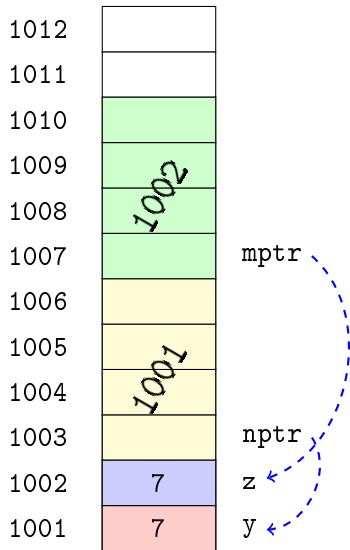


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado

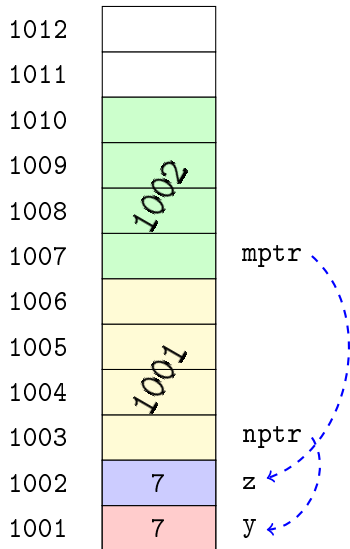


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```


Ejemplo anterior animado

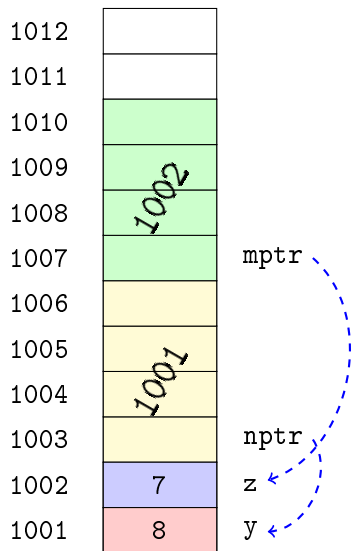


```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Ejemplo anterior animado



```

char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;

```

Operadores relacionales

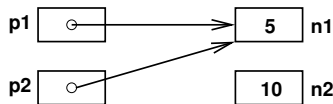
- Los operadores `<`, `>`, `<=`, `>=`, `!=`, `==` son aplicables a punteros.
- El valor del puntero (la dirección que almacena) se comporta como un número entero.

Operadores `!=` y `==`

- `p1 == p2`: comprueba si ambos punteros apuntan a la misma dirección de memoria (ambas variables guardan como valor la misma dirección)
- `*p1 == *p2`: comprueba si coincide lo almacenado en las direcciones apuntadas por ambos punteros

Operadores relacionales

```
int *p1, *p2, n1 = 5, n2 = 10;  
p1 = &n1;  
p2 = p1;  
if (p1 == p2)  
    cout << "Punteros iguales\n";  
else  
    cout << "Punteros diferentes\n";  
if (*p1 == *p2)  
    cout << "Valores iguales\n";  
else  
    cout << "Valores diferentes\n";
```



Operadores relacionales: Ejemplo anterior animado

1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;

// Se asignan los punteros
p1=&n1;
p2=p1

// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

Operadores relacionales: Ejemplo anterior animado

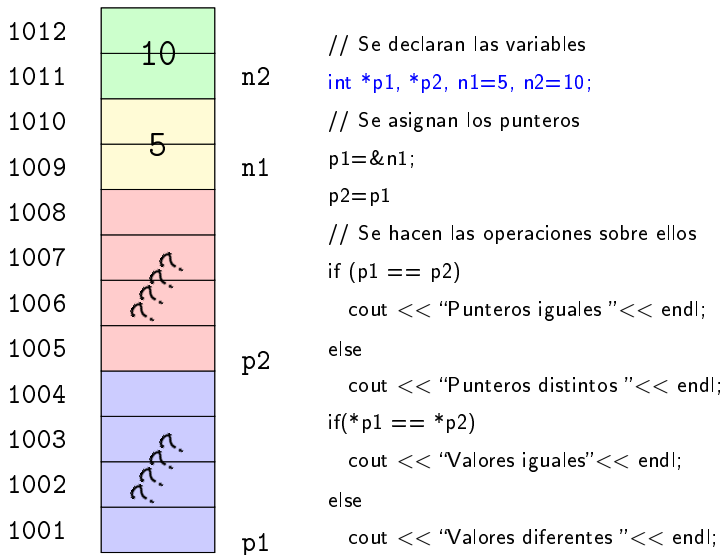
1012	
1011	
1010	
1009	
1008	
1007	
1006	
1005	
1004	
1003	
1002	
1001	

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;

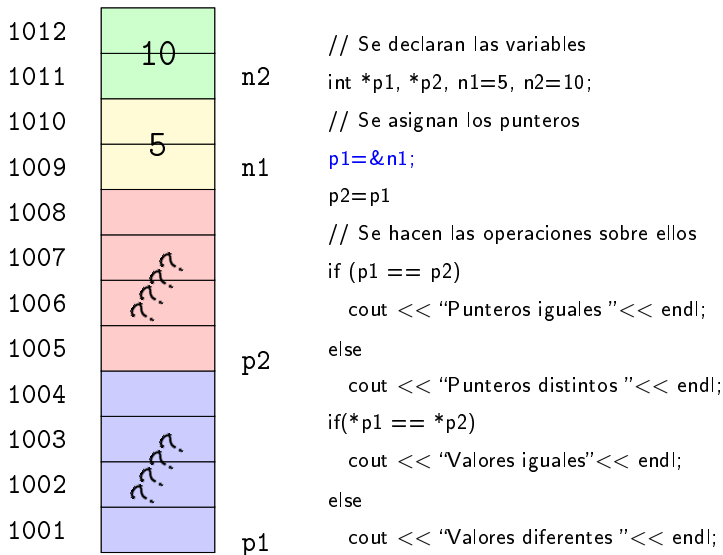
// Se asignan los punteros
p1=&n1;
p2=p1

// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

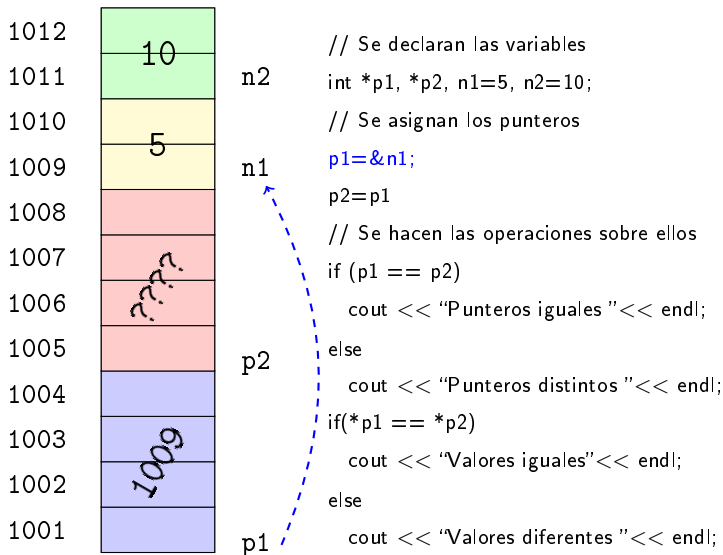
Operadores relacionales: Ejemplo anterior animado



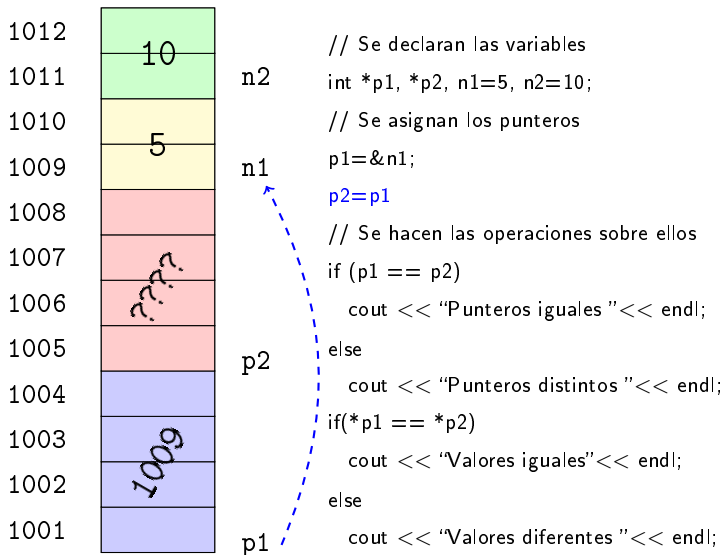
Operadores relacionales: Ejemplo anterior animado



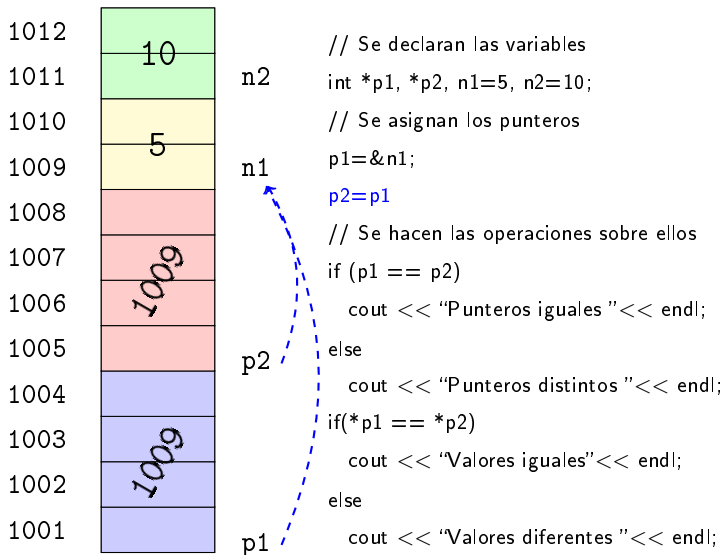
Operadores relacionales: Ejemplo anterior animado



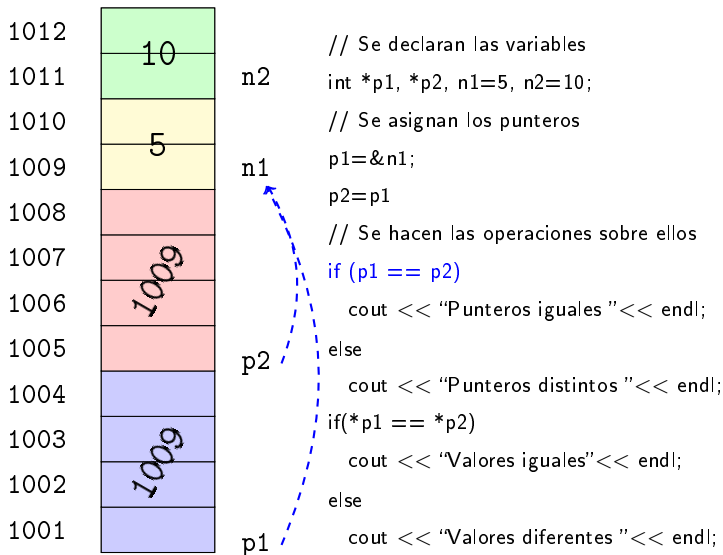
Operadores relacionales: Ejemplo anterior animado



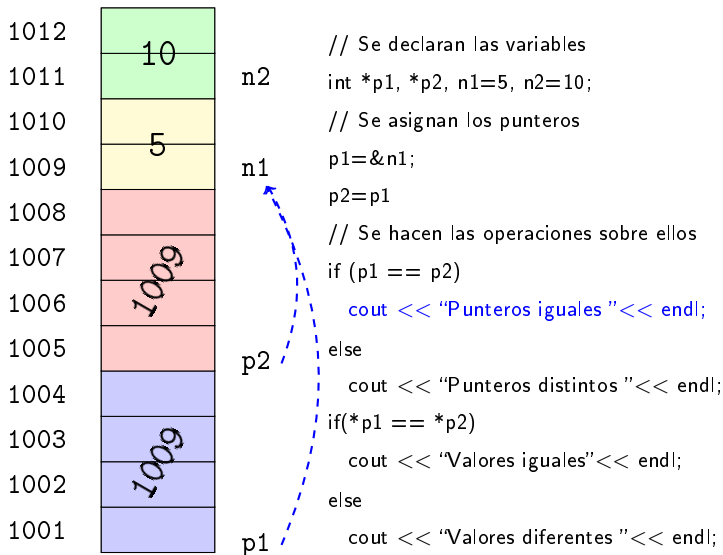
Operadores relacionales: Ejemplo anterior animado



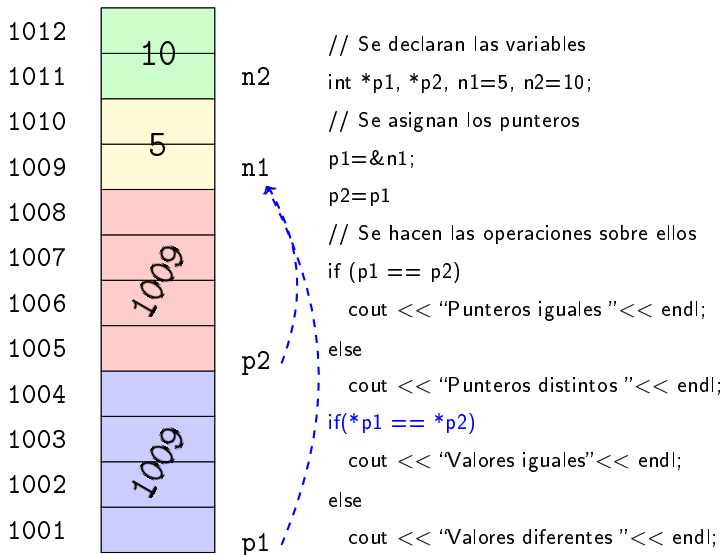
Operadores relacionales: Ejemplo anterior animado



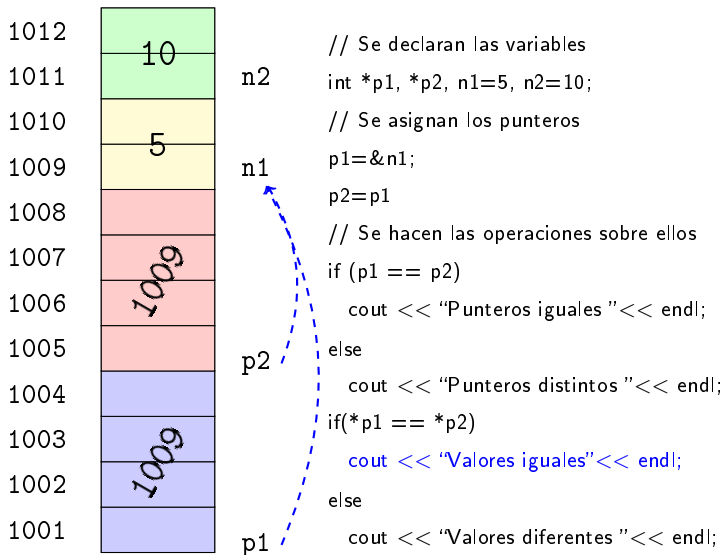
Operadores relacionales: Ejemplo anterior animado



Operadores relacionales: Ejemplo anterior animado

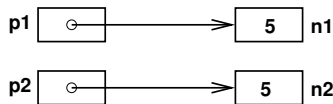


Operadores relacionales: Ejemplo anterior animado



Operadores relacionales: otro ejemplo

```
int *p1, *p2, n1 = 5, n2 = 5;
p1 = &n1;
p2 = &n2;
if (p1 == p2)
    cout << "Punteros iguales\n";
else
    cout << "Punteros diferentes\n";
if (*p1 == *p2)
    cout << "Valores iguales\n";
else
    cout << "Valores diferentes\n";
```



Operadores relacionales: otro ejemplo (ej. animado)

1012

1011

1010

1009

1008

1007

1006

1005

1004

1003

1002

1001

// Se declaran las variables

`int *p1, *p2, n1=5, n2=5;`

// Se asignan los punteros

`p1=&n1;``p2=&n2;`

// Se hacen las operaciones sobre ellos

`if (p1 == p2)``cout << "Punteros iguales " << endl;``else``cout << "Punteros distintos " << endl;``if(*p1 == *p2)``cout << "Valores iguales" << endl;``else``cout << "Valores diferentes " << endl;`

Operadores relacionales: otro ejemplo (ej. animado)

1012

1011

1010

1009

1008

1007

1006

1005

1004

1003

1002

1001

// Se declaran las variables

`int *p1, *p2, n1=5, n2=5;`

// Se asignan los punteros

`p1=&n1;``p2=&n2;`

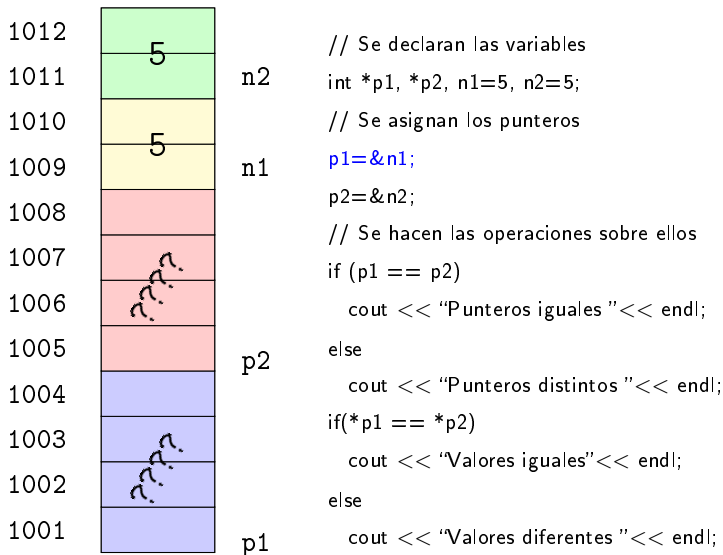
// Se hacen las operaciones sobre ellos

`if (p1 == p2)``cout << "Punteros iguales "<< endl;``else``cout << "Punteros distintos "<< endl;``if(*p1 == *p2)``cout << "Valores iguales"<< endl;``else``cout << "Valores diferentes "<< endl;`

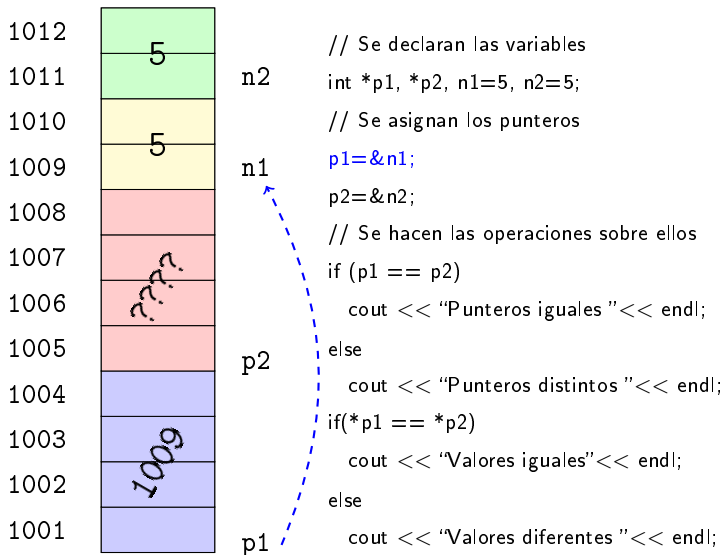
Operadores relacionales: otro ejemplo (ej. animado)

1012	5	n2	<pre>// Se declaran las variables int *p1, *p2, n1=5, n2=5; // Se asignan los punteros p1=&n1; p2=&n2; // Se hacen las operaciones sobre ellos if (p1 == p2) cout << "Punteros iguales "<< endl; else cout << "Punteros distintos "<< endl; if(*p1 == *p2) cout << "Valores iguales"<< endl; else cout << "Valores diferentes "<< endl;</pre>
1011			
1010	5	n1	
1009			
1008			
1007			
1006			
1005		p2	
1004			
1003			
1002			
1001		p1	

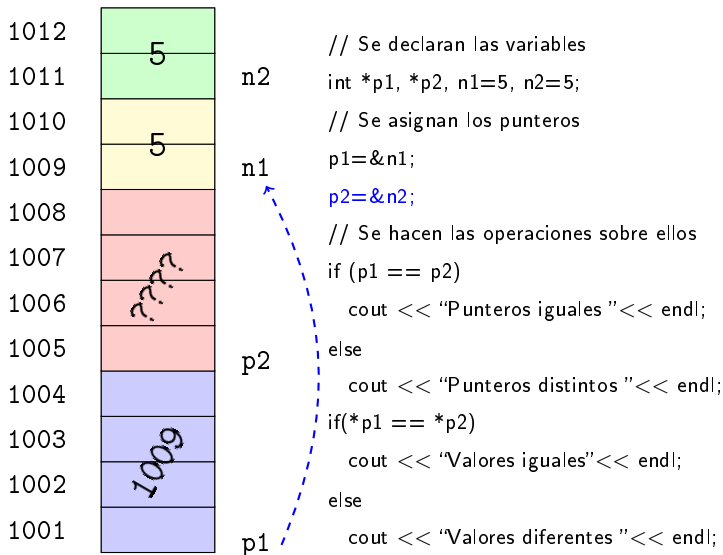
Operadores relacionales: otro ejemplo (ej. animado)



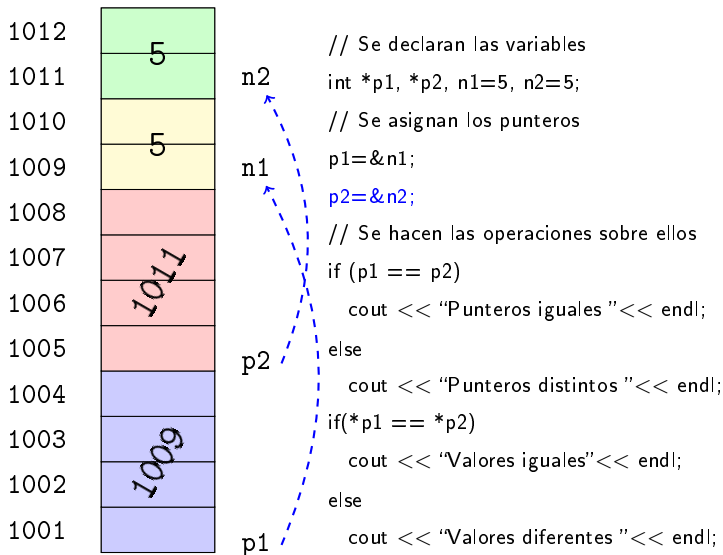
Operadores relacionales: otro ejemplo (ej. animado)



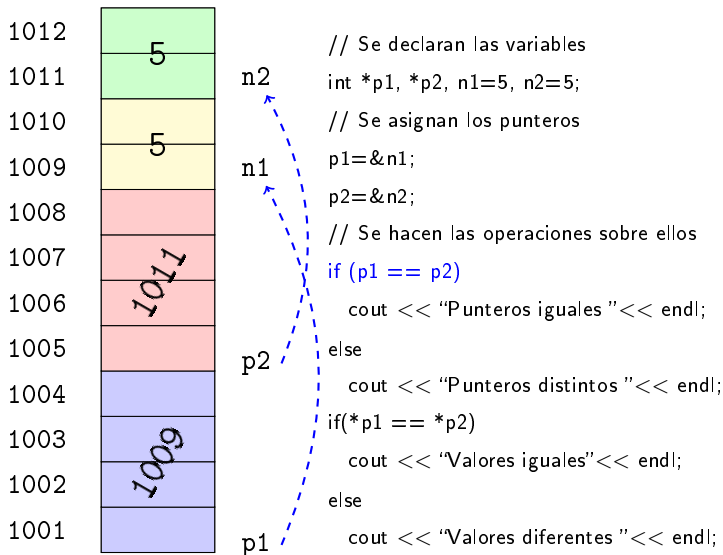
Operadores relacionales: otro ejemplo (ej. animado)



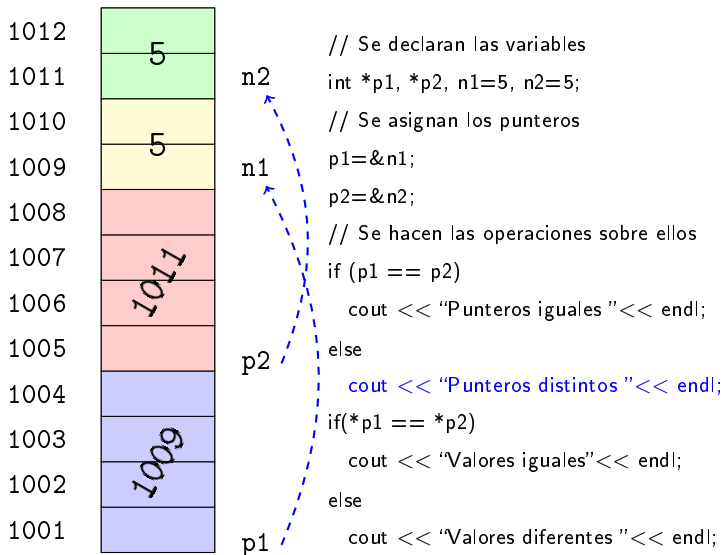
Operadores relacionales: otro ejemplo (ej. animado)



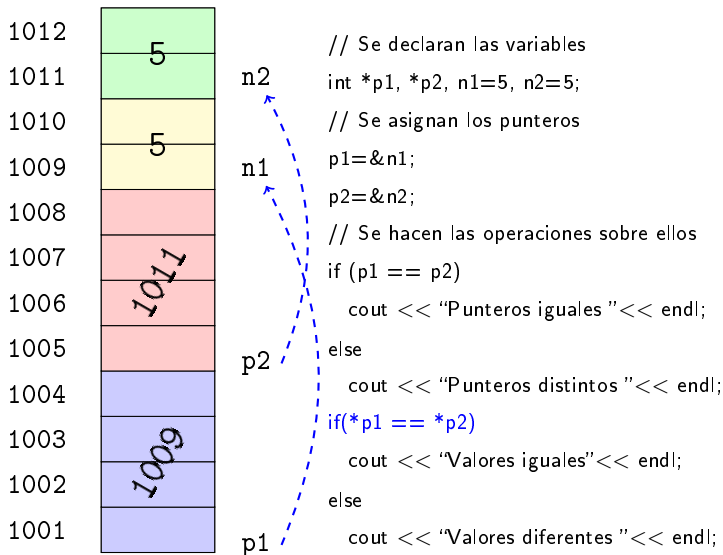
Operadores relacionales: otro ejemplo (ej. animado)



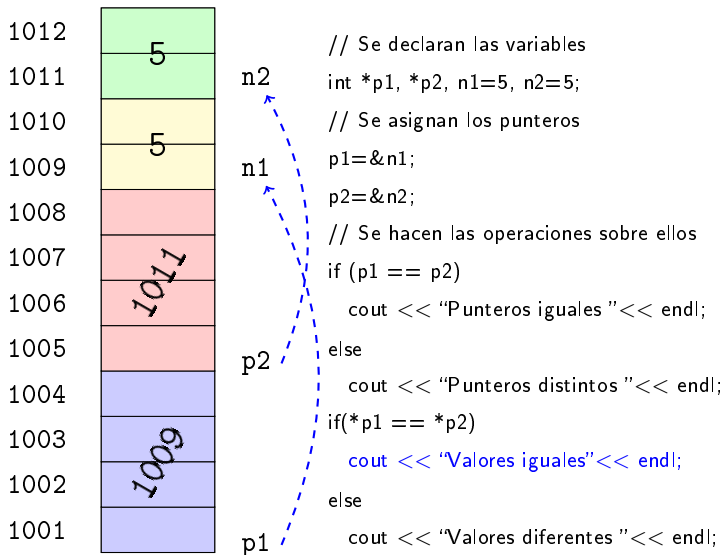
Operadores relacionales: otro ejemplo (ej. animado)



Operadores relacionales: otro ejemplo (ej. animado)



Operadores relacionales: otro ejemplo (ej. animado)

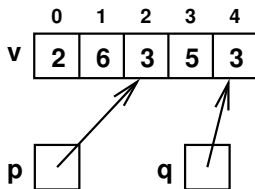


Operadores relacionales

Operadores $<$, $>$, $<=$, $>=$

- Los operadores $<$, $>$, $<=$ y $>=$ tienen sentido para conocer la posición relativa de un objeto respecto a otro en la memoria.
- Sólo son útiles si los dos punteros apuntan a objetos cuyas posiciones relativas guardan relación (por ejemplo, elementos del mismo array).

```
p = &v[2];  
q = &v[4];
```



<code>p==q</code>	false
<code>p!=q</code>	true
<code>*p==*q</code>	true
<code>p<q</code>	true
<code>p>q</code>	false
<code>p<=q</code>	true
<code>p>=q</code>	false

Operadores aritméticos

- Los operadores `+`, `-`, `++`, `--`, `+=` y `-=` son aplicables a punteros.
- Al usar estos operadores, el valor del puntero (la dirección que almacena) se comporta `CASI` como un número entero.
- Al sumar o restar un número `N` al valor del puntero, éste se incrementa o decrementa un determinado número de posiciones, en función del tipo de dato apuntado, según la fórmula:

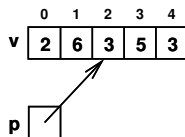
$$N * \text{sizeof}(\text{tipobase})$$

- Esto proporciona una forma rápida de acceso a los elementos de un array, aprovechando que todos sus elementos se almacenan en posiciones sucesivas.

Operadores aritméticos

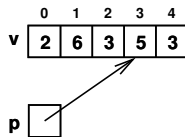
- Situación inicial:

```
int v [5] = {2, 6, 3, 5, 3};
int *p;
p = &v[2];
```



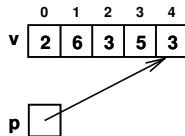
- Si sumamos 1 a p:

```
p++; // p=p+1
```



- Si sumamos 2 a p:

```
p+=2; // p=p+2
```

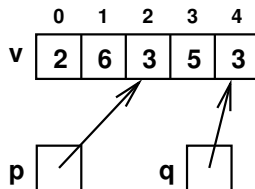


Operadores aritméticos

- ¿Qué devuelve $q - p$?

```
p = &v[2];
```

```
q = &v[4];
```



Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays**
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros y arrays

Los punteros y los arrays están estrechamente vinculados.

Al declarar un array

```
<tipo> <identif>[<n_elem>]
```

- 1 Se reserva memoria para almacenar <n_elem> elementos de tipo <tipo>.
- 2 Se crea un puntero CONSTANTE llamado <identif> que apunta a la primera posición de la memoria reservada.

Por tanto, el identificador de un array, es un puntero CONSTANTE a la dirección de memoria que contiene el primer elemento. Es decir, v es igual a $\&(v[0])$.

Podemos usar arrays como punteros al primer elemento.

```
int v[5] = {2, 6, 3, 5, 3};
cout << *v << endl;
cout << *(v+2) << endl;
```

	0	1	2	3	4
v	2	6	3	5	3

- `*v` es equivalente a `v[0]` y a `*(&v[0])`.
- `*(v+2)` es equivalente a `v[2]` y a `*(&v[2])`.

Podemos usar un puntero a un elemento de un array como un array que comienza en ese elemento

- De esta forma, los punteros pueden poner subíndices y utilizarse como si fuesen arrays: `v[i]` es equivalente a `ptr[i]`.

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           ———— 6

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```


Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;                      6
p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           ———— 6

p=v+2;

cout << *p << endl;

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           _____ 6

p=v+2;

cout << *p << endl;           _____ 3

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           _____ 6

p=v+2;

cout << *p << endl;           _____ 3

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           _____ 6

p=v+2;

cout << *p << endl;           _____ 3

p++;

cout << *p << endl;

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           _____ 6

p=v+2;

cout << *p << endl;           _____ 3

p++;

cout << *p << endl;           _____ 5

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
```

```
int v[5]={2, 6, 3, 5, 3};
```

```
// Se crea el puntero
```

```
int *p;
```

```
// Se asigna
```

```
p=&(v[1]);
```

```
cout << *p << endl;           _____ 6
```

```
p=v+2;
```

```
cout << *p << endl;           _____ 3
```

```
p++;
```

```
cout << *p << endl;           _____ 5
```

```
p=&(v[3])-2;
```

```
cout << p[0] << " " << p[2] << endl;
```

Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};

// Se crea el puntero
int *p;

// Se asigna
p=&(v[1]);

cout << *p << endl;           _____ 6

p=v+2;

cout << *p << endl;           _____ 3

p++;

cout << *p << endl;           _____ 5

p=&(v[3])-2;

cout << p[0] << " " << p[2] << endl;
```


Punteros y arrays: ejemplo

```
// Se declara el array
```

```
int v[5]={2, 6, 3, 5, 3};
```

```
// Se crea el puntero
```

```
int *p;
```

```
// Se asigna
```

```
p=&(v[1]);
```

```
cout << *p << endl;           _____ 6
```

```
p=v+2;
```

```
cout << *p << endl;           _____ 3
```

```
p++;
```

```
cout << *p << endl;           _____ 5
```

```
p=&(v[3])-2;
```

```
cout << p[0] << " " << p[2] << endl;           _____ 6 5
```

Algunos Ejemplos I

```

❶ int v[3]={1,2,3};
   int *p;
   p = v;           // v como int*
   cout << *p;      // Escribe 1
   cout << p[1];    //Escribe 2
   v = p;           //ERROR

❷ void CambiaSigno (double *v, int n){
    for (int i=0; i<n; i++)
        v[i]=-v[i];
}

int main(){
    double m[5]={1,2,3,4,5};
    CambiaSigno(m,5);
}

```

Algunos Ejemplos II

- 3 Recorrer e imprimir los elementos de un array:

```
int v[10] = {3,5,2,7,6,7,5,1,2,5};
for (int i=0; i<10; i++)
    cout << v[i] << endl;
```

- 4 Recorrer e imprimir los elementos de un array:

```
int v[10] = {3,5,2,7,6,7,5,1,2,5};
int *p=v;
for (int i=0; i<10; i++)
    cout << *(p++) << endl;
```

Algunos Ejemplos III

- 5 Recorrer e imprimir los elementos de un array:

```
int v[10] = {3,5,2,7,6,7,5,1,2,5};
```

```
for (int *p=v; p<v+10; ++p)  
    cout << *p << endl;
```

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas**
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros y cadenas

- Según vimos en el tema anterior:

Una cadena de caracteres estilo C es un array de tipo `char` de un tamaño determinado acabado en un carácter especial, el carácter `'\0'` (carácter nulo), que marca el fin de la cadena.

- También se vio que:

Un literal de cadena de caracteres es un array constante de `char` con un tamaño igual a su longitud más uno.

"Hola" de tipo `const char[5]`

"Hola mundo" de tipo `const char[11]`

- Realmente, C++ considera que un literal cadena de caracteres es de tipo `const char *`

Ejemplos de uso

- Calcular longitud cadena:

```
const char *cadena="Hola"; // Se reservan 5
const char *p;
int i=0;
for(p=cadena;*p!='\0';++p)
    ++i;
cout << "Longitud: " << i << endl;
```

- Eliminar los primeros caracteres de la cadena:

```
const char *cadena="Hola Adios";
cout << "Original: " << cadena << endl
    << "Sin la primera palabra: " << cadena+5;
```

Inicialización de cadenas

Notación de corchetes

- Se copia el contenido del literal en el array.
- Es posible modificar caracteres de la cadena.

```
char cad1[]="Hola"; // Copia literal "Hola" en cad1  
cad1[2] = 'b'; // cad1 contiene ahora "Hoba"
```

Notación de punteros

- Copia la dirección de memoria de la constante literal en el puntero.
- No es posible modificar caracteres de la cadena.

```
const char *cad2="Hola"; // Se asignan los punteros  
cad2[2] = 'b'; // Error
```

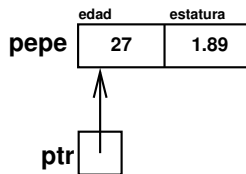

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class**
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros a objetos struct o class

Un puntero también puede apuntar a un **objeto de estructura** o clase:

```
struct Persona{  
    int edad;  
    double estatura;  
};  
Persona pepe;  
Persona *ptr;  
pepe.edad=27;  
pepe.estatura=1.89;  
ptr = &pepe;  
cout << (*ptr).edad << endl;
```

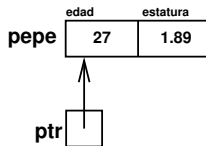


Punteros a objetos struct o class

Igualmente un puntero puede apuntar a un **objeto de una clase**:

```
class Persona{
    int edad;
    double estatura;
public:
    int getEdad() const;
    double getEstatura() const;
    void setEdad(int anios);
    void setEstatura(double metros);
};

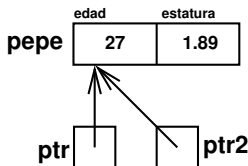
Persona pepe, *ptr;
pepe.setEdad(27); pepe.setEstatura(1.89);
// pepe.edad=27; CUIDADO: no valido desde fuera
//de metodo de la clase, edad es privado
ptr = &pepe;
cout << (*ptr).getEdad() << endl;
// cout << (*ptr).edad << endl; CUIDADO: no valido
//desde fuera de metodo de la clase, edad es privado
```



Punteros a objetos struct o class

La asignación entre punteros funciona igual cuando apuntan a un **objeto struct** o **class**.

```
struct Persona{  
    int edad;  
    double estatura;  
};  
  
Persona pepe;  
Persona *ptr, *ptr2;  
pepe.edad=27;  
pepe.estatura=1.89;  
ptr = &pepe;  
ptr2 = ptr;  
cout << (*ptr).edad << endl;  
cout << (*ptr2).edad << endl;
```

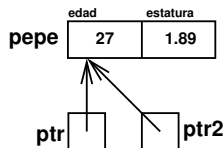


Punteros a objetos struct o class

La asignación entre punteros funciona igual cuando apuntan a un **objeto** struct o **class**.

```
class Persona{
    int edad;
    double estatura;
public:
    int getEdad() const;
    double getEstatura() const;
    void setEdad(int anios);
    void setEstatura(double metros);
};

Persona pepe, *ptr, *ptr2;
pepe.setEdad(27); pepe.setEstatura(1.89);
ptr = &pepe;
ptr2 = ptr;
cout << (*ptr).getEdad() << endl;
cout << (*ptr2).getEdad() << endl;
```



Operador `->`

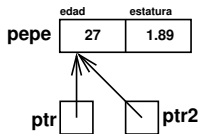
Si `p` es un puntero a un `struct` o `class` podemos acceder a sus miembros con:

- `(*p).miembro`: Cuidado con el paréntesis
- `p->miembro`

Ejemplo con struct

```
struct Persona{
    int edad;
    double estatura;
};

Persona pepe;
Persona *ptr, *ptr2;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
ptr2 = ptr;
cout << ptr->edad << endl;
cout << ptr2->edad << endl;
```



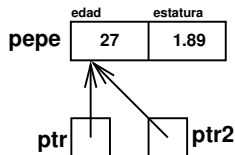
Ejemplo con class

```

class Persona{
    int edad;
    double estatura;
public:
    int getEdad() const;
    double getEstatura() const;
    void setEdad(int anios);
    void setEstatura(double metros);
};

Persona pepe, *ptr, *ptr2;
pepe.setEdad(27); pepe.setEstatura(1.89);
ptr = &pepe;
ptr2 = ptr;
cout << ptr->getEdad() << endl;
cout << ptr2->getEdad() << endl;

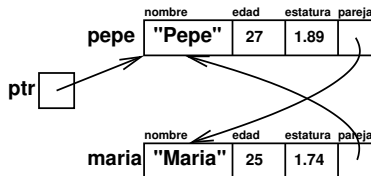
```



Un struct o class puede contener campos de tipo puntero.

```
struct Persona{
    string nombre;
    int edad;
    double estatura;
    Persona *pareja;
};

Persona pepe={"Pepe",27,1.89,0},
        maria={"Maria",25,1.74,0},
        *ptr=&pepe;
pepe.pareja=&maria;
maria.pareja=&pepe;
cout << "La pareja de "
      << ptr->nombre
      << " es "
      << ptr->pareja->nombre
      << endl;
```



Ejemplo con class

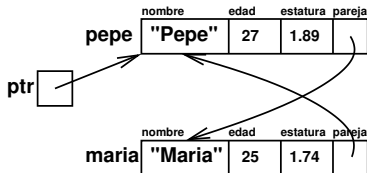
```
class Persona{
    string nombre;
    int edad;
    double estatura;
    Persona *pareja;

public:
    Persona(string name, int anios, double metros);
    int getEdad() const;
    double getEstatura() const;
    Persona *getPareja() const;
    void setPareja(Persona *compa);
    ...
};
```

```

Persona pepe("Pepe",27,1.89),
        maria("Maria",25,1.74),
        *ptr=&pepe;
pepe.setPareja(&maria);
maria.setPareja(&pepe);
cout << "La pareja de "
      << ptr->getNombre()
      << " es "
      << ptr->getPareja()->getNombre()
      << endl;

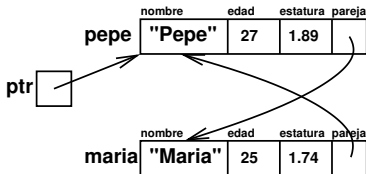
```



```

Persona::Persona(string name, int anios, double metros){
    nombre=name;
    edad=anios;
    estatura=metros;
    pareja=0;
}
Persona* Persona::getPareja() const{
    return pareja;
}
void Persona::setPareja(Persona *compa){
    pareja=compa;
}

```



Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones**
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros y funciones I

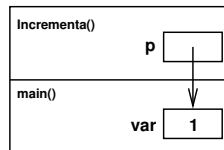
Un puntero puede ser un argumento de una función

- Puede usarse por ejemplo para simular el paso por referencia.

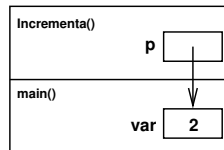
```

1 void incrementa(int* p){
2     (*p)++;
3 }
4 int main()
5 {
6     int var = 1;
7     cout << var << endl; // 1
8     incrementa(&var);
9     cout << var << endl; // 2
10 }
```

Situación en línea 1



Situación en línea 3



Punteros y funciones II

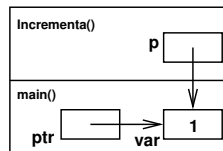
Otra posibilidad

```

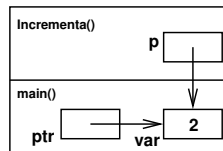
1 void incrementa(int* p){
2     (*p)++;
3 }
4 int main()
5 {
6     int var = 1;
7     int *ptr=&var;
8     cout << var << endl; // 1
9     incrementa(ptr);
10    cout << var << endl; // 2
11 }

```

Situación en línea 1



Situación en línea 3



Punteros y funciones III

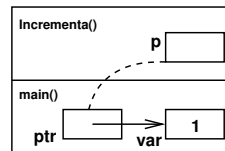
El puntero se puede pasar por referencia

Si deseamos modificar el puntero original, podemos usar paso por referencia.

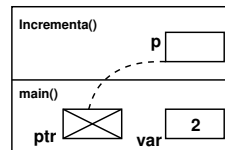
```

1 void incrementa(int* &p){
2     (*p)++;
3     p=0;
4 }
5 int main()
6 {
7     int var = 1;
8     int *ptr=&var;
9     cout << var << endl; // 1
10    incrementa(ptr);
11    cout << var << endl; // 2
12 }
```

Situación en línea 1



Situación en línea 4



Punteros y funciones IV

Devolución de punteros a datos locales

La devolución de punteros a datos locales a una función es un error típico: Los datos locales se destruyen al terminar la función.

```
int *doble(int x)
{
    int a;
    a = x*2;
    return &a;
}
int main(){
    int *x;
    x = doble(3);
    cout << *x << endl;
}
```


Punteros y funciones V

Otro ejemplo incorrecto

```
int *doble(int x)
{
    int a;
    int *p=&a;
    a = x*2;
    return p;
}

int main(){
    int *x;
    x = doble(3);
    cout << *x << endl;
}
```

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros**
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros a punteros

Un puntero a puntero es un puntero que contiene la dirección de memoria de otro puntero.

```
int a = 5;
```

```
int *p;
```


```
int **q;
```

```
p = &a;
```


```
q = &p;
```

1009	?	q
1005	?	p
1001	5	a

1009	?	q
1005	1001	p
1001	5	a



1009	1005	q
1005	1001	p
1001	5	a



En este caso, para acceder al valor de la variable a tenemos tres opciones: a, *p y **q.

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const**
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros y const I

- Cuando tratamos con punteros manejamos dos datos:
 - El dato puntero.
 - El dato que es apuntado.
- Pueden ocurrir las siguientes situaciones:

Ninguno sea const	<code>double *p;</code>
Sólo el dato apuntado sea const	<code>const double *p;</code>
Sólo el puntero sea const	<code>double *const p;</code>
Los dos sean const	<code>const double *const p;</code>

- Las siguientes expresiones son equivalentes:

<code>const double *p;</code>	<code>double const *p;</code>
-------------------------------	-------------------------------

Punteros y const II

- Es posible asignar un puntero no const a uno const, pero no al revés (en la asignación se hace una conversión implícita).

```
double a = 1.0;
double * const p=&a; // puntero constante a double
double * q;         // puntero no constante a double
q = p;              // BIEN: q puede apuntar a cualquier dato
p = q;              // MAL: p es constante
```

Error de compilación:

...error: asignación de la variable de sólo lectura 'p'

p ha quedado asignado en la declaración de la constante y no admite cambios posteriores (como buena constante.....)

Punteros y const III

- Un puntero a dato no const no puede apuntar a un dato const.

Ejemplo 1

El siguiente código da error ya que `&f` devuelve un `const double *`

```
double *p;  
const double f=5.2;  
p = &f;      // INCORRECTO, ya que permitiría cambiar el  
*p = 5.0;    // valor de f a través de p
```

Error de compilación:

...error: conversión inválida de 'const double*' a 'double*' [-fpermissive]

Nota: observad que de permitirse la operación se permitiría cambiar el valor de `f`, que fue declarada como constante.

Punteros y const IV

Ejemplo 2

El siguiente código da error ya que *p devuelve un const double

```
const double *p;  
double f;  
p = &f;    // (const double *) = (double *)  
*p = 5.0;  // ERROR: no se puede cambiar el valor
```

Error de compilación:

...error: asignación de la ubicación de sólo lectura '*p'

Punteros y const V

Ejemplo 3

El siguiente código da error ya que `&(vocales[2])` devuelve un `const char *`

```
const char vocales[5]={'a','e','i','o','u'};  
char *p;  
p = &(vocales[2]); // ERROR de compilación
```

Error de compilación:

...error: conversión inválida de 'const char*' a 'char*' [-fpermissive]

Punteros, funciones y const

Podemos llamar a una función que espera un puntero a dato const con uno a dato no const.

```
void HacerCero(int *p){
    *p = 0;
}
void EscribirEntero(const int *p){
    cout << *p;
}
int main(){
    const int a = 1;
    int b=2;
    HacerCero(&a);      // ERROR
    EscribirEntero(&a); // CORRECTO
    EscribirEntero(&b); // CORRECTO
}
```

Error de compilación:

...error: conversión inválida de 'const int*' a 'int*' [-fpermissive]

Punteros, arrays y const

Dada la estrecha relación entre arrays y punteros, podemos usar un array de constantes como un puntero a constantes, y al contrario:

```
const int matConst[5]={1,2,3,4,5};  
int mat[3]={3,5,7};  
const int *pconst;  
int *p;  
pconst = matConst;  // CORRECTO  
pconst = mat;        // CORRECTO  
p = mat;             // CORRECTO  
p = matConst;        // ERROR
```

Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros**
- 10 Punteros a funciones
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Arrays de punteros

Arrays de punteros

Un array donde cada elemento es un puntero

Declaración

Podemos declarar un array de punteros a enteros de la siguiente forma:

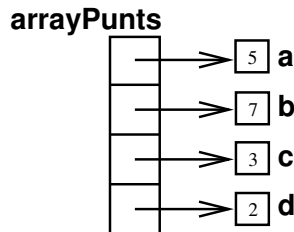
```
int* arrayPunts[4];
```

Arrays de punteros

Ejemplo de array de punteros a enteros

```
int* arrayPunts[4];  
int a=5, b=7, c=3, d=2;  
arrayPunts[0] = &a;  
arrayPunts[1] = &b;  
arrayPunts[2] = &c;  
arrayPunts[3] = &d;  
for(int i=0; i<4; i++){  
    cout << *arrayPunts[i] << " ";  
}  
cout << endl;
```

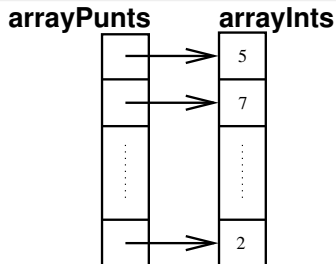
5 7 3 2



Arrays de punteros

Otro ejemplo de array de punteros a enteros

Podemos usar un array de punteros a los elementos de otro array para ordenar sus elementos sin modificar el array original.



Arrays de punteros

```
#include <iostream>
using namespace std;

void ordenacionPorSeleccion(const int * v[], int util_v){
    int pos_min;
    const int *aux;

    for (int i=0; i<util_v-1; i++){
        pos_min=i;
        for (int j=i+1; j<util_v; j++){
            if (*v[j] < *v[pos_min])
                pos_min=j;

            aux = v[i];
            v[i] = v[pos_min];
            v[pos_min] = aux;
        }
    }
}
```



```
int main(){
    const int DIMARRAY=100;
    const int* arrayPunts[DIMARRAY];
    const int arrayInts[DIMARRAY]={5,7,3,2};
    int utilArray=4;

    for(int i=0; i< utilArray; i++){
        arrayPunts[i] = &arrayInts[i];
    }

    cout<<"Array antes de ordenar (impreso con arrayPunts):"<<endl;
    for(int i=0; i< utilArray; i++){
        cout << *arrayPunts[i] << " ";
    }
    cout << endl;
```

```
ordenacionPorSeleccion(arrayPunts,utilArray);

cout<<"Array despues de ordenar (impreso con arrayPunts):"<<endl;
for(int i=0; i< utilArray; i++){
    cout << *arrayPunts[i] << " ";
}
cout << endl;

cout<<"Array despues de ordenar (impreso con arrayInts):"<<endl;
for(int i=0; i< utilArray; i++){
    cout << arrayInts[i] << " ";
}
cout << endl;
}
```

Arrays de punteros

```
Array antes de ordenar (impreso con arrayPunts):  
5 7 3 2  
Array despues de ordenar (impreso con arrayPunts):  
2 3 5 7  
Array despues de ordenar (impreso con arrayInts):  
5 7 3 2
```

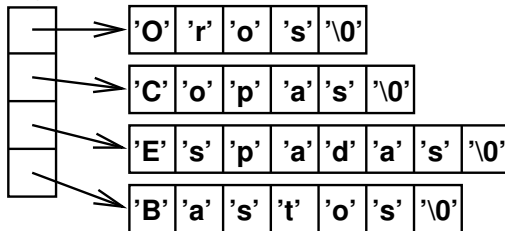


Arrays de punteros

Ejemplo de array de punteros a cadenas estilo C

Podemos usar un array de punteros a cadenas de caracteres estilo C.

palosBaraja



Arrays de punteros

```
#include <iostream>
using namespace std;

int main(){
    const char*  const palosBaraja[4]={"Oros", "Copas", "Espadas", "Bastos"};

    cout<<"Palos de la baraja: ";
    for(int i=0; i< 4; i++){
        cout << palosBaraja[i] << " ";
    }
    cout << endl;
}
```

Palos de la baraja: Oros Copas Espadas Bastos



Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones**
- 11 Errores comunes con punteros
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Punteros a funciones

Puntero a función

Contiene la dirección de memoria de una función, o sea la dirección donde comienza el código que realiza la tarea de la función apuntada.

Con estos punteros podemos hacer las siguientes operaciones:

- Usarlos como parámetro a una función.
- Ser devueltos por una función con `return`.
- Crear arrays de punteros a funciones.
- Asignarlos a otras variables puntero a función.
- Usarlos para llamar a la función apuntada.

Declaración de variables o parámetro puntero a función

Declaración de variables o de parámetros puntero a función

Puntero a función que devuelve `bool` y que tiene dos parámetros de tipo `int`:

```
bool ( *comparar )( int, int );
```

Los paréntesis alrededor de `*comparar` son obligatorios para indicar que es un puntero a función.

Cuidado con los paréntesis

Si no incluimos los paréntesis, estaríamos declarando una función que recibe dos enteros y devuelve un puntero a un valor `bool`.

```
bool *comparar( int, int );
```


Ejemplo de punteros a funciones

Ordenación de un array ascendente o descendente

Construimos una función con un parámetro puntero a función para permitir ordenar ascendente o descendente.

```
bool ascendente( int a, int b ){
    return a < b;
}

bool descendente( int a, int b ){
    return a > b;
}

void ordenarPorSeleccion(int arrayInts[], const int utilArrayInts,
                        bool (*comparar)( int, int ) ){
    ...
    if (!(*comparar) (arrayInts[masPequenoOMasGrande], arrayInts[index]))
    ...
}
```

Ejemplo de punteros a funciones

```
int main(){  
    const int DIMARRAY = 10;  
    int array[DIMARRAY] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };  
  
    ...  
    // Ordena ascendente  
    ordenarPorSeleccion(array, DIMARRAY, ascendente );  
  
    ...  
    // Ordena descendente  
    ordenarPorSeleccion(array, DIMARRAY, descendente );  
}
```

Llamada a la función apuntada por un puntero a función

Llamada a la función apuntada por un puntero a función

Usaremos la sintaxis:

```
(*comparar)( valorEntero1, valorEntero2 );
```

Cuidado con los paréntesis

Son obligatorios los paréntesis alrededor de `*comparar`.

Alternativa para la llamada a la función apuntada por un puntero a función

```
comparar( valorEntero1, valorEntero2 );
```

Pero es recomendable la primera forma, ya que indica explícitamente que `comparar` es un puntero a función. En el segundo caso, parece que `comparar` es el nombre de alguna función del programa.

Ejemplo de punteros a funciones I

Ordenación de un array ascendente o descendientemente (código completo)

Mostramos a continuación el código completo para este problema.

```

1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  // prototipos
6  void ordenarPorSeleccion( int [], const int, bool (*)( int, int ) );
7  void intercambiar( int * const, int * const );
8  bool ascendente( int, int ); // implementa orden ascendente
9  bool descendente( int, int ); // implementa orden descendente
10
11 int main(){
12     const int DIMARRAY = 10;
13     int orden; // 1 = ascendente, 2 = descendente
14     int contador; // indice del array
15     int array[DIMARRAY] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
16

```

Ejemplo de punteros a funciones II

```

17  do{
18      cout << "Introduce 1 para ordenar en orden ascendente,\n"
19          << "Introduce 2 para ordenar en orden descendente: ";
20      cin >> orden;
21  } while(orden!=1 && orden!=2);
22
23      cout << "\nElementos en el orden original\n";
24
25      for ( contador = 0; contador < DIMARRAY; ++contador )
26          cout << setw( 4 ) << array[contador];
27
28      if ( orden == 1 ){
29          ordenarPorSeleccion( array, DIMARRAY, ascendente );
30          cout << "\nElementos en el orden ascendente\n";
31      }
32      else {
33          ordenarPorSeleccion( array, DIMARRAY, descendente );
34          cout << "\nElementos en el orden descendente\n";
35      }
36

```

Ejemplo de punteros a funciones III

```

37     for ( contador = 0; contador < DIMARRAY; ++contador )
38         cout << setw( 4 ) << array[contador];
39     cout << endl;
40 }
41
42 void ordenarPorSeleccion( int arrayInts[], const int utilArrayInts,
43                          bool (*comparar)( int, int ) ){
44     int masPequenoOMasGrande;
45     for ( int i = 0; i < utilArrayInts - 1; ++i ){
46         masPequenoOMasGrande = i;
47         for (int index = i + 1; index < utilArrayInts; ++index)
48             if (!(*comparar)(arrayInts[masPequenoOMasGrande],
49                             arrayInts[index]))
50                 masPequenoOMasGrande = index;
51         intercambiar(&arrayInts[masPequenoOMasGrande], &arrayInts[i]);
52     }
53 }
54
55
56

```

Ejemplo de punteros a funciones IV

```
57 void intercambiar( int * const elemento1Ptr, int * const elemento2Ptr ){
58     int aux = *elemento1Ptr;
59     *elemento1Ptr = *elemento2Ptr;
60     *elemento2Ptr = aux;
61 }
62
63 bool ascendente( int a, int b ){
64     return a < b; // devuelve true si a es menor que b
65 }
66
67 bool descendente( int a, int b ){
68     return a > b; // devuelve true si a es mayor que b
69 }
70
71
72
```

Ejemplo de punteros a funciones

```
Introduce 1 para ordenar en orden ascendente,  
Introduce 2 para ordenar en orden descendente: 1
```

```
Elementos en el orden original
```

```
2   6   4   8  10  12  89  68  45  37
```

```
Elementos en el orden ascendente
```

```
2   4   6   8  10  12  37  45  68  89
```

```
Introduce 1 para ordenar en orden ascendente,  
Introduce 2 para ordenar en orden descendente: 2
```

```
Elementos en el orden original
```

```
2   6   4   8  10  12  89  68  45  37
```

```
Elementos en el orden descendente
```

```
89  68  45  37  12  10   8   6   4   2
```



Contenido del tema

- 1 Definición y declaración de variables
- 2 Operaciones con punteros
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 6 Punteros y funciones
- 7 Punteros a punteros
- 8 Punteros y const
- 9 Arrays de punteros
- 10 Punteros a funciones
- 11 Errores comunes con punteros**
- 12 Estructura de la memoria
- 13 Gestión dinámica de la memoria
- 14 Objetos dinámicos simples
- 15 Objetos dinámicos compuestos
- 16 Ejemplo: Objetos dinámicos autoreferenciados
- 17 Arrays dinámicos
- 18 Matrices dinámicas

Algunos errores comunes

- Asignar puntero de distinto tipo

```
int a=10, *ptri;
double b=5.0, *ptrf;
```

```
ptri = &a;
ptrf = &b;
ptrf = ptri; // Error en compilación
```

- Uso de punteros no inicializados

```
char y=5, *nptr;
*nptr=5; // ERROR
```

- Asignación de valores al puntero y no a la variable.

```
char y=5, *nptr =&y;
nptr = 9; // Error de compilación
```