

# Metodología de la Programación

## Tema 2. Punteros y memoria dinámica

Andrés Cano Utrera  
(acu@decsai.ugr.es)

Departamento de Ciencias de la Computación e I.A.



Curso 2018-2019

# Contenido del tema

## Parte I: Tipo de Dato Puntero

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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

# Contenido del tema

## 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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**.

# 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**.

# 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**.

# 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**.

# 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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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.

# 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';
```

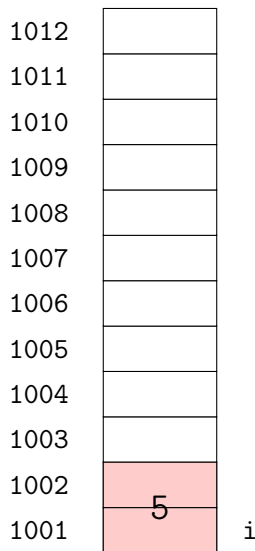
// Se declara puntero a entero

```
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
```

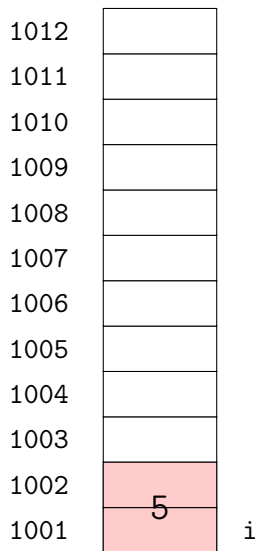
```
int * ptri;
```

```
// 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';
```

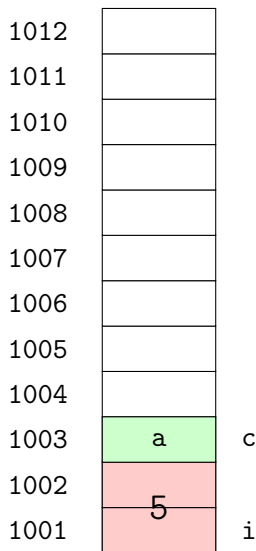
```
// Se declara puntero a entero
```

```
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';
```

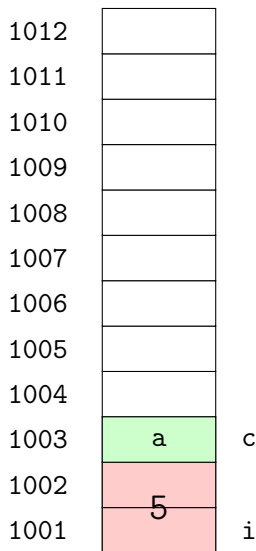
// Se declara puntero a entero

```
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';
```

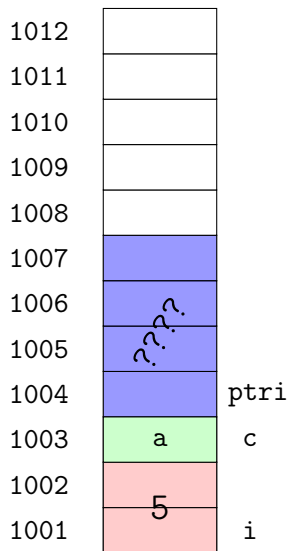
// Se declara puntero a entero

```
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';
```

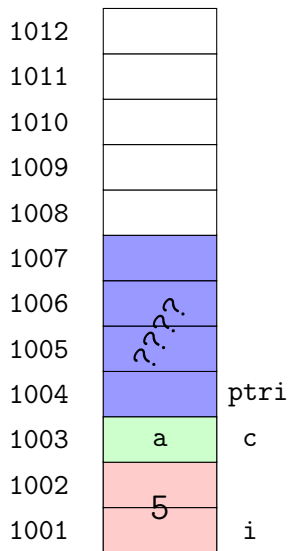
```
// Se declara puntero a entero
```

```
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';
```

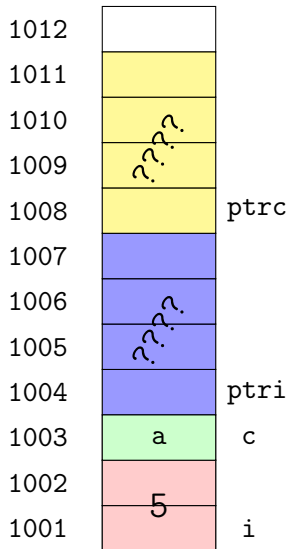
```
// Se declara puntero a entero
```

```
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).

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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - **Operador de dirección &**
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Operador de dirección &

## Operador de dirección &

`&<var>` devuelve la dirección de la variable `<var>` (o sea, un puntero).

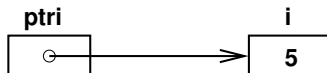
# Operador de dirección &

## 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;
```



# Operador de dirección &

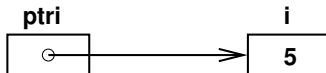
## Operador de dirección &

&<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.

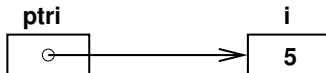
# Operador de dirección &

## 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.

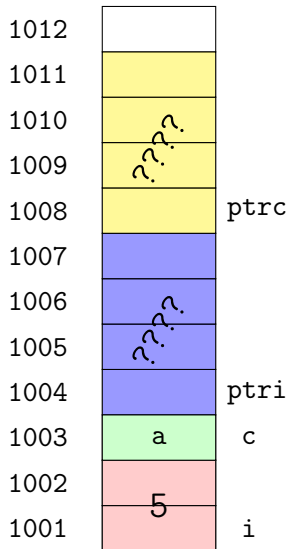
```
int i = 5, *ptri;  
ptri = &i;
```



- 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;
```

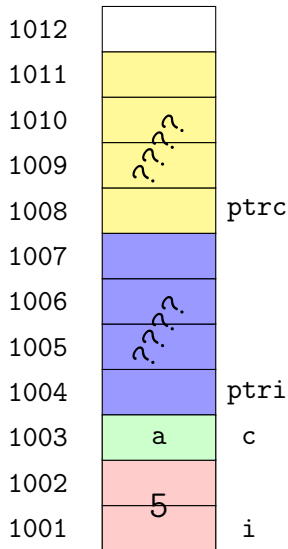
```
// Se declara el puntero a char
```

```
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
```

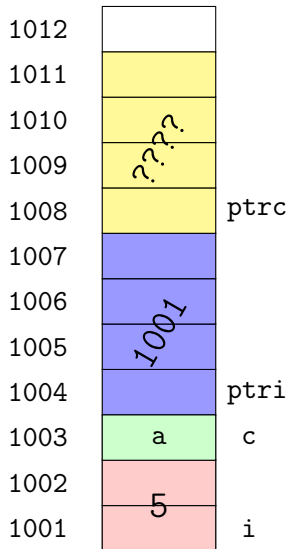
```
char * ptrc;
```

```
// 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;
```

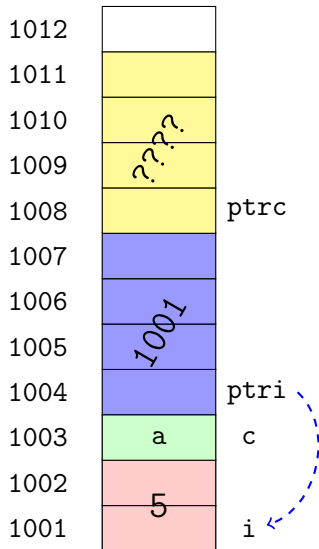
```
// Se declara el puntero a char
```

```
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;
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - **Operador de indirección \***
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Operador de indirección \*

## Operador de indirección \*

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

- Ejemplo:

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

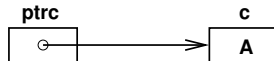
```
.....
```

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

```
ptrc = &c;
```

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

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



# Operador de indirección \*

## Operador de indirección \*

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

- Ejemplo:

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

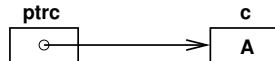
```
.....
```

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

```
ptrc = &c;
```

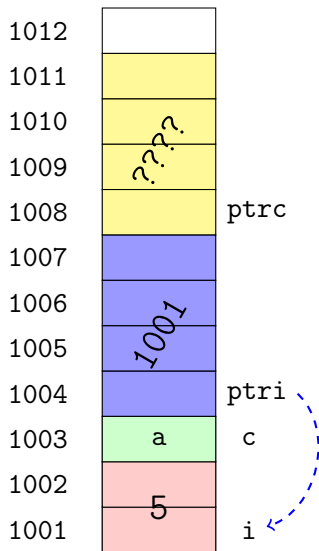
```
// Cambiamos contenido de c mediante ptrc
```

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



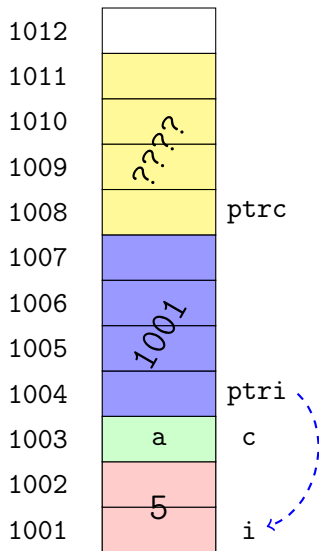
- ptrc es un puntero a carácter que contiene la dirección de c, por tanto, \*ptrc es el objeto apuntado por el puntero, es decir, c.

# Operador de indirección \*



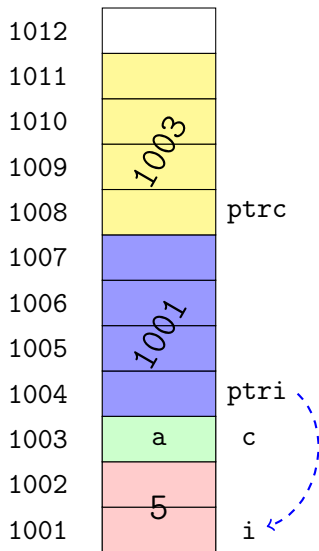
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# 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';
```

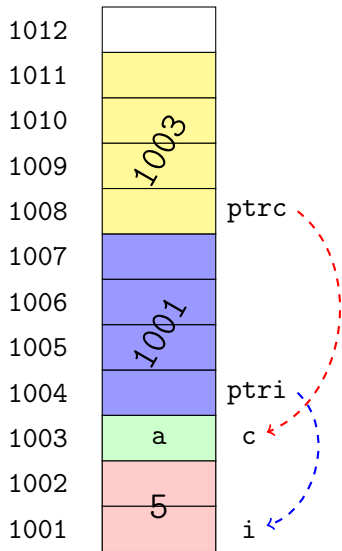
# Operador de indirección \*



```
// 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';
```

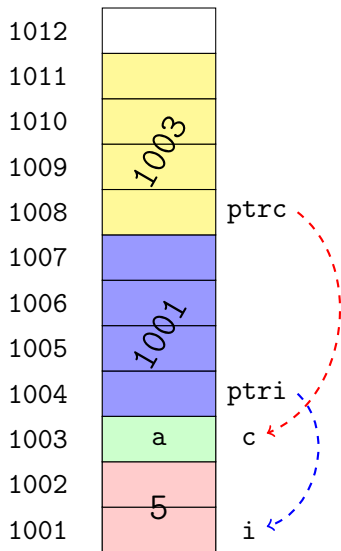


# Operador de indirección \*



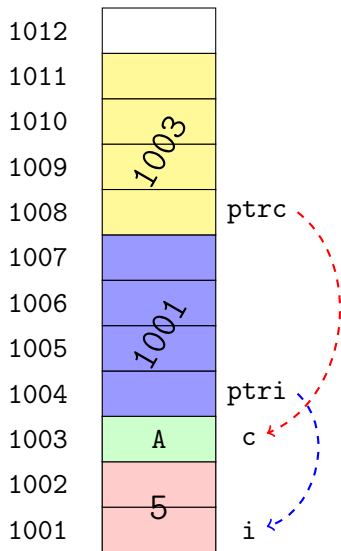
```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# 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';
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - **Asignación e inicialización de punteros**
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Asignación e inicialización de punteros

## Inicialización de un puntero

Un puntero se puede inicializar con la dirección de una variable.

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

## Asignación de punteros

A un puntero se le puede asignar una dirección de memoria de otra variable. La única dirección de memoria que se puede asignar directamente (valor literal) a un puntero es la dirección nula.

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

# Asignación e inicialización de punteros

## Inicialización de un puntero

Un puntero se puede inicializar con la dirección de una variable.

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

## Asignación de punteros

A un puntero se le puede asignar una dirección de memoria de otra variable. La única dirección de memoria que se puede asignar directamente (valor literal) a un puntero es la dirección nula.

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



# Asignación e inicialización de punteros

- La asignación solo 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;
```



# 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)

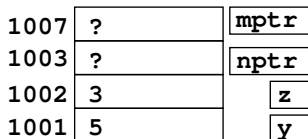


- Es conveniente inicializar los punteros en la declaración, con el puntero nulo: 0

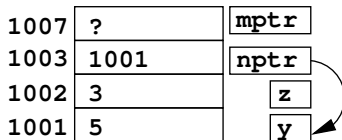
```
int *p2=0;
```

# Ejemplo

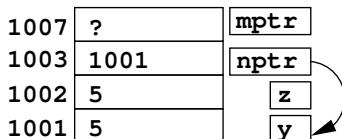
```
int main() {
    char y = 5, z = 3;
    char *nptr;
    char *mptr;
```



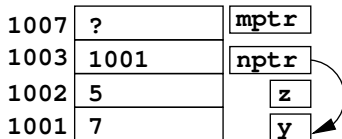
```
nptr = &y;
```



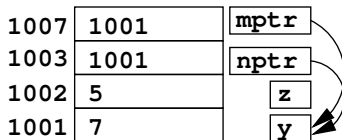
```
z = *nptr;
```



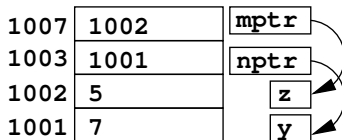
```
*nptr = 7;
```



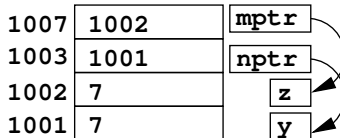
```
mptr = nptr;
```



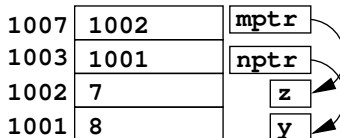
```
mptr = &z;
```



```
*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;
```

```
nptra = &y;
```

```
z = *nptra;
```

```
*nptra=7;
```

```
mptra = nptra;
```

```
mptra = &z;
```

```
*mptra = *nptra;
```

```
y = (*mptra)+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;
```

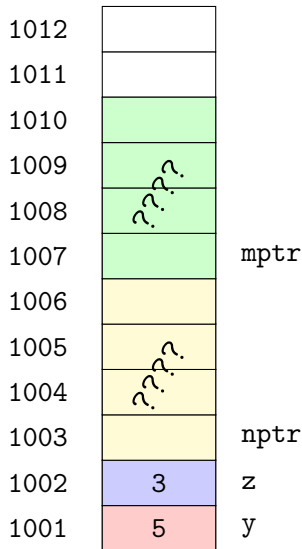
```
y = (*mptr)+1;
```

## Ejemplo anterior animado

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

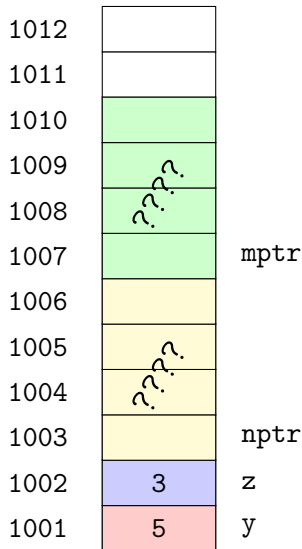
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



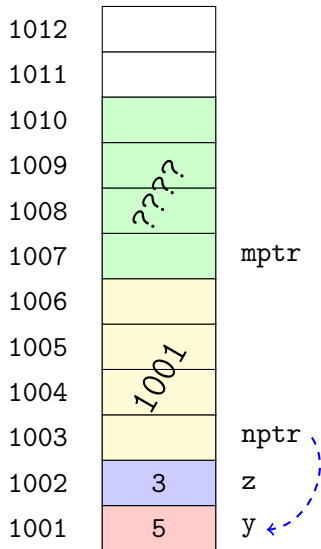
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



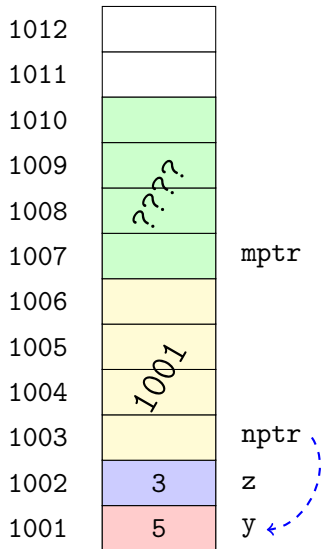
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



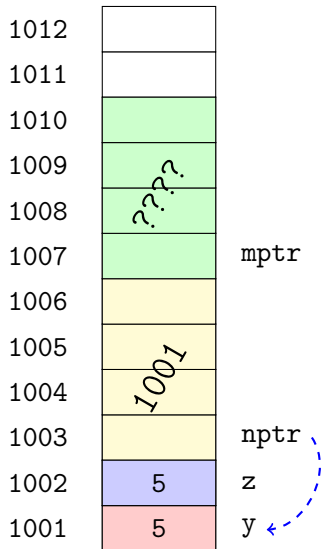
```
char y = 5, z = 3;  
char * nptr;  
char * mptra;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptra = nptr;  
mptra = &z;  
*mptra = *nptr;  
y = (*mptra)+1;
```

## Ejemplo anterior animado



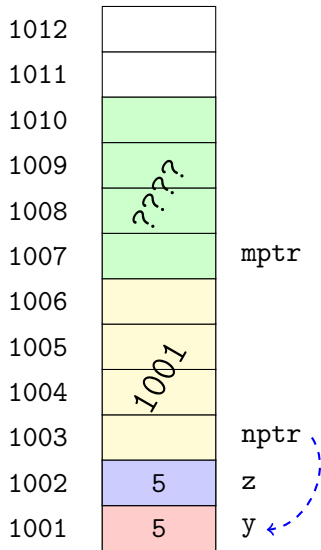
```
char y = 5, z = 3;  
char * nptr;  
char * mptra;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptra = nptr;  
mptra = &z;  
*mptra = *nptr;  
y = (*mptra)+1;
```

## 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;
```

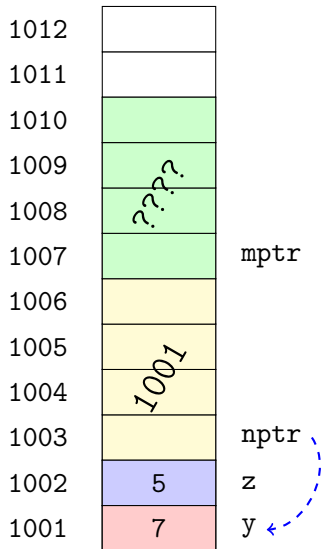
## Ejemplo anterior animado



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

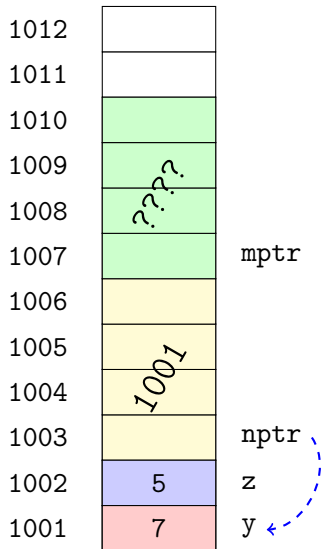


## Ejemplo anterior animado



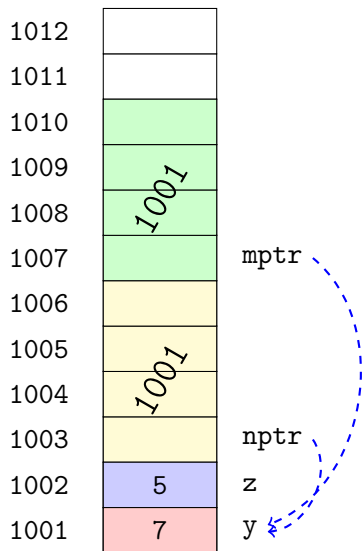
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



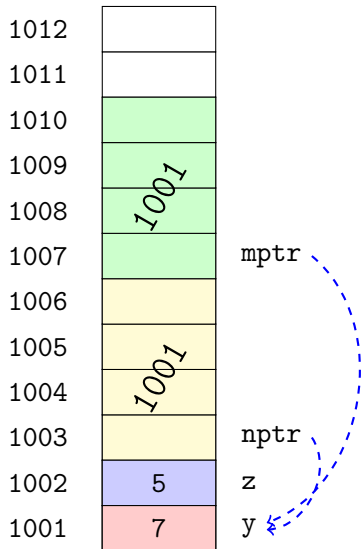
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptra = &y;  
z = *nptra;  
*nptra=7;  
mptra = nptra;  
mptra = &z;  
*mptra = *nptra;  
y = (*mptra)+1;
```

## Ejemplo anterior animado



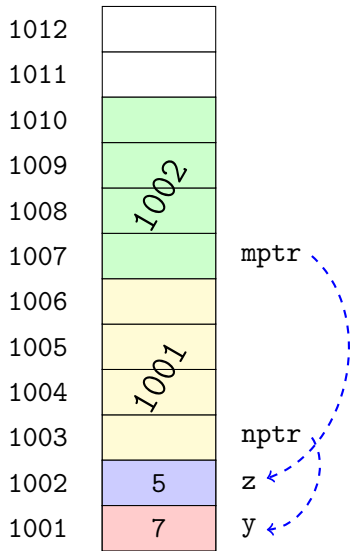
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



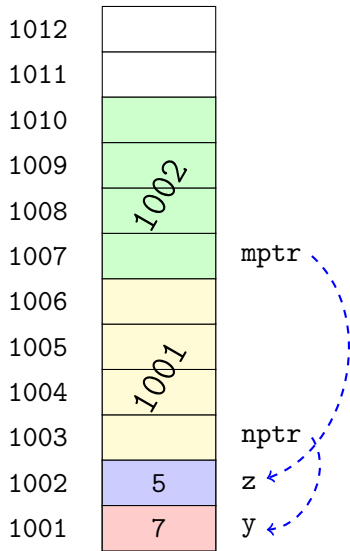
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



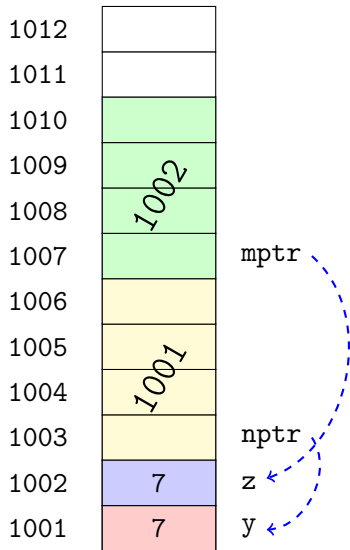
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## Ejemplo anterior animado



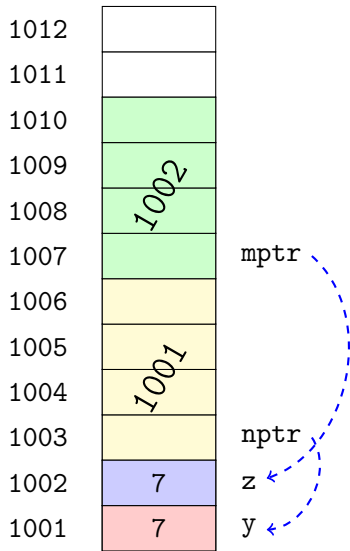
```
char y = 5, z = 3;  
char * nptr;  
char * mptr;  
nptr = &y;  
z = *nptr;  
*nptr=7;  
mptr = nptr;  
mptr = &z;  
*mptr = *nptr;  
y = (*mptr)+1;
```

## 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;
```

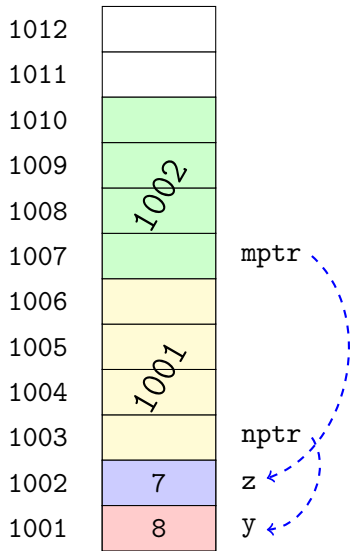
## Ejemplo anterior animado



```
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;
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - **Operadores relacionales**
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Operadores relacionales

## Operadores relacionales

Los operadores relacionales  $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $!=$ ,  $==$  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

## Operadores relacionales

Los operadores relacionales  $<$ ,  $>$ ,  $<=$ ,  $>=$ ,  $!=$ ,  $==$  son aplicables a punteros.

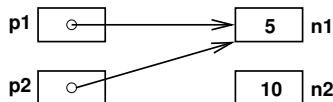
El valor del puntero (la dirección que almacena) se comporta como un número entero.

## 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;
```

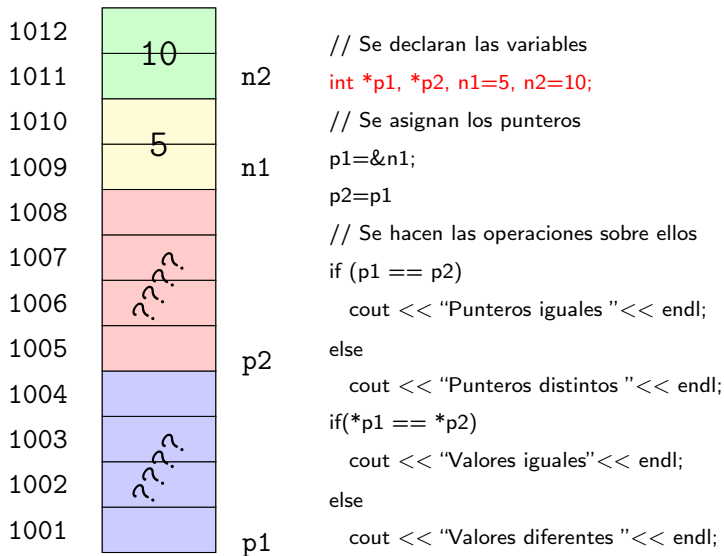
```
if(*p1 == *p2)
```

```
    cout << "Valores iguales" << endl;
```

```
else
```

```
    cout << "Valores diferentes " << endl;
```

# Operadores relacionales: Ejemplo anterior animado



```
// 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)
```

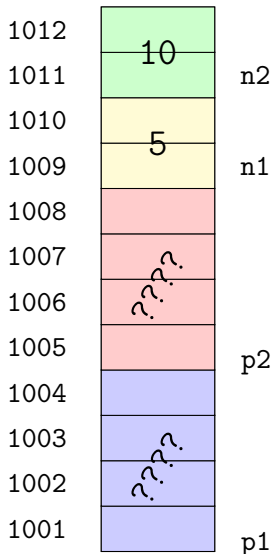
```
    cout << "Valores iguales" << endl;
```

```
else
```

```
    cout << "Valores diferentes " << endl;
```



# Operadores relacionales: Ejemplo anterior animado



// 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;
```

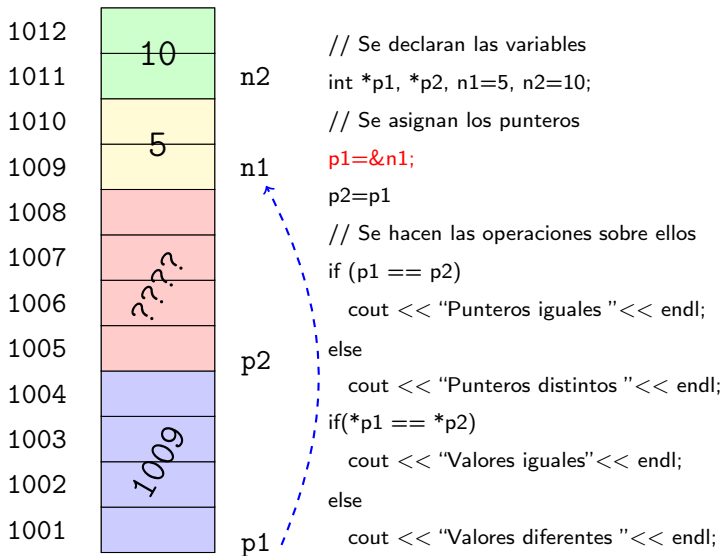
```
if(*p1 == *p2)
```

```
    cout << "Valores iguales" << endl;
```

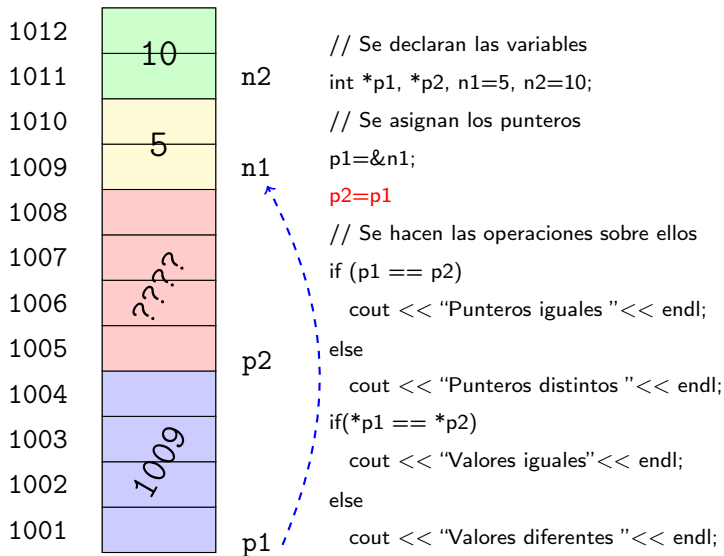
```
else
```

```
    cout << "Valores diferentes " << endl;
```

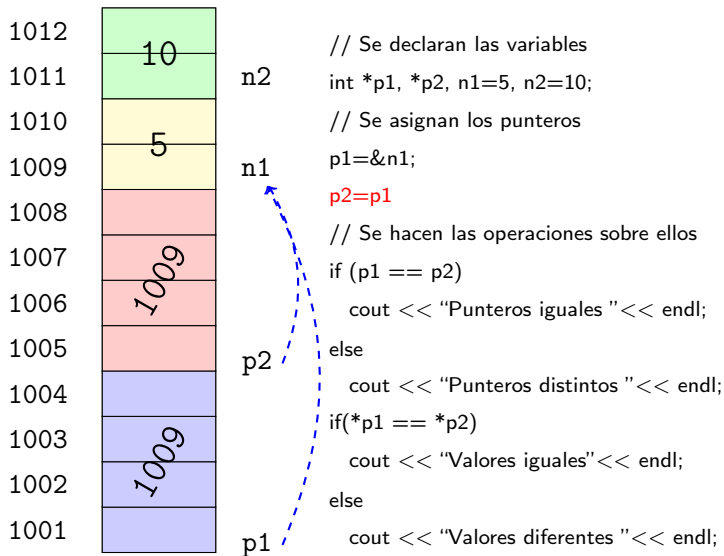
## Operadores relacionales: Ejemplo anterior animado



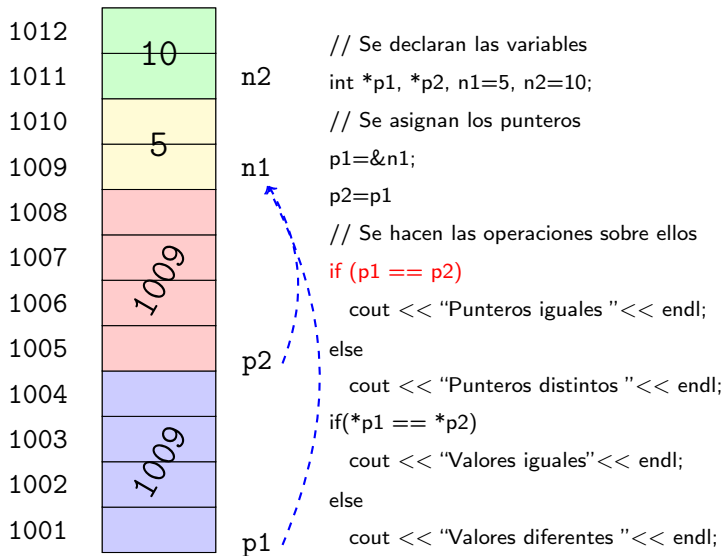
# Operadores relacionales: Ejemplo anterior animado



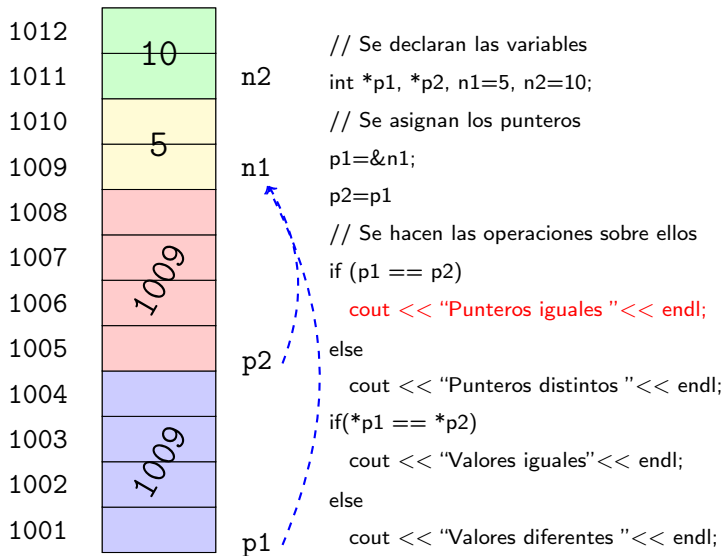
# Operadores relacionales: Ejemplo anterior animado



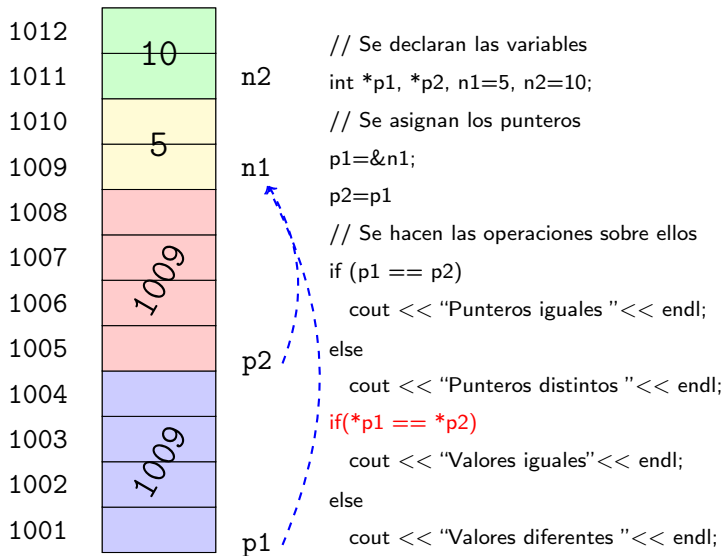
# Operadores relacionales: Ejemplo anterior animado



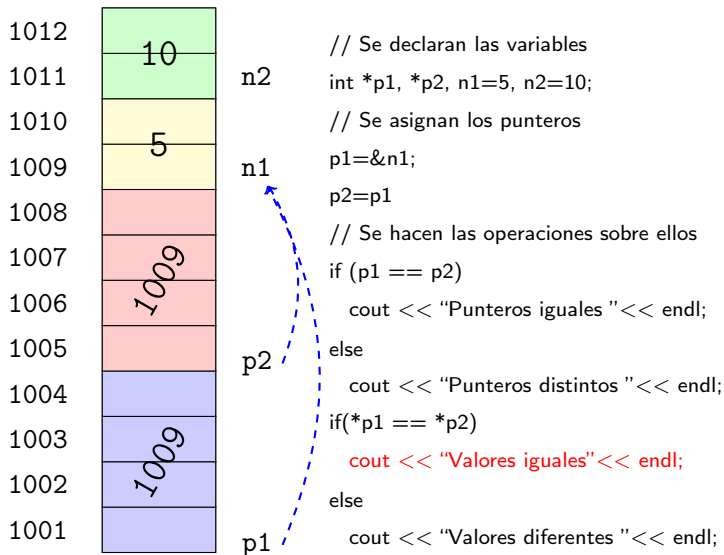
# Operadores relacionales: Ejemplo anterior animado



# Operadores relacionales: Ejemplo anterior animado



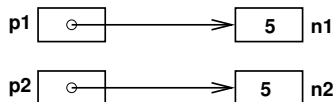
# Operadores relacionales: Ejemplo anterior animado





# Operadores relacionales: 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;
```

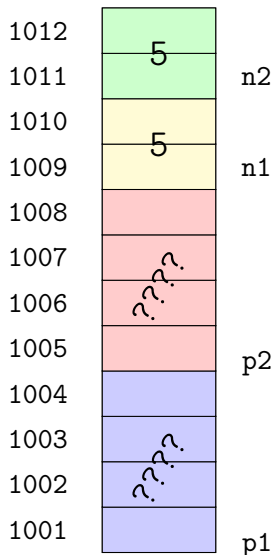
```
if(*p1 == *p2)
```

```
    cout << "Valores iguales" << endl;
```

```
else
```

```
    cout << "Valores diferentes " << endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



// 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;
```

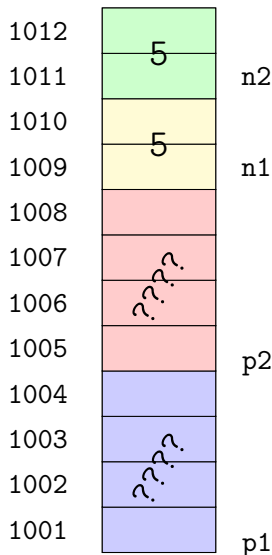
```
if(*p1 == *p2)
```

```
    cout << "Valores iguales" << endl;
```

```
else
```

```
    cout << "Valores diferentes " << endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



```
// 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;
```

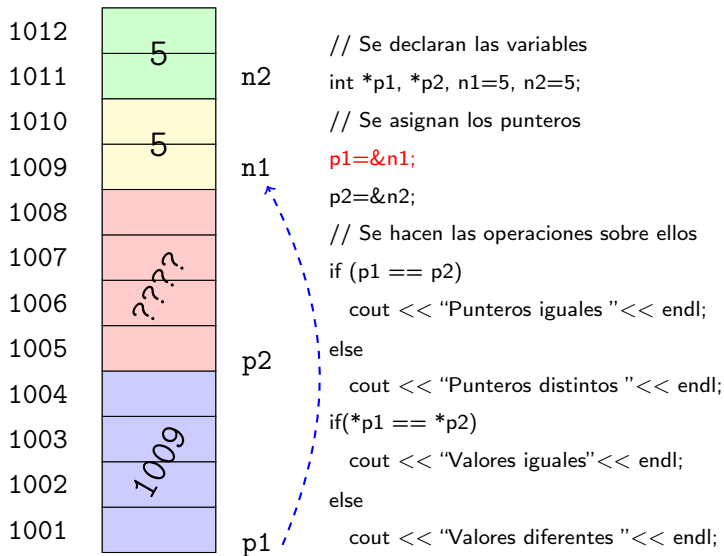
```
if(*p1 == *p2)
```

```
    cout << "Valores iguales" << endl;
```

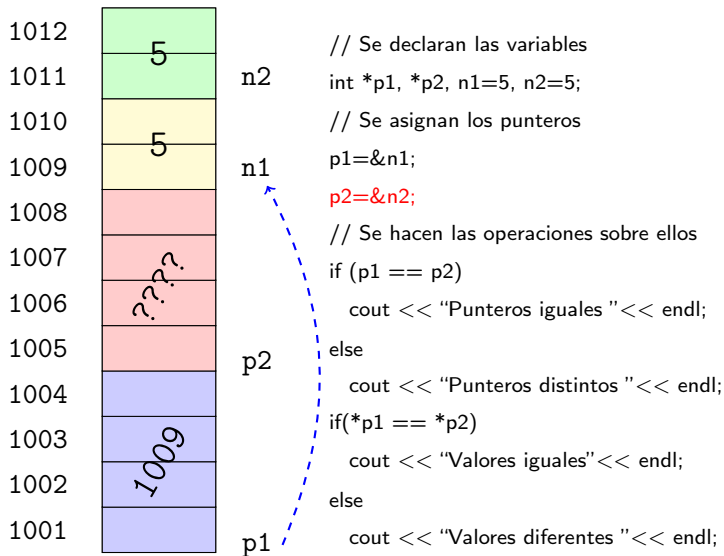
```
else
```

```
    cout << "Valores diferentes " << endl;
```

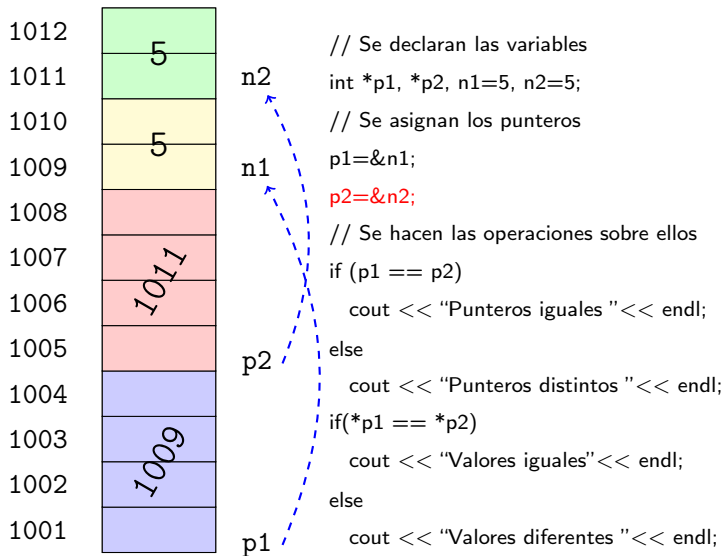
# Operadores relacionales: otro ejemplo (ej. animado)



# Operadores relacionales: otro ejemplo (ej. animado)

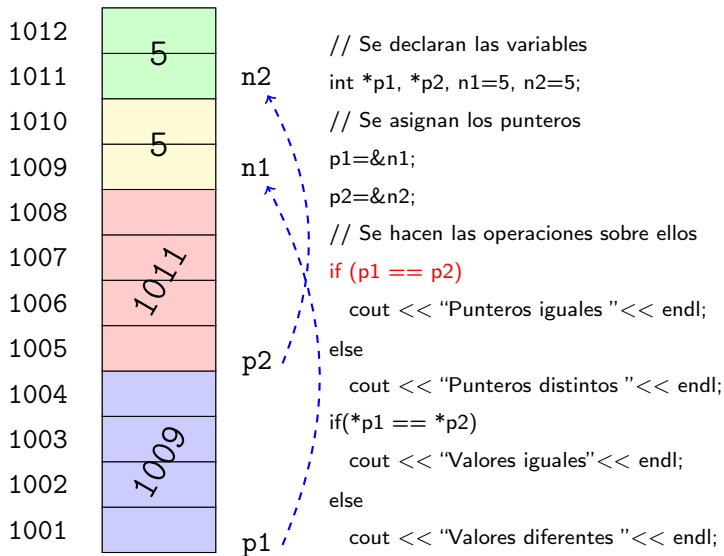


# Operadores relacionales: otro ejemplo (ej. animado)

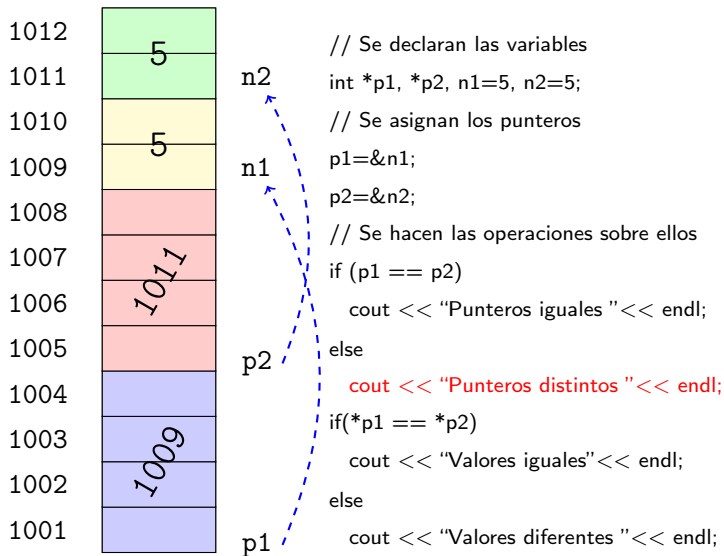




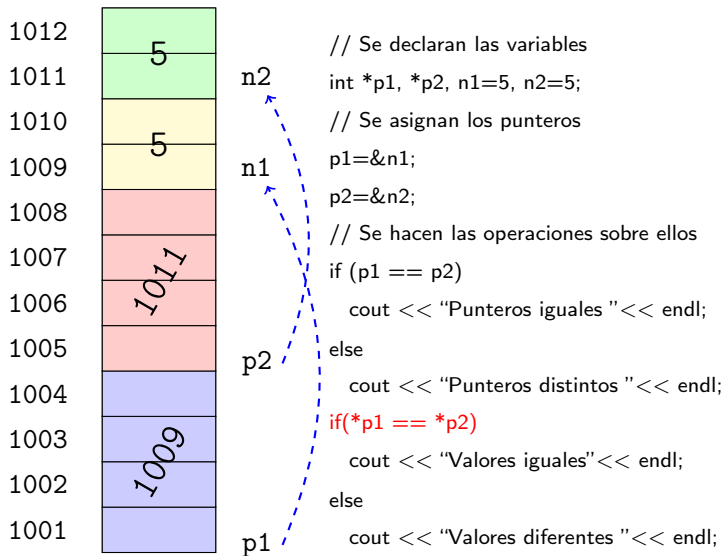
# Operadores relacionales: otro ejemplo (ej. animado)



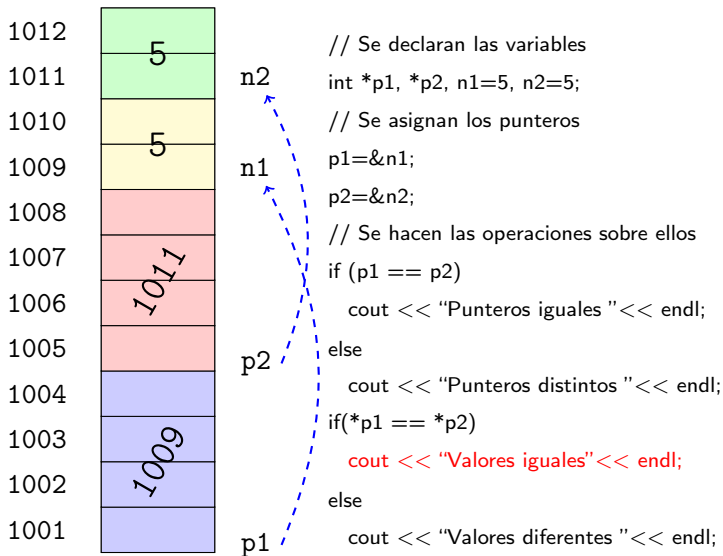
# Operadores relacionales: otro ejemplo (ej. animado)



# Operadores relacionales: otro ejemplo (ej. animado)



# Operadores relacionales: otro ejemplo (ej. animado)



# Operadores relacionales

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

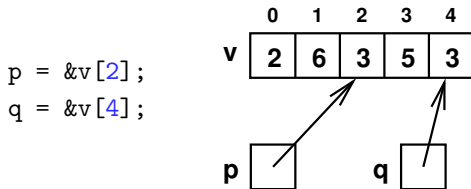
Permiten conocer la posición relativa de un objeto respecto a otro en la memoria.

# Operadores relacionales

## Operadores `<`, `>`, `<=`, `>=`

Permiten conocer la posición relativa de un objeto respecto a otro en la memoria.

- Solo son útiles si los dos punteros apuntan a objetos cuyas posiciones relativas guardan relación (por ejemplo, elementos del mismo array).



<code>p==q</code>	false
<code>p!=q</code>	true
<code>*p==*q</code>	true
<code>p&lt;q</code>	true
<code>p&gt;q</code>	false
<code>p&lt;=q</code>	true
<code>p&gt;=q</code>	false

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Operadores aritméticos

## Operadores aritméticos +, -, ++, --, += y -=

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.
- Al usar estos operadores, el valor del puntero (la dirección que almacena) se comporta **CASI** como un número entero.



# Operadores aritméticos

## Operadores aritméticos $+$ , $-$ , $++$ , $--$ , $+=$ y $-=$

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.
- Al usar estos operadores, el valor del puntero (la dirección que almacena) se comporta **CASI como un número entero**.

# Operadores aritméticos

## Operadores aritméticos $+$ , $-$ , $++$ , $--$ , $+=$ y $-=$

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:

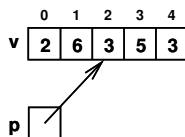
$$N * \text{sizeof}(\text{tipobase})$$

- Esto proporciona una forma rápida de **acceso a los elementos de un array**, aprovechando que todos sus elementos se almacenan en posiciones sucesivas.
- Al usar estos operadores, el valor del puntero (la dirección que almacena) se comporta **CASI como un número entero**.

# Operadores aritméticos: ejemplos

- Situación inicial:

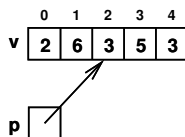
```
int v [5] = {2, 6, 3, 5, 3};  
int *p;  
p = &v[2];  
cout << p[1];
```



# Operadores aritméticos: ejemplos

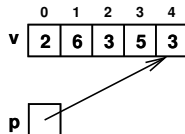
- Situación inicial:

```
int v [5] = {2, 6, 3, 5, 3};  
int *p;  
p = &v[2];  
cout << p[1];
```



- Si sumamos 2 a `p`:

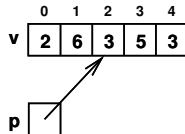
```
p+=2; // p=p+2
```



# Operadores aritméticos: ejemplos

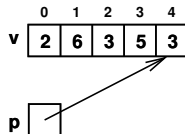
- Situación inicial:

```
int v [5] = {2, 6, 3, 5, 3};  
int *p;  
p = &v[2];  
cout << p[1];
```



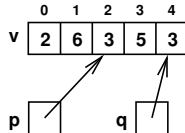
- Si sumamos 2 a `p`:

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



- ¿Qué devuelve `q - p`?

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



# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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])`.

# 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])`.



## Usar arrays como punteros

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])`.

## Usar punteros como arrays

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]`.

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

## Usar punteros como arrays

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]`.

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

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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Punteros y cadenas

## Cadena de caracteres

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.

## Literal cadena de caracteres

También vimos 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 *`

# Punteros y cadenas

## Cadena de caracteres

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.

## Literal cadena de caracteres

También vimos 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 *`



# Punteros y cadenas

## Cadena de caracteres

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.

## Literal cadena de caracteres

También vimos 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;
```

# 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 de compilación
```



# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Punteros a objetos struct o class

## Punteros a objetos

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

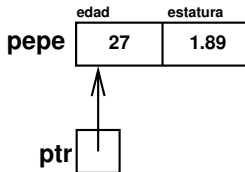
# Punteros a objetos struct o class

## Punteros a objetos

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

## Ejemplo: puntero a struct

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

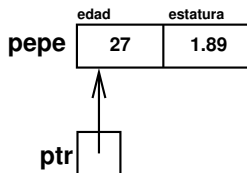


# Punteros a objetos struct o class

## Ejemplo: puntero a 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 válido desde fuera
//de método de la clase, edad es privado
ptr = &pepe;
cout << (*ptr).getEdad() << endl;
// cout << (*ptr).edad << endl; CUIDADO: no válido
//desde fuera de método de la clase, edad es privado
```





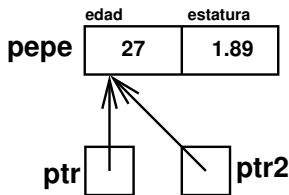
# Punteros a objetos struct o class

## Asignación de punteros a objetos

La asignación entre punteros funciona igual cuando apuntan a un **objeto struct** o **class** que cuando apuntan a datos de tipo primitivo.

## Ejemplo: Asignación de punteros a struct

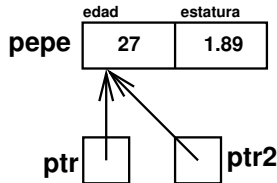
```
struct Persona{  
    int edad;  
    double estatura;  
};  
Persona pepe;  
Persona *ptr, *ptr2;  
pepe.edad=27;  
pepe.estatura=1.89;  
ptr = &pepe;  
ptr2 = ptr;  
cout << (*ptr).edad << endl;  
cout << (*ptr2).edad << endl;
```



# Punteros a objetos struct o class

## Ejemplo: Asignación de punteros a 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, *ptr2;
pepe.setEdad(27); pepe.setEstatura(1.89);
ptr = &pepe;
ptr2 = ptr;
cout << (*ptr).getEdad() << endl;
cout << (*ptr2).getEdad() << endl;
```



# Punteros a objetos struct o class: operador ->

## Operador – >

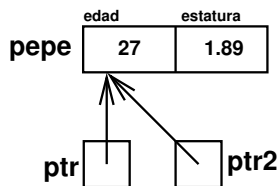
Si `p` es un puntero a un objeto `struct` o `class` podemos acceder a sus datos miembro de dos formas:

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

# Punteros a objetos struct o class: operador ->

## 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;
```

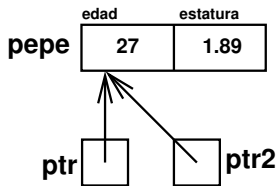


# Punteros a objetos struct o class: operador ->

## 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;
```



# Punteros a objetos struct o class

Struct y class con datos de tipo puntero

Un struct o class puede contener campos de tipo puntero.

## Ejemplo con struct

```

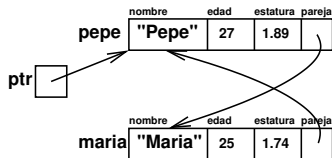
struct Persona{
    string nombre;
    int edad;
    double estatura;
    Persona *pareja;
};

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;

```



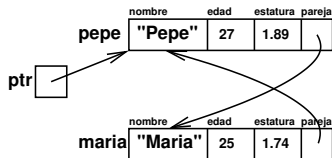


## Ejemplo con class

```

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::Persona(string name, int anios, double metros){
    nombre=name;
    edad=anios;
    estatura=metros;
    pareja=0;
}

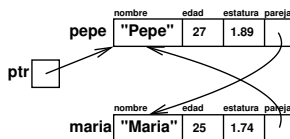
Persona* Persona::getPareja() const{
    return pareja;
}

void Persona::setPareja(Persona *compa){
    pareja=compa;
}
```

```

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;

```



# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Punteros y funciones I

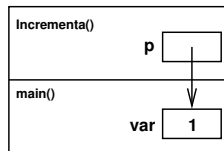
Un puntero puede ser un argumento de una función

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

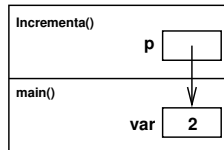
```

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

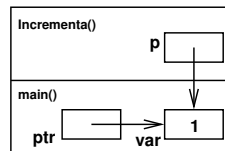
## Otra posibilidad

```

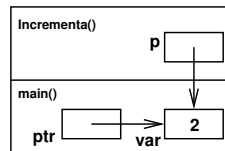
1 void incrementa(int* p){
2     (*p)++;
3 }
4 int main()
5 {
6     int var = 1;
7     int *ptr=&var;
8     cout << var << endl; // 1
9     incrementa(ptr);
10    cout << var << endl; // 2
11 }

```

### Situación en línea 1



### Situación en línea 3



# Punteros y funciones III

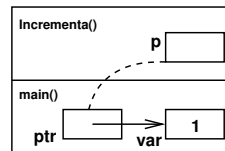
Un puntero se puede pasar por referencia

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

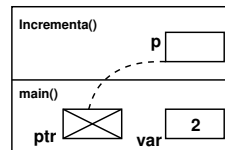
```

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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Punteros y const

- 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>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo 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

- 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>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo 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

- 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>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo 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

- 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>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo 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

- 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>
Solo el dato apuntado sea const	<code>const double *p;</code>
Solo 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 const y no const

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 solo lectura 'p'

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

## Punteros const y no const

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 solo lectura 'p'

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

## Puntero a dato no const

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.

## Puntero a dato no const

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.

## 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 solo lectura '\*p'

### 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

## Funciones con parámetro puntero a dato 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

## Array de constantes y puntero a dato 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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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:

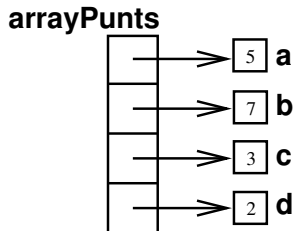
```
int* arrayPunts[4];
```

# 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;
```

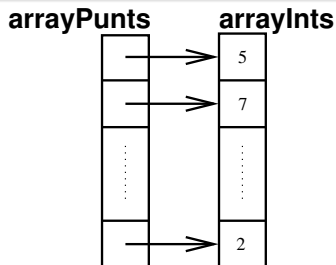
5 7 3 2



# 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
```



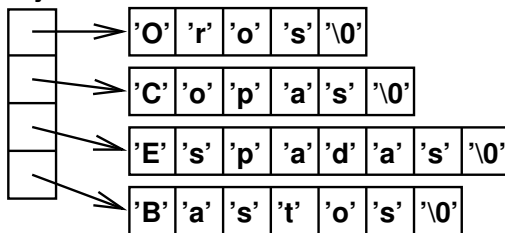


# Arrays de punteros

## 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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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 descendentemente

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

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

    ...
    ordenarPorSeleccion(array, DIMARRAY, ascendente ); // Ordena ascendentemente
    ...
    ordenarPorSeleccion(array, DIMARRAY, descendente ); // Ordena descendentemente
}
```

# 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

## Ordenación de un array ascendente o descendientemente (código completo)

Mostramos a continuación el código completo para este problema.

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

// prototipos
void ordenarPorSeleccion( int [], const int, bool (*)( int, int ) );
void intercambiar( int * const, int * const );
bool ascendente( int, int ); // implementa orden ascendente
bool descendente( int, int ); // implementa orden descendente

int main()
{
    const int DIMARRAY = 10;
    int orden; // 1 = ascendente, 2 = descendente
    int contador; // indice del array
    int array[DIMARRAY] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

    cout << "Introduce 1 para ordenar en orden ascendente,\n"
          << "Introduce 2 para ordenar en orden descendente: ";
    cin >> orden;
```



# Ejemplo de punteros a funciones

```
cout << "\nElementos en el orden original\n";
for ( contador = 0; contador < DIMARRAY; ++contador )
    cout << setw( 4 ) << array[contador];
if ( orden == 1 )
{
    ordenarPorSeleccion( array, DIMARRAY, ascendente );
    cout << "\nElementos en el orden ascendente\n";
}
else
{
    ordenarPorSeleccion( array, DIMARRAY, descendente );
    cout << "\nElementos en el orden descendente\n";
}
for ( contador = 0; contador < DIMARRAY; ++contador )
    cout << setw( 4 ) << array[contador];

cout << endl;
}
```

# Ejemplo de punteros a funciones

```

void ordenarPorSeleccion( int arrayInts[], const int utilArrayInts,
                        bool (*comparar)( int, int ) )
{
    int masPequenoOMasGrande;
    for ( int i = 0; i < utilArrayInts - 1; ++i )
    {
        masPequenoOMasGrande = i;
        for ( int index = i + 1; index < utilArrayInts; ++index )
            if ( !(*comparar)( arrayInts[ masPequenoOMasGrande ], arrayInts[ index ] ) )
                masPequenoOMasGrande = index;
        intercambiar( &arrayInts[ masPequenoOMasGrande ], &arrayInts[ i ] );
    }
}

void intercambiar( int * const elemento1Ptr, int * const elemento2Ptr )
{
    int aux = *elemento1Ptr;
    *elemento1Ptr = *elemento2Ptr;
    *elemento2Ptr = aux;
}

bool ascendente( int a, int b )
{
    return a < b; // devuelve true si a es menor que b
}

bool descendente( int a, int b )
{
    return a > b; // devuelve true si a es mayor que b
}

```

## 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
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# 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
```

# 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
```

# 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
```

## Parte II

# Gestión Dinámica de Memoria



# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Estructura de la memoria asociada a un programa

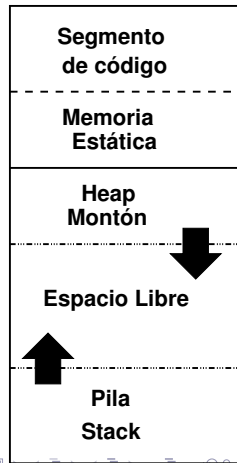
Gracias a la gestión de memoria del Sistema Operativo, los programas tienen una visión más simplificada del uso de la memoria, la cual ofrece una serie de componentes bien definidos.

## Segmento de código

Es la parte de la memoria asociada a un programa que contiene las instrucciones ejecutables del mismo.

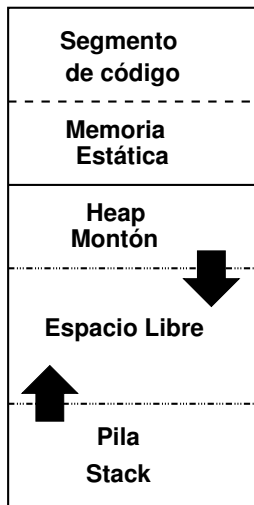
## Memoria estática

- Reserva antes de la ejecución del programa
- Permanece fija
- No requiere gestión durante la ejecución
- El sistema operativo se encarga de la reserva, recuperación y reutilización.
- Variables globales y static.



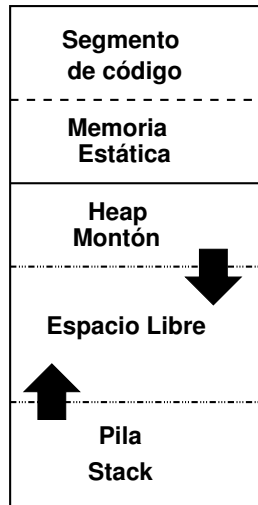
## La pila (Stack)

- Es una zona de memoria que gestiona las llamadas a funciones durante la ejecución de un programa.
- Cada vez que se realiza una llamada a una función en el programa, se crea un **entorno de programa**, que se libera cuando acaba su ejecución.
- La reserva y liberación de la memoria la realiza el S.O. de forma automática durante la ejecución del programa.
- Las variables locales no son variables estáticas. Son un tipo especial de variables dinámicas, conocidas como **variables automáticas**.



## El montón (Heap)

- Es una zona de memoria donde se reservan y se liberan “trozos” durante la ejecución de los programas según sus propias necesidades.
- Esta memoria surge de la necesidad de los programas de “crear nuevas variables” en tiempo de ejecución con el fin de optimizar el almacenamiento de datos.



## Ejemplo

Supongamos que se desea realizar un programa que permita trabajar con una lista de datos relativos a una persona.

```
struct Persona{  
    char nombre[80];  
    int DNI;  
    image foto;  
};
```

¿Qué inconvenientes tiene la definición `Persona arrayPersona[100]`?

- Si el número de posiciones usadas es mucho menor que 100, tenemos reservada memoria que no vamos a utilizar.
- Si el número de posiciones usadas es mayor que 100, el programa no funcionará correctamente.

"Solución": Ampliar la dimensión del array y volver a compilar.

## Ejemplo

Supongamos que se desea realizar un programa que permita trabajar con una lista de datos relativos a una persona.

```
struct Persona{  
    char nombre[80];  
    int DNI;  
    image foto;  
};
```

¿Qué inconvenientes tiene la definición `Persona arrayPersona[100]`?

- Si el número de posiciones usadas es mucho menor que 100, tenemos reservada memoria que no vamos a utilizar.
- Si el número de posiciones usadas es mayor que 100, el programa no funcionará correctamente.

**"Solución":** Ampliar la dimensión del array y volver a compilar.

## Consideraciones:

- La utilización de variables estáticas o automáticas para almacenar información cuyo tamaño no es conocido a priori (solo se conoce exactamente en tiempo de ejecución) resta generalidad al programa.
- La alternativa válida para solucionar estos problemas consiste en la posibilidad de reservar la memoria justa que se precise (y liberarla cuando deje de ser útil), **en tiempo de ejecución**.
- Esta memoria se reserva en el Heap y, habitualmente, se habla de **variables dinámicas** para referirse a los bloques de memoria del Heap que se reservan y liberan en tiempo de ejecución.

# Contenido del tema

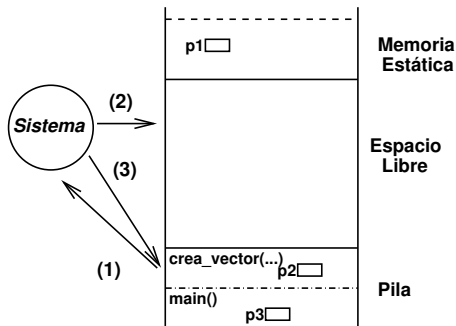
- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 3 Punteros y arrays
- 4 Punteros y cadenas
- 5 Punteros, struct y class
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas



# Gestión dinámica de la memoria

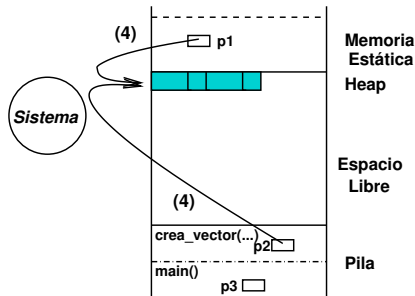
El sistema operativo es el encargado de controlar la memoria que queda libre en el sistema.

- (1) Petición al S.O. (tamaño)
- (2) El S.O. comprueba si hay suficiente espacio libre.
- (3) Si hay espacio suficiente, devuelve la ubicación donde se encuentra la memoria reservada, y marca dicha zona como memoria ocupada.

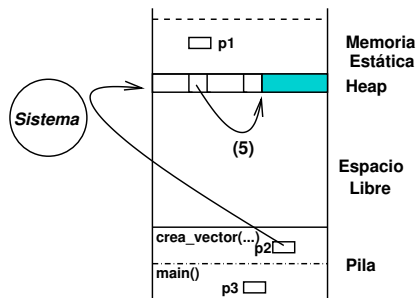


## Reserva de memoria

- (4) La ubicación de la zona de memoria se almacena en una variable estática (**p1**) o en una variable automática (**p2**).  
Por tanto, si la petición devuelve una dirección de memoria, **p1** y **p2** deben ser variables de tipo *puntero* al tipo de dato que se ha reservado.

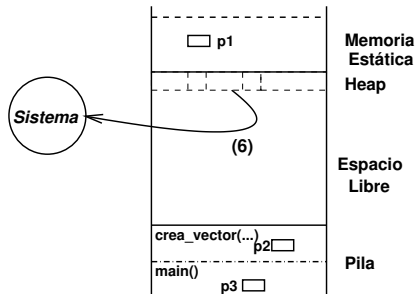


- 5 A su vez, es posible que las nuevas variables dinámicas creadas puedan almacenar la dirección de nuevas peticiones de reserva de memoria.



## Liberación de memoria

6 Finalmente, una vez que se han utilizado las variables dinámicas y ya no se van a necesitar más, es necesario liberar la memoria que se está utilizando e informar al S.O. que esta zona de memoria vuelve a estar libre para su utilización.



## ¡ RECORDAR LA METODOLOGÍA !

- 1 Reservar memoria.
- 2 Utilizar memoria reservada.
- 3 Liberar memoria reservada.

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# El operador new

## Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (`sizeof(tipo)` bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;
p = new <tipo>;
```

- Si `new` no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

## Otra opción (no recomendable)

```
<tipo> *p;
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (0).

# El operador new

## Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof(*tipo*)** bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;
p = new <tipo>;
```

- Si **new** no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

## Otra opción (no recomendable)

```
<tipo> *p;
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (0).



# El operador new

## Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (**sizeof(*tipo*)** bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;  
p = new <tipo>;
```

- Si **new** no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

## Otra opción (no recomendable)

```
<tipo> *p;  
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (0).

# El operador new

## Operador new

Reserva una zona de memoria en el Heap del tamaño adecuado para almacenar un dato del tipo *tipo* (`sizeof(tipo)` bytes), devolviendo la dirección de memoria donde empieza la zona reservada.

```
<tipo> *p;
p = new <tipo>;
```

- Si `new` no puede reservar espacio (p.e. no hay suficiente memoria disponible), se provoca una excepción y el programa termina.
- Por ahora supondremos que siempre habrá suficiente memoria.

## Otra opción (no recomendable)

```
<tipo> *p;
p = new (nothrow) <tipo>;
```

En caso de que no se haya podido hacer la reserva devuelve el puntero nulo (0).

## Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```

### Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

## Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```

### Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

## Ejemplo

```
int main(){  
    int *p;  
  
    p = new int;  
    *p = 10;  
}
```

### Notas:

- Observad que **p** se declara como un puntero más.
- Se pide memoria en el Heap para guardar un dato **int**. Si hay espacio para satisfacer la petición, **p** apuntará al principio de la zona reservada por **new**. Asumiremos que siempre hay memoria libre para asignar.
- Se trabaja, como ya sabemos, con el objeto referenciado por **p**.

# El operador delete

## Operador delete

Libera la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

```
delete puntero;
```

### Ejemplo

```
int main(){  
    int *p, q=10;  
  
    p = new int;  
    *p = q;  
    .....  
    delete p;  
}
```

### Notas:

- El objeto referenciado por **p** deja de ser “operativo” y la memoria que ocupaba está disponible para nuevas peticiones con **new**.

# El operador delete

## Operador delete

Libera la memoria del Heap que previamente se había reservado y que se encuentra referenciada por un puntero.

```
delete puntero;
```

## Ejemplo

```
int main(){  
    int *p, q=10;  
  
    p = new int;  
    *p = q;  
    .....  
    delete p;  
}
```

## Notas:

- El objeto referenciado por **p** deja de ser “operativo” y la memoria que ocupaba está disponible para nuevas peticiones con **new**.

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas



# Objetos dinámicos compuestos

## Objetos dinámicos compuestos

Con objetos `struct` y `class` la metodología a seguir es la misma:

- Operador `new`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los datos del objeto.
  - Y llama al **constructor** de la clase para inicializar los datos del objeto.
- Operador `delete`:
  - Llama al **destructor** de la clase. (lo veremos en tema 4)
  - Y después **libera la memoria** de todos y cada uno de los campos del objeto.

# Objetos dinámicos compuestos

## Ejemplo con struct

```
struct Persona{  
    char nombre[80];  
    char DNI[10];  
};  
  
int main(){  
    Persona *yo;  
  
    yo = new Persona;  
    lee_linea((*yo).nombre,80);  
    lee_linea((*yo).DNI,10);  
    .....  
    delete yo;  
}
```

# Objetos dinámicos compuestos

## Ejemplo con class

```
class Estudiante {  
    string nombre;  
    int nAsignaturasMatricula;  
    vector<int> codigosAsignaturasMatricula;  
public:  
    Estudiante();  
    Estudiante(string name);  
  
    void setNombre(string nuevoNombre);  
    string getNombre() const;  
    void insertaAsignatura(int codigo);  
    int getNumeroAsignaturas() const;  
    int getCodigoAsignatura(int index) const;  
    ...  
};
```

# Objetos dinámicos compuestos

```
int main() {  
    Estudiante* ramon;  
    ramon=new Estudiante("Ramón Rodríguez Ramírez");  
    ramon->insertaAsignatura(302);  
    ramon->insertaAsignatura(307);  
    ramon->insertaAsignatura(205);  
    ...  
    delete ramon;  
}
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos**
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Arrays dinámicos

## Motivación

- Hasta ahora, solo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener. P.e.  
`int vector[20];`
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.

## Array dinámico

Usando memoria dinámica, podemos crear arrays dinámicos que tengan **justo el tamaño necesario**.

Podemos, además, crearlos **justo en el momento** en el que lo necesitamos y destruirlos cuando dejen de ser útiles.

# Arrays dinámicos

## Motivación

- Hasta ahora, solo podíamos crear un array conociendo *a priori* el número máximo de elementos que podría llegar a tener. P.e.  
`int vector[20];`
- Esa memoria está ocupada durante la ejecución del módulo en el que se realiza la declaración.

## Array dinámico

Usando memoria dinámica, podemos crear arrays dinámicos que tengan **justo el tamaño necesario**.

Podemos, además, crearlos **justo en el momento** en el que lo necesitamos y destruirlos cuando dejen de ser útiles.

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 **Arrays dinámicos**
  - **Arrays dinámicos de datos de tipo primitivo**
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas



# Arrays dinámicos

## Operador new[]

Reserva una zona de memoria en el Heap para almacenar **num** datos de tipo **<tipo>**, devolviendo la dirección de memoria inicial.

**num** es un entero estrictamente mayor que 0.

```
<tipo> *p;
```

```
p = new <tipo> [num];
```

## Operador delete[]

Libera (pone como disponible) la zona de memoria **previamente reservada** por una orden **new[]**, zona referenciada por puntero.

```
delete [] puntero;
```

# Arrays dinámicos

## Operador new[]

Reserva una zona de memoria en el Heap para almacenar `num` datos de tipo `<tipo>`, devolviendo la dirección de memoria inicial.

`num` es un entero estrictamente mayor que 0.

```
<tipo> *p;  
p = new <tipo> [num];
```

## Operador delete[]

Libera (pone como disponible) la zona de memoria **previamente reservada** por una orden `new[]`, zona referenciada por puntero.

```
delete [] puntero;
```

# Arrays dinámicos: Ejemplo I

## Ejemplo de creación y destrucción de array dinámico

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     int *v=0, n;
6
7     cout << "Numero de casillas: ";
8     cin >> n;
9     // Reserva de memoria
10    v = new int [n];
```



# Ejemplo I

```
1  for (int i= 0; i<n; i++) {    // Lectura del vector dinamico
2      cout << "Valor en casilla "<<i<< ": ";
3      cin >> v[i];
4  }
5  cout << endl;
6
7  for (int i= 0; i<n; i++) // Escritura del vector dinamico
8      cout << "En la casilla " << i
9          << " guardo: "<< v[i] << endl;
10
11  delete [] v; // Liberar memoria
12  v = 0;
13 }
```

## Ejemplo

Una función que devuelve una copia de un array automático (o dinámico) en un array dinámico.

```
1 #include <iostream>
2 using namespace std;
3
4 int *copia_vector(const int v[], int n){
5     int *copia = new int[n];
6     for (int i=0; i<n; i++)
7         copia[i]=v[i];
8     return copia;
9 }
10 int main(){
11     int v1[30], *v2=0, m;
12     cout << "Numero de casillas: ";
13     cin >> m;
```



```
14  for (int i=0; i<m; i++) { // Rellenar el vector
15      cout << "Valor en casilla " << i << ": ";
16      cin >> v1[i];
17  }
18  cout << endl;
19
20  // Copiar en v2 (dinámico) el vector v1
21  v2 = copia_vector(v1,m);
22
23  for (int i=0; i<m; i++) // Escribir vector v2
24      cout << "En la casilla " << i
25          << " guardo: " << v2[i] << endl;
26
27  delete [] v2; // Liberar memoria
28  v2 = 0;
29 }
```

## ¡Cuidado!

Un **error** muy común a la hora de construir una función que copie un array es el siguiente:

```
int *copia_vector(const int v[], int n){  
    int copia[100];  
    for (int i=0; i<n; i++)  
        copia[i]=v[i];  
    return copia;  
}
```

## ¡Cuidado!

Al ser copia una variable local no puede ser usada fuera del ámbito de la función en la que está definida.

## ¡Cuidado!

Un **error** muy común a la hora de construir una función que copie un array es el siguiente:

```
int *copia_vector(const int v[], int n){  
    int copia[100];  
    for (int i=0; i<n; i++)  
        copia[i]=v[i];  
    return copia;  
}
```

## ¡Cuidado!

Al ser copia una variable local no puede ser usada fuera del ámbito de la función en la que está definida.



## Ejemplo: Ampliación del espacio ocupado por un array dinámico

```
void redimensionar (int* &v, int& tama, int aumento){
    if(tama+aumento > 0){
        int *v_ampliado = new int[tama+aumento];

        for (int i=0; (i<tama) && (i<tama+aumento); i++)
            v_ampliado[i] = v[i];
        delete[] v;
        v = v_ampliado;
        tama=tama+aumento;
    }
}
```

Cuestiones a tener en cuenta:

- `v` y `tama` se pasan por referencia porque se van a modificar.
- Es necesario liberar `v` antes de asignarle el valor de `v_ampliado`.
- El aumento de tamaño puede ser positivo o negativo.

## Ejemplo: Ampliación del espacio ocupado por un array dinámico

```
void redimensionar (int* &v, int& tama, int aumento){
    if(tama+aumento > 0){
        int *v_ampliado = new int[tama+aumento];

        for (int i=0; (i<tama) && (i<tama+aumento); i++)
            v_ampliado[i] = v[i];
        delete[] v;
        v = v_ampliado;
        tama=tama+aumento;
    }
}
```

Cuestiones a tener en cuenta:

- `v` y `tama` se pasan por referencia porque se van a modificar.
- Es necesario liberar `v` antes de asignarle el valor de `v_ampliado`.
- El aumento de tamaño puede ser positivo o negativo.

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos**
  - Arrays dinámicos de datos de tipo primitivo
  - **Arrays dinámicos de objetos**
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Array dinámico de objetos

## Array dinámico de objetos

Usando el operador `new[]` y `delete[]` podemos crear y destruir también arrays dinámicos de objetos `struct` y `class`

- Operador `new[]`:
  - **Reserva la memoria** necesaria para almacenar todos y cada uno de los objetos del array.
  - Y llama al **constructor** para cada objeto del array.
- Operador `delete[]`:
  - Llama al **destructor** de la clase con cada objeto del array.
  - Y después **libera la memoria** ocupada por el array de objetos.

# Array dinámico de objetos

## Ejemplo con class

```
class Estudiante {  
    string nombre;  
    int nAsignaturasMatricula;  
    vector<int> codigosAsignaturasMatricula;  
public:  
    Estudiante();  
    Estudiante(string name);  
  
    void setNombre(string nuevoNombre);  
    string getNombre() const;  
    void insertaAsignatura(int codigo);  
    int getNumeroAsignaturas() const;  
    int getCodigoAsignatura(int index) const;  
    ...  
};
```

# Array dinámico de objetos

```
int main() {  
    Estudiante* arrayEstudiantes;  
    arrayEstudiantes=new Estudiante[50];  
    arrayEstudiantes[0].setNombre("Ramón Rodríguez Ramírez");  
    arrayEstudiantes[0].insertaAsignatura(302);  
    arrayEstudiantes[0].insertaAsignatura(307);  
    arrayEstudiantes[0].insertaAsignatura(205);  
    ...  
    delete[] arrayEstudiantes;  
}
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica**
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas

# Clases que contienen datos en memoria dinámica

## Clases que contienen datos en memoria dinámica

Una clase puede contener datos miembro punteros que pueden usarse para alojar datos en memoria dinámica. Para ello:

- Los **constructores** pueden reservar la memoria dinámica al crear los objetos.
- Otros métodos podrían aumentar o disminuir el tamaño de la memoria dinámica necesaria.
- El **destructor** liberará automáticamente la memoria dinámica que contenga el objeto.

Lo veremos en tema 4. Por ahora, lo haremos explícitamente usando un método `liberar()`.



# Clases que contienen datos en memoria dinámica

## Ejemplo: clase Poligono

Contiene un array dinámico con los vértices (objetos Punto).

```
class Punto{
    double x;
    double y;
public:
    Punto(){x=0; y=0;};
    Punto(int x, int y){this->x=x; this->y=y};
    double getX(){return x;} const;
    double getY(){return y;} const;
    double setXY(int x, int y){this->x=x; this->y=y};
};
```

# Clases que contienen datos en memoria dinámica

```
class Poligono{
    int nVertices;
    Punto* vertices;
public:
    Poligono();
    ~Poligono(); // destructor (lo veremos en tema 4)
    Poligono(const Poligono& otro); // constructor copia (tema 4)
    Poligono& operator=(const Poligono& otro); // op asignación (tema 4)
    int getNumeroVertices() const;
    Punto getVertice(int index) const;
    void addVertice(Punto v);
    ...
};
```

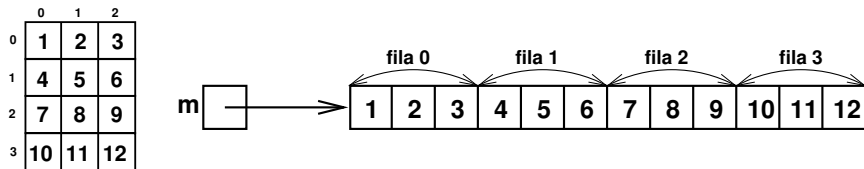
# Clases que contienen datos en memoria dinámica

```
int main() {  
    Punto punto;  
    Poligono poligono;  
    punto.setXY(10,10);  
    poligono.addVertice(punto);  
    ...  
    poligono.destruir();  
}
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas**
- 19 Lista de celdas enlazadas

# Matriz 2D usando un array 1D



- Creación de la matriz:

```
int *m;
int nfil, ncol;
m = new int[nfil*ncol];
```

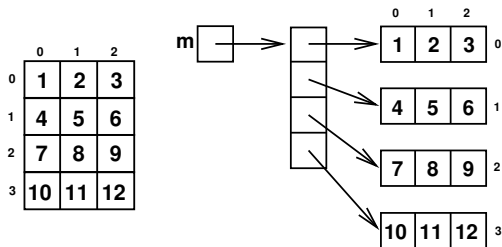
- Acceso al elemento f,c:

```
int a;
a = m[f*ncol+c];
```

- Liberación de la matriz:

```
delete[] m;
```

# Matriz 2D usando un array 1D de punteros a arrays 1D



- Creación de la matriz:

```
int **m;
int nfil, ncol;
m = new int*[nfil];
for (int i=0; i<nfil;++i)
    m[i] = new int[ncol];
```

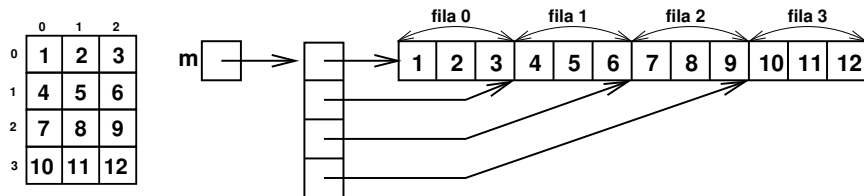
- Acceso al elemento f,c:

```
int a;
a = m[f][c];
```

- Liberación de la matriz:

```
for(int i=0;i<nfil;++i)
    delete[] m[i];
delete[] m;
```

# Matriz 2D usando un array 1D de punteros a un único array



- Creación de la matriz:

```
int **m;
int nfil, ncol;
m = new int*[nfil];
m[0] = new int[nfil*ncol];
for (int i=1; i<nfil;++i)
    m[i] = m[i-1]+ncol;
```

- Acceso al elemento  $f, c$ :

```
int a;
a = m[f][c];
```

- Liberación de la matriz:

```
delete[] m[0];
delete[] m;
```

# Contenido del tema

- 1 Definición y Declaración de variables
- 2 Operaciones con punteros
  - Operador de dirección &
  - Operador de indirección \*
  - Asignación e inicialización de punteros
  - Operadores relacionales
  - Operadores aritméticos
- 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 Arrays dinámicos
  - Arrays dinámicos de datos de tipo primitivo
  - Arrays dinámicos de objetos
- 17 Clases que contienen datos en memoria dinámica
- 18 Matrices dinámicas
- 19 Lista de celdas enlazadas**

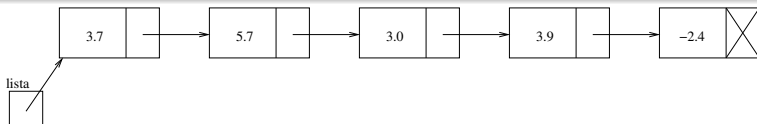


# Lista de celdas enlazadas

## Lista de celdas enlazadas

Es una **estructura de datos lineal** que nos permite guardar un conjunto de elementos del mismo tipo usando celdas enlazadas.

- Cada celda se alojará en el Heap.
- Usaremos punteros para enlazar una celda con la siguiente.



```
struct Celda{  
    double dato;  
    Celda* sig;  
}
```

# Lista de celdas enlazadas

```
#include <iostream>
using namespace std;
struct Celda{
    double dato;
    Celda* sig;
};

int main(){
    Celda* lista;
    double valor;

    lista = 0;
    cin >> valor;
    while(valor != 0.0){ // Creación de las celdas de la lista
        Celda* aux = new Celda;
        aux->dato = valor;
        aux->sig = lista;
        lista = aux;
        cin >> valor;
    }
```

# Lista de celdas enlazadas

```
// Mostrar la lista en salida estándar
aux = lista;
while(aux != 0){
    cout << aux -> dato << " ";
    aux = aux->sig;
}
cout << endl;

while (lista != 0) { // Destrucción de la lista
    Celda* aux = lista;
    lista = aux->sig;
    delete aux;
}
}
```

# Lista de celdas enlazadas

## Función para insertar al principio de la lista

```
void insertarPrincipioLista(Celda* &lista, double valor){  
    Celda* aux = new Celda;  
    aux->dato = valor;  
    aux->sig = lista;  
    lista = aux;  
}
```

## Función para mostrar el contenido de la lista

```
void mostrarLista(Celda* lista){  
    Celda* aux = lista;  
    while(aux != 0){  
        cout << aux -> dato << " ";  
        aux = aux->sig;  
    }  
    cout << endl;  
}
```

# Lista de celdas enlazadas

## Función para insertar al principio de la lista

```
void insertarPrincipioLista(Celda* &lista, double valor){
    Celda* aux = new Celda;
    aux->dato = valor;
    aux->sig = lista;
    lista = aux;
}
```

## Función para mostrar el contenido de la lista

```
void mostrarLista(Celda* lista){
    Celda* aux = lista;
    while(aux != 0){
        cout << aux -> dato << " ";
        aux = aux->sig;
    }
    cout << endl;
}
```

# Lista de celdas enlazadas

## Función para destruir la lista

```
void destruirLista(Celda* &lista){  
    while (lista != 0) {  
        Celda* aux = lista;  
        lista = aux->sig;  
        delete aux;  
    }  
}
```

# Lista de celdas enlazadas

## Función para insertar al final de la lista

- Si la lista está vacía, inserto al principio.
- Si la lista no esta vacía
  - Busco puntero p a última celda.
  - Inserto después de posición p.

# Lista de celdas enlazadas

## Función para insertar después de una celda apuntada por un puntero p

- Hacer que aux (puntero auxiliar) apunte a nueva celda.
- Asignar a aux->dato, el nuevo dato.
- Asignar a aux->sig, el valor de p->sig.
- Asignar a p->sig el valor de aux.



# Lista de celdas enlazadas

## Función para insertar antes de una celda apuntada por un puntero p

- Si se quiere insertar al principio o la lista está vacía, insertar al principio.
- En caso contrario:
  - Buscar un puntero aux que apunte a celda anterior a la apuntada por p
  - Hacer que aux2 (puntero auxiliar) apunte a nueva celda.
  - Asignar a aux2->dato, el nuevo dato.
  - Asignar a aux2->sig, el valor de p.
  - Asignar a aux->sig, el valor de aux2.

## Función para borrar la celda apuntada por un puntero p

# Lista de celdas enlazadas

## Función para insertar antes de una celda apuntada por un puntero p

- Si se quiere insertar al principio o la lista está vacía, insertar al principio.
- En caso contrario:
  - Buscar un puntero aux que apunte a celda anterior a la apuntada por p
  - Hacer que aux2 (puntero auxiliar) apunte a nueva celda.
  - Asignar a aux2->dato, el nuevo dato.
  - Asignar a aux2->sig, el valor de p.
  - Asignar a aux->sig, el valor de aux2.

## Función para borrar la celda apuntada por un puntero p