# Metodología de la Programación

## Tema 2. Punteros y memoria dinámica

## Parte 1. Punteros

Sylvia Acid (acid@decsai.ugr.es)
derivado de la obra de Andrés Cano
Departamento de Ciencias de la Computación e I.A.




Curso 2025-26

# Índice I

# Motivación

- En muchos problemas es difícil saber en tiempo de compilación la cantidad de memoria que se va a necesitar para almacenar los datos presentes.
- Este problema tendría solución si pudieramos definir variables cuyo espacio se reserva en tiempo de ejecución.
- La memoria dinámica permite ajustar el espacio requerido a los datos presentes, en tiempo de ejecución.
- En C++, la gestión de esta memoria es responsabilidad del programador.
- Necesitamos un nuevo tipo de dato tipo puntero.

# Indice

# El tipo puntero a ...

El par **identificador-valor**. El identificador hace referencia a una zona de memoria donde se almacena el valor.

## Una variable de tipo puntero a ... va a contener direcciones de memoria

- Valores válidos de memoria y una dirección especial llamada *dirección nula*, la constante `nullptr` desde C++11.

## Sintaxis

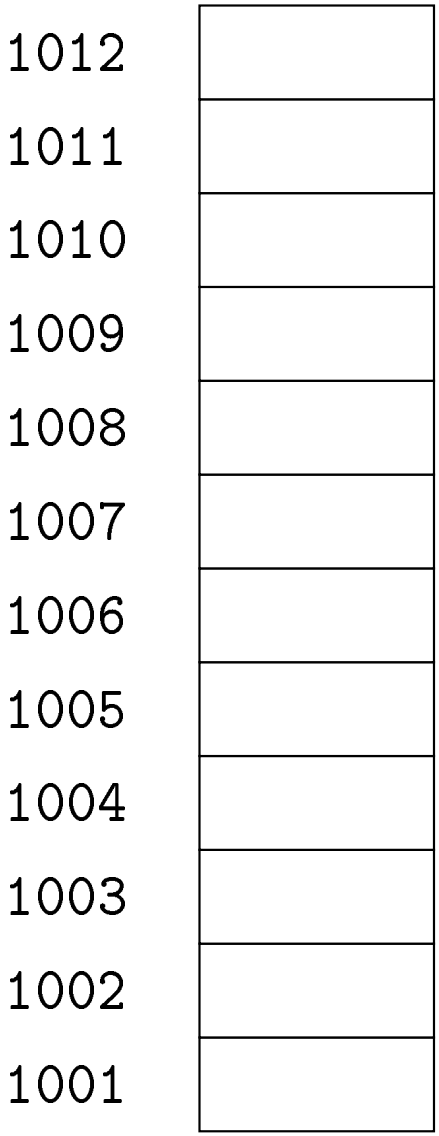
```
<tipo> *<identificador>;
```

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

# Ejemplo: Declaración de punteros

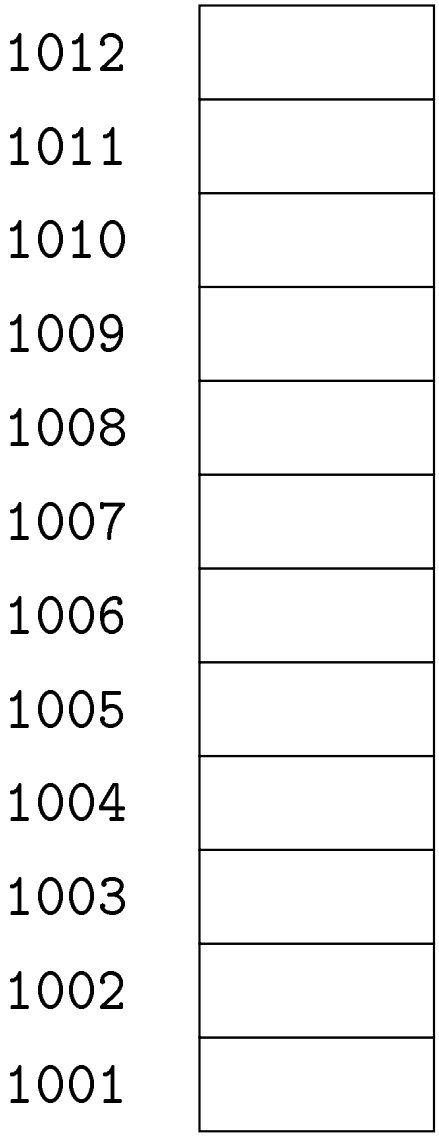
```
1
2     ..................
3
4     // Se declara variable de tipo entero
5     int i=5;
6
7     // Se declara variable de tipo char
8     char c='a';
9
10    // Se declara puntero a entero
11    int * ptri;
12
13    // Se declara puntero a char
14    char * ptrc;
15
16    ..................
17
```

# Ejemplo: Declaración de punteros



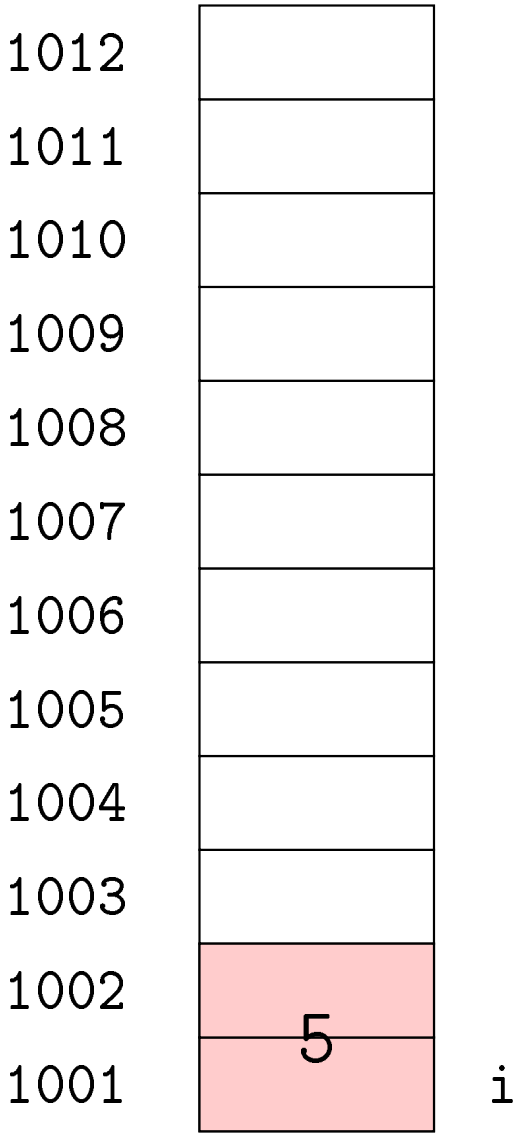

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



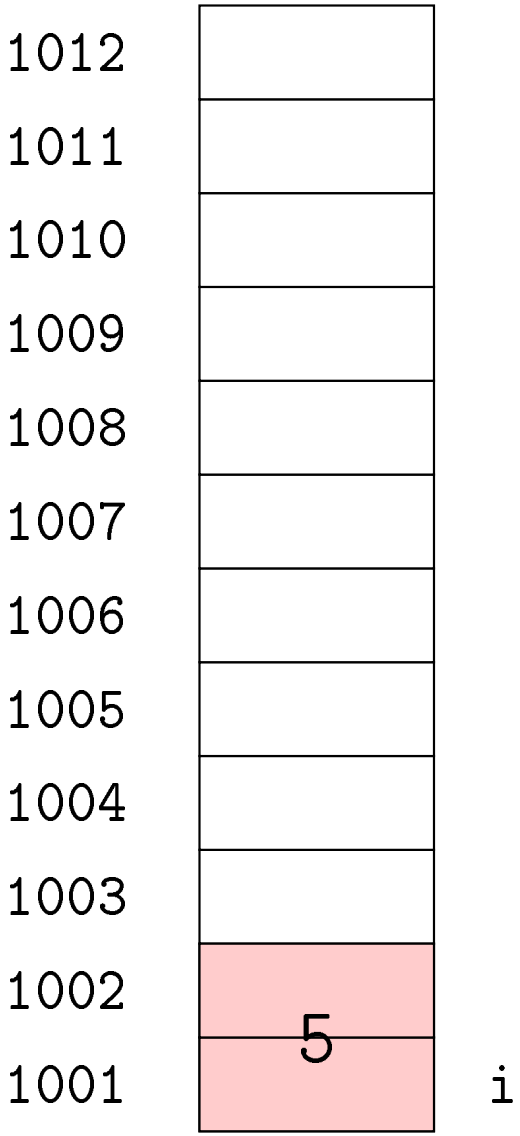

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



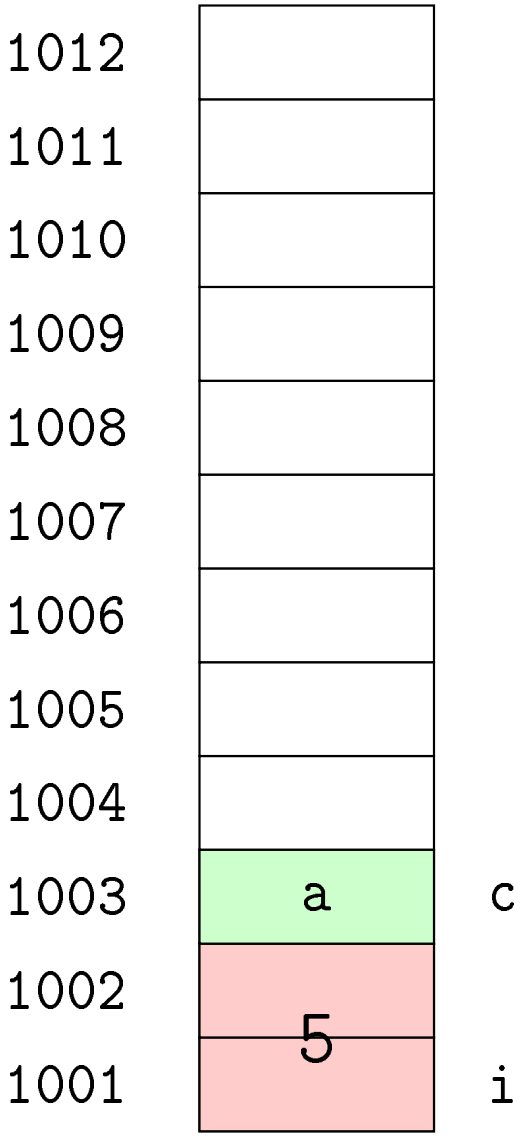

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



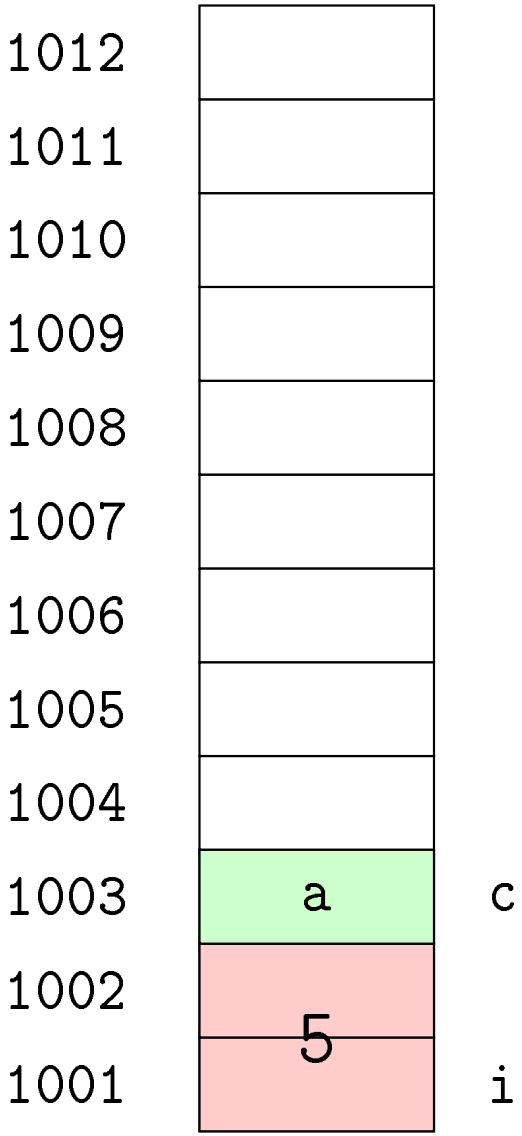

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



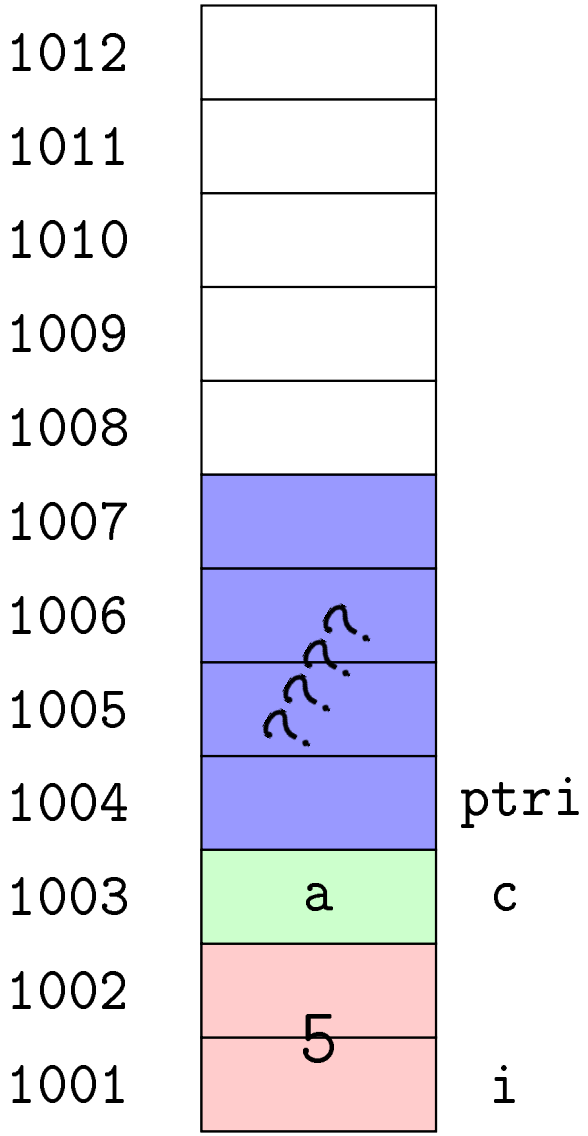

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



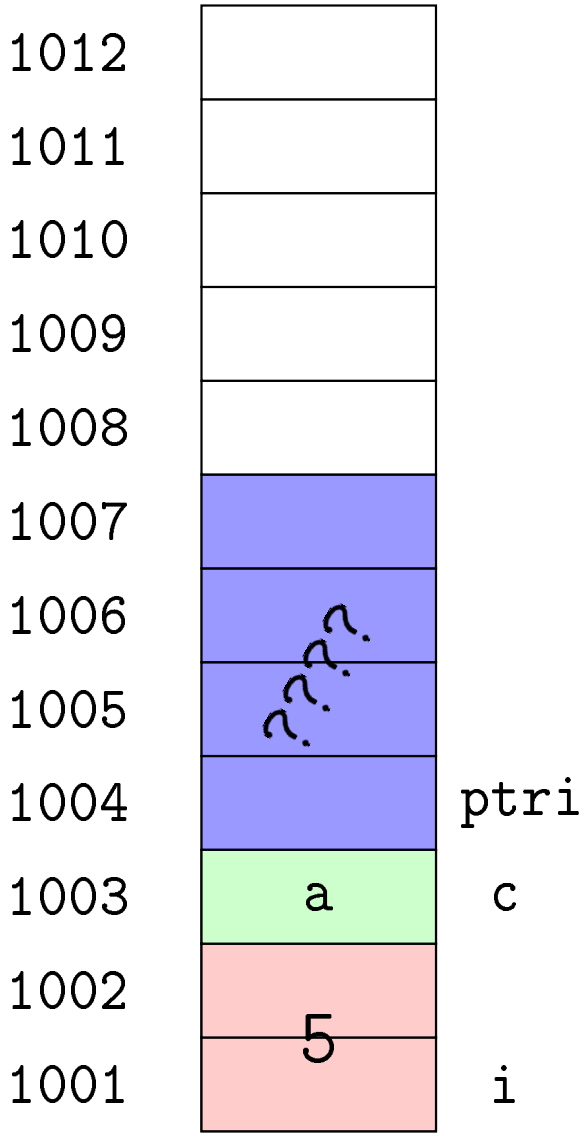

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# Ejemplo: Declaración de punteros



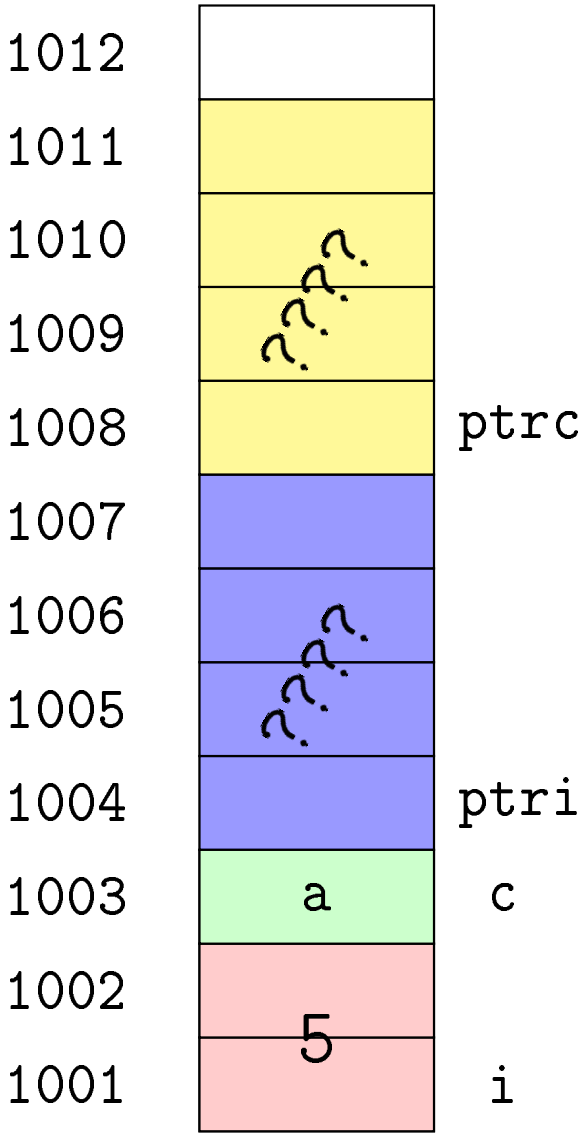

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
```

# 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;
```

# 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;
```

- `int *ptri;`
- `char *ptrc;`

## Puntero a

Al declarar un puntero se debe especificar el tipo al que apunta. El tipo puntero no **es genérico**.

## La variable en memoria

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

## El espacio en memoria

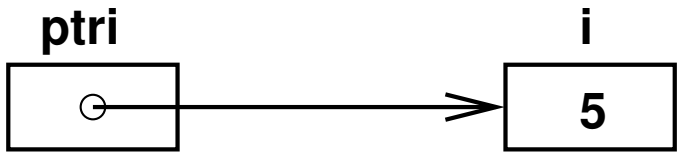
El tamaño de memoria reservado para albergar un puntero-a, es el mismo independientemente del tipo de dato al que pueda apuntar (espacio para albergar una dirección de memoria, 32 ó 64 bits, dependiendo del tipo de procesador usado).

# Indice

# Operador de dirección &

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

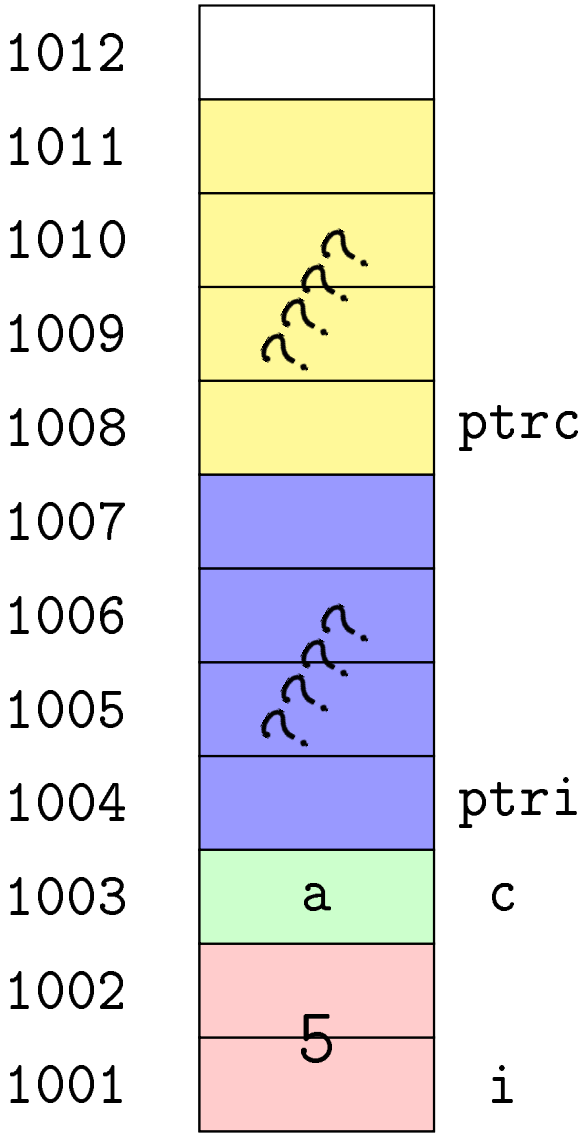
```
int i = 5, *ptri;
ptri = &i;
```




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

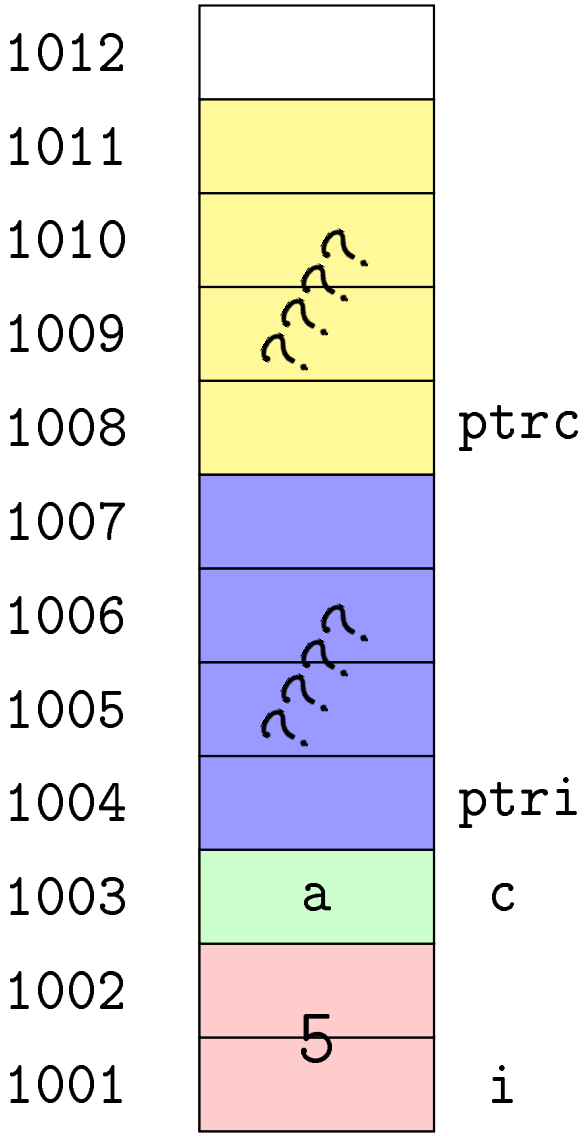
Se dice que ptri *apunta* o *referencia* a i.

# Operador de dirección &



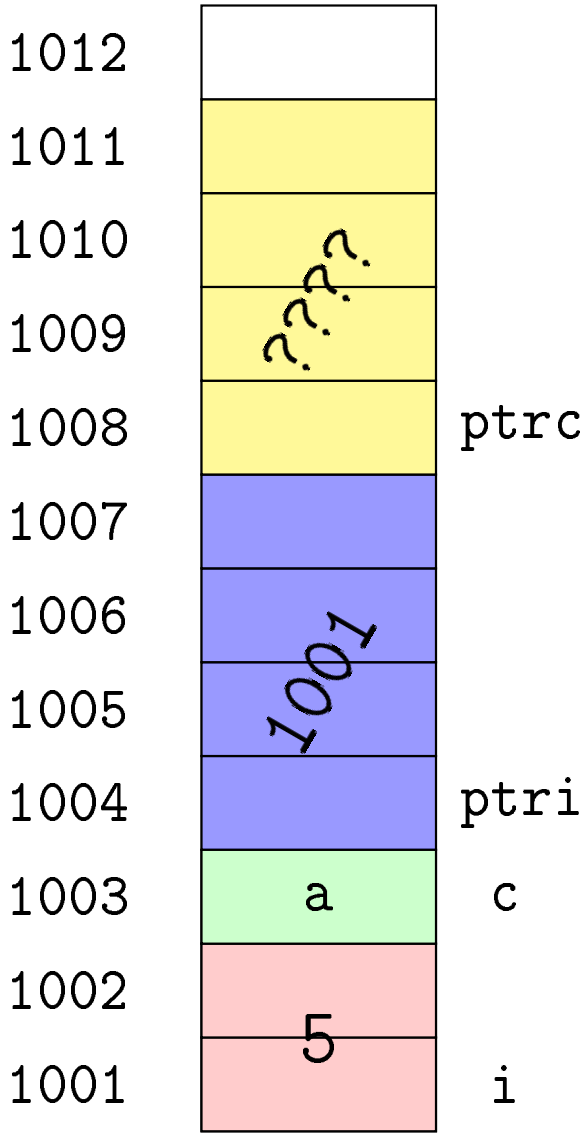

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
```

# Operador de dirección &



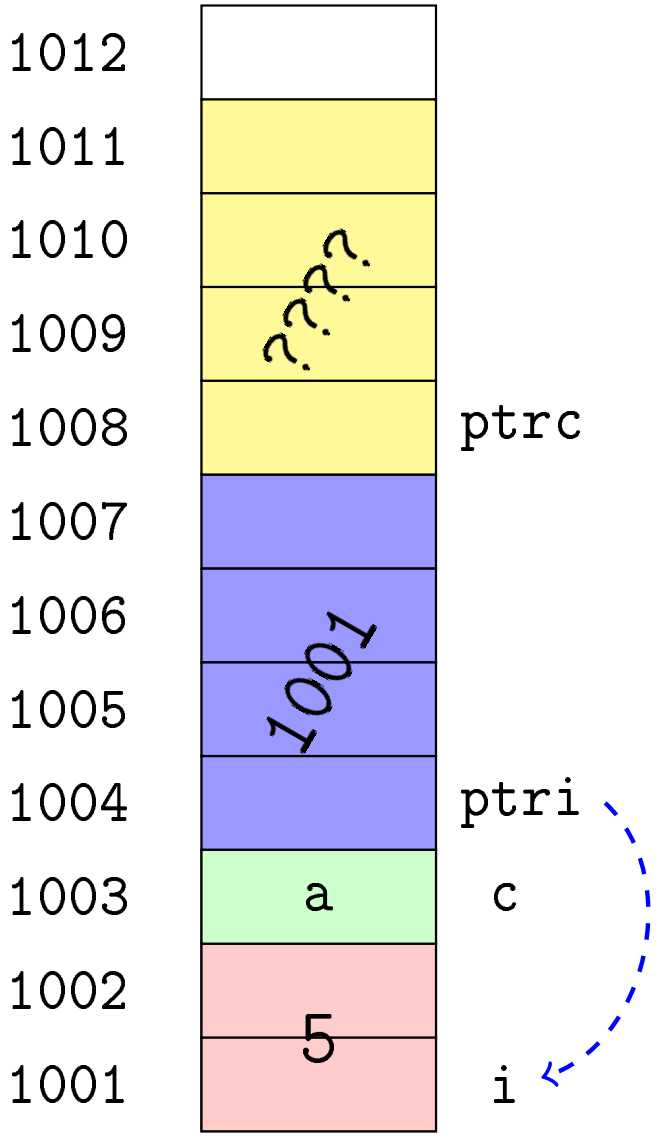

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
```

# Operador de dirección &




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

# Operador de dirección &

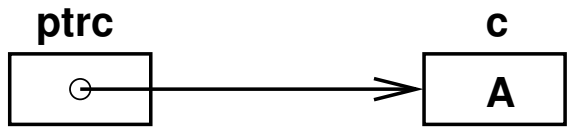



```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
```

# Operador de indirección *

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

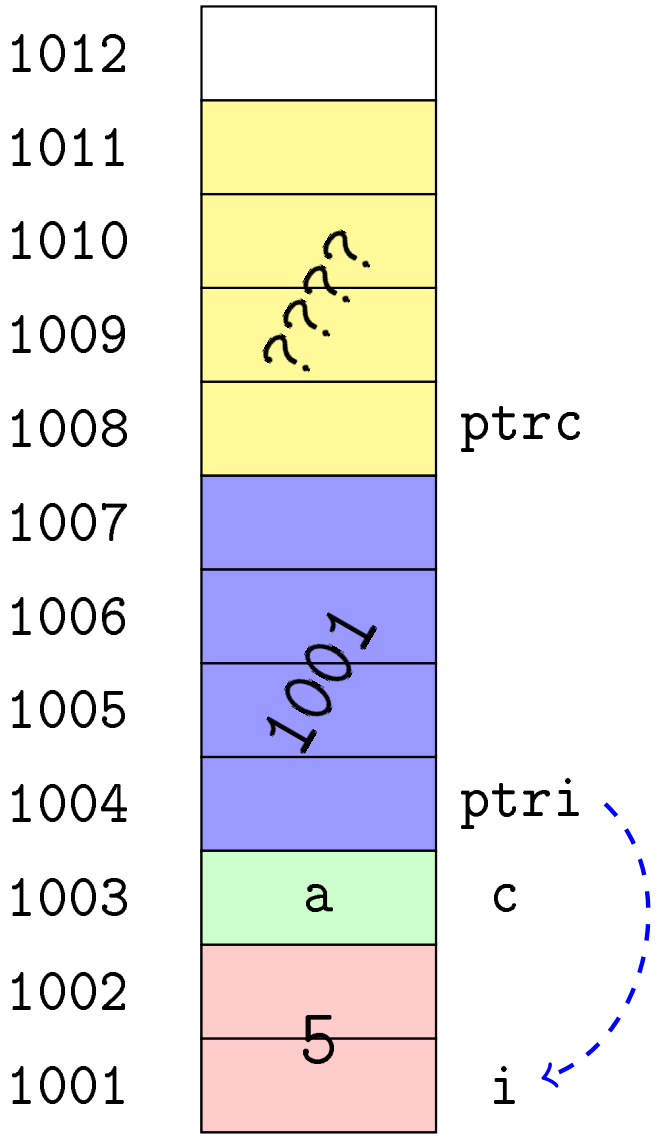
```
char c, *ptrc;
.............
// Hacemos que el puntero apunte a c
ptrc = &c;
// Cambiamos contenido de c mediante ptrc
*ptrc = 'A'; // equivale a c = 'A'
```




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

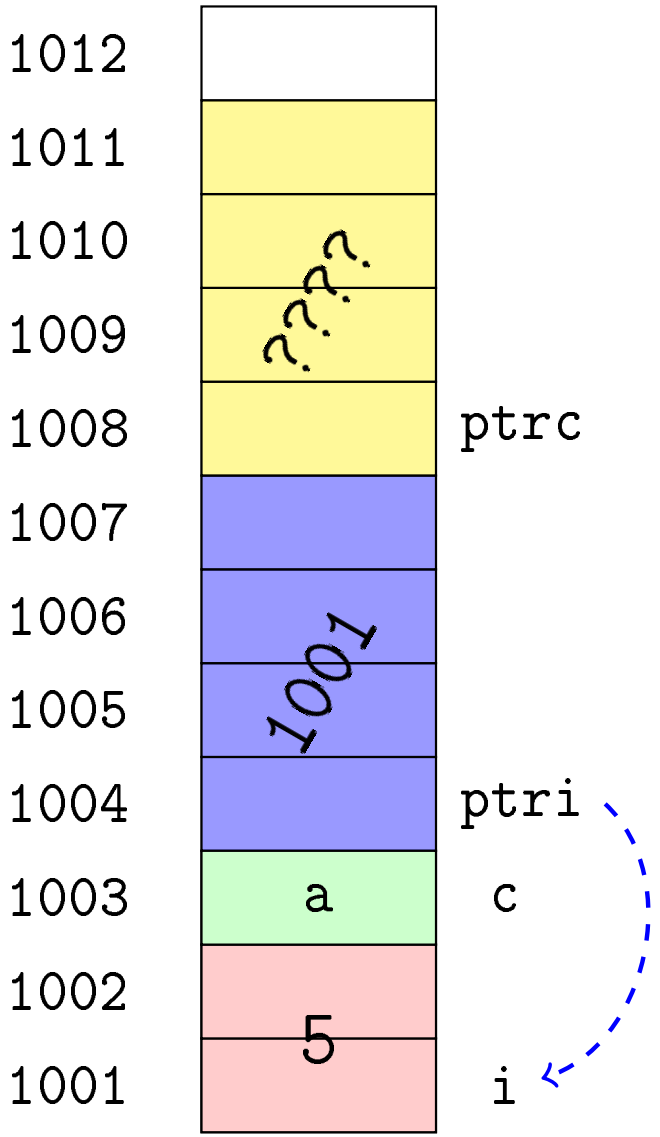
Un puntero contiene una dirección de memoria y se puede interpretar como un número entero aunque un puntero no es un número entero. Existen un conjunto de operadores que se pueden aplicar sobre punteros (como veremos más adelante): $+, -, ++, --, !=, ==$

# Operador de indirección *



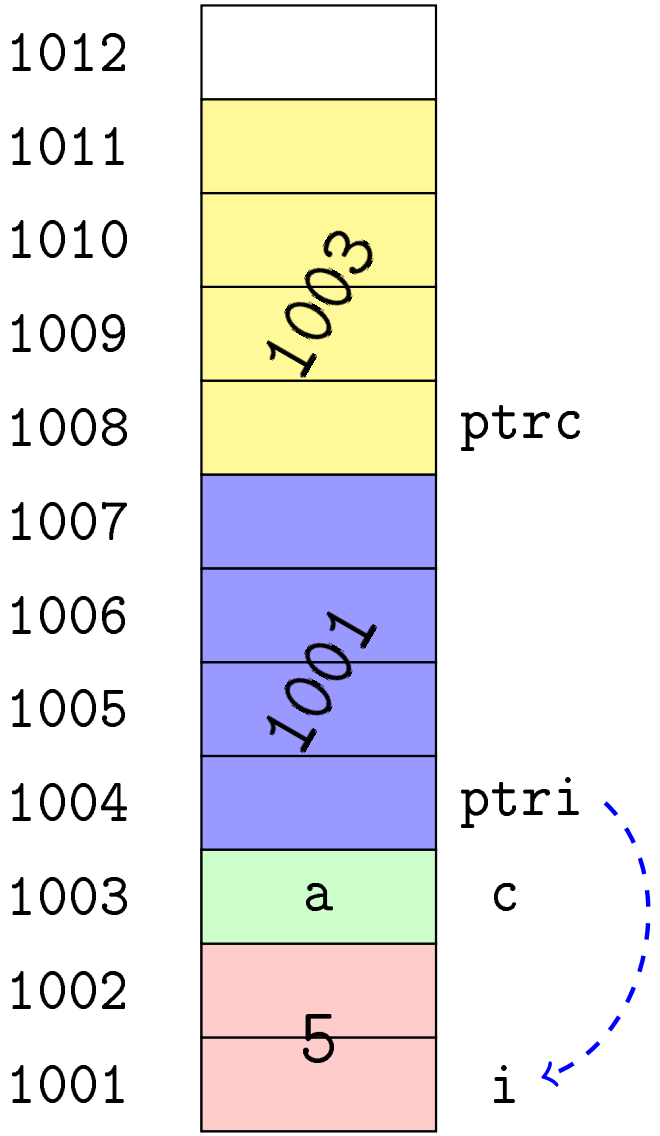

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# Operador de indirección *



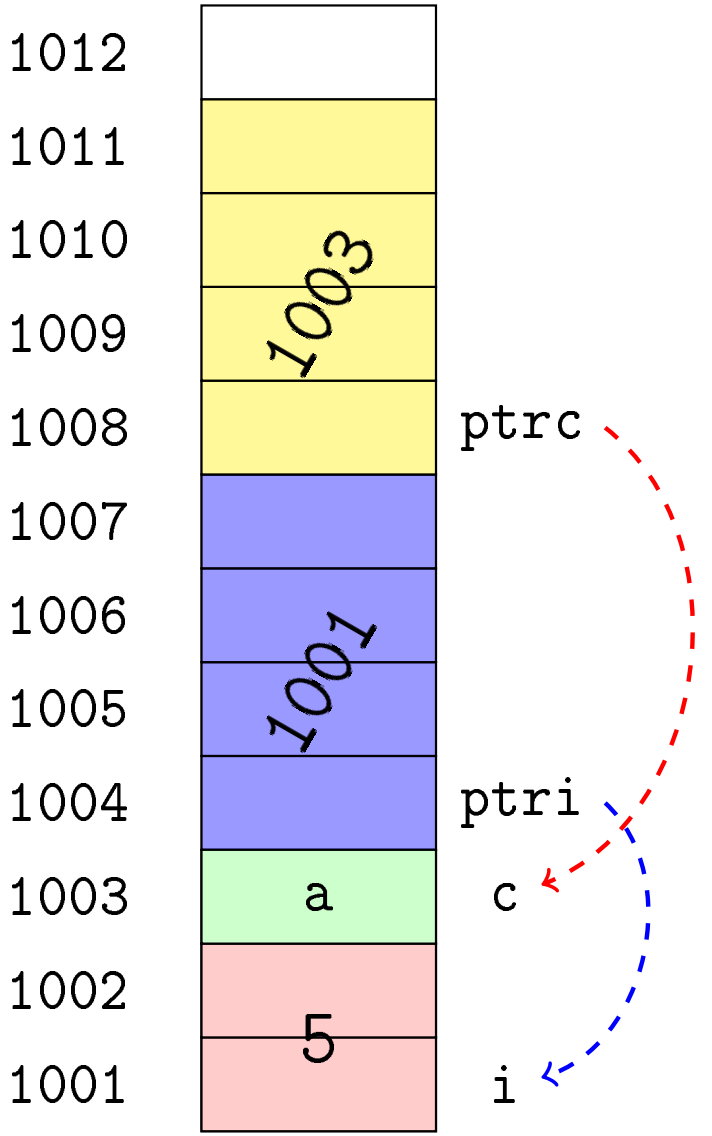

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# Operador de indirección *



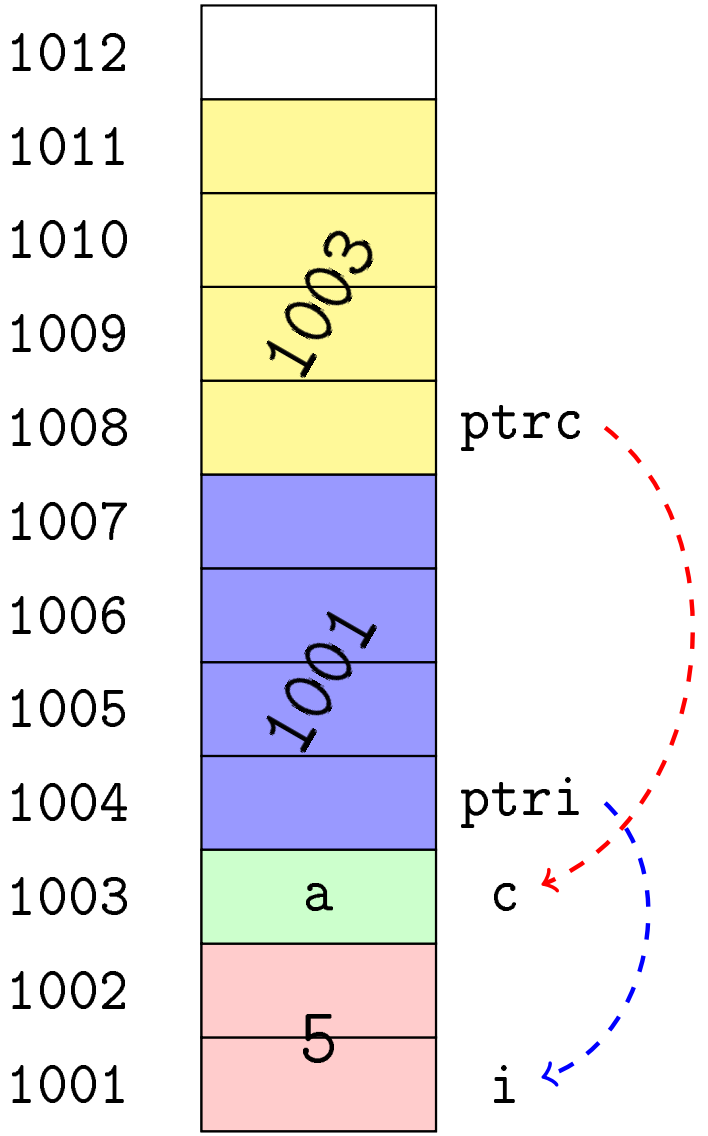

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# Operador de indirección *



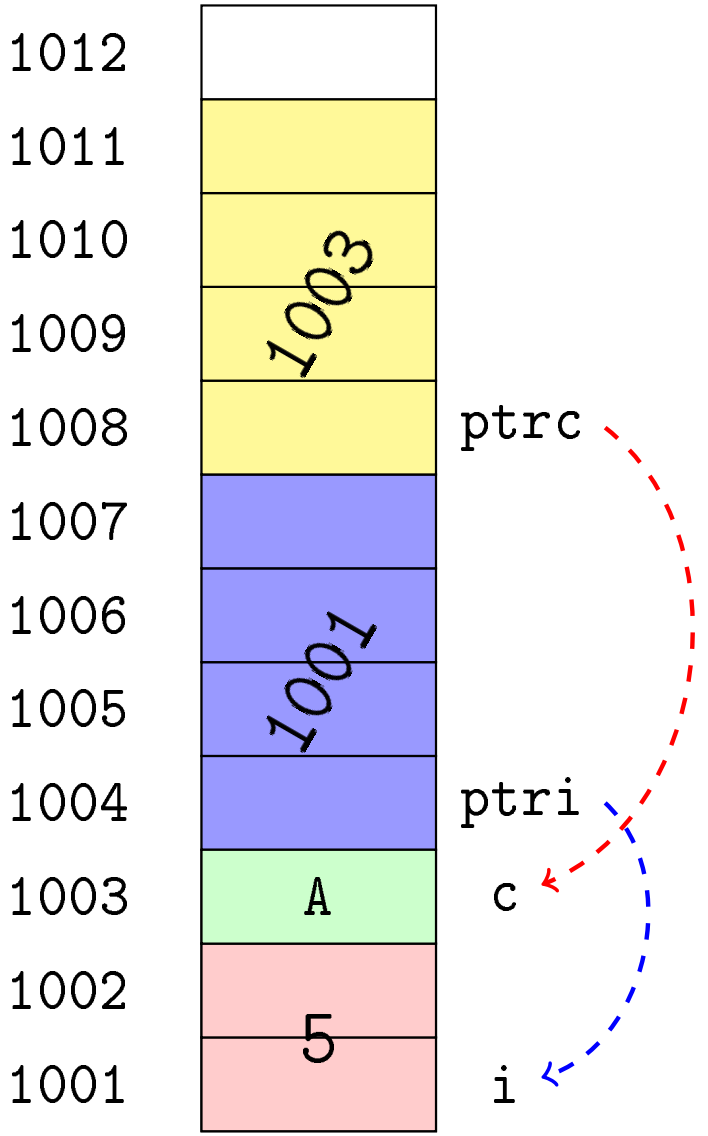

```
// Se declara la variable de tipo entero
int i=5;
// Se declara la variable de tipo char
char c='a';
// Se declara puntero a entero
int * ptri;
// Se declara el puntero a char
char * ptrc;
// ptri apunta a la variable i
ptri=&i;
// ptrc apunta a c
ptrc=&c;
//cambia contenido con ptrc
*ptrc='A';
```

# Operador de indirección *




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

# Operador de indirección *




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

# Operador de asignación, =

## Inicialización

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

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

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

```
int *ptri = nullptr;
```

# Operador de asignación, =

## Asignación

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

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

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

# Asignación e inicialización de punteros

- Un puntero debe estar correctamente inicializado antes de usarse

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

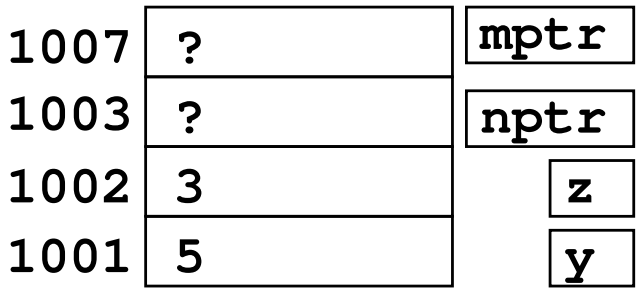
```
Violación de segmento (`core' generado)
```

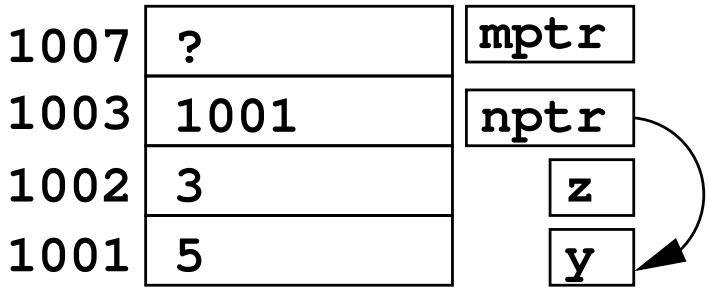
- Es conveniente inicializar los punteros en la declaración, con el puntero nulo, con la constante `nullptr`.

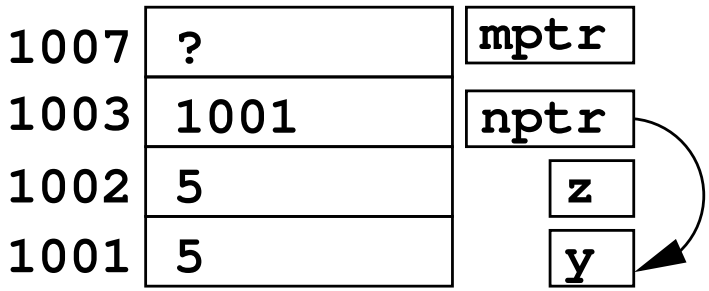
```
int *p2 = nullptr;
```

# Ejemplo

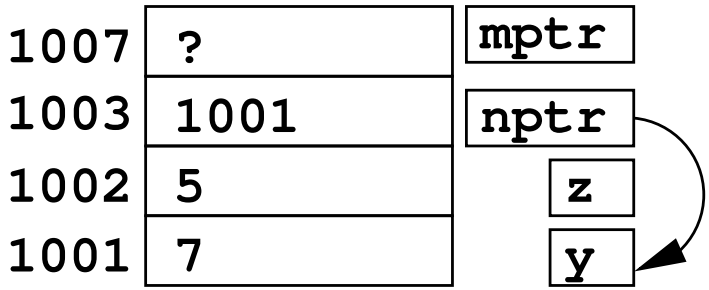
```
int main() {
  int y = 5, z = 3;
  int *nptr;
  int *mptr;

  nptr = &y;

  z = *nptr;
```



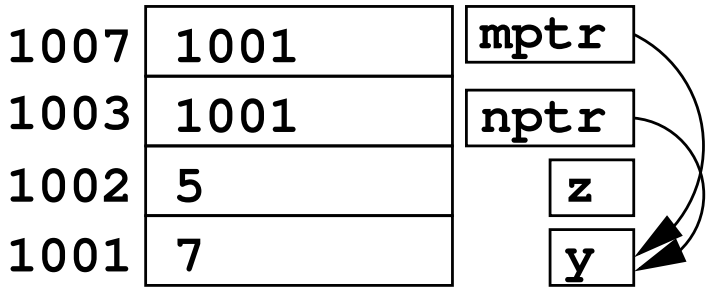



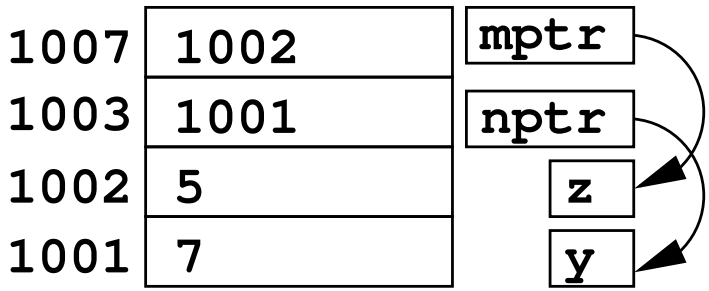
```
*nptr = 7;
```



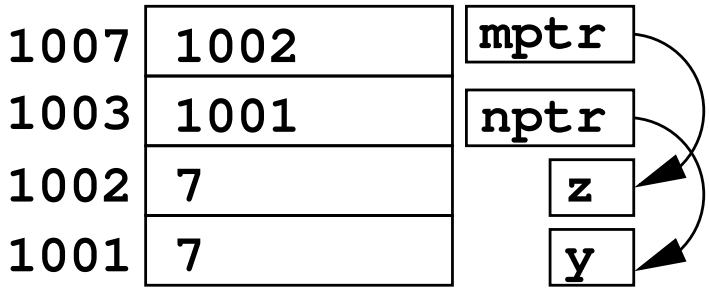

```
mptr = nptr;
```



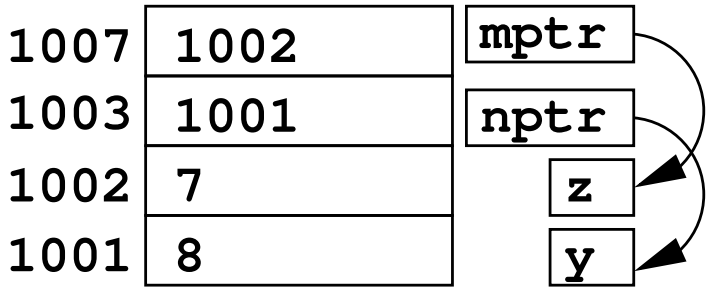

```
mptr = &z;
```



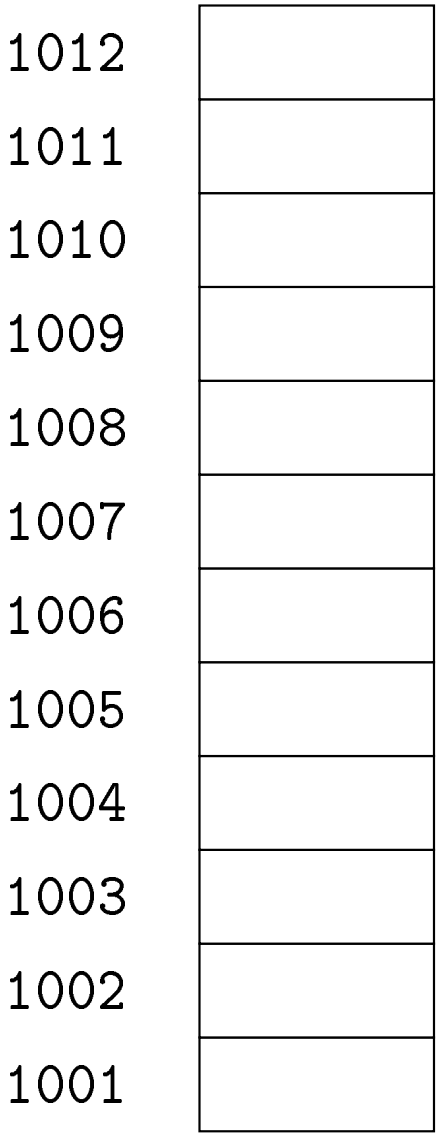
```
   *mptr = *nptr;
```




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

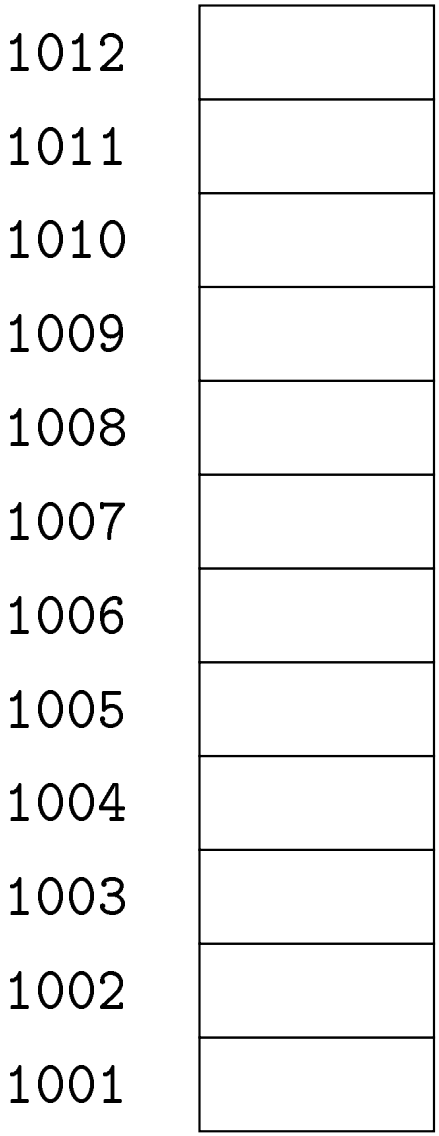


# Ejemplo anterior animado



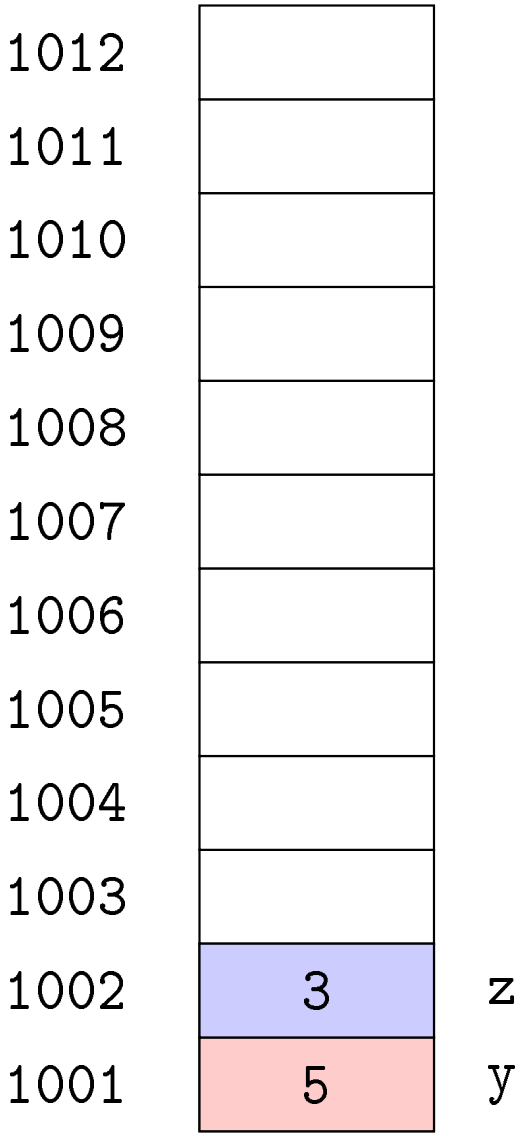

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



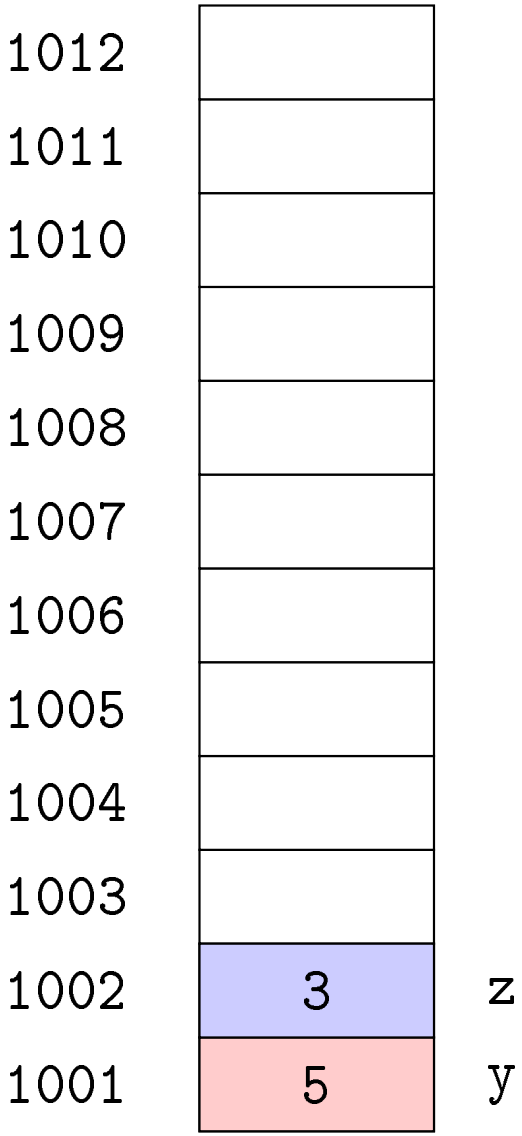

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



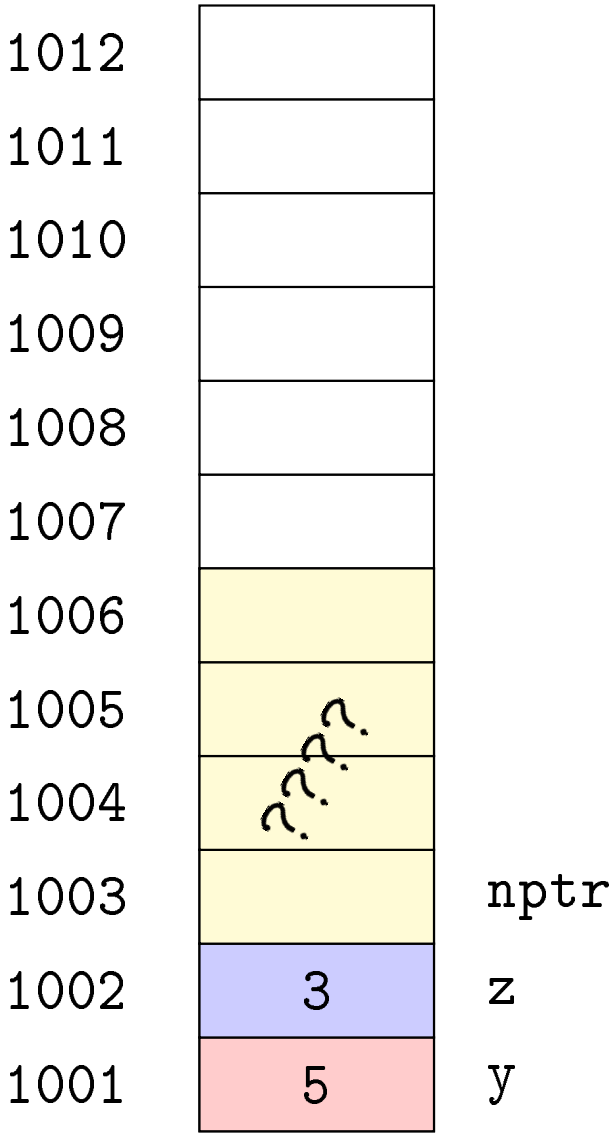

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



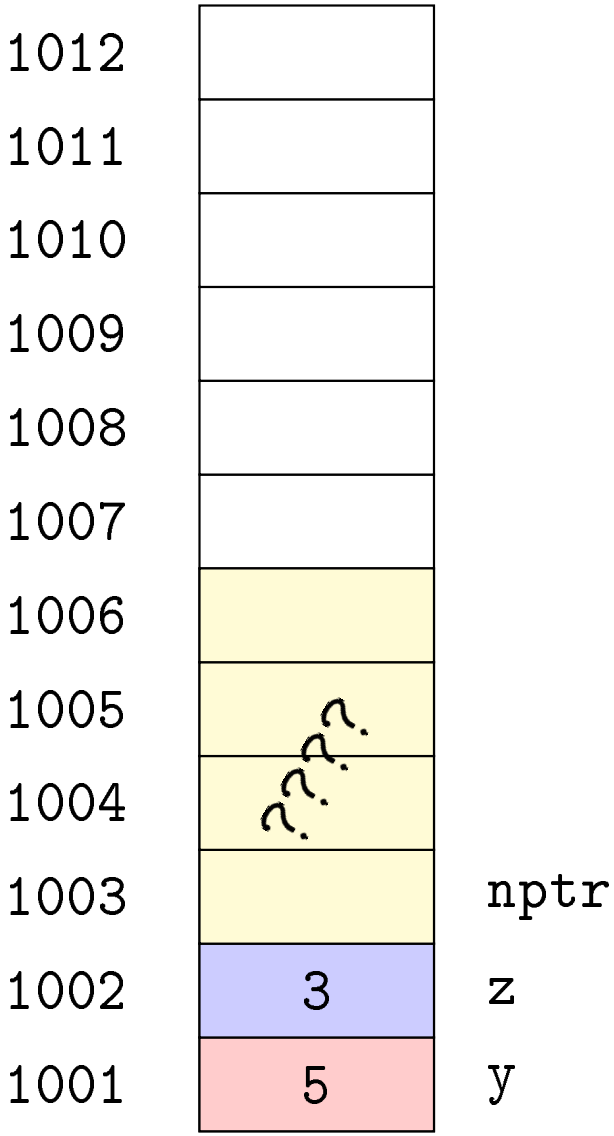

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



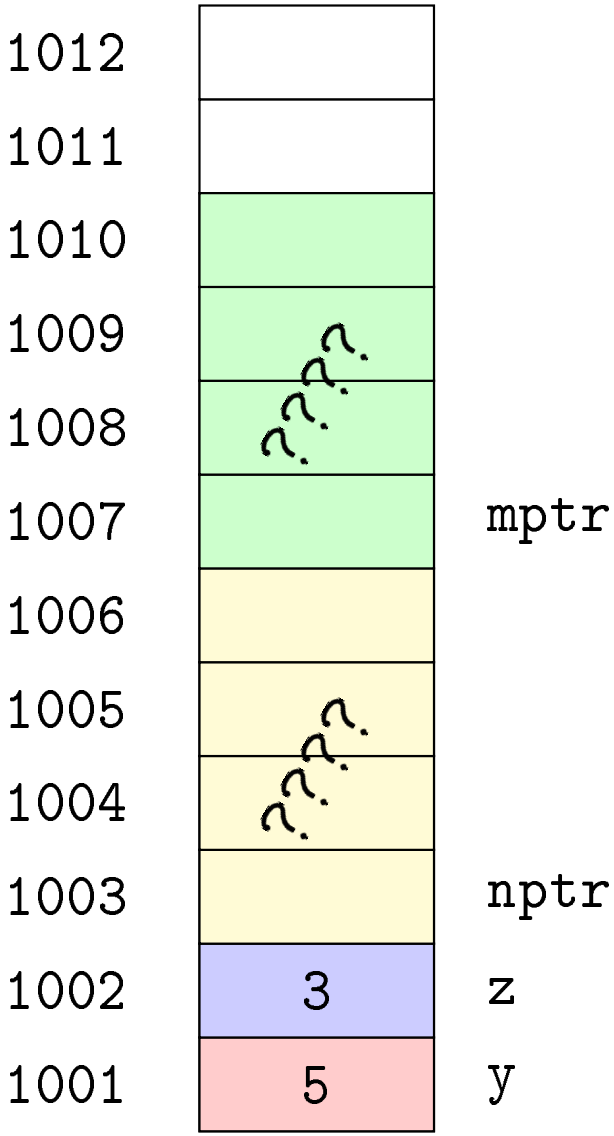

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



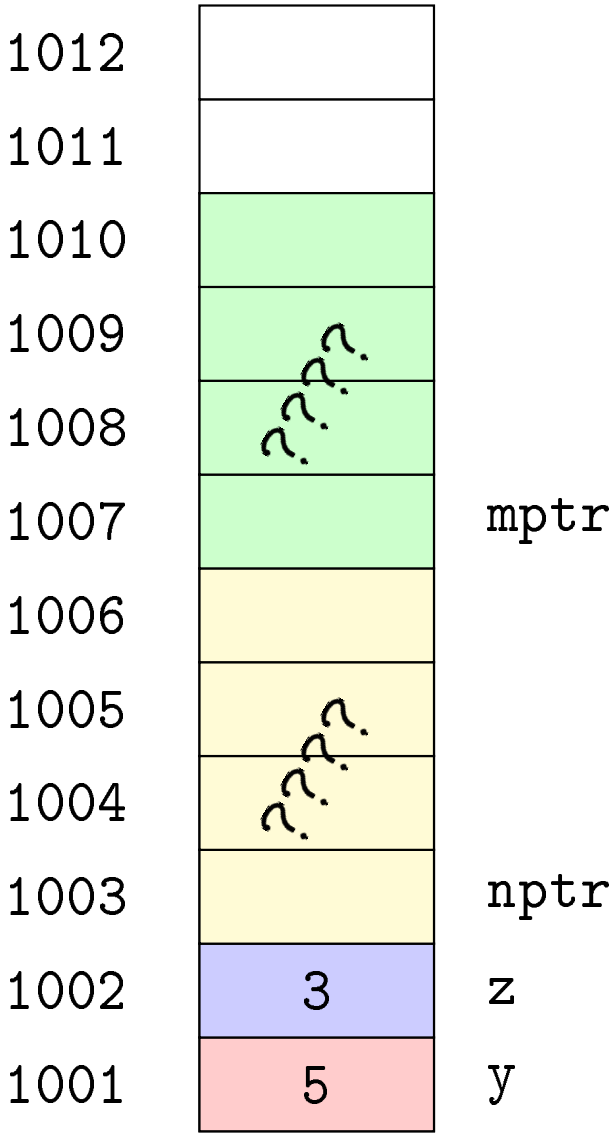

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



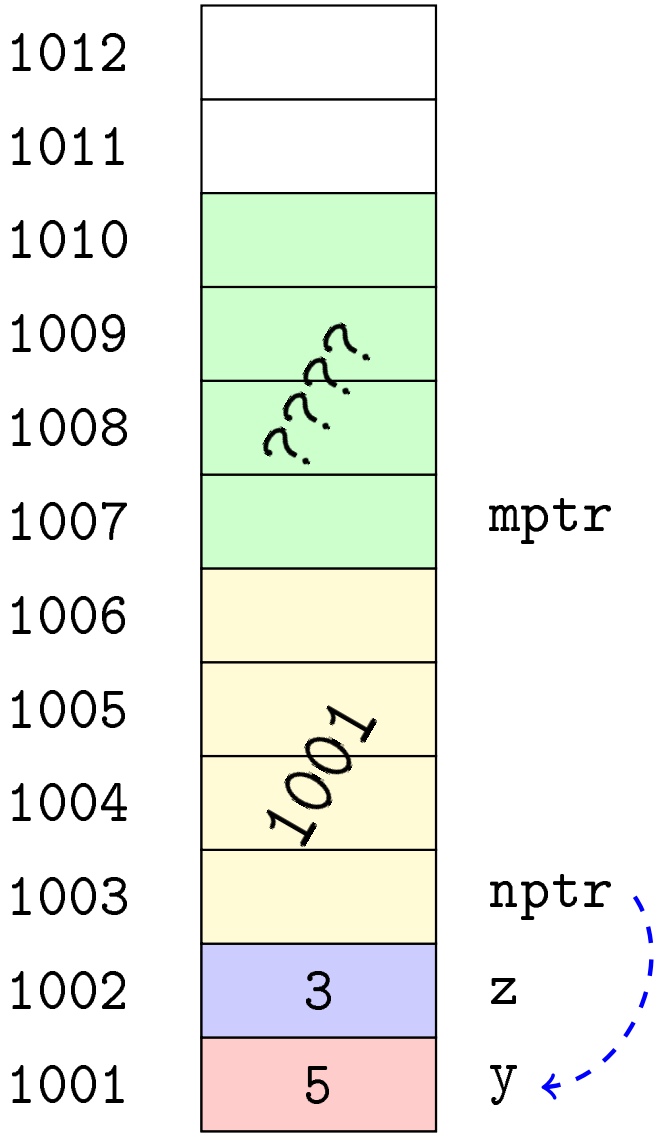

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



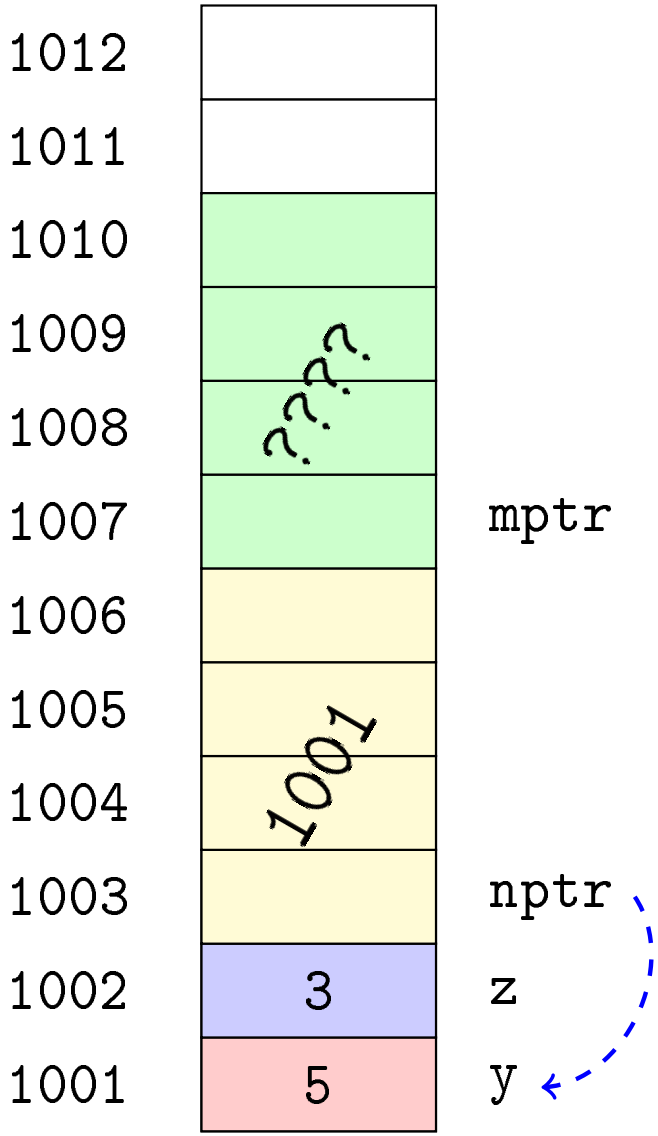

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



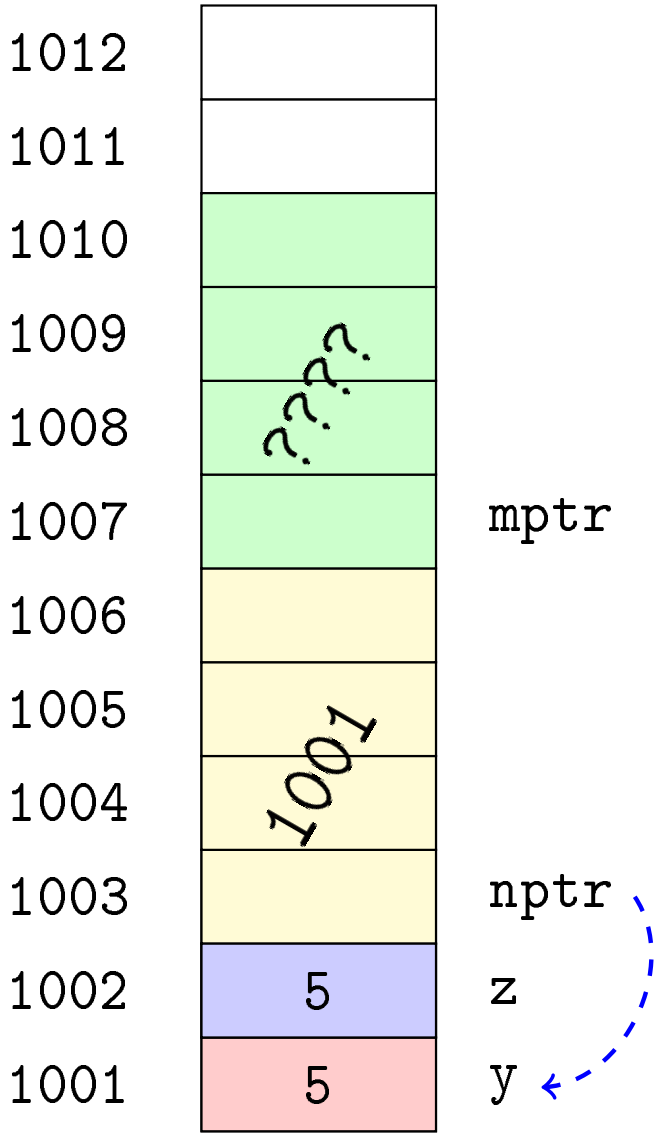

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



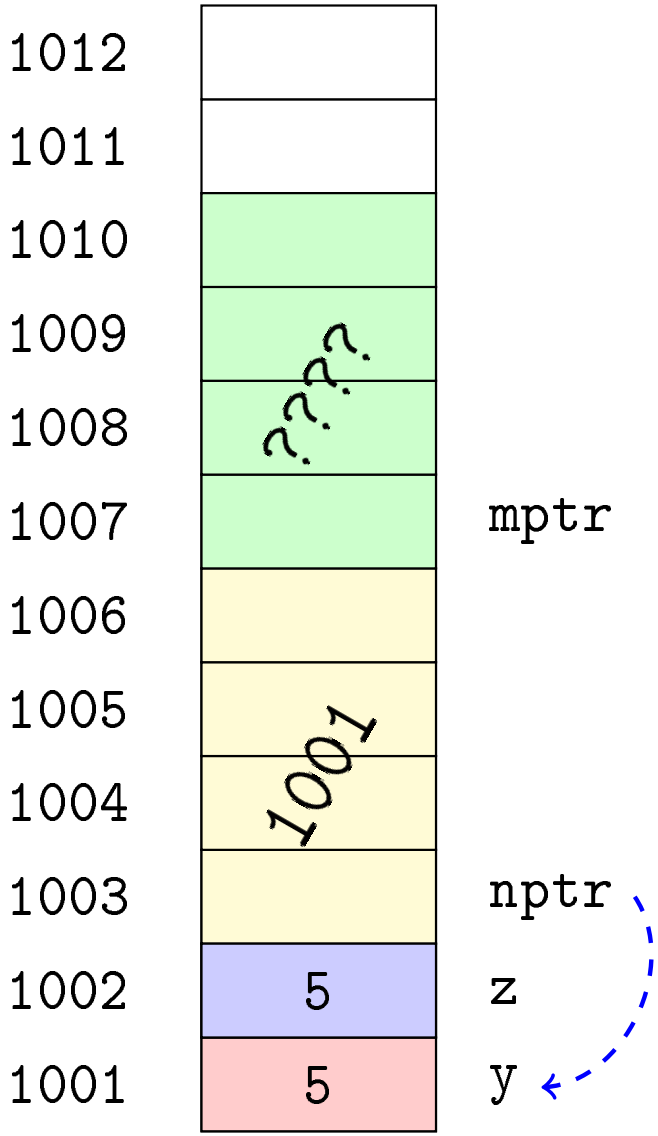

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



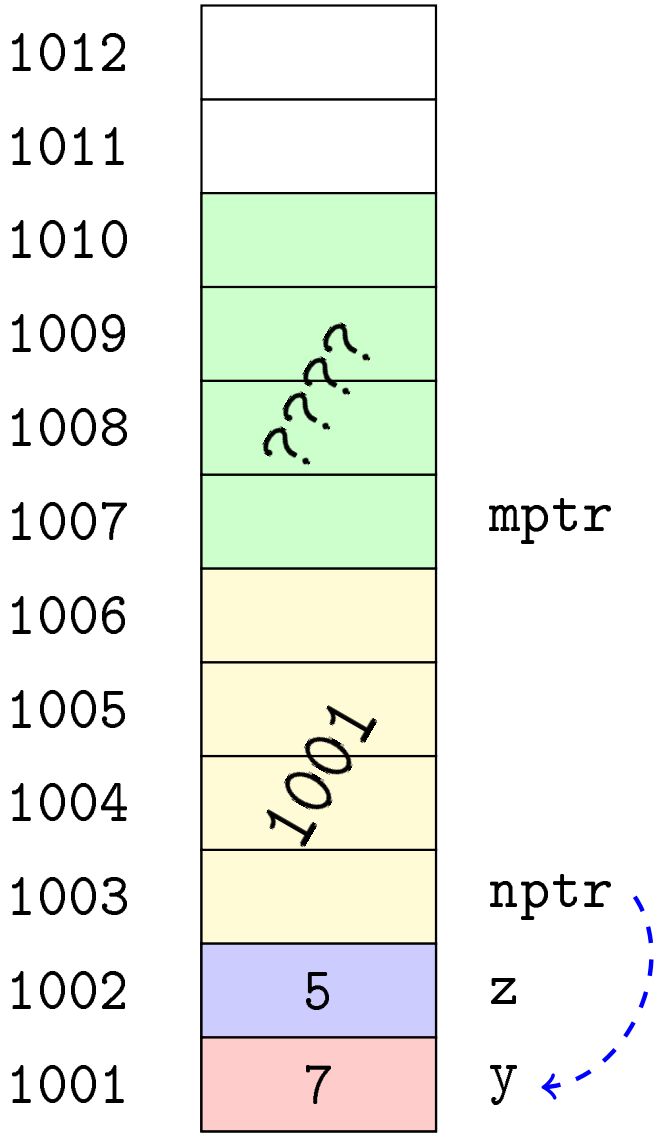

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



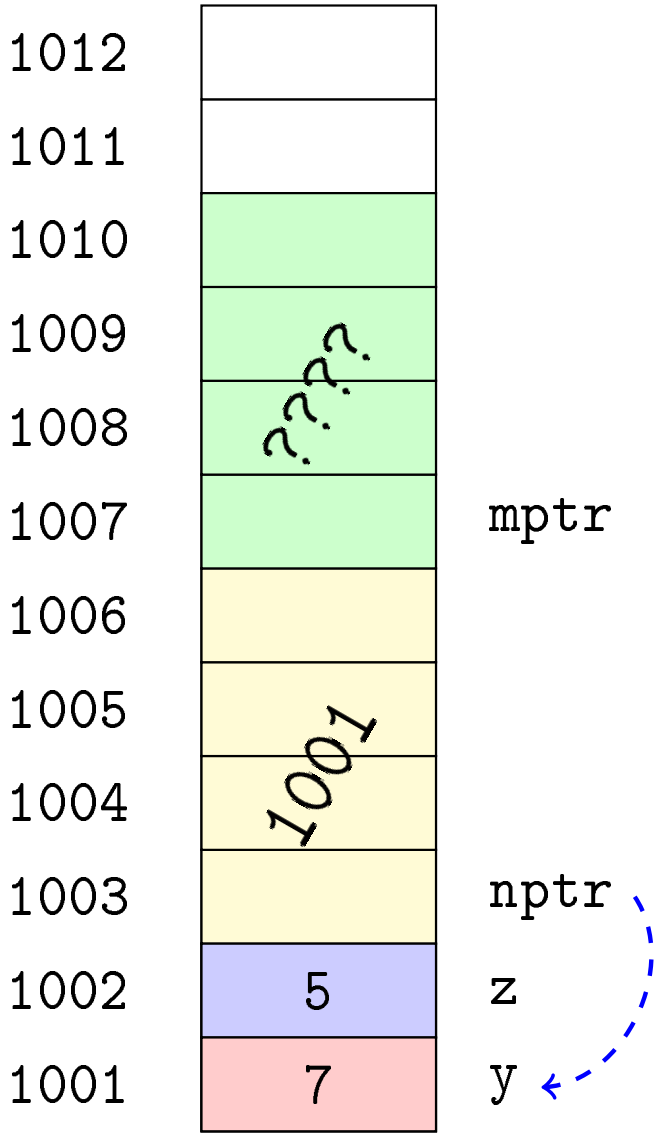

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



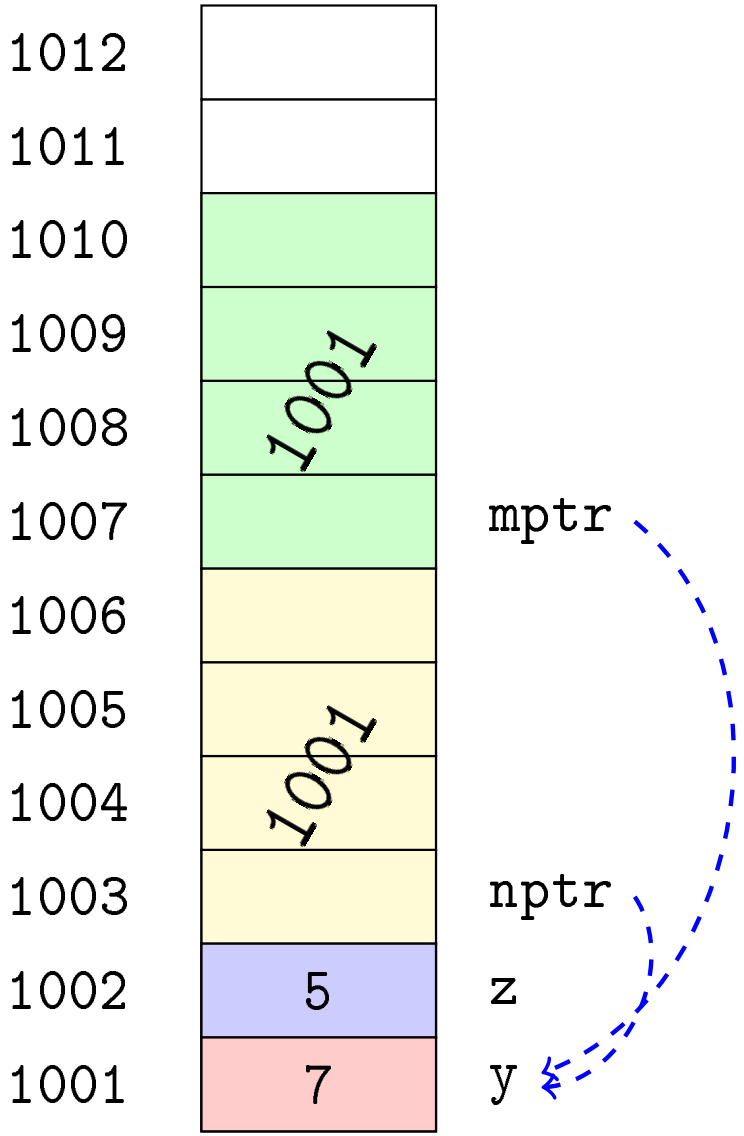

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



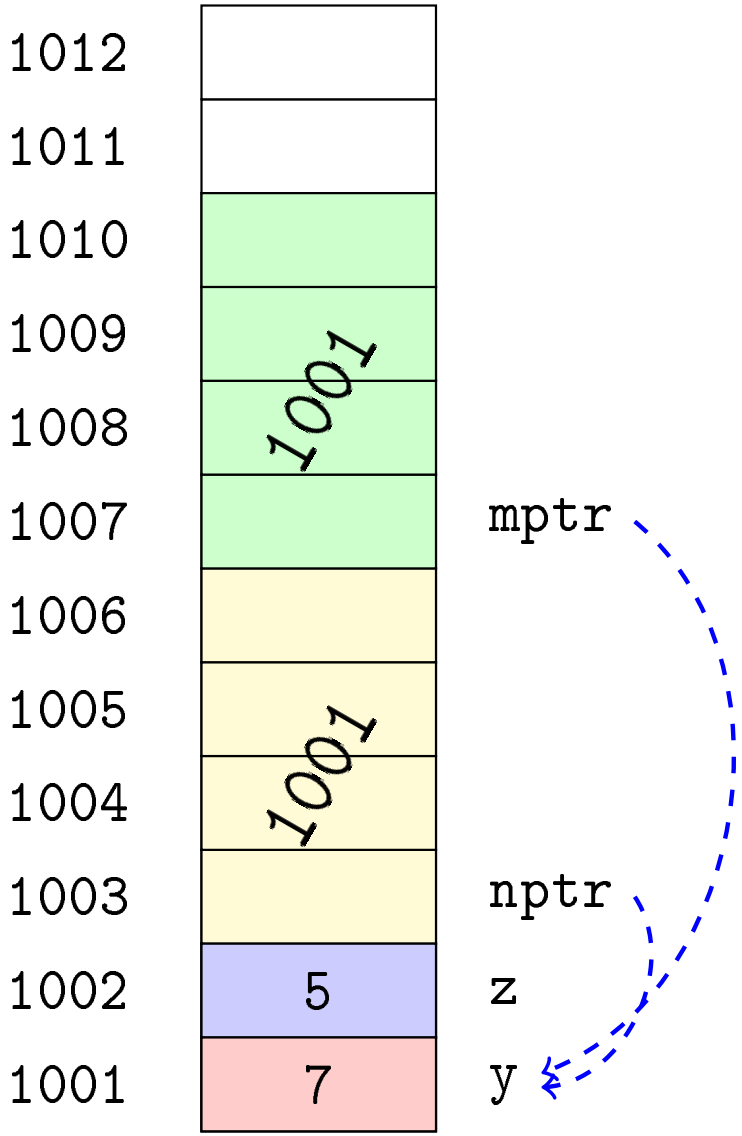

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



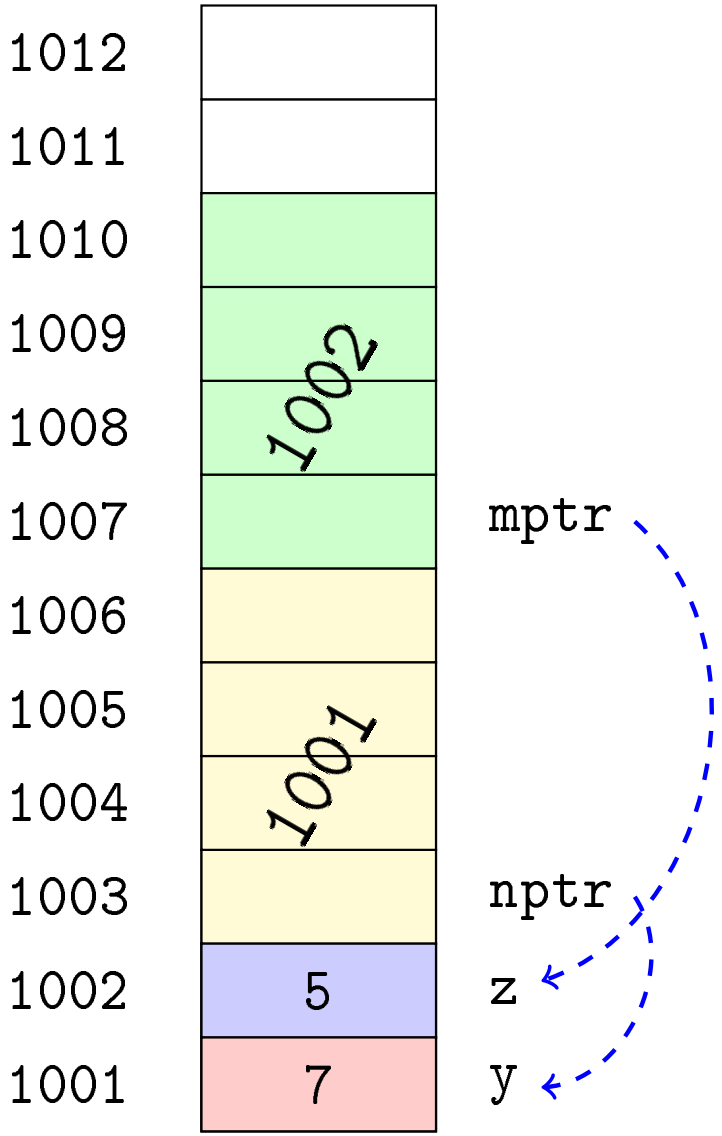

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



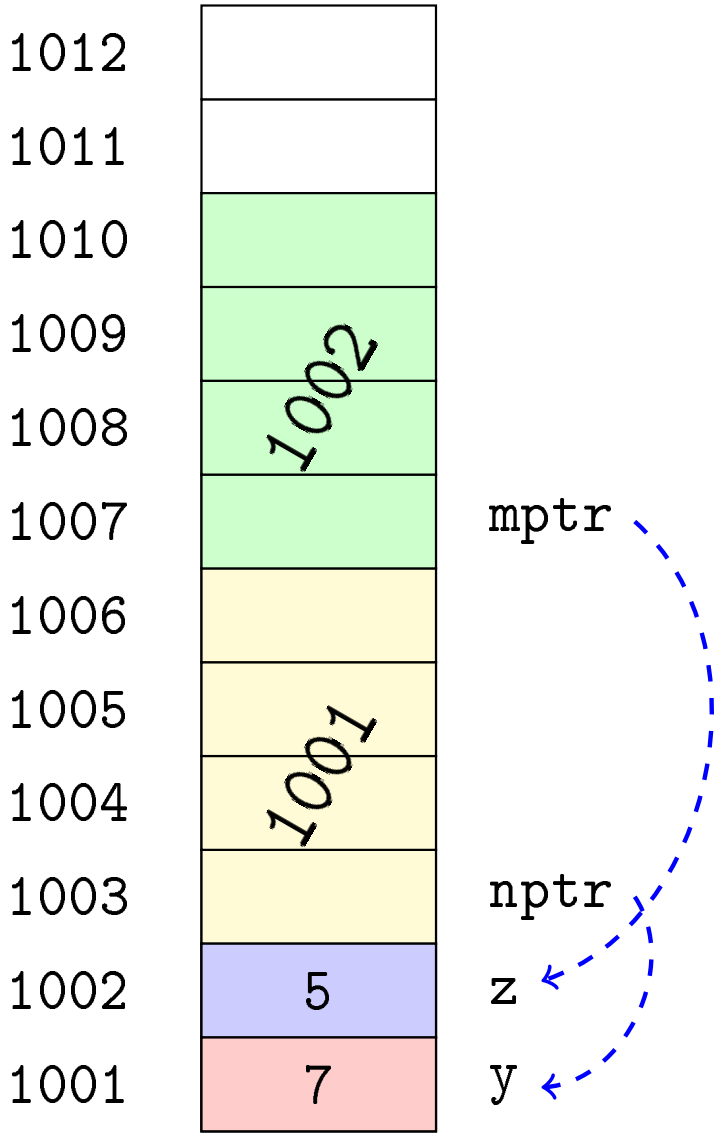

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



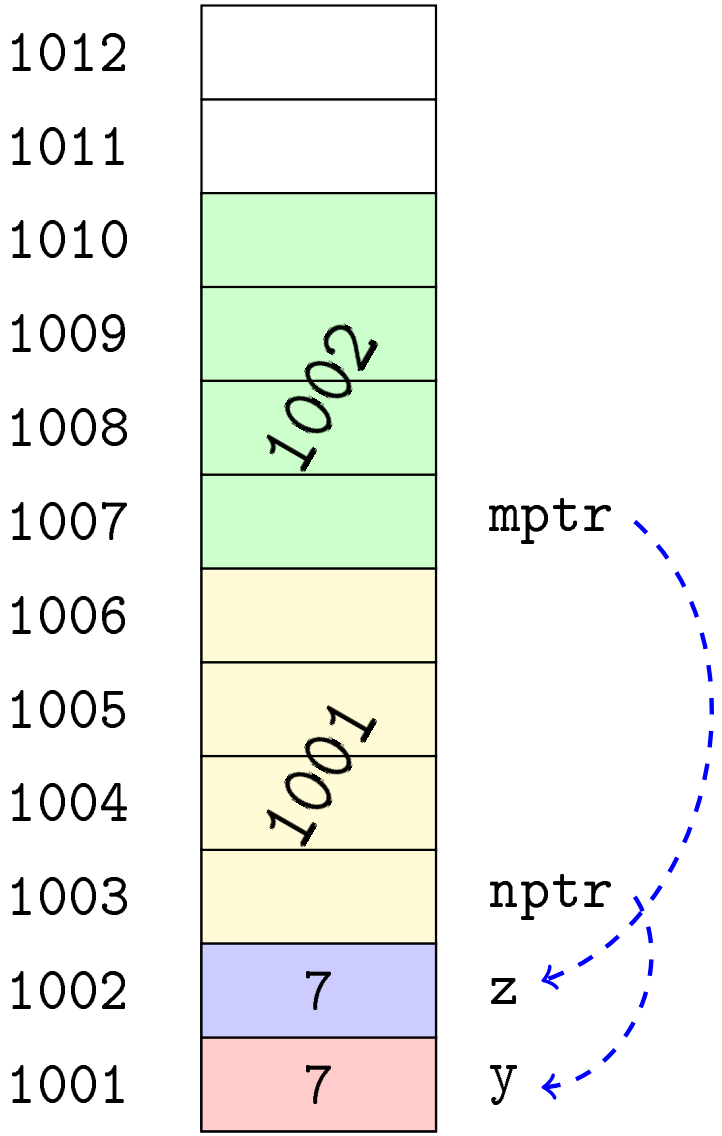

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



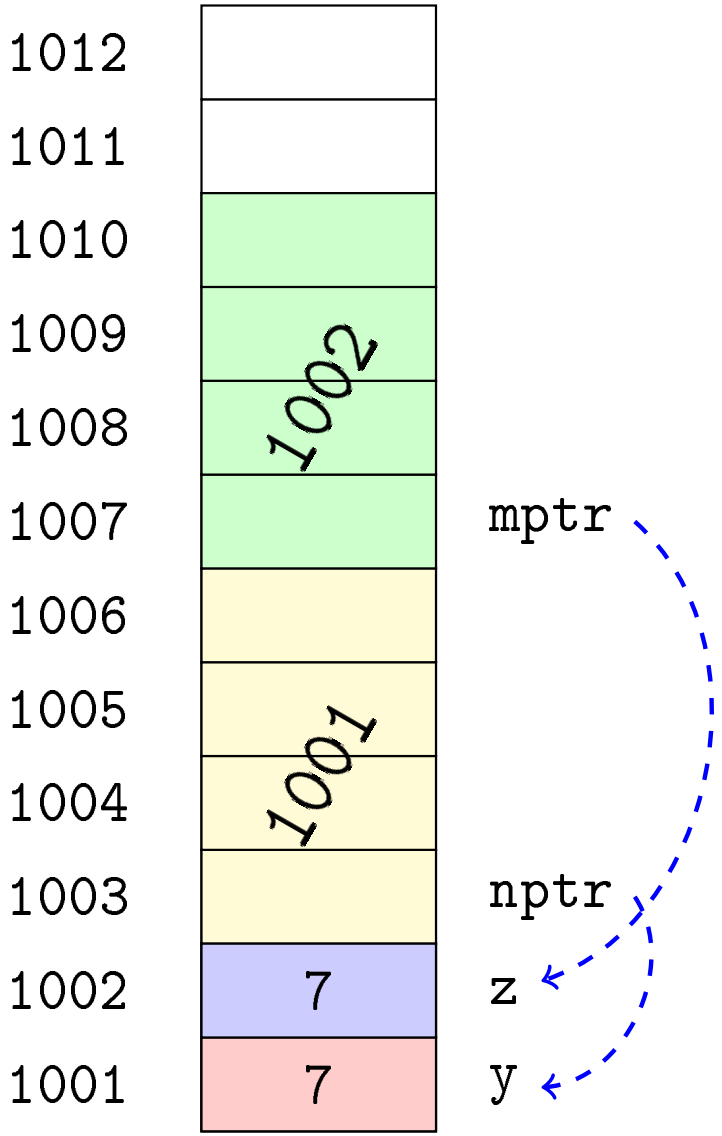

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado



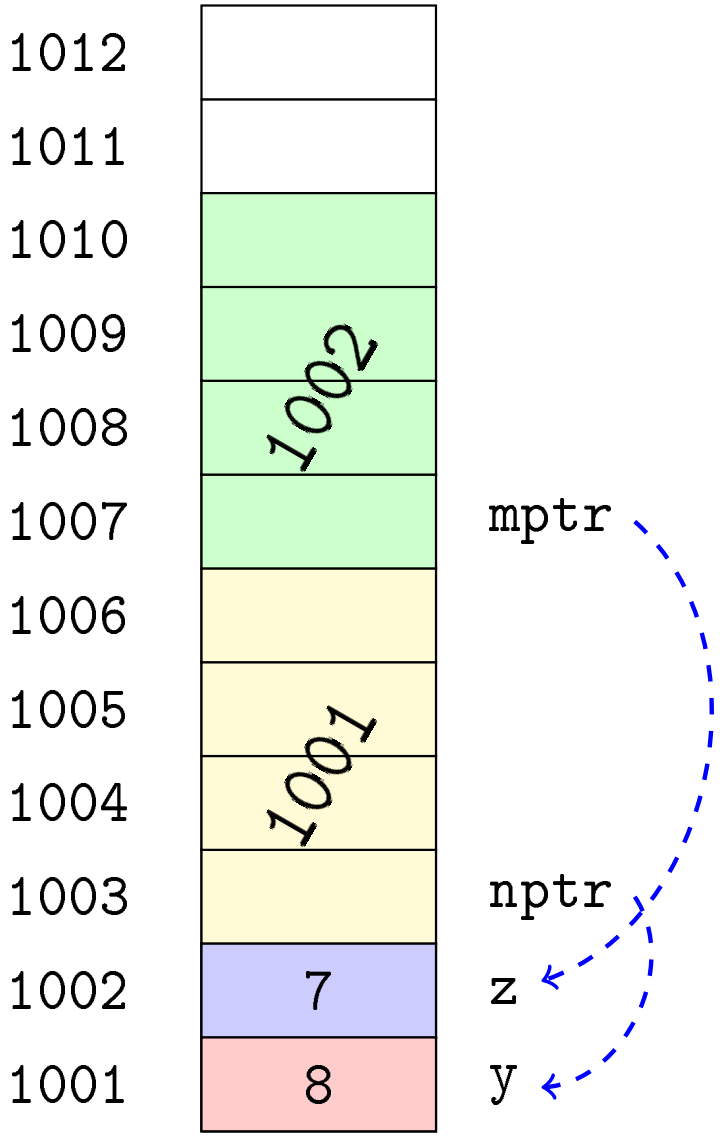

```
char y = 5, z = 3;
char * nptr;
char * mptr;
nptr = &y;
z = *nptr;
*nptr=7;
mptr = nptr;
mptr = &z;
*mptr = *nptr;
y = (*mptr)+1;
```

# Ejemplo anterior animado




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

# Ejemplo anterior animado




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

# Operadores relacionales

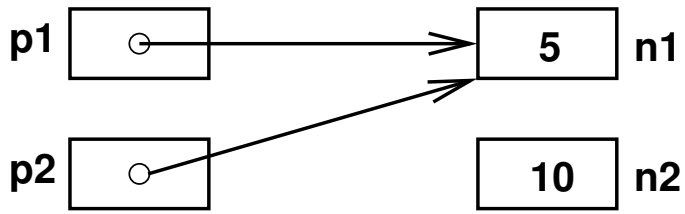
- Los operadores <, >, <=, >=, !=, == son aplicables a punteros.
- El valor del puntero (la dirección que almacena) se comporta como un número entero.

Operadores != y ==

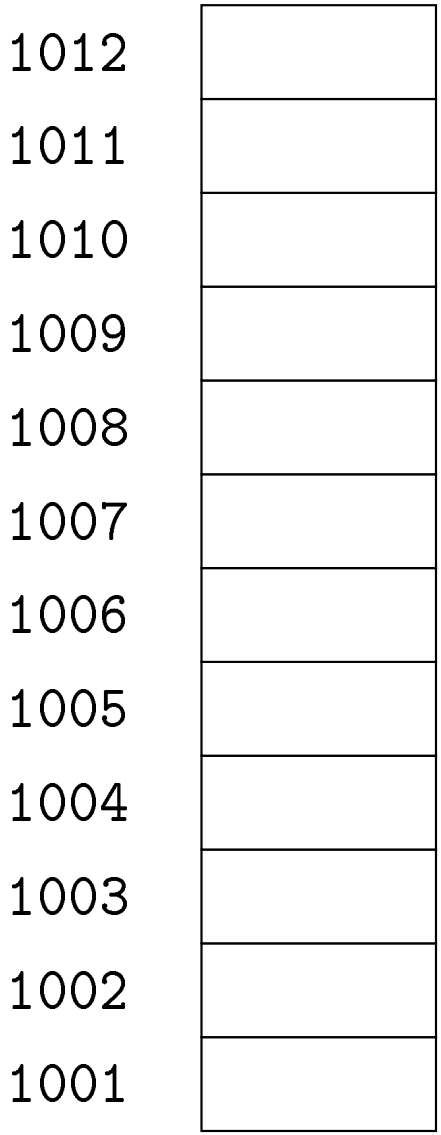
- $p1 == p2$: comprueba si ambos punteros apuntan a la **misma dirección** de memoria (ambas variables guardan como valor la misma dirección)
- $*p1 == *p2$: comprueba si coinciden **los dos datos** apuntados por ambos punteros

# 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";
```

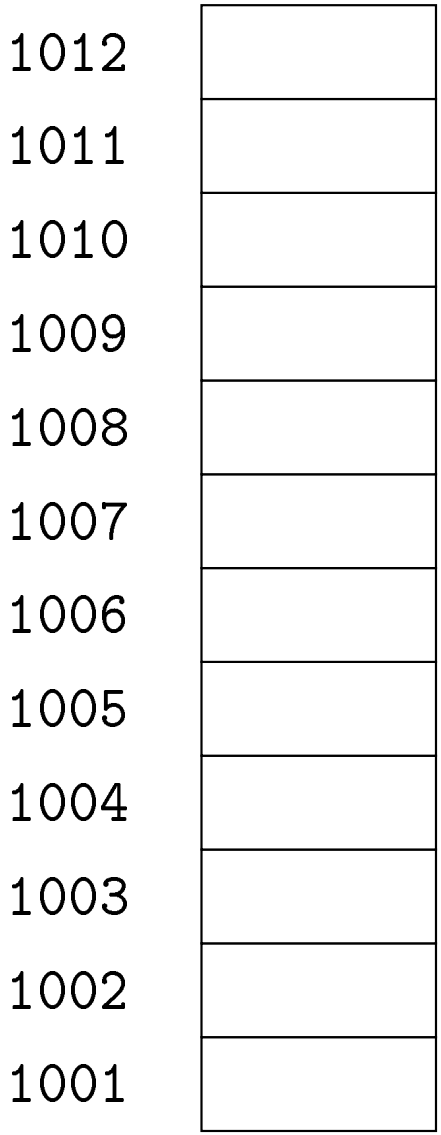


# Operadores relacionales: Ejemplo anterior animado



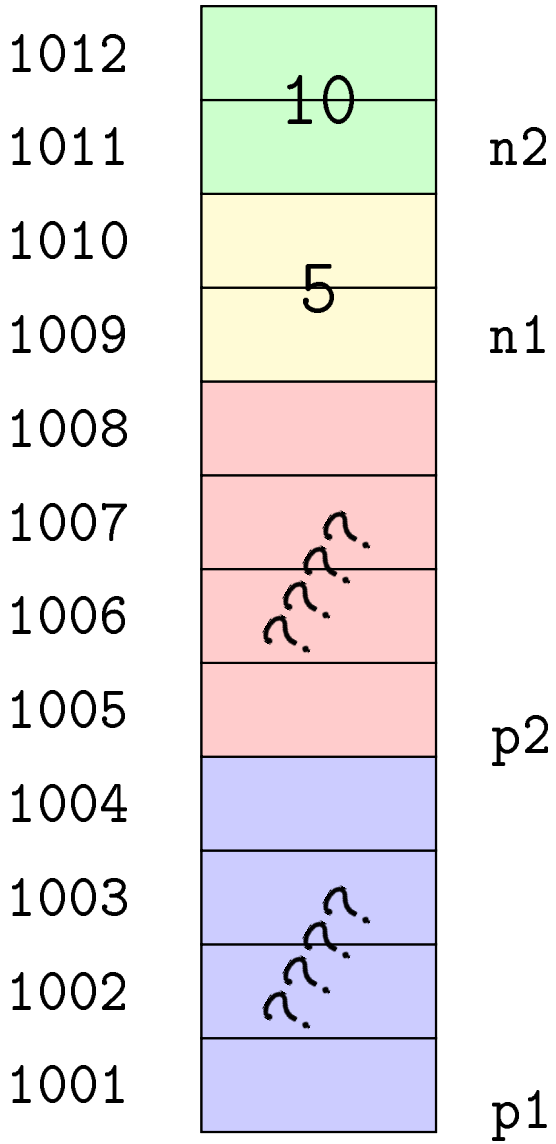

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



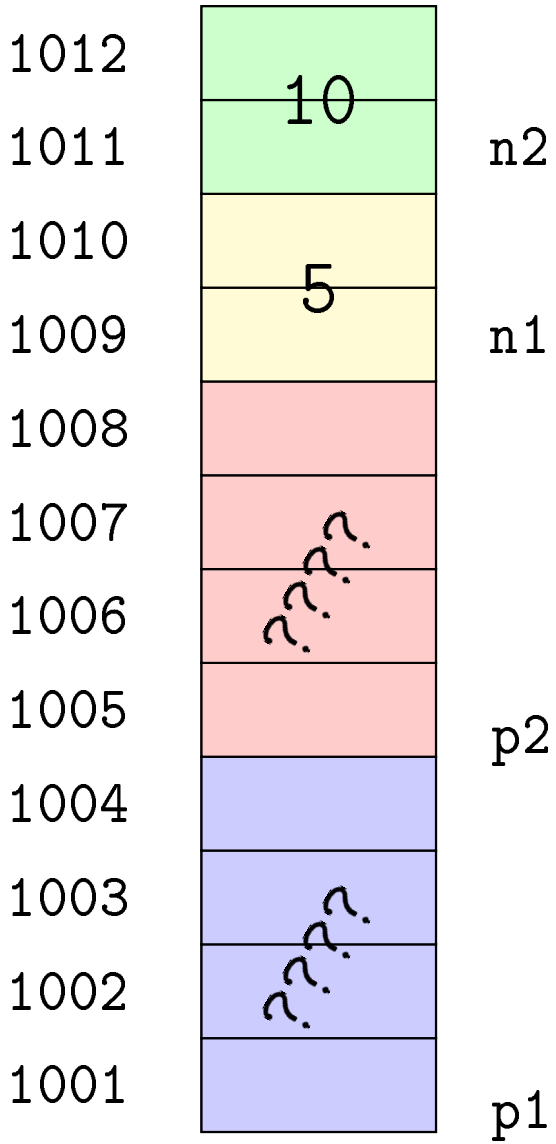

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



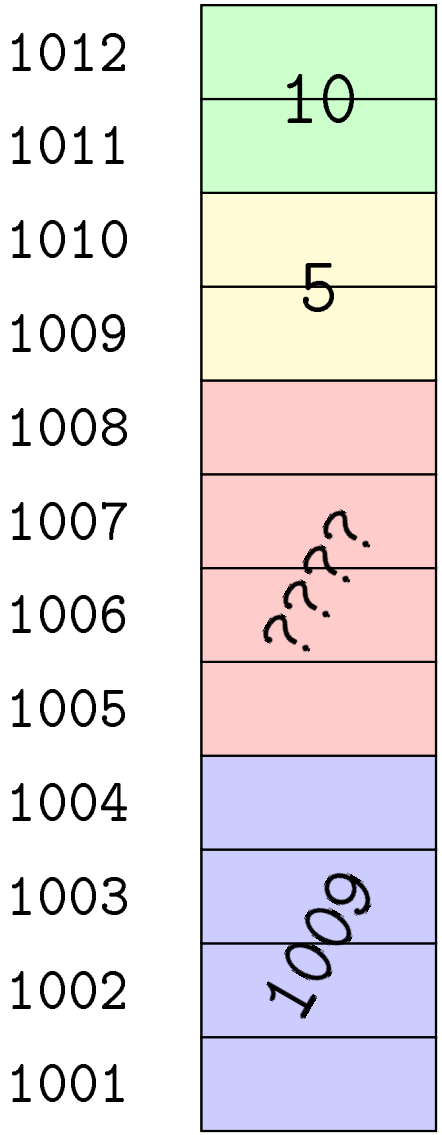

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



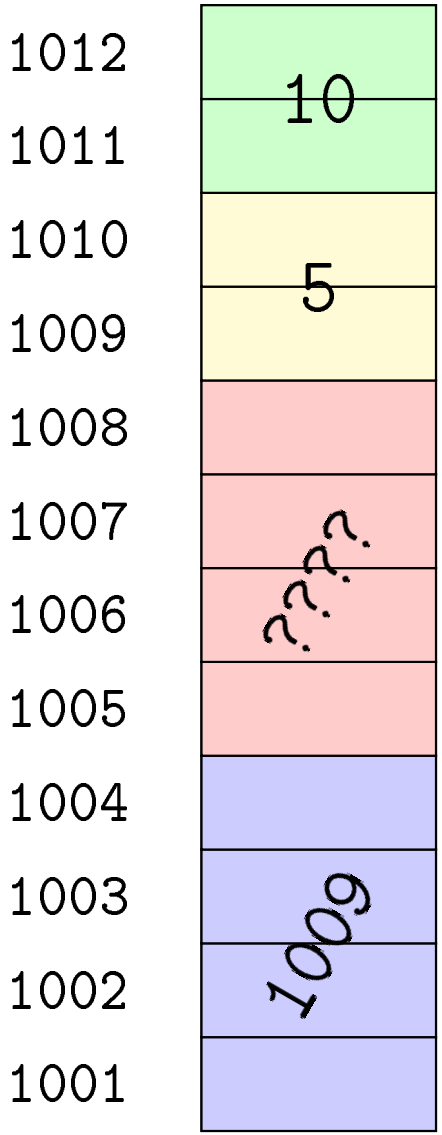

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



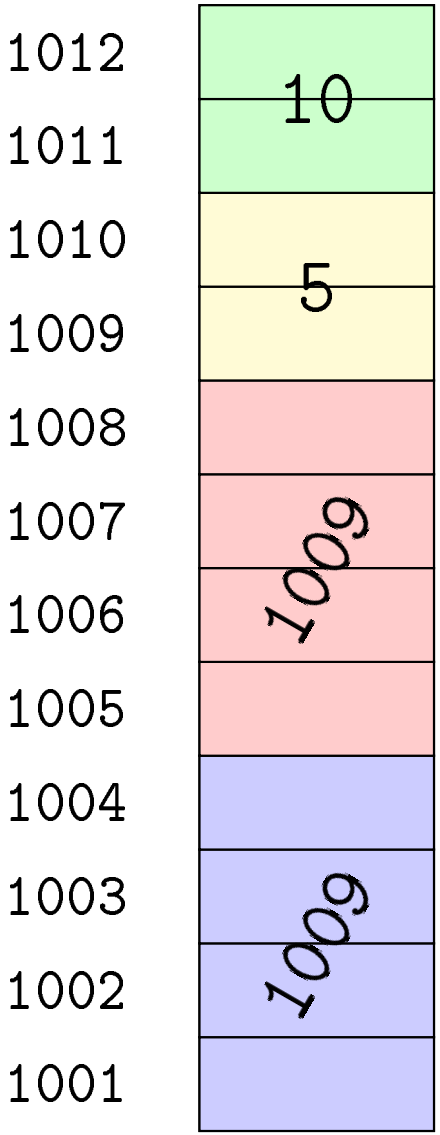

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



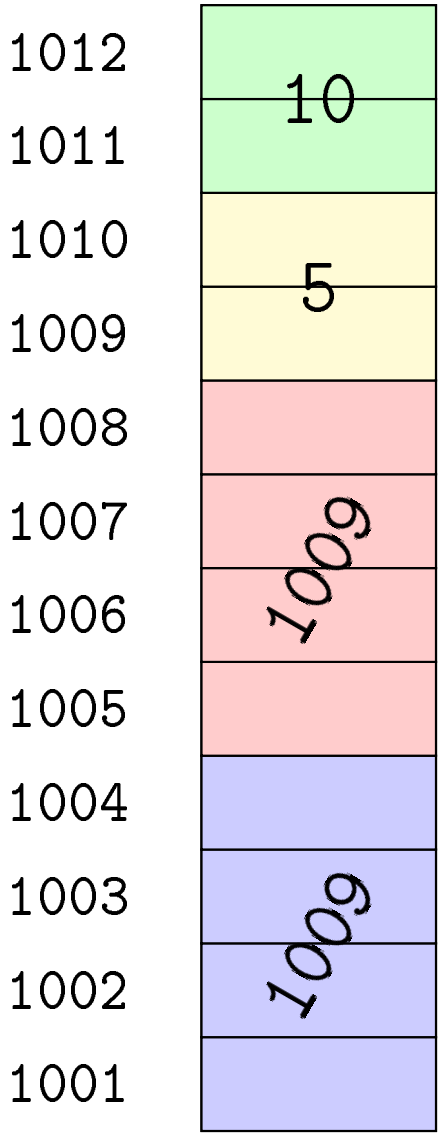

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



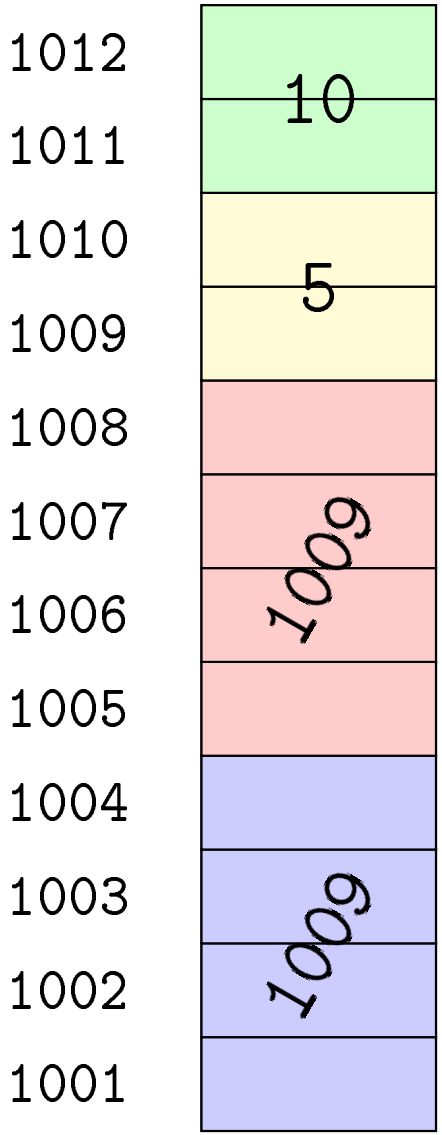

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



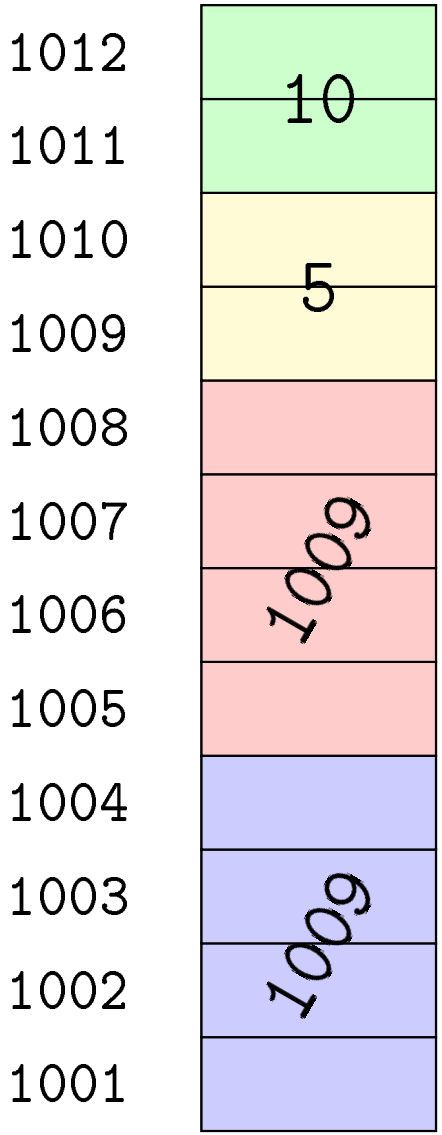

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado



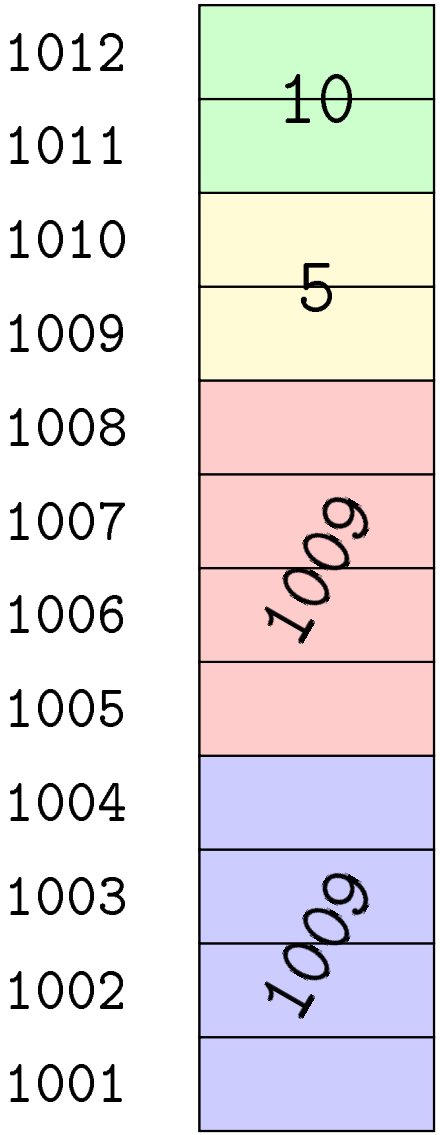

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: Ejemplo anterior animado




```
// 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;
```

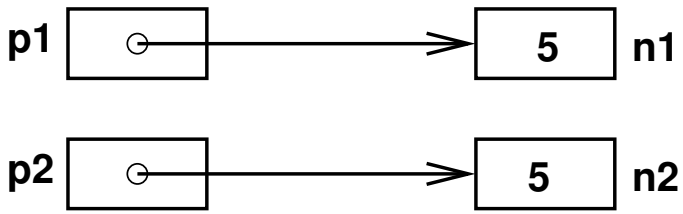
# Operadores relacionales: Ejemplo anterior animado



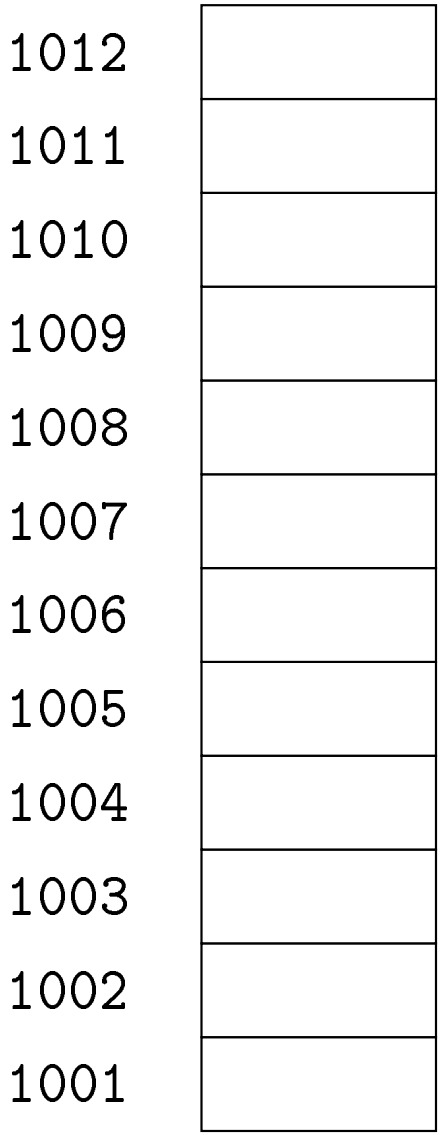

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=10;
// Se asignan los punteros
p1=&n1;
p2=p1
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: 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";
```

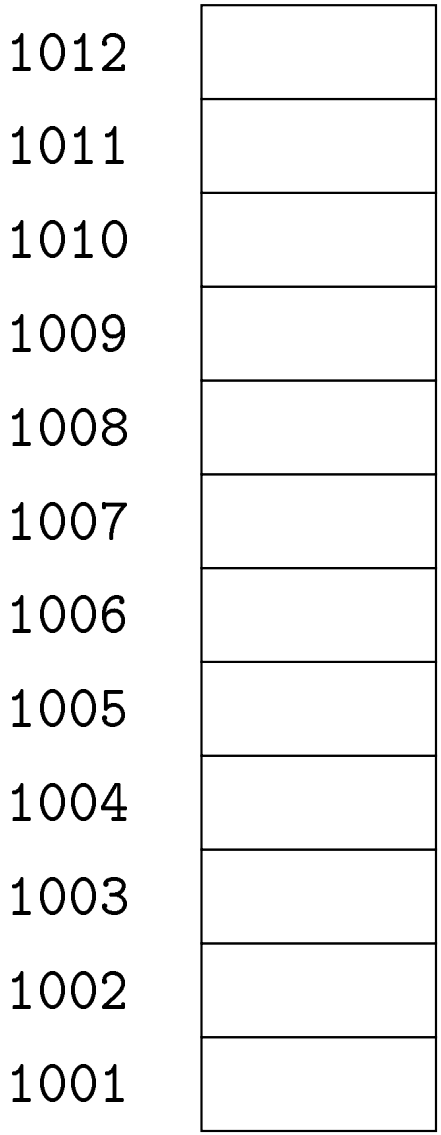


# Operadores relacionales: otro ejemplo (ej. animado)



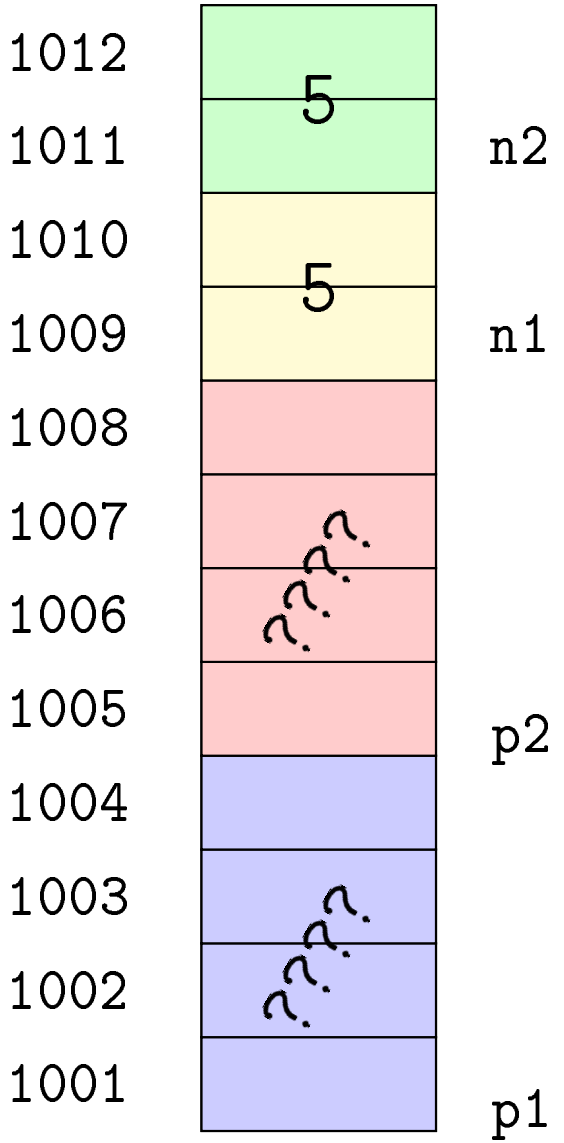

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



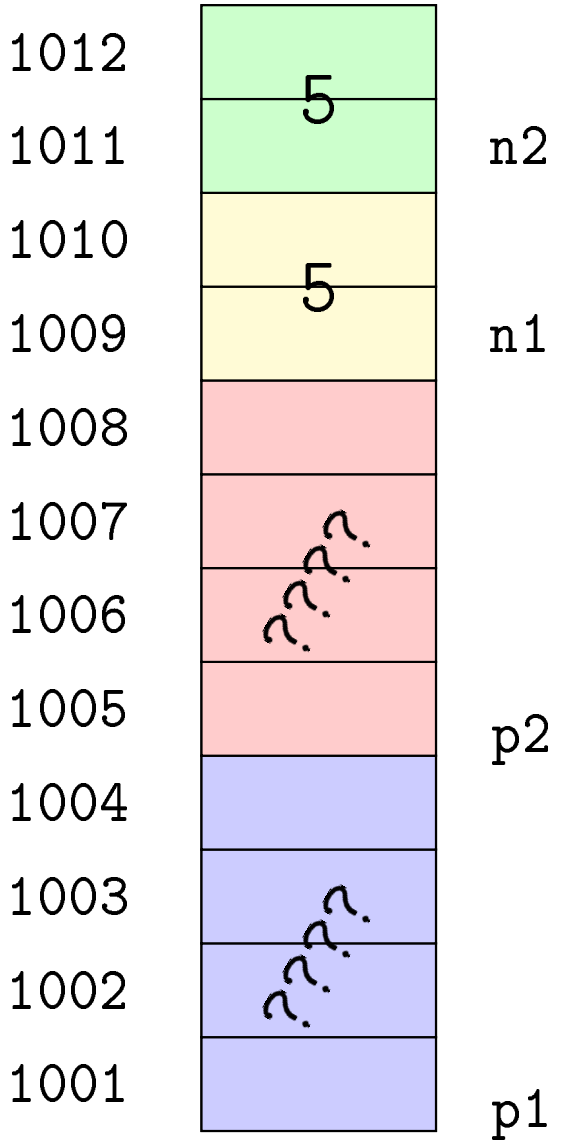

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



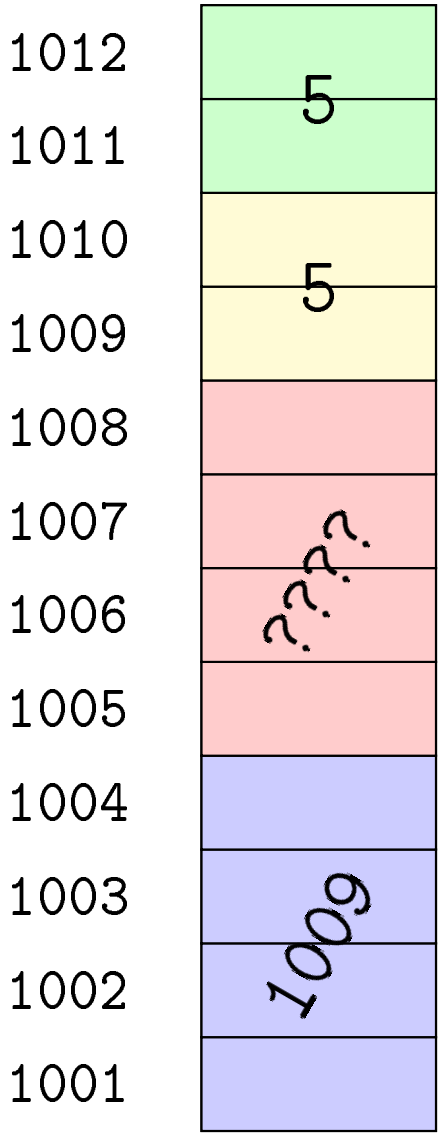

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



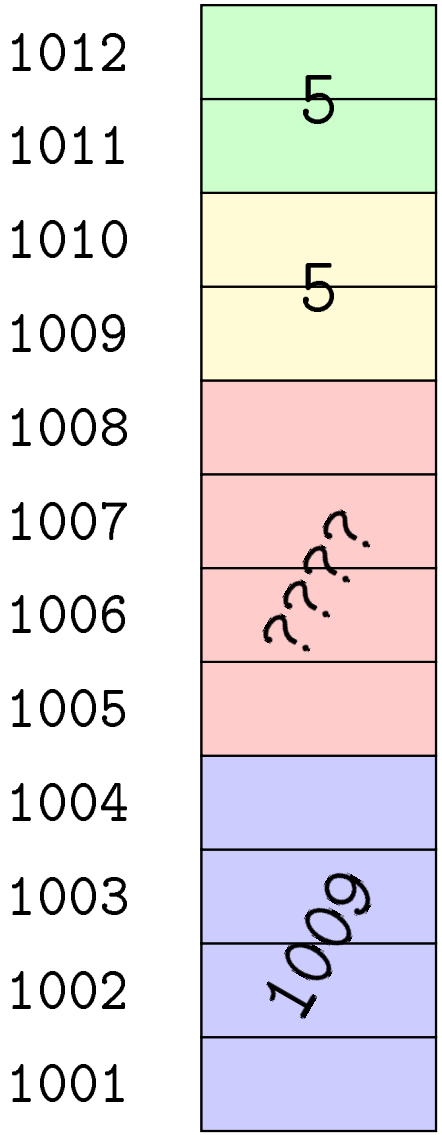

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



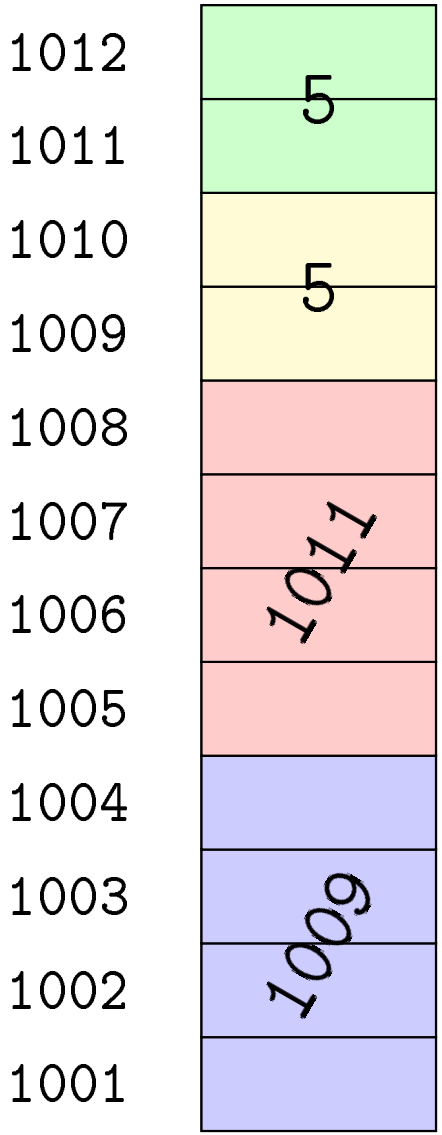

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



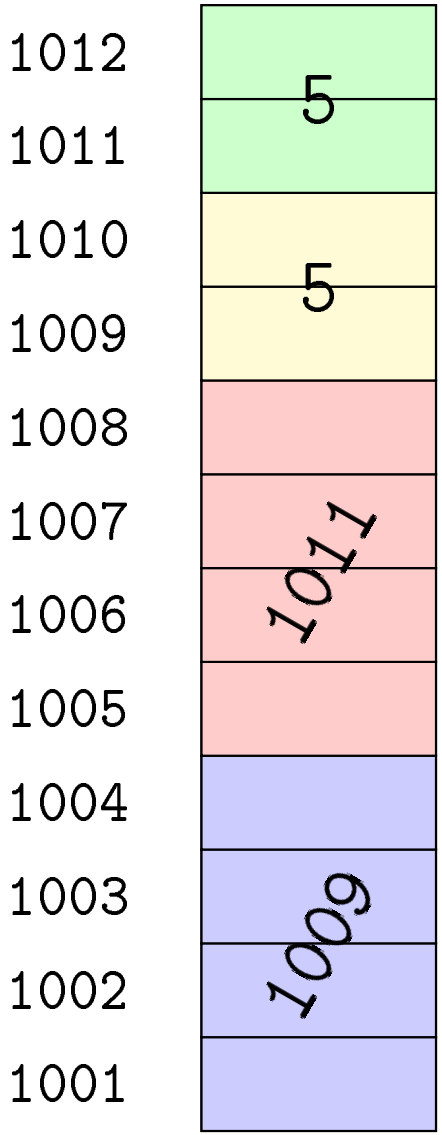

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



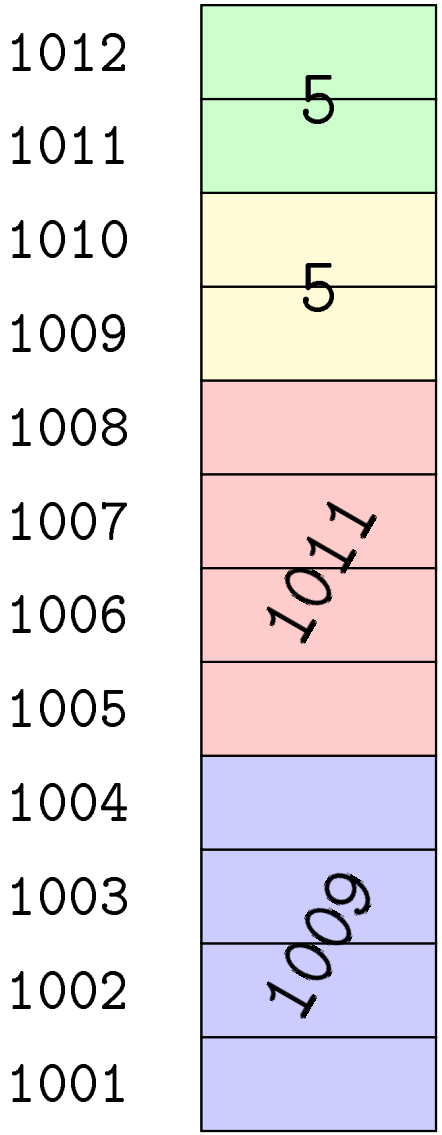

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



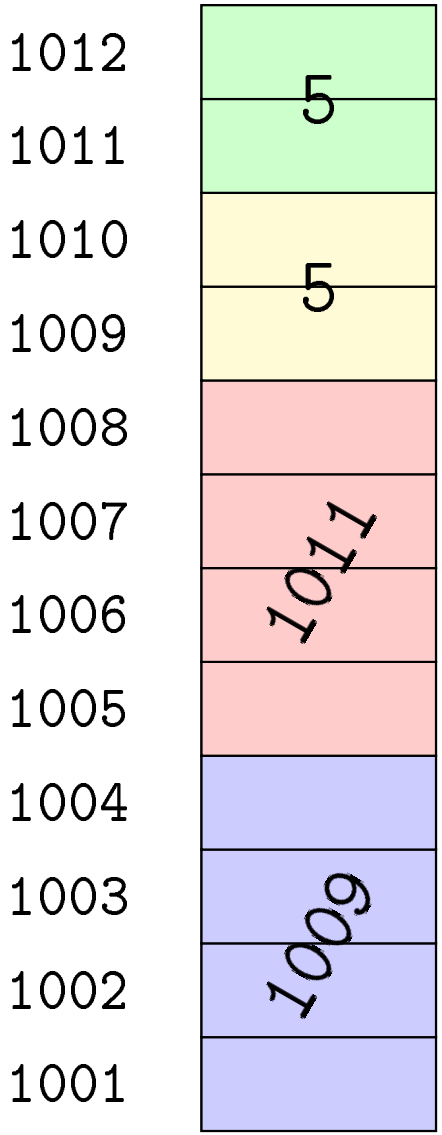

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
   cout << "Punteros iguales "<< endl;
else
   cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
   cout << "Valores iguales"<< endl;
else
   cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)



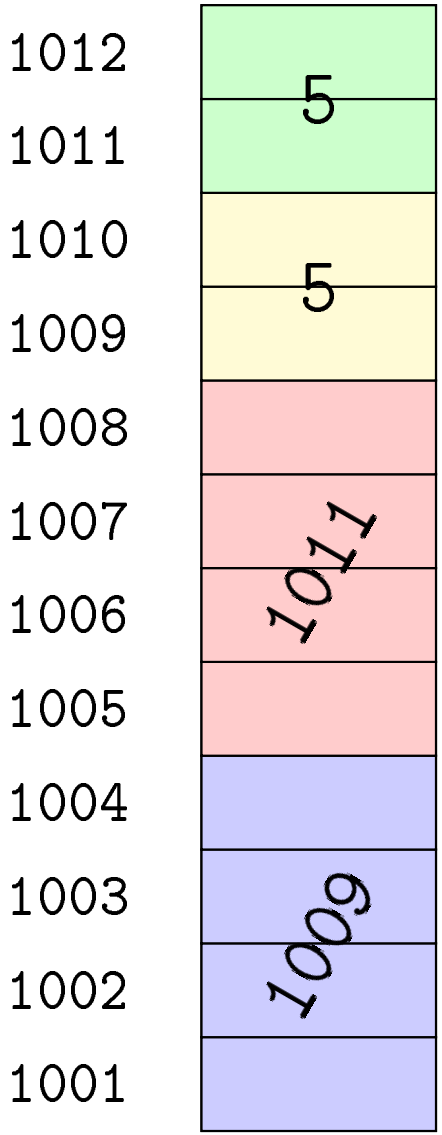

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales: otro ejemplo (ej. animado)




```
// 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)



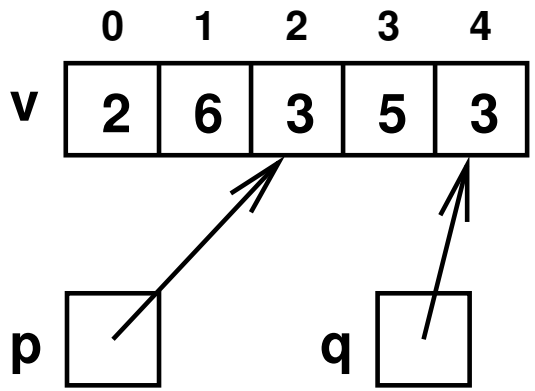

```
// Se declaran las variables
int *p1, *p2, n1=5, n2=5;
// Se asignan los punteros
p1=&n1;
p2=&n2;
// Se hacen las operaciones sobre ellos
if (p1 == p2)
    cout << "Punteros iguales "<< endl;
else
    cout << "Punteros distintos "<< endl;
if(*p1 == *p2)
    cout << "Valores iguales"<< endl;
else
    cout << "Valores diferentes "<< endl;
```

# Operadores relacionales

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

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

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




| | |
|---|---|
| p==q | false |
| p!=q | true |
| *p==*q | true |
| p<q | true |
| p>q | false |
| p<=q | true |
| p>=q | false |

# Operadores aritméticos

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

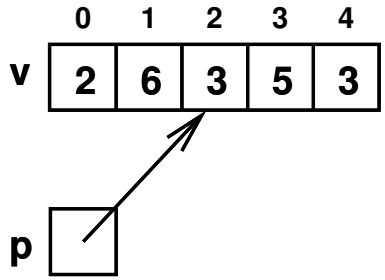
$$N * sizeof(tipobase)$$

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

# Operadores aritméticos

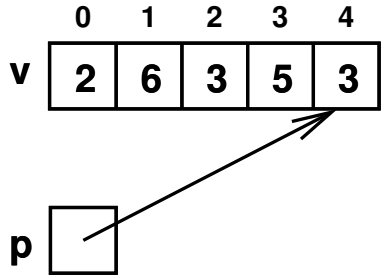
- Situación inicial:

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



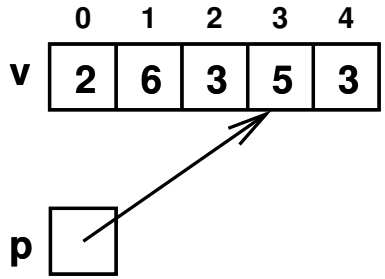

- Si sumamos 2 a p:

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




- Si restamos 1 a p:

```
p--; // p=p-1
```

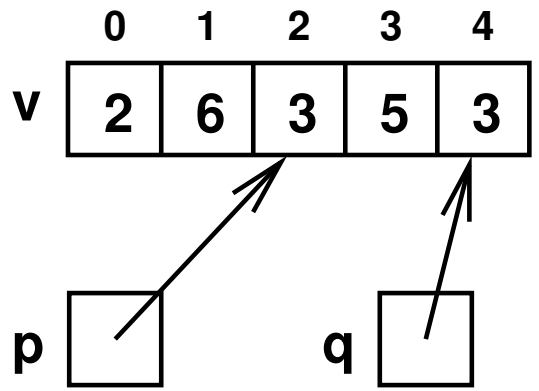


# Operadores aritméticos

Cuidado con expresiones que combinan $*$, & **(monarios)** y operadores aritméticos. Se sigue la regla de la precedencia de los operadores.

- *p++; r = *p+1; ¿$p$, $*p$ ?
- ¿Qué devuelve q - p?

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



# Indice

# Punteros y arrays

Los punteros y los arrays están estrechamente vinculados.

## Al declarar un array

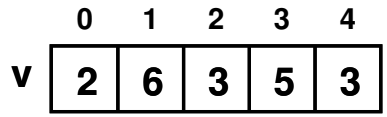
```
<tipo> <identif>[<n_elem>]
```

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

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

Podemos usar arrays como punteros al primer elemento.

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




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

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

```
int *ptr = v;
```

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

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ——————— 6
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;                    ———————— 5
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;                    ———————— 5
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———————— 6
p=v+2;
cout << *p << endl;                    ———————— 3
p++;
cout << *p << endl;                    ———————— 5
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;
```

# Punteros y arrays: ejemplo

```
// Se declara el array
int v[5]={2, 6, 3, 5, 3};
// Se crea el puntero
int *p;
// Se asigna
p=&(v[1]);
cout << *p << endl;                    ———— 6
p=v+2;
cout << *p << endl;                    ———— 3
p++;
cout << *p << endl;                    ———— 5
p=&(v[3])-2;
cout << p[0] << " " << p[2] << endl;   ———— 6  5
```

# Algunos Ejemplos I

1. 
```
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
```

2. 
```
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:

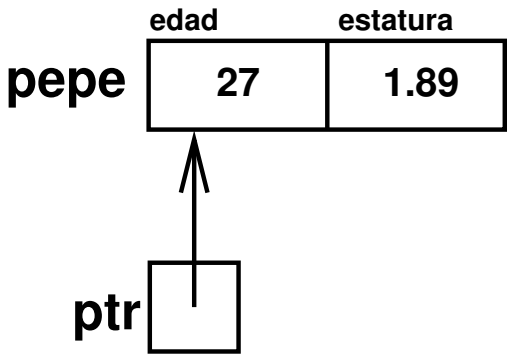
```
int v[10] = {3,5,2,7,6,7,5,1,2,5};

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

# Indice

# Operadores =, &, * con objetos

Un puntero también puede apuntar a objetos **de tipo struct o clase**:

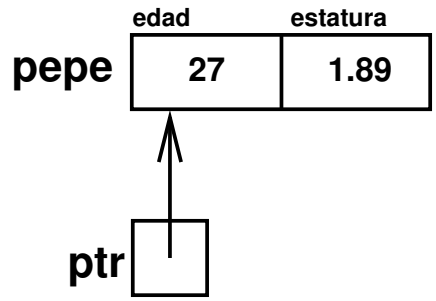
```
struct Persona{
    int edad;
    double estatura;
};
Persona pepe;
Persona *ptr;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
cout << (*ptr).edad << endl;
```



# Operadores =, &, * con objetos

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

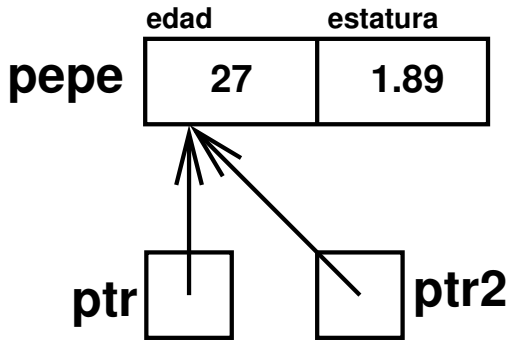
```
class Persona{
    int edad;
    double estatura;
public:
    int getEdad() const;
    double getEstatura() const;
    void setEdad(int anios);
    void setEstatura(double metros);
};
Persona pepe, *ptr;
pepe.setEdad(27); pepe.setEstatura(1.89);
// pepe.edad=27; CUIDADO: no valido desde fuera
//de metodo de la clase, edad es privado
ptr = &pepe;
cout << (*ptr).getEdad() << endl;
// cout << (*ptr).edad << endl; CUIDADO: no valido
//desde fuera de metodo de la clase, edad es privado
```



# Operadores =, & * con objetos

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

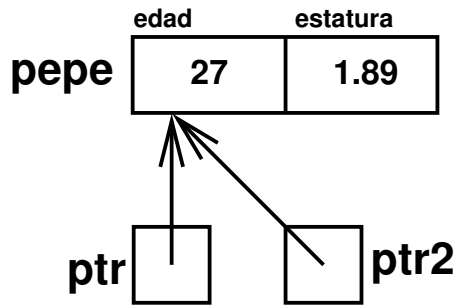
```
struct Persona{
    int edad;
    double estatura;
};
Persona pepe;
Persona *ptr, *ptr2;
pepe.edad=27;
pepe.estatura=1.89;
ptr = &pepe;
ptr2 = ptr;
cout << (*ptr).edad << endl;
cout << (*ptr2).edad << endl;
```



# Operadores =, &, * con objetos

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

```
class Persona{
    int edad;
    double estatura;
public:
    int getEdad() const;
    double getEstatura() const;
    void setEdad(int anios);
    void setEstatura(double metros);
};
Persona pepe, *ptr, *ptr2;
pepe.setEdad(27); pepe.setEstatura(1.89);
ptr = &pepe;
ptr2 = ptr;
cout << (*ptr).getEdad() << endl;
cout << (*ptr2).getEdad() << endl;
```



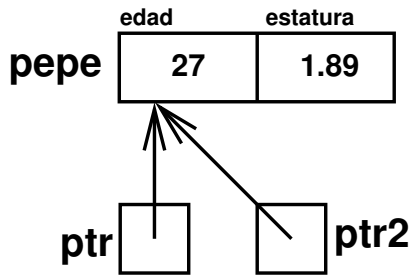
## Operador ->

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

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

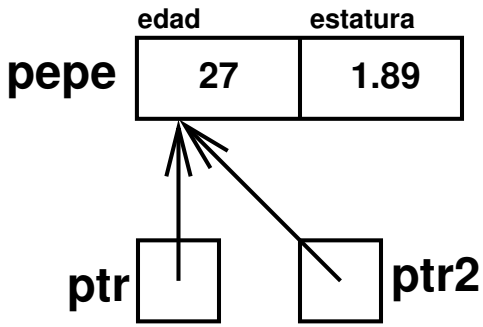
## Ejemplo de uso de -> con struct

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



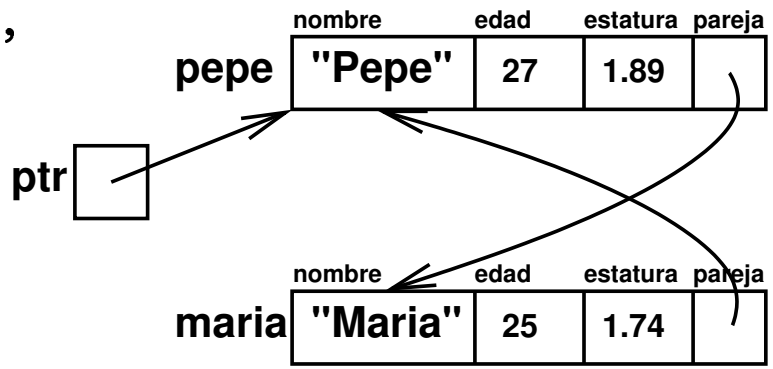
## Ejemplo de uso de $->$ con class

```
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;
```



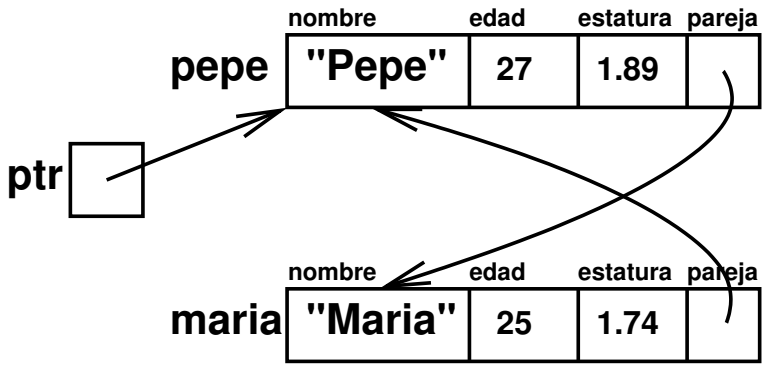
# Campos de tipo puntero a ..., en struct

```
Persona *p;
 struct Persona{
     string nombre; int edad;
     double estatura;
     Persona *pareja;
 };
 Persona pepe={"Pepe",27,1.89,nullptr},
         maria={"Maria",25,1.74,nullptr},
         *ptr=&pepe;
 pepe.pareja=&maria;
 maria.pareja=&pepe;
 cout << "La pareja de "
      << ptr->nombre
      << " es "
      << ptr->pareja->nombre <<
 endl;
```

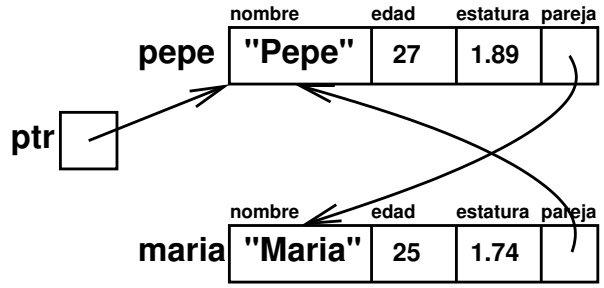


# Campos de tipo puntero a ..., en class

```
class Persona{
    string nombre;
    int edad;
    double estatura;
    Persona *pareja;
public:
    Persona(string name, int anios, double metros);
    int getEdad() const;
    double getEstatura() const;
    Persona *getPareja() const;
    void setPareja(Persona *compa);
    ...
};
Persona pepe("Pepe",27,1.89),
        maria("Maria",25,1.74),
        *ptr=&pepe;
pepe.setPareja(&maria);
maria.setPareja(&pepe);
cout << "La pareja de "
     << ptr->getNombre()
     << " es "
     << ptr->getPareja()->getNombre()<< endl;
```



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



# Indice

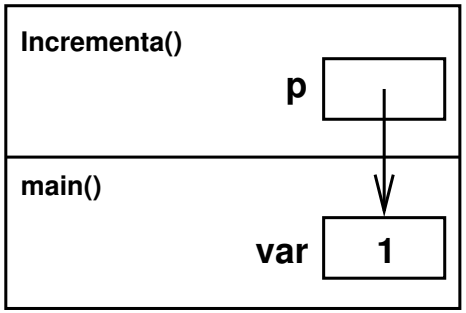
# Paso por valor de punteros I

## Puntero como argumento de entrada de una función

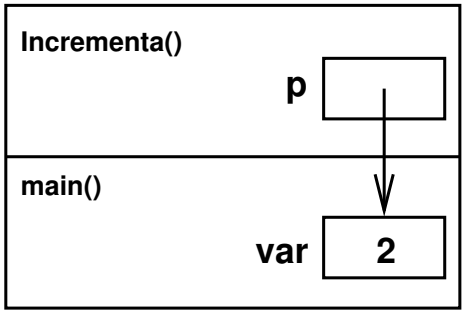
- El puntero permite simular el paso por referencia de una variable.

```
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**



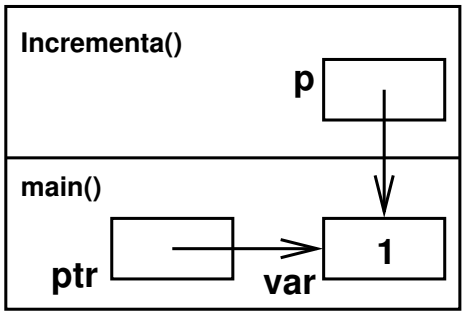
# Paso por valor de punteros II

## Puntero como argumento de entrada de una función

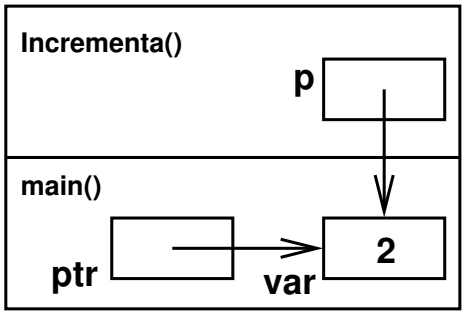
- El puntero permite simular el paso por referencia de una variable.

```
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**




Cambia la llamada respecto al código anterior.

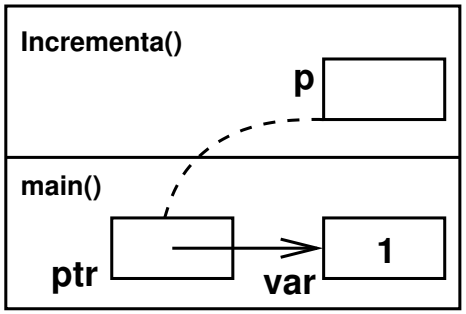
# Paso por referencia de punteros I

## Puntero como argumento de salida de una función

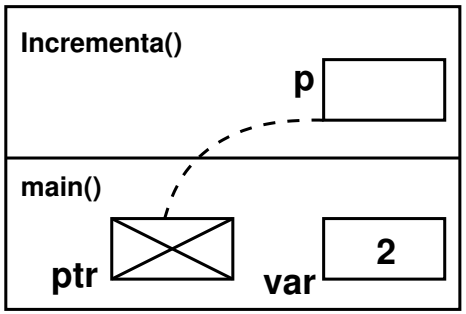
- Si deseamos modificar el puntero original, debemos usar el paso por referencia.

```
1  void incrementa(int* &p){
2      (*p)++;
3      p=nullptr;
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**



# Error por devolución de punteros I

## Puntero 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;
}
```

# Error por devolución de punteros II

## Otro ejemplo de error

La devolución de punteros a **datos locales** a una función.

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

# Error por devolución de punteros III

## Otro ejemplo de error

La devolución de punteros a **datos locales** a una función.

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

# Error por devolución de punteros IV

## Otro ejemplo de error

La devolución de punteros a **datos locales** a una función.

```
const int DIM = 10;
int *doble() {
    int v[DIM]={1,3,5,7,9,0};
    int *p;
    for (p=v; *p; p++)
        *p *= 2;
    return v;
}
int main(){
    int *vv;
    vv = doble();
```

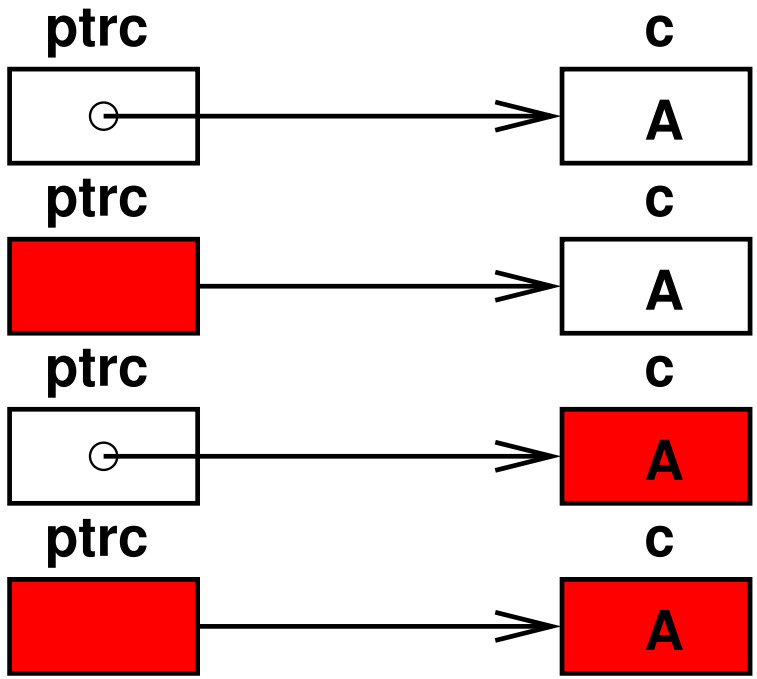
# Error por devolución de punteros V

```
cout << *vv << endl;
return 0; }
```

# Indice

# Punteros y const I

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



# Punteros y const II

| | |
|---|---|
| Ninguno sea const | `double *p;` |
| Sólo el **dato** apuntado sea **const** | `const double *p;` |
| Sólo el **puntero** sea **const** | `double *const p;` |
| **Puntero y dato** sean **const** | `const double * const p;` |

- Las siguientes expresiones son equivalentes:

| | |
|---|---|
| `const double *p;` | `double const *p;` |

# Asignación entre punteros y **const** punteros I


Es posible asignar un const puntero a un puntero no const, pero no al revés.

- * puntero = * **const** puntero. Sí

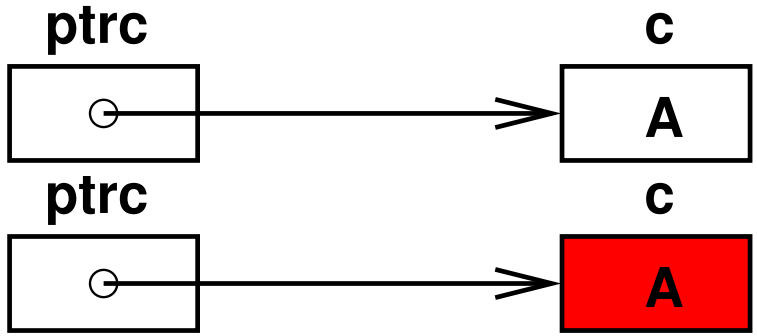
```
double a = 1.0;
double * const p=&a; // puntero constante a double
double * q;          // puntero no constante a double
q = p;       // BIEN: q puede apuntar a cualquier dato
p = q;       // MAL: p es constante
```

Error de compilación:

...error: asignación de la variable de sólo lectura ‘p’

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

# Asignación entre punteros a no **const** y punteros a **const** I




Un puntero a dato **no const** no puede apuntar a un dato **const**, pero al revés sí. (En la asignación se hace una conversión implícita).

# Asignación entre punteros a no **const** y punteros a **const** II

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: de permitirse la operación de asignación permitiría cambiar el valor de f, que fue declarada **const**.

# Error de compilación I

## Asignación incorrecta

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

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

## Error de compilación:

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

# Error de compilación II

## Asignación incorrecta

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

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

## Error de compilación:

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

# Punteros, funciones y const

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

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

Error de compilación:

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

# Punteros, arrays y const

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

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

# Indice

# Ejemplos de uso con variables cadena

Cadena C, un array de char con centinela

- Mostrar caracteres no 'e' no 'a' de una cadena C

```
char cadena[] = "metesematasa"; // cadena C
char *p;
for (p = cadena; *p ; p++)      // *p != '\0'
 if (*p != 'a' and *p != 'e')
     cout << *p;
cout << endl;
```

- Calcular longitud de una cadena:

```
char cadena[] = "metesematasa"; // cadena C
char *p;
int l=0;
for (p=cadena; *p!='\0';++p)
      l++;
cout << "Longitud: " << l << endl;
```

# Punteros y literales, un caso particular de cadena

- 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]`

- C++ trata un literal cadena de caracteres como `const char *`

# Ejemplos de uso con literales

- Calcular longitud de un literal:

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

- Se pierden 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.

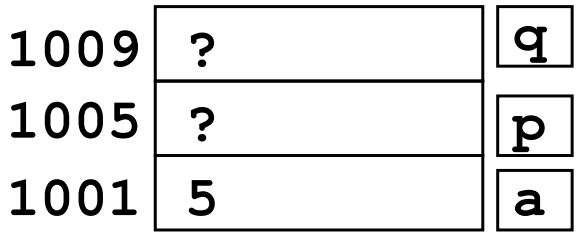
```
const char *cad2="Hola"; // Se asignan los punteros
cad2[2] = 'b'; // Error
```

# Indice

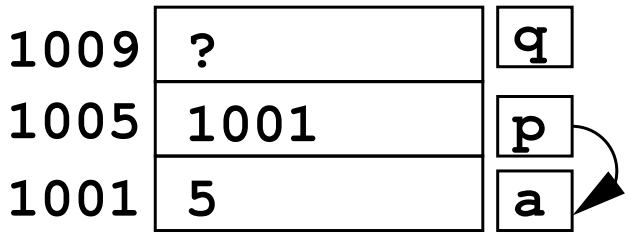
# Punteros a punteros

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

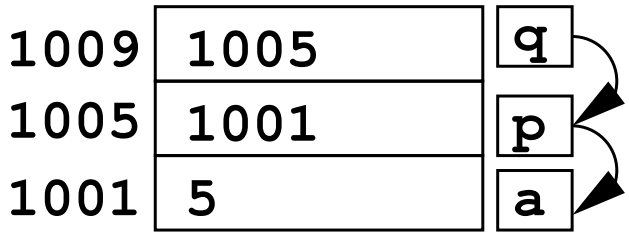
```
int a = 5;
int *p;
int **q;
```




```
p = &a;
```




```
q = &p;
```




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

# Indice

# 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:

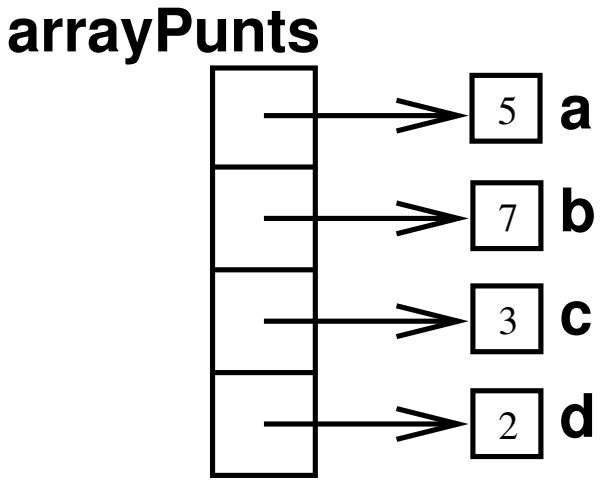
```
int* arrayPunts[4];
```

# 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;
```

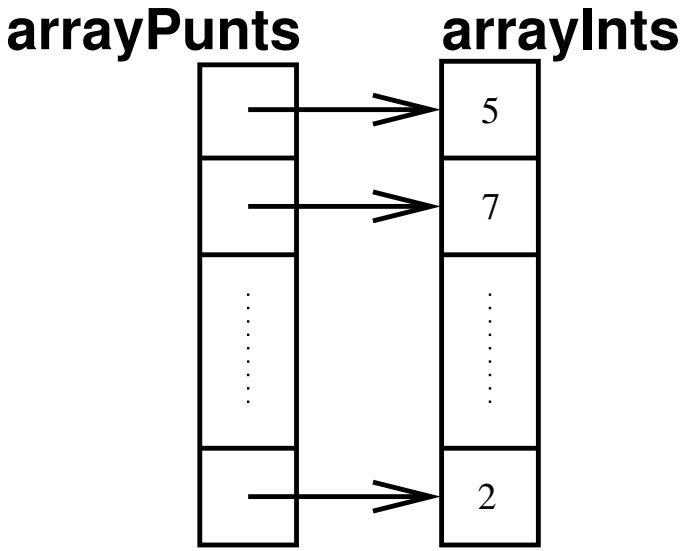
```
5 7 3 2
```



# Arrays de punteros

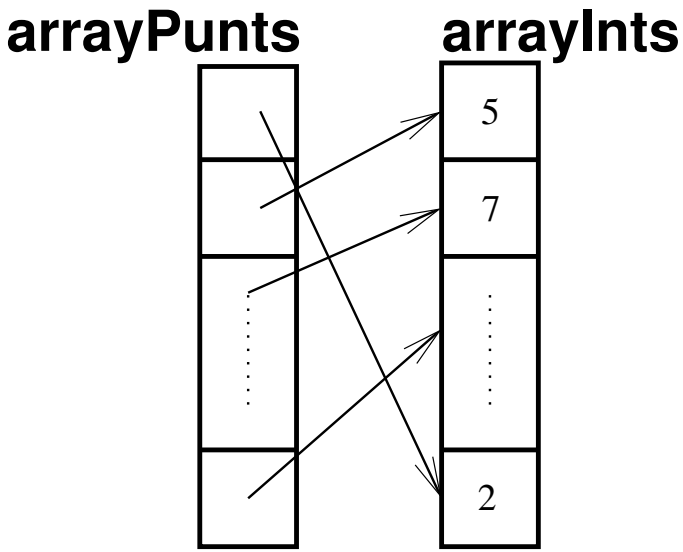
## Aplicación de array de punteros a enteros

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




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

# Arrays de punteros




```
for (int i=0; i < utilArray; i++)
    cout << *arrayPunts[i] << " ";
```

```
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++){     # inicializacion
      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] << " ";
```

# 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
```

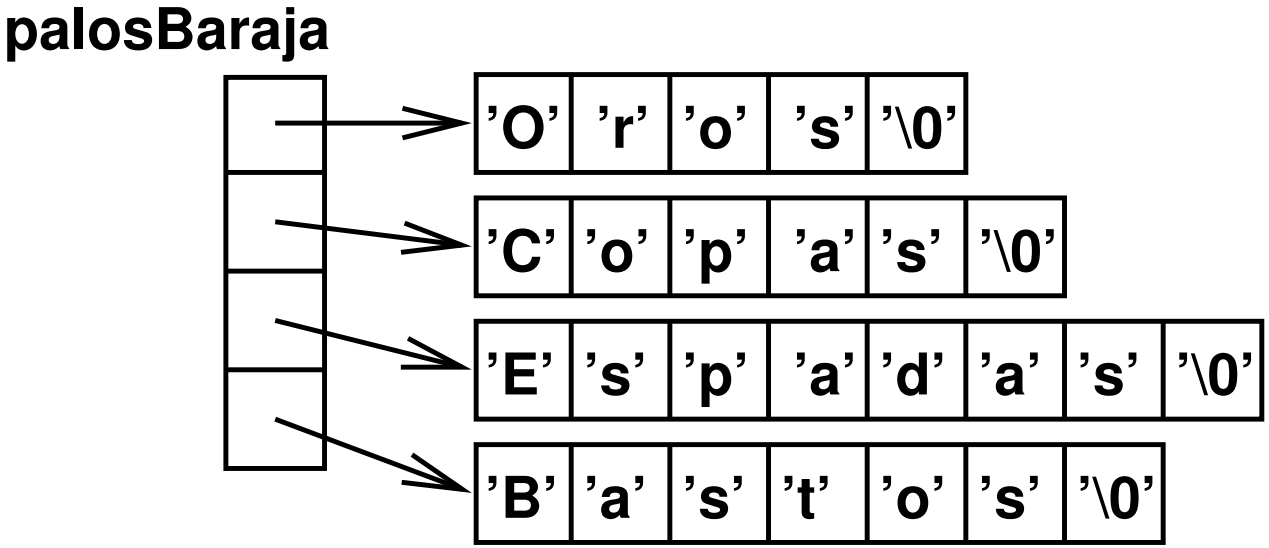
# Arrays de punteros

```
#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;
   }
}
```

# Arrays de punteros

## Aplicación: Array de cadenas estilo C

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



# 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
```

# Indice

# 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 descendentemente (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
```

# Indice

# 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
```

- Sea `double * p;` Responde a las preguntas y si alguna es cierta pon un ejemplo.
  1. `p` puede apuntar a una variable automática?
  2. `p` puede apuntar a una variable estática?
  3. `p` puede apuntar a un espacio de memoria alojado en el heap?
  4. `p` puede apuntarse a sí mismo?
- Definir las cabeceras de las funciones `f` para el código siguiente:

```
1
2     int main(){
3     double medidas[10] = { 1.5, 2.0, 0.3};
4     util = 3;
5     f1(medidas,util); // no se cambia nada
6     f2(medidas,util); // se cambia medidas
7     f3(medidas,util); // añade un componente
8     .......
9
```


creative
commons


BY NC SA