

Punteros en C++, referencias, vectores de punteros, memoria dinámica y vectores 3D... para quien no tiene npi. (En construcción)

Atención. Este documento es muy extenso y cubre mucho más de lo necesario en los ciclos formativos de informática.

Una ayuda que te puede venir bien: Cuando se creó el lenguaje C, se necesitó usar símbolos para indicar muchas cosas, desde operaciones aritméticas hasta caracteres de escape. Por ejemplo: "+", "*", "&", "/", "\", etc. Todos esos símbolos tienen un uso o significado.

Se necesitaban gastar tantos símbolos que los inventores de C, en un intento de ahorrar, decidieron utilizar el mismo símbolo para usos diferentes si eso era posible. Por ejemplo, las comillas (") siempre significan inicio o fin de cadena de caracteres, pero el símbolo ampersand (&) puede usarse para hacer la And lógica de dos operandos, la operación AND bit a bit entre dos operandos, para obtener la posición de memoria de una variable y para indicar que se está declarando una referencia.

El efecto del caracter depende de la situación precisa del mismo (¿Se está declarando una variable o no? ¿ Está entre dos variables o antepuesta a una variable?) Según sea la respuesta, su efecto será uno u otro

Esto te puede desconcertar si vas sin previo aviso

Declaración y uso básico

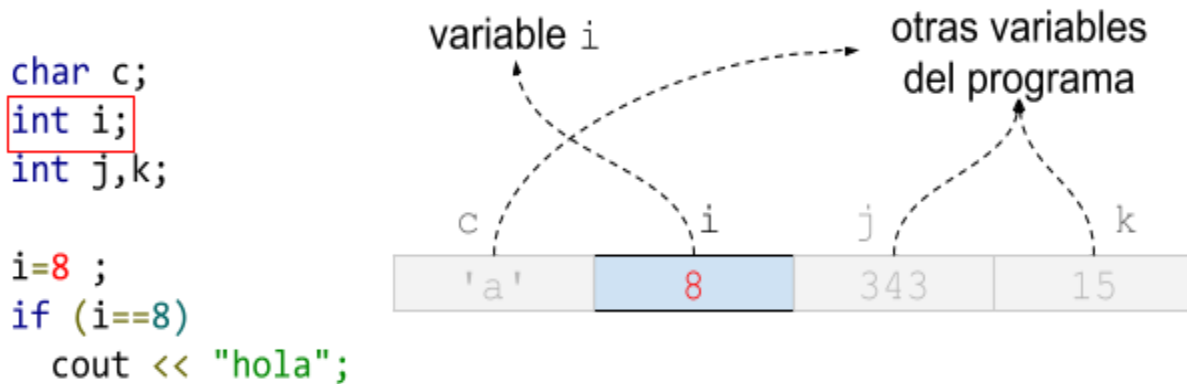
Piensa que todas las variables de un programa han de residir en la memoria durante la ejecución de éste. Dependiendo del tipo de variable, el espacio ocupado será diferente. El compilador se encarga de preparar el programa para que -al iniciar su ejecución- establezca espacios de memoria para cada variable. También decimos que *"el programa reserva espacios de memoria para las variables"*

Por ejemplo, cuando realizamos una declaración como la siguiente:

```
int i;
```

El programa -al iniciar su ejecución- asignará o reservará un espacio de memoria para la variable i. En ese espacio almacenará un dato numérico. En el programa en C, a partir del punto donde está esa declaración (`int i;`), "i" hará referencia al número almacenado y para todos los efectos podremos considerar a "i" como a un número. Podremos consultar y cambiar

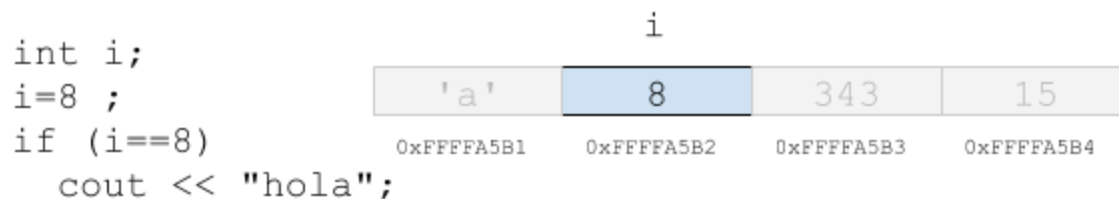
el valor de ese número "cambiando la *i*". Esto ocurre con todas las variables del programa. Observa y comprende la siguiente imagen descriptiva:



Dirección de memoria de una variable

En C es posible saber la posición de memoria de una variable (también llamada "dirección de memoria"). Es decir, puedes saber en qué celdas está almacenado un dato correspondiente a una variable como "i",

Realmente puedes pensar "*¿Y para qué? Si lo que quiero es almacenar un número, no me importa un bledo dónde esté ese número*". Verás la utilidad más adelante. La idea es que puedes saber la dirección de memoria de una variable. Pero seguramente la dirección en sí, no te diga nada. Por ejemplo, en un computador de 32 bits, la dirección de memoria podría ser algo así (en Hexadecimal) 0xFFFFFA5B2. Difícilmente algo así ayude en nada.



Fíjate cómo las variables del programa están almacenadas en direcciones de memoria consecutivas (esto es una simplificación válida de la realidad)

Sea como sea, la dirección de memoria puede ser sabida y almacenada en otra variable. Hemos de pensar entonces que esta nueva variable va a almacenar la dirección de memoria donde hay un entero. Ahora jugamos con dos variables, la "original i" que guarda un número y la nueva que almacena la dirección donde está ese número.

Modificamos el programa anterior y declaramos esta nueva variable cuyo tipo de dato es "dirección de memoria donde se guarda un entero". Esta declaración es muy similar a la de un entero como la variable "i" de antes. Tan sólo hay que reflejar que "no es un entero en sí, sino la dirección de un entero". Observa la declaración de la siguiente imagen:

ref

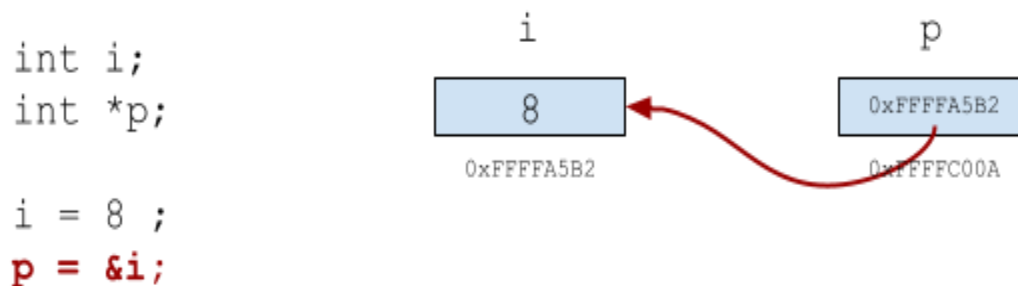
La variable "p" está declarada casi como la "i". Tan sólo aparece un **asterisco** como diferencia. Sin embargo, este asterisco es el que está diciendo que "p" almacenará la dirección

de memoria de un entero. Podemos pensar en la variable `p` como algo que apunta, referencia o te dirige a otro entero. Llamamos a esta variable **puntero**, porque se usa para *apuntar* a otra variable -en este caso de tipo `int`-. La palabra "puntero" es sinónima de "dirección de memoria". Además, observa que la variable `p` es eso: una variable, que tiene un espacio propio reservado en memoria.

Por ello, cuando dibujamos diagramas con las variables de un programa, a los punteros suele pintársele una flecha que parte de la variable y llega a otra parte. Sin embargo, si la variable está declarada y no inicializada -como cualquier otra variable- no sabemos qué valor tiene almacenado y no sabemos a dónde apunta. De ahí que la flecha en el diagrama anterior está dejada caer apuntando a nada en particular.

Para lograr que la variable `p` almacene la dirección de otra variable y por tanto apunte a esa otra variable, hemos de asignarle el valor correspondiente a la dirección de memoria de una variable y como se ha dicho antes: ¡Sí! ¡Podemos saber la dirección de memoria de una variable cualquiera!

La dirección de una variable se obtiene usando el operador ampersand "&" antepuesto a la variable. Observa el siguiente diagrama



En la línea roja se realiza lo siguiente:

1. Se recupera la dirección de memoria de la variable `i`. Gracias al "&"
2. Se almacena ese dato en la variable `p`

Ahora, **"p" apunta a "i"**. Es decir, `p` contiene la dirección de memoria de la variable `i` como puedes comprobar en el dibujo. La flecha en rojo, refuerza la visión de la situación que describen los números hexadecimales que aparecen, ya que deja claro que el contenido de `p` te lleva a `i`. A partir de ahora, no pondremos más números hexadecimales, porque es más claro dejar dibujadas las flechas

Lo importante es que ahora, gracias a `p`, podemos cambiar el valor numérico de la variable `i` sin usar la variable `i` (aunque suene a estupidez). Supongamos que deseamos almacenar el valor 11 en la variable `i`, pero usando la variable puntero `p`. Quizá estés pensando en hacer:

```
p = 11;
```

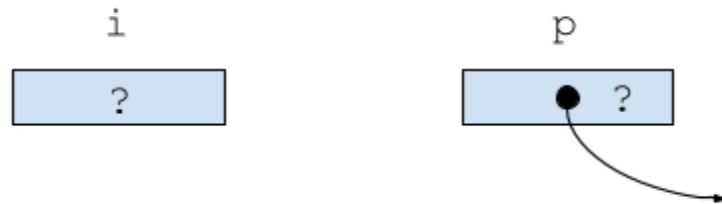
Esto sería un grave error. Lo que realmente estarías haciendo es modificando la variable p únicamente, e indicando que apunte a la dirección 11 de memoria. ¡ A saber qué hay en la dirección 11 de memoria !

Es necesario usar otro operador o símbolo para acceder a i a través de p. Observa el diagrama con un bloque de código que comentamos con dibujos. Inicialmente se declara un entero (con valor desconocido) y un puntero (con valor igualmente desconocido)

```
int i;
int *p;

i=8 ;
p = &i;

*p = 11;
```

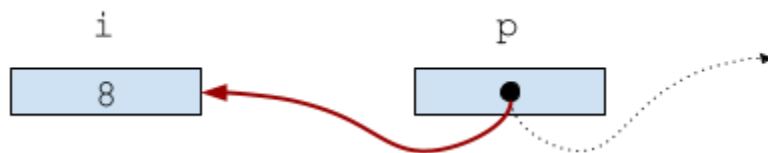


Seguidamente asignamos el valor 8 al entero i (esto no tendrá consecuencias aquí)y la dirección de memoria de i a p.

```
int i;
int *p;

i=8 ;
p = &i;

*p = 11;
```



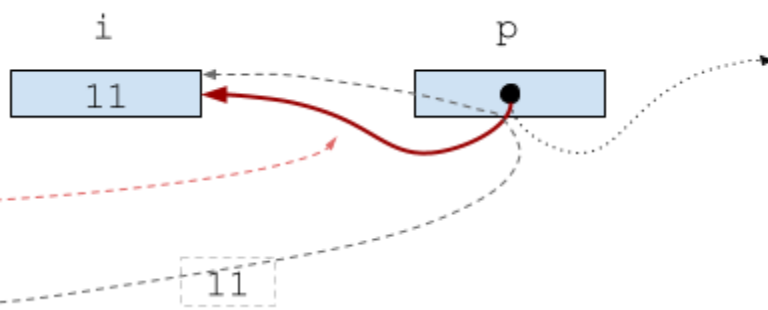
Fíjate que p ya no apunta a algo desconocido, está inicializada a la dirección de p, apunta a p

Finalmente vamos a usar el puntero para cambiar el valor de i. Esto ocurre en la última línea

```
int i;
int *p;

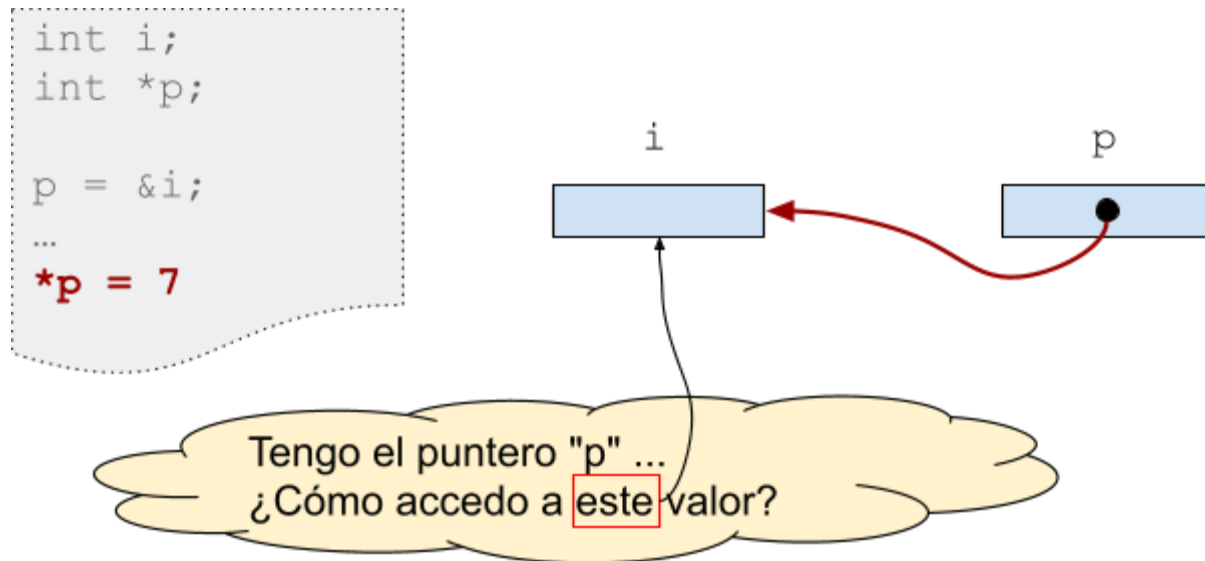
i=8 ;
p = &i;

*p = 11;
```



La línea en rojo (`*p = 11 ;`) es la que consigue almacenar un número 11 en la variable `i`, cambiando su valor como haría `"i = 11;"`. El **asterisco** en este caso, significa "lo apuntado por". De forma que hay que entender que no se desea acceder a `p`, sino a lo apuntado por `p`. Por ello la línea gris quiere expresar que el valor 11 "Viaja" hasta `p`, para descubrir donde está el dato entero y depositar allí el valor.

Es decir se desea acceder a la dirección de memoria que `p` tiene guardada, que resulta ser la posición donde está `i`... o sea, el valor de la variable "`i`".



Prueba finalmente el programa que demuestra estos conceptos básicos de punteros

```
#include <stdio.h>
#include <iostream>

using namespace std;

int main (int argc, char *argv[] ) {
    int i;
    int *p;

    i=8;
    p = &i;
    int aux;

    aux = *p;
    // p se ha usado para copiar i en aux.
```

```
cout << "i vale " << aux << endl;

*p = 10;
cout << "i vale ahora " << i << endl;
}
```

Manipulando mal un puntero

Observa el siguiente bloque de código donde algunas cosas han cambiado .

```
#include <stdio.h>
#include <iostream>

using namespace std;

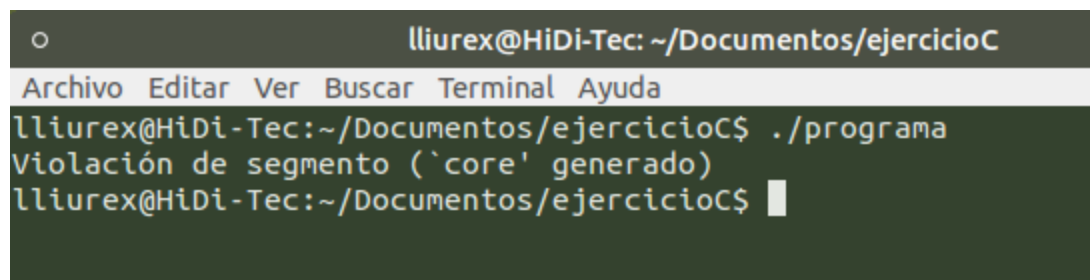
int main (int argc, char *argv[] ) {
    int i;
    int *p;

    i=8;
    *p = i;

    cout << "i vale " << i << endl;
}
```

Al compilarlo y ejecutarlo, obtenemos el siguiente resultado.

(se compila con `g++ -o programa fichero.cpp`)

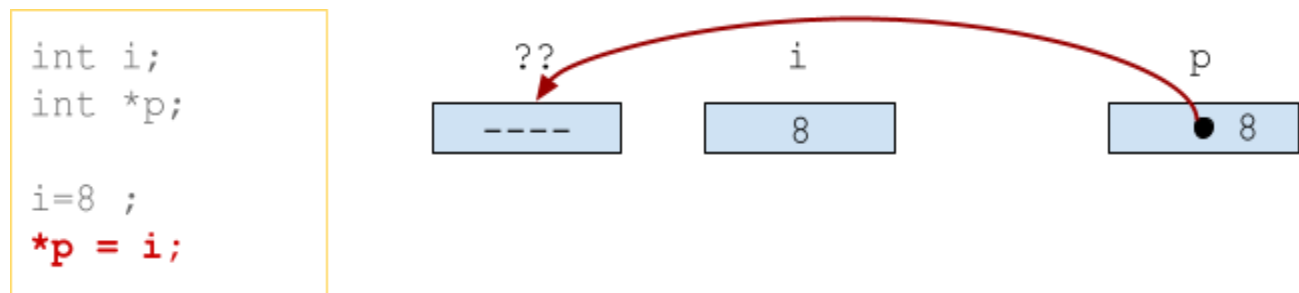
A screenshot of a terminal window. The title bar shows a window icon and the text 'lliurex@HiDi-Tec: ~/Documentos/ejercicioC'. The menu bar contains 'Archivo', 'Editar', 'Ver', 'Buscar', 'Terminal', and 'Ayuda'. The terminal text shows the user running './programa' in the directory '~/Documentos/ejercicioC', which results in a 'Violación de segmento ('core' generado)' error. The prompt returns to 'lliurex@HiDi-Tec:~/Documentos/ejercicioC\$' with a cursor.

```
lliurex@HiDi-Tec: ~/Documentos/ejercicioC
Archivo Editar Ver Buscar Terminal Ayuda
lliurex@HiDi-Tec:~/Documentos/ejercicioC$ ./programa
Violación de segmento ('core' generado)
lliurex@HiDi-Tec:~/Documentos/ejercicioC$
```

¿Qué ha pasado?

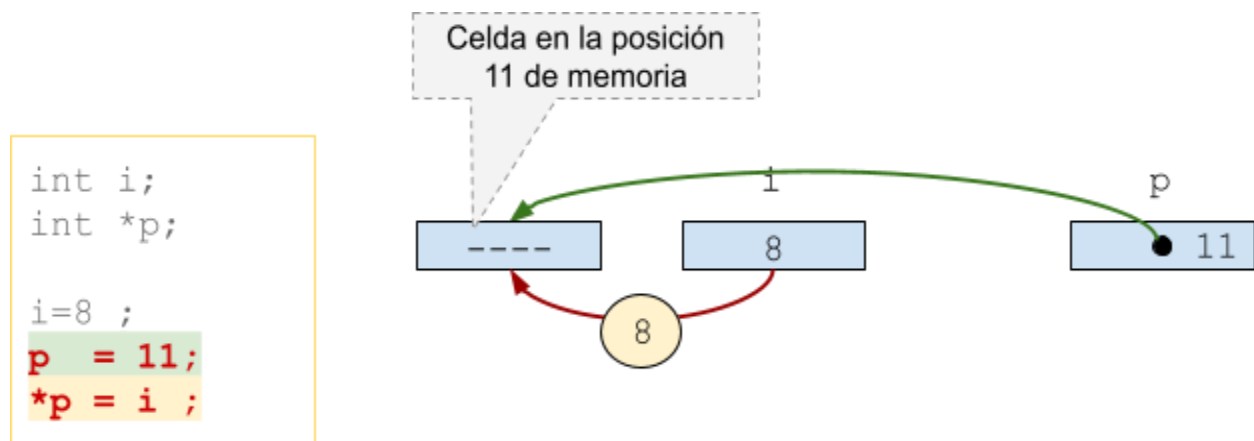
Primero, observa que el compilador no se ha quejado de nada. No encuentra errores. Lo que vemos, sólo ocurre al ejecutar el programa. Es un error de ejecución .

El problema lo manifiesta la línea $*p = 8$. ¿Dónde va ese valor 8?... donde apunte p. Y ¿Dónde apunta? No lo sabemos...



Un puntero debe ser inicializado a una dirección conocida (debe "apuntar" a una variable del programa) siempre antes de ser accedido. En el ejemplo anterior, la instrucción en rojo es correcta de por sí, pero no así el valor de p y por ello el lugar donde se quiere depositar el valor es un lugar desconocido y *prohibido*.

El siguiente caso es también un error, y ocurre con algo de frecuencia. El error es asignar un valor numérico a una variable de tipo puntero. Observa la instrucción sombreada en verde que se corresponde con la línea verde del siguiente diagrama:



Al asignar el valor 11 directamente a la variable p, estamos indicando que p apunta a la dirección 11 de memoria. Aunque de por sí, eso no daría error, no sabemos ahí qué es lo que hay y cuando intentamos usar p para almacenar algo en esa dirección 11, el programa falla, puesto que **no podemos escribir datos en ubicaciones de memoria que no sean variables de nuestro programa**.

Ejercicio: El siguiente programa lee el valor de dos variables desde la entrada estándar :

```
int a;
int b;

cout << "introduce el valor primero: " ;
cin >> a;
cout << endl << "introduce el valor segundo: " ;
cin >> b;
cout << endl;

if (a > b) {

}
else {

}
```

Se pide que sin utilizar las expresiones

a =

b =

Consigas incrementar en uno la variable en la que el usuario ha introducido el máximo valor
<aquí aparece el primer error severo

*p++ // explicar que lo correcto es : (*p)++

Punteros y funciones

Los punteros pueden ser especificados como argumento de funciones y como valor de retorno.

Paso por puntero para consultar el valor

Supón que tenemos una función que indica si un número es positivo o negativo. Es una función muy simple tanto de implementar como de utilizar.

```
bool esPositivo ( int dato) {
    if ( datp < 0 ) return false ;
    return true;
```



```

    }

int main(int argc, char *argv[]) {

    int i;

    cout << "Introduce un número" << endl;
    cin >> i ;

    if ( esPositivo(i) )
        cout << " Has introducido un número positivo" << endl;
    else
        cout << "Has introducido un número negativo" << endl;

}

```

Esta función no requiere cambios, es simple y funcionará siempre. Pero pese a todo vamos a cambiarla haciendo que en vez de recibir un número, reciba la dirección de un número e igualmente indique si el número en cuestión es positivo o no. Empezaremos por cambiar el argumento, que es lo que la función recibe.

Pasamos de

```
bool esPositivo ( int dato) {
```

a

```
bool esPositivo ( int * p) {
```

Date cuenta del asterisco ("*") que se antepone al nombre del argumento. Ello nos indica que el argumento es un puntero . La palabra "int", ahora por su parte, nos indica que es un puntero a un entero.

Este cambio hace que en el interior de la función debamos cambiar la forma de acceder al argumento, dado que ya no tenemos un argumento de tipo entero, sino puntero a entero.

```

bool esPositivo ( int * p) {
    if ( *p < 0 ) return false
    return true;
}

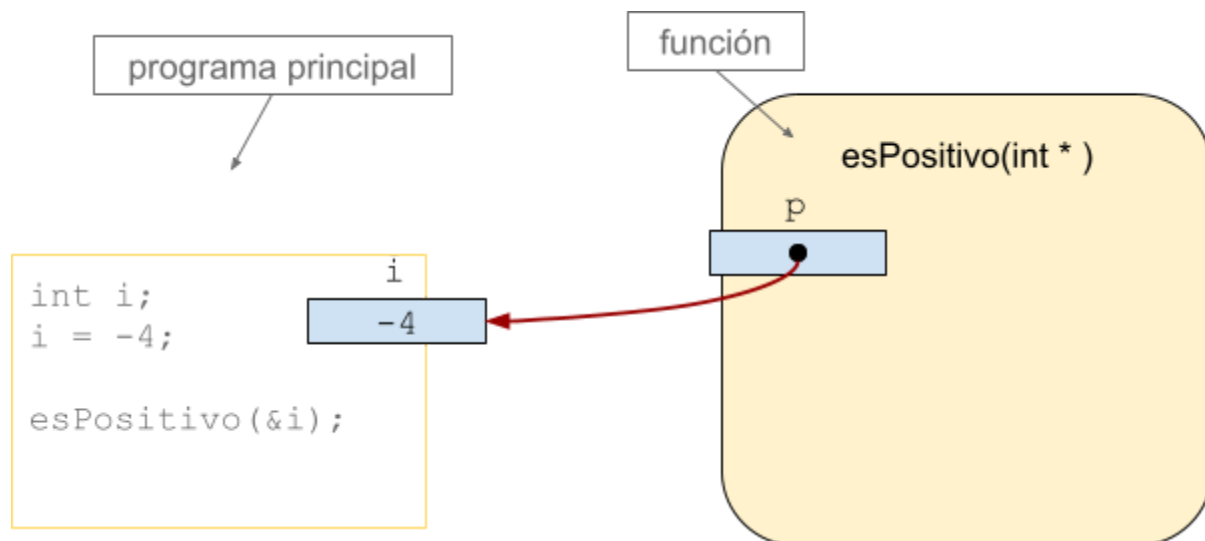
```

El argumento sigue la misma regla para ser declarado que las variables normales. Igual es también la forma de acceder al valor del número almacenado.

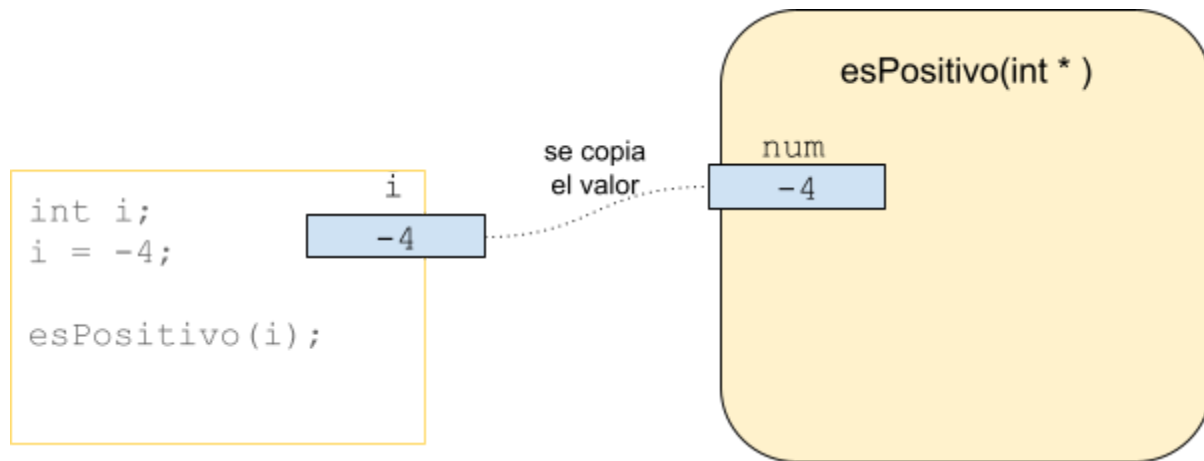
En estos casos es interesante estudiar cómo se realizan las llamadas a dichas funciones desde otro lugar.

```
int i;  
i = -4 ;  
bool negativo;  
negativo = esPositivo(&i);  
  
if (negativo)  
    cout << "Todo correcto, i es negativa, nunca positiva" << endl;
```

Lo interesante es mostrar una imagen de memoria cuando esté ejecutándose la función (las variables aparecen en azul):



Como puedes ver, es más sencillo de programar, se quitan los asteriscos * y el ampersand &, se cambia el nombre de la variable "p" por "num" (de "número") y se entiende todo mejor. Vamos a dibujar la situación creada en la llamada a la función porque estas diferencias son las que importa entender y no hemos de escatimar en sobreentenderlas:



Lo importante (importantísimo) es ver que la variable "num" tiene una copia del valor de la variable i. Son variables distintas, pero por efecto de la llamada se ha copiado su valor.

Paso por puntero par alterar el valor

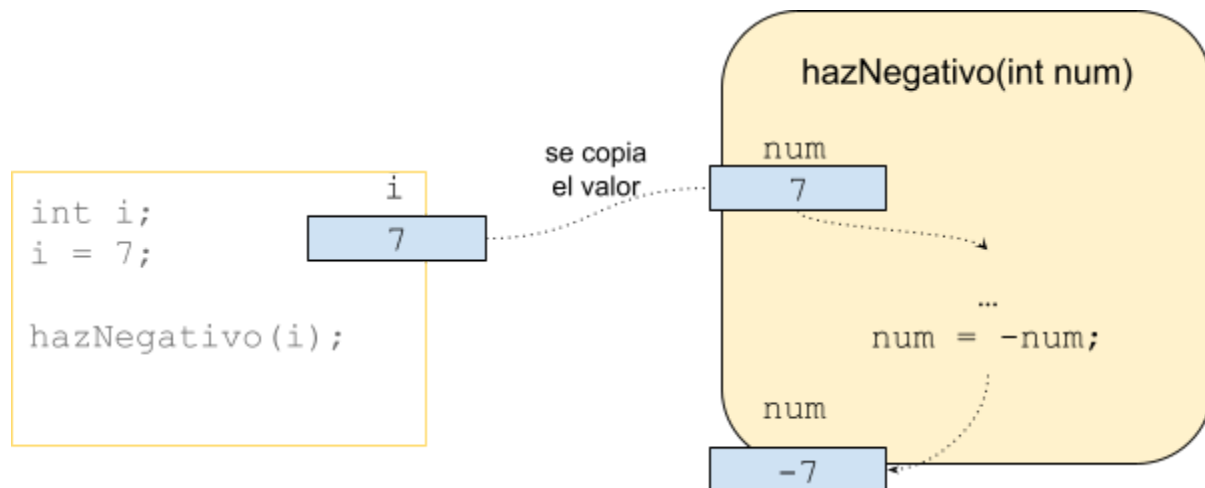
Ahora, imagina que quieres hacer una función para hacer negativa una variable (o para cambiarle el signo). Llamemos a la función "hazNegativo(??)". Como no sé qué tipo de dato pasarle, de momento, voy a dejarlo sin indicar. Se desearía usar la función como se puede ver a continuación :

```
int i = 7;
hazNegativo (  i  );
if (i != -7 )
    cout << "La función hazNegativo no funciona" << endl;
```

En un primer intento podríamos implementar la función de la siguiente manera:

```
void hazNegativo(int num) {
    if (num >= 0 ) num = - num ;
}
```

El problema es que al igual que ha ocurrido antes, la variable `num` (que es la que cambia) es una copia de la variable `i` que es la que se desea cambiar con la función

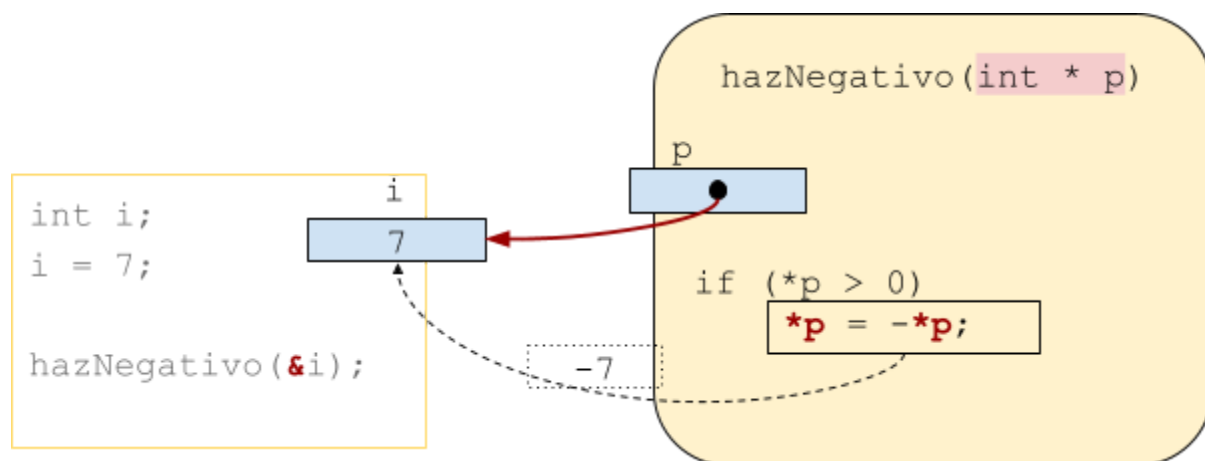


"i" sigue valiendo 7 después de la llamada a la función

Veámos ahora cómo se podría implementar esta función usando punteros. La idea clave es que *"en vez de recibir un número vamos a recibir la dirección de memoria de un número"*. Para implementarla, casi podemos hacer un proceso mecánico: "poner asteriscos delante del parámetro", y cambiar el nombre del parámetro -recomendable- para indicar que se trata de un puntero.

```
void hazNegativo(int * p) {
    if (*p >= 0 ) *p = - *p ;
}
```

La función ya no recibe un número, sino la dirección donde hay un número. Hemos de cambiar la llamada a la función, ya no podemos pasar la variable "i" como antes, sino *la dirección de la variable i*.



El diagrama anterior muestra la situación. La llamada a la función se realiza usando "&i" (hazNegativo(&i)) para acceder a la dirección de la variable. Haciendo este cambio, cumplimos con la función que estamos usando, concretamente con la parte sombreada en rojo

((void hazNegativo (int *)). Ahí, estamos diciendo que en la función, vamos a recibir la dirección de una variable que contendrá un entero... "llamada y declaración ¡encajan!"

Dentro de la función tenemos dos opciones:

- Usar el argumento "p" para indicar la dirección donde hay un entero
- Usar " *p " para acceder a "lo apuntado". Es decir para manipular el número existente y que corresponde -en este caso- con la variable i

Evidentemente, dentro de la función no nos interesa ni necesitamos para nada, saber o cambiar cuál es la dirección de la variable a la que tenemos que hacer negativa. Dónde esté concretamente esa variable en memoria ni nos importa ni se ha de cambiar.

Sin embargo, sí que debemos leer el valor numérico que allí existe y eventualmente cambiarlo. Esto se logra con el " * " antepuesto a " p ". Esto es lo que ocurre en la instrucción (*p = - *p). Se cambia el valor de un entero cuya dirección es p... y resulta que p apunta a i. Luego se cambia el valor de i. Esto se indica mediante una línea discontinua.

Ejercicio. Obtener la variable máxima entre dos

Para terminar con todas las posibilidades de usar punteros simples en las funciones, vamos a plantear y resolver un ejercicio. En él se pone de manifiesto la utilidad de devolver una dirección de memoria como valor de retorno de una función

Ejercicio: *Obtener el máximo de dos números de forma que **el resultado de la función pueda ser usado para cambiar el valor de la variable que tenía el máximo.*** (iremos poco a poco)

Empecemos a analizar el enunciado:

Queremos una función que reciba dos números, y que nos devuelva el máximo de esos dos números.

Sin embargo, el enunciado nos dice que el resultado de la función debe poder ser usada para alterar la variable que tenía el valor máximo. Es decir, con lo que la función devuelve podemos alterar el valor de alguna de las variables pasadas. Esto se puede poner de manifiesto en cualquiera de los siguientes ejemplos que son aproximados:

```
int num1 = 9;
int num2 = 11;
int * mayor;

mayor = max(num1, num2);
(*mayor) = 20
// ahora num2 debería valer 20
```

La idea también podría ser la siguiente:

```
int num1 = 9;
int num2 = 11;

*(max(num1, num2)) = 20;

// ahora num2 debería valer 20
```

(no te preocupes si no entiendes el código anterior, se te hará más claro conforme sigas leyendo)

En principio la primera función que nos viene a la cabeza es ésta:

```
int max(int a, int b) {
    if (a > b) return a;
    else return b;
}
```

La función anterior podría ser usada de la siguiente manera:

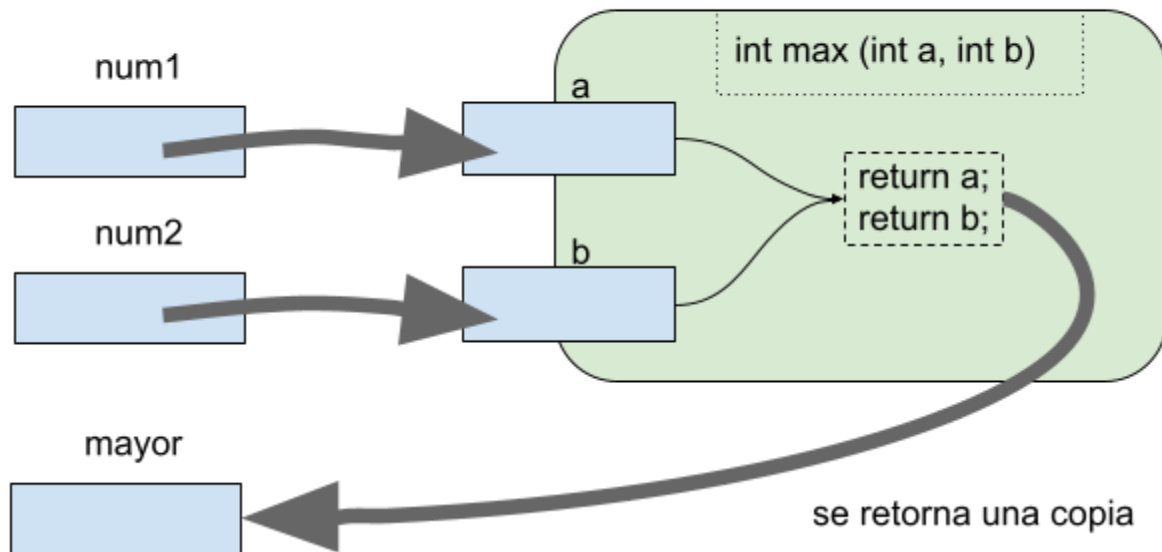
```
int num1 = 9;
int num2 = 11;
int mayor;

mayor = max(num1, num2);
mayor = 20

// ¿Qué vale num2 ?
```

. El siguiente diagrama muestra lo que ocurre con las variables y los valores (el recuadro verde representa la función max):

```
mayor = max(num1, num2)
```



El valor de `num1` se copia en la variable `a` local a la función. "`a`" es un argumento, un parámetro, pero en el fondo, "`a`" es una variable local a la función y tiene el valor copiado de `num1` cuando la función es invocada. Con `num2` y `b` ocurre lo mismo. Por otra parte, la función retornará un número. Ese valor será copiado al finalizar la función a la variable `mayor` tal como está establecido en la línea `mayor = max(num1, num2)`.

La única manera de usar la función `max` para modificar la variable que tenía el valor máximo es la que puedes ver en el siguiente código

```
int num1 = 9;
int num2 = 11;
int mayor;

mayor = max(num1, num2);
if (num1 == mayor) num1 = 20;
else /*if (num2 == mayor) */ num2 = 20;
```

Pero esto no ayuda nada, es una tontería que nos permite entender algo que no es lo que buscamos.

Aunque hay varios problemas o impedimentos en lo que hemos realizado, dediquémonos a examinar uno primero: La función devuelve un valor que se copia a la variable `mayor`. Lo que deseamos es que esa variable no tenga un valor numérico, sino que sea una referencia, un puntero a otra variable. Empezaremos cambiando el tipo de dato de la variable `mayor`.

```
int * mayor;
```

Ahora mayor almacenará la dirección de memoria de otra variable. La idea es que la variable mayor, que es un puntero, apuntará a num1 o num2, dependiendo de cuál sea mayor. Este cambio tiene consecuencias que producen otros cambios necesarios. En concreto, debe analizarse detenidamente la línea

```
mayor = max(num1, num2);
```

A los dos lados de la igualdad, los tipos deben coincidir. Acabamos de cambiar el tipo de "mayor" y estamos seguros de ese cambio. Es hora de ajustar, consecuentemente, el tipo devuelto por la función max para que sea el mismo que la variable mayor, de lo contrario esa línea no compilará. Los cambios puedes verlos a continuación en rojo.

```
int * max(int a, int b) {  
    if (a > b) return &a;  
    else return &b;  
}  
...  
int num1 = 9;  
int num2 = 11;  
int *mayor;  
  
mayor = max(num1, num2);
```

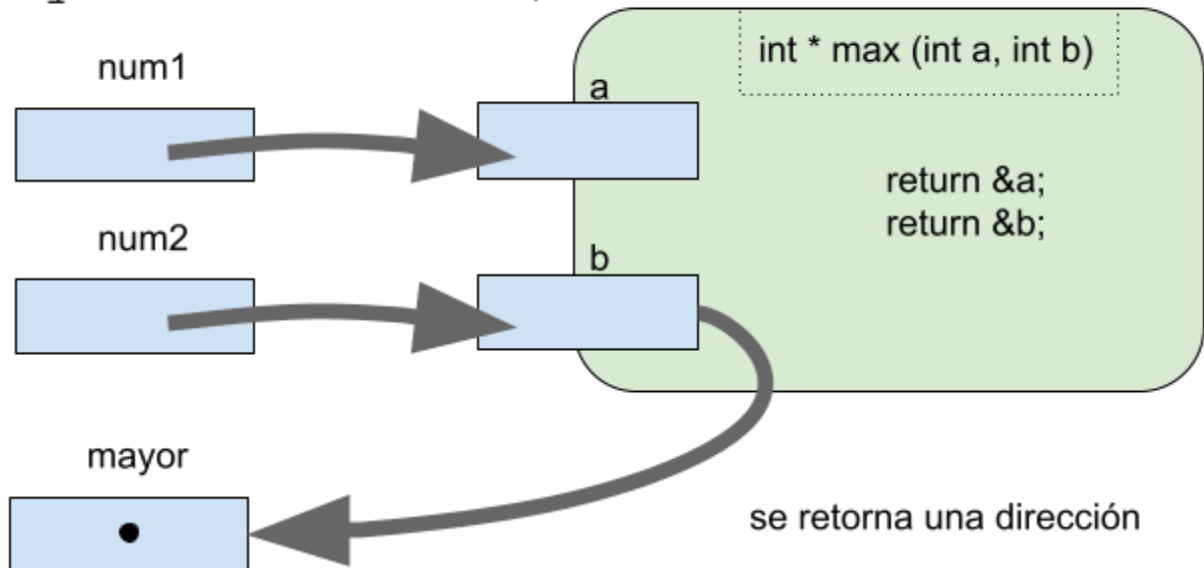
Los cambios se han propagado. Debido al nuevo tipo de la variable mayor, hemos cambiado el tipo de retorno y a su vez, debido a ello, hemos ajustado los "return" para que cumplan con lo que se pretende devolver en la función.

Recuerda:

El tipo devuelto por una función coincide con el tipo de elemento puesto a continuación de `return` y a su vez con la variable a la que se asigna la función en su llamada

La situación de la función y su llamada se muestra en la siguiente imagen:

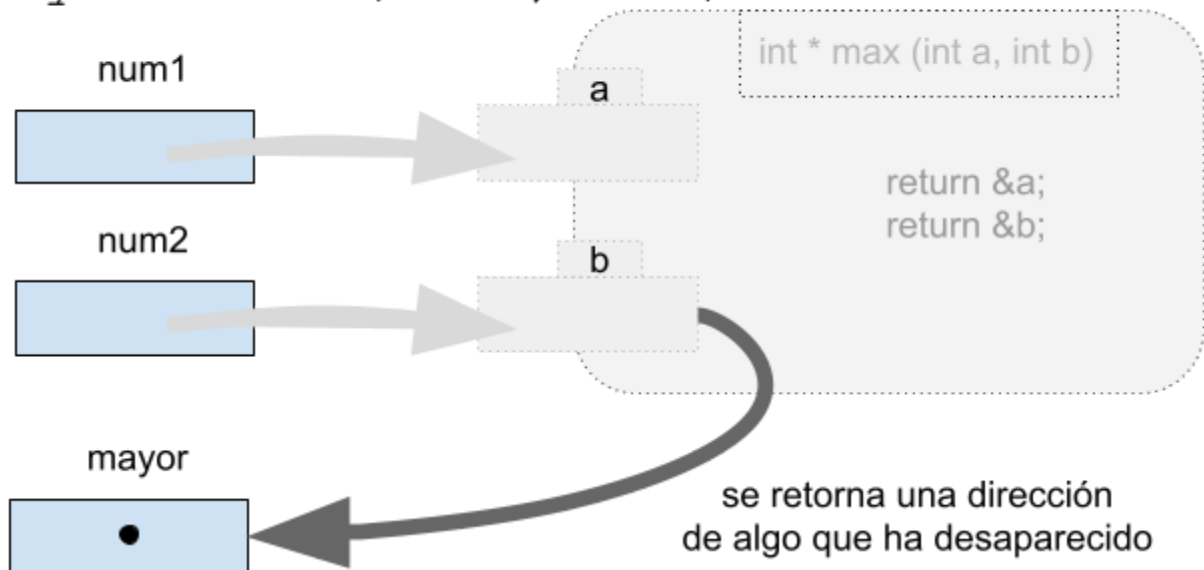

```
int * mayor;  
mayor = max(num1, num2)
```



Quizá puedas darte cuenta. Hay algo que no encaja: devolvemos la dirección de "a" o "b", que son parámetros locales de la función. Hay dos cosas muy mal en esta solución:

- `mayor` no termina apuntando a `num1` o `num2`
- `mayor` termina apuntando a una variable ("a" ó "b") que ¡¡ no existen !! No existen porque cuando se termine de ejecutar la función, las variables `a` y `b` se destruyen; son locales a la función (una variable local sólo existe durante la ejecución de la función donde está).

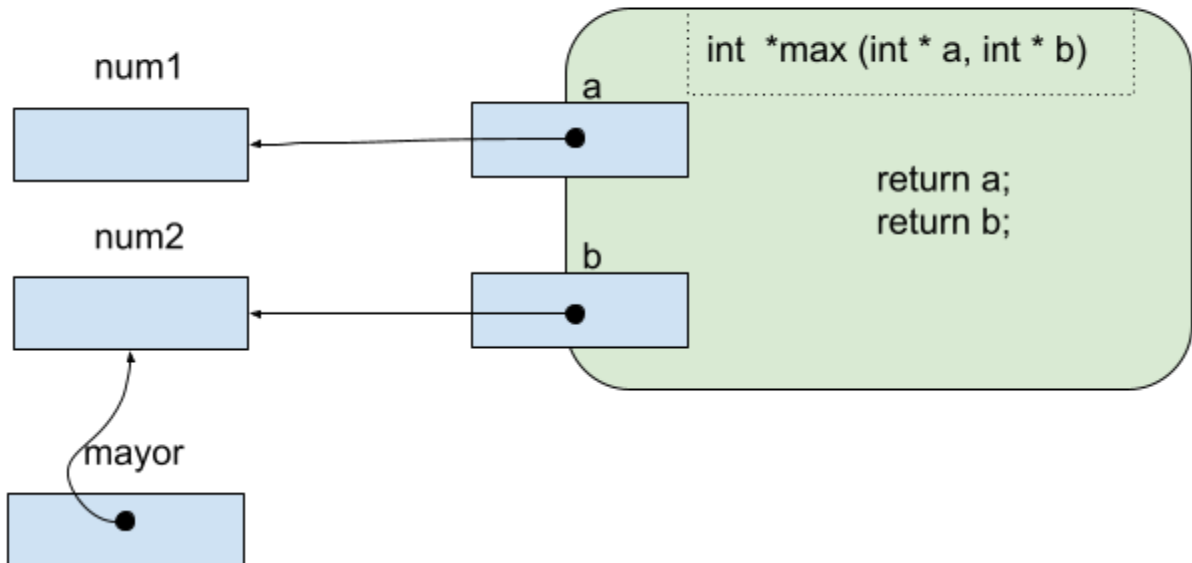
```
int * mayor;  
mayor = max(num1, num2)
```



Hay que replantearse cómo hacer este ejercicio y atacar uno de sus problemas: "Si pasamos copias de valores de las variables a una función, sólo podremos recuperar valores o referencias a variables locales de dicha función". Es necesario, desde el programa principal, pasar las variables en sí, no sus valores y ello sólo se logra con punteros. Los punteros permiten que la función acceda completamente a la variable pasada (y por tanto también a su valor).

La idea a conseguir es expresada en la siguiente imagen.

```
int * mayor;
mayor = max(&num1, &num2)
```



Observa también cómo definimos la función y la usamos. ¡¡ Estamos recibiendo punteros a variables !! Variables que existen fuera de la función.

```
int * max(int *a, int *b) {
    if (*a > *b) return a;
    else return b;
}
...
int num1 = 9;
int num2 = 11;
int *mayor;

mayor = max(&num1, &num2);
```

Ya no pasamos un número, sino que pasamos la dirección de la variable num1 y num2 (eso es lo que consigue el "&"). Dentro de la función se accede al valor de las variables num1 y num2 porque a y b son respectivamente punteros a ellas. Con " *a " accedemos al valor de *lo apuntado por a*. Finalmente, la función devuelve una dirección de memoria, que resulta ser la que tenía el valor más alto. Lo que recogemos en la variable mayor, es la dirección de memoria de num1 o num2. Y sabiendo su dirección podemos acceder a su valor y cambiarlo.

```
mayor = max(&num1, &num2);  
*mayor = 20; //nuevo valor
```

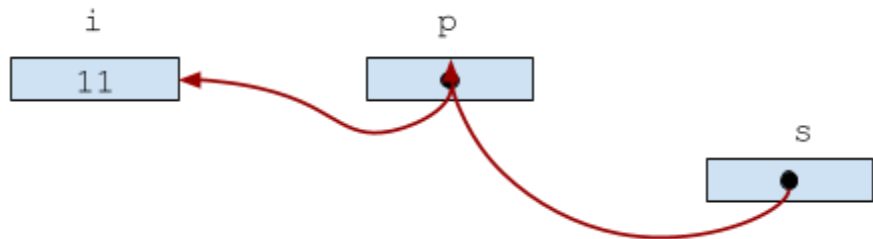
Punteros a punteros (opcional)

Se puede ir más allá con los punteros, ya que existe la posibilidad de utilizar punteros que apuntan a otros punteros. Antes de nada, veámos un ejemplo simple de cómo se utilizan

```
int i;  
int *p;  
int **s;
```

```
i=8 ;  
p = &i;  
s = &p;
```

```
**s = 11;
```



Al igual que muchas veces, esto parece inútil, pero no lo es del todo. En ciertas ocasiones se puede necesitar. Vamos a verlo con el siguiente ejercicio:

Ejercicio, repetir el ejercicio anterior de encontrar el máximo de dos variables pero atendiendo a la llamada siguiente:

```
int num1 = 11 , num2 =9;  
int *mayor;
```

```
max(&num1,&num2,&mayor);  
// mayor debe apuntar ahora a num1
```

Así como antes la función max devolvía un puntero a la variable con el valor más alto, ahora dicho puntero debe ser obtenido pasándolo a la función como argumento: la variable "mayor" se pasa por puntero para que la función cambie el valor de dicha variable y poder recoger desde fuera de la función.

Recuerda el ejemplo anterior con la función "hazNegativo(int *)". Se le ha tenido que pasar un puntero a una variable para permitir que la función pudiese acceder a la variable original y modificar su valor (y no una copia).

Este caso es similar, la variable mayor debe ser alterada. Antes de la ejecución de la función tiene un valor (quizá sin inicializar) pero después de la función debe apuntar a la

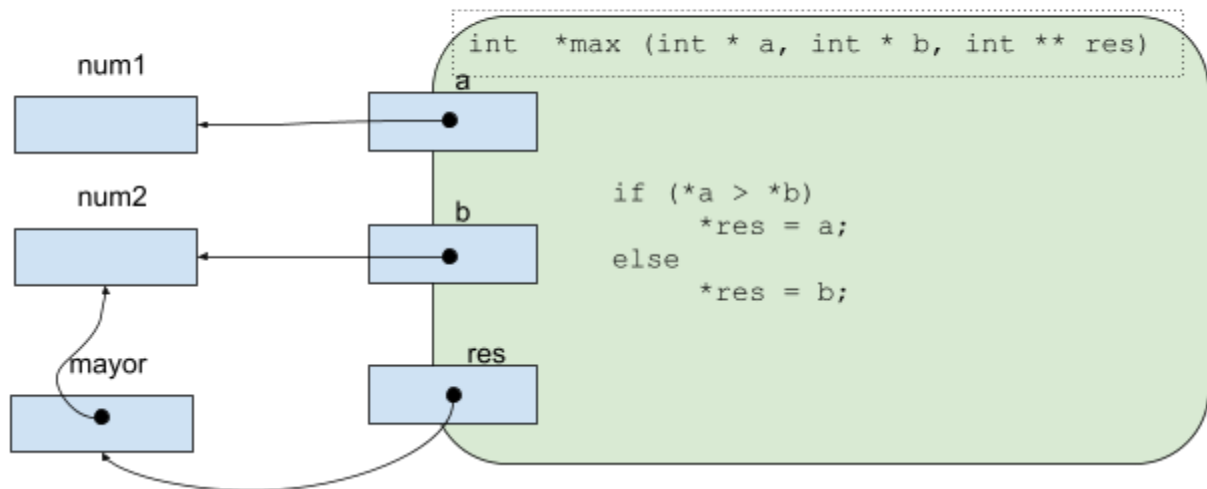
variable num1 ó num2. Igual que con la función `hazNegativo()`, habrá que pasar un puntero a la variable mayor. Para esta circunstancia, da igual que mayor ya sea un puntero. si no pasamos su dirección, desde el interior de la función `mayor()` no se podrá alterar su valor. Fíjate en la llamada cómo se pasa "&mayor".

Ahora, viendo la llamada, podemos hacernos una idea de cómo especificar los argumentos de la función `mayor()` (recuerda: los argumentos coinciden en la especificación y en la llamada)

```
void mayor ( int * a, int * b, int ** resultado)
```

¡Sí! ¡ El argumento "resultado" lleva dos asteriscos delante ! Ello indica que aquí se va a recibir la dirección de una variable cuyo contenido es la dirección de una variable entera. Pasamos la dirección de la variable mayor para cambiar su contenido (la dirección donde apunta).

```
int * mayor;  
mayor = max(&num1, &num2, &mayor)
```



Date cuenta de que dentro de la función " `mayor` " es equivalente a " `*res` "

Referencias

Las referencias son exclusivas de C++ (no aparecen en C como los punteros). Se compilan siempre con el compilador de C++ (g++ en linux). En principio no tienen relación con los punteros desde el punto de vista del programador. Esto no es una continuación de los puntos anteriores.

Una referencia es básicamente un nombre alternativo a una variable o dato existente.

Añadamos una referencia *tonta* para las variables `i` y `j`

```
int main (int argc, char *argv[] ) {  
  
    int i,j;  
  
    int &iAlt = i;  
    int &jAlt = j;
```

¿Qué es la variable iAlt ?....No es más que :

- una variable que es la misma variable que i
- otro nombre para la variable i

El motivo de llamar antes "tonta" a la referencia que se ha creado, es que no aporta nada nuevo, ni tiene utilidad alguna especial. (Como siempre, sigue leyendo y verás la utilidad)

Interpretación del símbolo "& "

El uso del símbolo "&" difiere del visto hasta ahora. Hay que ser muy conscientes de que estamos declarando una variable en las líneas donde lo usamos para diferenciar su efecto de aquellas líneas donde estamos usándolo. En esas circunstancias, el símbolo "&" no significa nada relacionado con la dirección de memoria de nada. Tiene el significado de estar declarando sólo un nuevo nombre de algo existente. **El tipo de la variable declarada es el mismo que tendría si no estuviese el &**

```
1    int i;  
2    int *p = &i;  
  
3    int &k = i;
```

En la línea 2 estamos declarando un puntero p. En esa misma línea estamos asignado a p la dirección de la variable i. Para obtener su dirección usamos el símbolo "&". Observa que ahí justamente no estamos declarando ya nada (la declaración ocurre antes del símbolo "=").

En la línea 3, estamos usando el símbolo "&" en una declaración. Por tanto se trata de una referencia, y estamos indicando que "k" es la misma variable que "i"

El hecho de estar bautizando con otro nombre algo ya existente, es tan esencial que resulta imposible declarar una referencia sin -en la misma línea- indicar la variable que va a referenciar

```
int & iAlt; //-> da error de compilación. La referencia se debe crear sobre algo
```

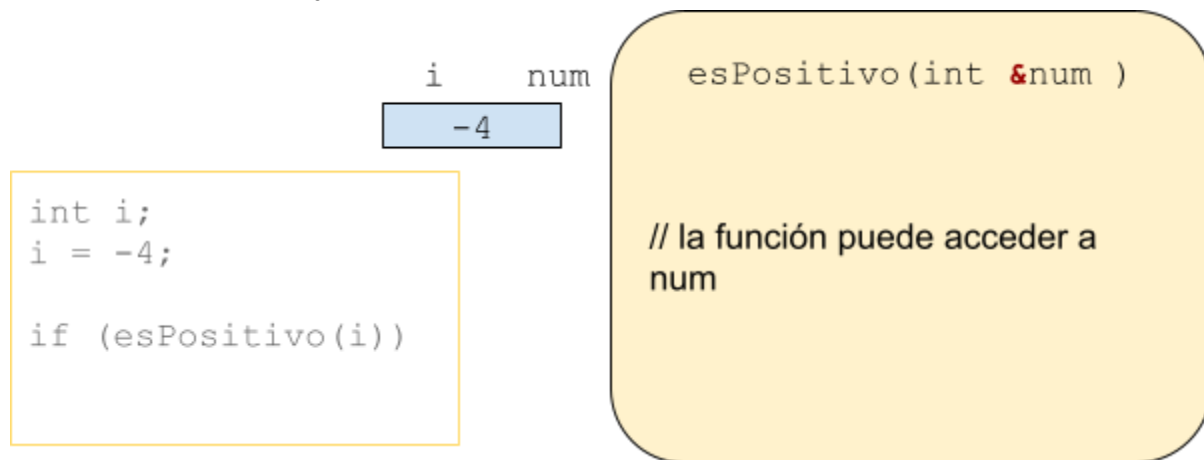
Uso de referencia en funciones

Si sospechas que las referencias son inútiles, ¡Tienes razón! Hasta el momento no sirven para nada. Sin embargo, usándolas como argumentos y valor de retorno en funciones, revelan su utilidad.

Supongamos que declaramos una referencia como argumento a una función. Usaremos la función `esPositivo` conocida

```
bool esPositivo ( int & num) {  
    if (num >= 0) return true;  
}
```

La situación podría dibujarse así:



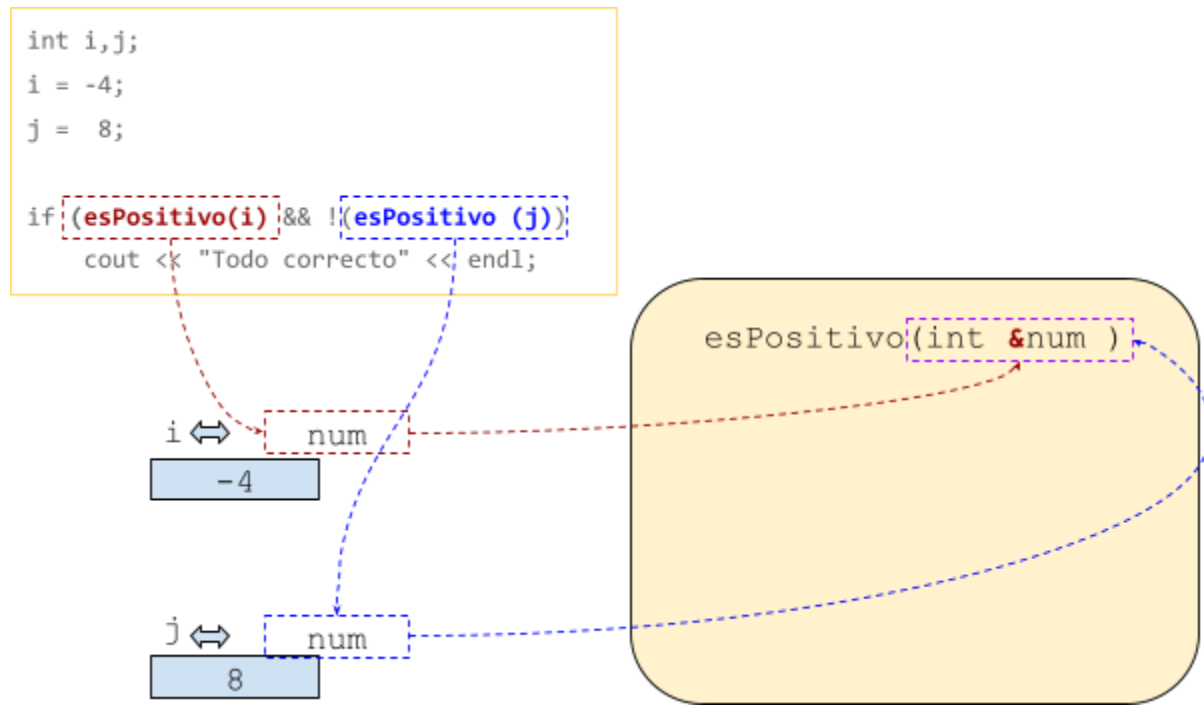
Antes de explicar qué ocurre, hay que decir que resulta difícil dibujar bien la idea del paso de parámetros por referencia. Lo que ocurre es:

- No se crea ninguna variable nueva como siempre había ocurrido antes cuando se declaraban parámetros. Observa que a diferencia de diagramas anteriores, no hay un recuadro azul de variable o puntero creados en la función.
- "num" es un nombre accesible desde la función, pero el valor al que se accede está fuera de la función realmente.
- "num" es la misma variable que `i`. Si cambia `num`, se cambia `i`. inevitablemente

Si hacemos dos llamadas a la función con variables diferentes, `num` será en cada llamada un sinónimo de diferentes variables.

```
int i,j;  
i = -4;  
j = 8;
```

```
if (esPositivo(i) && !(esPositivo (j))
    cout << "Todo correcto" << endl;
```



Paso por referencia para modificar la variable pasada

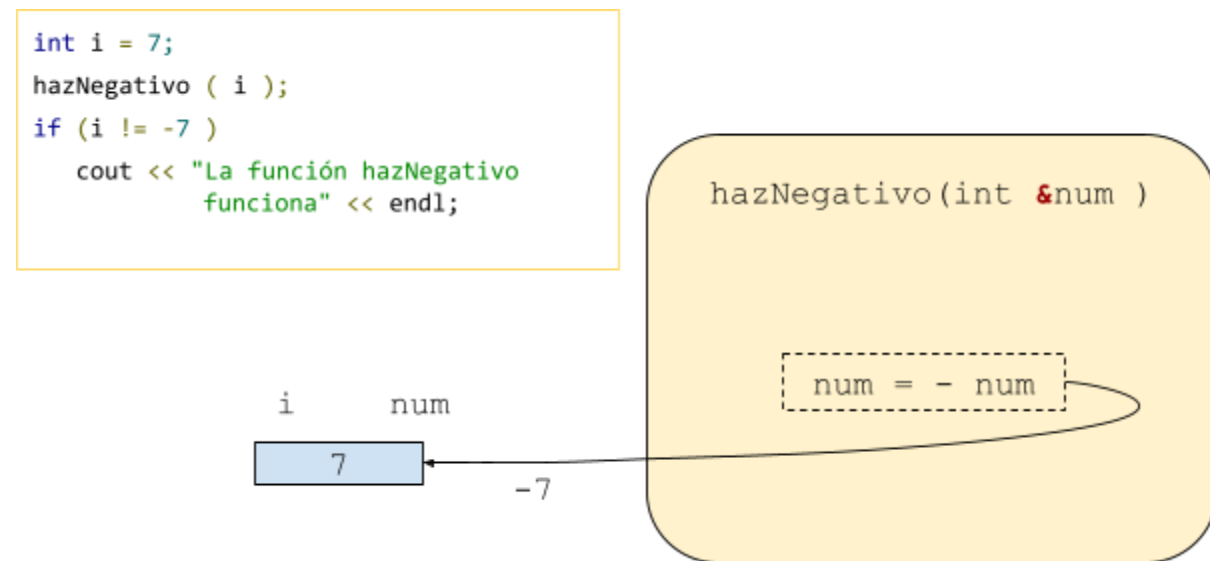
En la función `esPositivo()`, el uso de referencias no aporta ninguna mejora respecto a pasar una copia. Pero en la función `hazNegativo(int)` es clave. Si usamos referencias, podemos alterar el valor de la variable original sin usar punteros.

```
void hazNegativo(int & num) {
    if (num >= 0 )    num = - num ;
}
```

Sea cual sea la llamada, el parámetro `num` será una variable externa a la función accesible desde la función y no una mera copia como ocurre sin referencias. Cambiar el valor de `num` supone cambiar la variable externa.

```
int i = 7;
hazNegativo ( i );
if (i != -7 )
    cout << "La función hazNegativo funciona" << endl;
```


El programa anterior funcionaría como esperamos ya que realmente no estamos pasando una copia de la variable `i`, sino que la estamos haciendo accesible desde la función. Este paso de parámetros no es percible mirando sólo la llamada, hay que verlo en la declaración de la función. Con los punteros sí que se veía explícitamente que estábamos pasando una dirección de memoria ("`hazNegativo (& i)`" era la llamada y se puede deducir que la función recibe un puntero)



Retorno por referencia

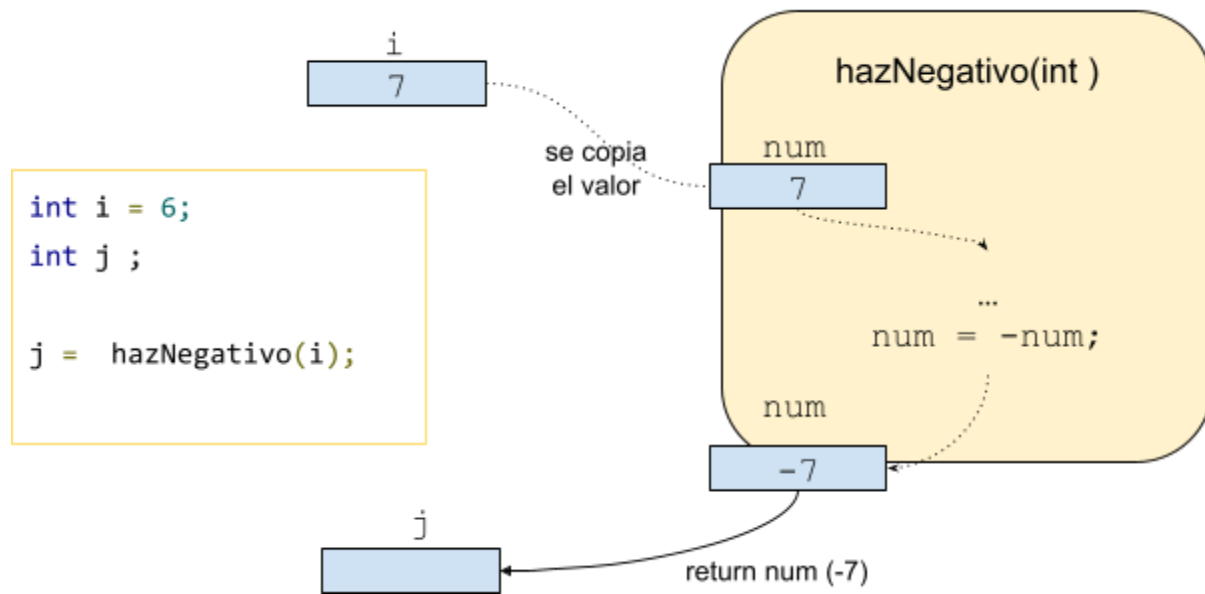
Otro uso de las referencias se da cuando una función retorna una referencia. Hasta ahora, la función que devuelve un valor, realiza una copia en alguna variable. Vamos a ver el caso de la función `hazNegativo(int)` pero con una variante diferente en la que se devuelve el valor negativo del número pasado:

```
int hazNegativo(int num) {
    if (num >= 0 )    num = - num ;
    return num
}
```

La llamada a esta función se realizará de la siguiente manera:

```
int i = 6;
int j ;

j = hazNegativo(i);
```

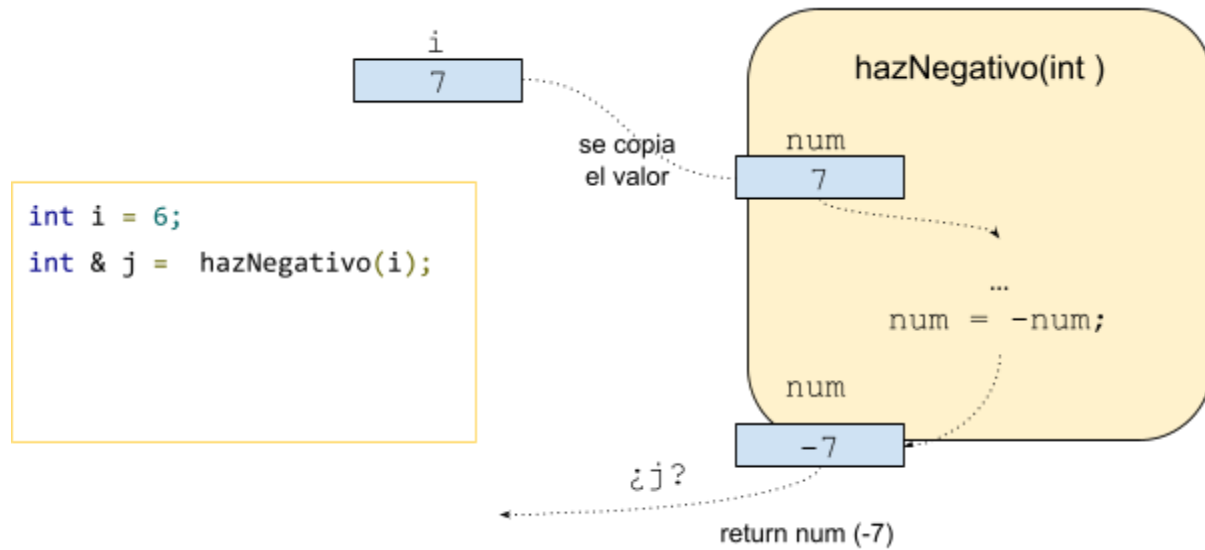


El return hace una copia de valores. El contenido de la variable num es copiado a la variable j. El caso concreto anterior no supone efectivamente ningún cambio respecto devolver un entero con normalidad. Hay un error en la función, y sin embargo la manera en la que se realiza la llamada y la asignación encubren el error y aparentemente todo funciona.

Vamos a cambiar el código y probar algo un poco especial:

```
int i = 6;
int & j = hazNegativo(i);
```

Este caso, j es una referencia a un entero. Es decir, es un nombre alternativo a una variable que ya existe.



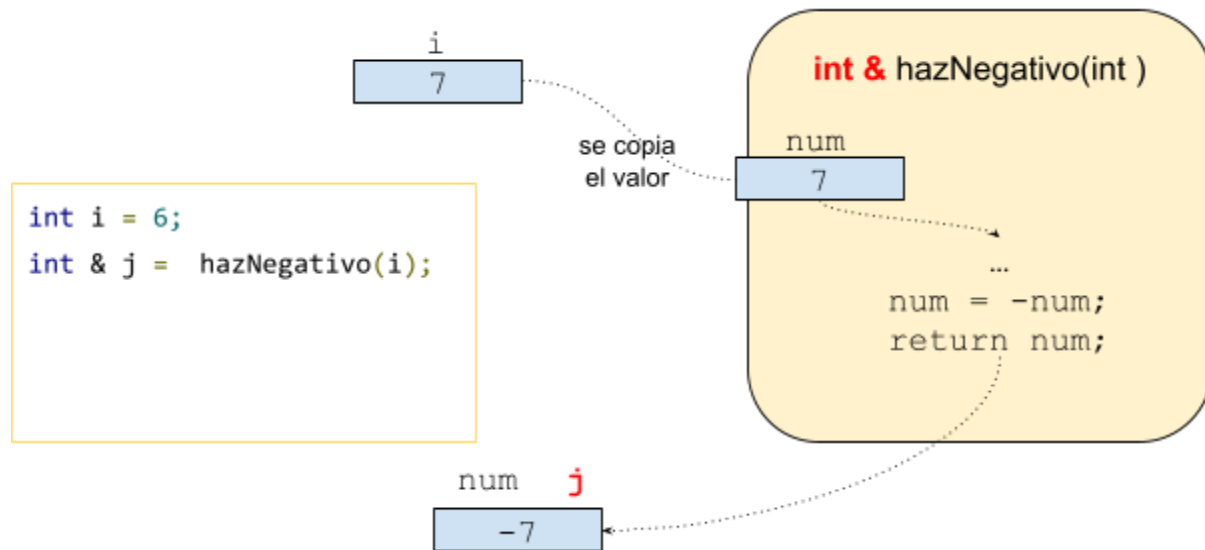
La instrucción `return` hace una copia del valor de `num` y la deposita en la variable `j`. Sin embargo, `j` no es una variable, es sólo un sinónimo de algo... ¿De qué? De un valor que se está copiando desde la variable local `num` a ninguna parte.

Evidentemente esta situación es errónea e indeseable. Aquí la función no debe devolver una copia sino una variable tal cual. Eso es el retorno por referencia. Hagámoslo:

```
int & hazNegativo(int num) {
    if (num >= 0 ) num = - num ;
    return num
}
```

La declaración de la función es muy similar, tan sólo se añade el símbolo `&` después de `int`. La llamada a la función vamos a mantenerla como antes:

```
int i = 6;
int & j = hazNegativo(i);
```



Al devolver por referencia "return num" devuelve "la referencia a la variable". Este hecho combinado con la declaración de `j` como referencia causan un cambio importante. La variable `num` y `j` serán la misma variable. El retorno por referencia aquí se complementa con la referencia `j` que se crea en la propia línea

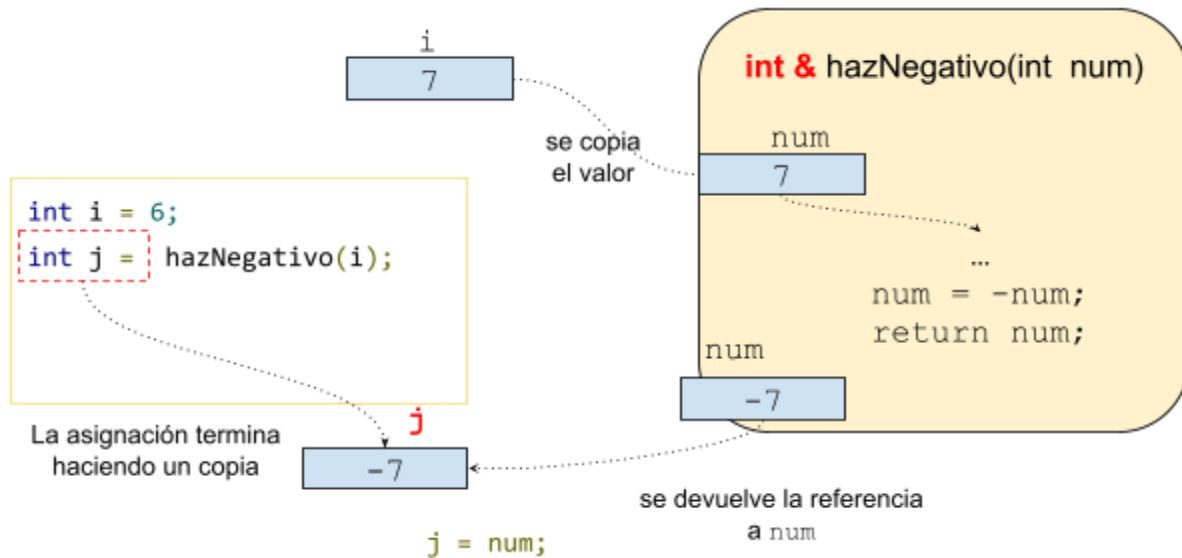
Hay, sin embargo, un gravísimo problema: `num` es una variable local de la función, que es destruida tan pronto la función termina. Por ello, en este ejemplo, `j` es el sinónimo de una variable que deja de existir. Antes de solucionar el problema, vamos a contemplar otro caso, que supone un pasito atrás pero da luz sobre cómo funcionan las referencias (o la ausencia de referencias) y dejamos para después la solución al problema que tenemos entre manos.

```

int & hazNegativo(int num) {
    if (num >= 0 )    num = - num ;
    return num
}

int i = 6;
int j = hazNegativo(i);

```



En este caso no existe el problema anterior de referenciar una variable que desaparece, ya que al ser `j` una variable normal diferente de otras, la copia sí se realiza, aunque se haga de la referencia obtenida por la función `hazNegativo()`.

Retorno de una variable perdurable.

Vamos a finiquitar el asunto (... de momento) haciendo el programa correcto. Recuerda el código puesto antes:

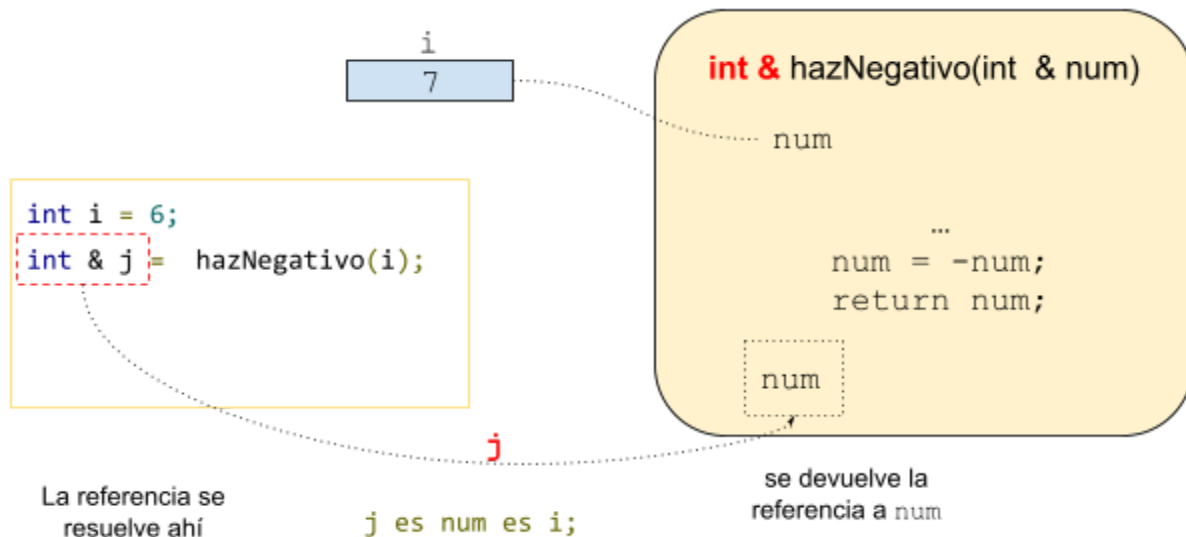
```
int i = 6;
int & j = hazNegativo(i);
```

Párrafo Importante: Sea lo que sea lo que devuelva `hazNegativo(i)`; debe ser algo que perdure, que exista, y no simplemente un valor que se copia. Por ello, hay algo seguro: la función debe devolver una referencia. Pero la referencia debe ser a algo que no se borre al terminar la función. Es decir no se puede devolver una referencia a una variable local a la función o un parámetro, sino que debe devolverse una variable que ya existiese antes de la función y que seguirá existiendo después.

Como se ha visto antes, la única manera de acceder a variables existentes fuera de la función es recibéndolas por referencia como argumento (sin usar punteros). Considera la siguiente función:

```
int & hazNegativo(int & num) {
    if (num >= 0 )    num = - num ;
    return num
}
```

Se devuelve una referencia a la variable num, que resulta ser un argumento recibo por referencia. Es decir "num" es un sinónimo de otra variable externa a la función y ella es la que se devuelve por referencia.



La historia es bonita: `i` es pasada por referencia. Por tanto "num" es la misma variable que `i`. Al ejecutar `return`, se devuelve la referencia a `num` y ésta referencia se usa para inicializar `j` (que es otra referencia). Por tanto "j" es "num" (que es "i").

Un ejemplo completo

El ejemplo completo de referencias se puede realizar con la función `max` anterior

```
int & max(int & a, int& b) {
    if (a > b) return a;
    else return b;
}

int num1 = 9;
int num2 = 11;
int & mayor = max (num1, num2);
```

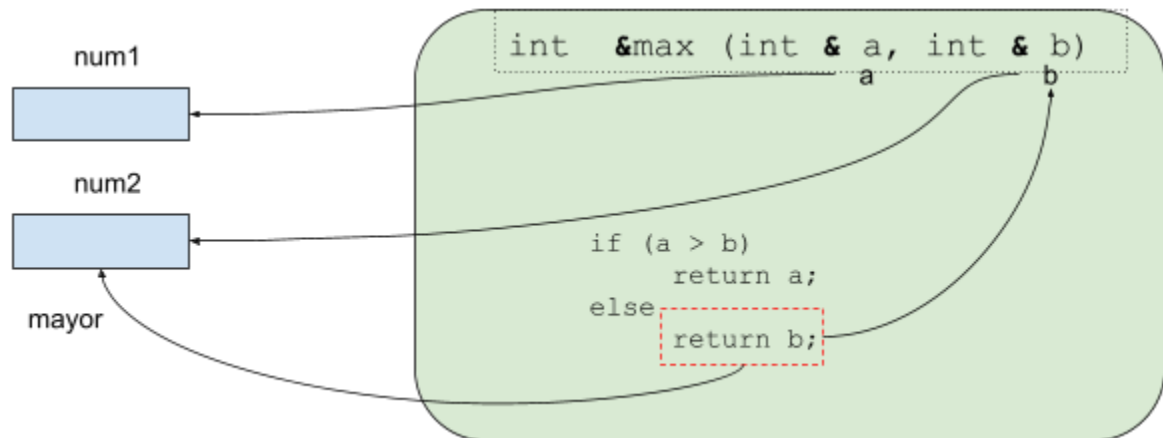
`mayor` es una referencia que será `num1` o `num2` dependiendo de cuál sea la variable con valor más alto.

```

int & max(int & a, int& b) {
    if (a > b) return a;
    else return b;
}

int num1 = 9;
int num2 = 11;
int & mayor = max (num1, num2);

```



Las claves son:

- La función recibe dos referencias, por lo que no se crean variables nuevas para los argumentos, sino que se usan variables ya existentes (`num1` y `num2`) que se rebautizan dentro de la función como "a" y "b"
- La función devuelve por referencia, bien "a" o bien "b". Estas son variables externas que existirán después de la ejecución de esta función
- `mayor` es una referencia que es inicializada a lo que la función devuelve, que es una referencia a una variable existente anterior (`num1` o `num2`). Por tanto "mayor" termina referenciando a `num1` o a `num2`

Uso de punteros para referenciar a objetos creados durante la ejecución

Si recapacitas sobre los ejemplos anteriores en los que se han tratado los punteros, observarás que hasta ahora, éstos se han usado para apuntar a otras variables del programa. El puntero puede cambiar de apuntar de una variable a otra.

```

int a=8, b=6;
int * p = &a;
p = &b ;

```

Hemos encontrado utilidad en los punteros para varios menesteres. Sin embargo, el principal uso de los punteros todavía está por venir, cuando vemos que los punteros pueden apuntar a elementos no asignados o referenciados por ninguna variable

Estructuras y Vectores estáticos

Cuando un programador realiza un programa, elige manipular variables que almacenarán los datos del programa. Estas variables, en ocasiones, son vectores y estructuras de datos complejas, incluso mezcladas. Observa el siguiente ejemplo

```
typedef struct {  
    int numeroDeCuenta;  
    float saldo;  
    string nombreDelCliente;  
} CuentaBancaria;  
  
CuentaBancaria cuentas[10];
```

El anterior ejemplo muestra la creación de los datos necesarios en un programa que gestiona las cuentas bancarias de un imaginario banco. Primero, hemos creado una estructura de datos para almacenar todos los datos necesarios que conciernen a una cuenta bancaria. Seguidamente, hemos creado un vector de 10 elementos, cada uno de los cuales es una cuenta bancaria tal como hemos definido antes.



Si tomamos en serio este código, veremos que la declaración del vector crea una limitación o un gasto inútil de memoria. En el vector, se crean 10 cuentas bancarias en memoria. Considera que en un ejemplo más real, podrían ser muchas más, hoy en día ninguna oficina bancaria tiene como máximo 10 clientes. Por tanto, el vector seguramente se quede pequeño para las necesidades del programa. Si elegimos curarnos en salud y hacer el vector mucho más grande, provocaremos por otra parte, un desperdicio importante de memoria. Ya que al dimensionar el vector exageradamente grande, con millones de elementos, ocuparemos muchísima memoria que probablemente no sea utilizada.

Este problema no se tendría si el programador supiese cuántos clientes exactamente va a tener el banco durante toda su existencia. Evidentemente, ni el programador puede saberlo, ni preverlo, ni el número de clientes permanecerá constante. Por ello la solución a este problema exige una gestión de la memoria y de las variables diferente. La idea es permitir crear más "cuentas" conforme haga falta durante la ejecución, sin haber previsto un número concreto durante la programación o compilación.

Creación dinámica de objetos

La idea básica que se toma como punto de partida es la siguiente

```
CuentaBancaria *p;
```

Ahora **p es un puntero que no apunta a ningún sitio**. Sin embargo, sabemos que podríamos hacer que p apuntase a una Cuenta. Pero para ello deberíamos declarar una variable de tipo Cuenta y después asignar a p la dirección de la variable

```
CuentaBancaria *p;  
CuentaBancaria unaCuenta;
```

```
p = & unaCuenta;
```

Esta declaración anterior tiene un problema muy serio. Supón, por ejemplo, que está dentro de una función, llamada nuevaCuenta que sirve para crear una cuenta nueva y devolverla por puntero.

```
Cuenta * nuevaCuenta() {  
    /* ...      declaramos variables y preguntamos  
                al usuario el número de cuenta y  
                titular d ela nueva cuenta a crear */  
  
    Cuenta c;  
    c.titular = nuevotitular; c.numCuenta = nuevoNumcuenta; c.saldo  
    = 0;  
  
    return &c;  
}
```

El programa no funcionará, porque la declaración

```
Cuenta c;
```

Crea una variable que sólo existe durante la ejecución de la función. Y por tanto, cuando se ejecute

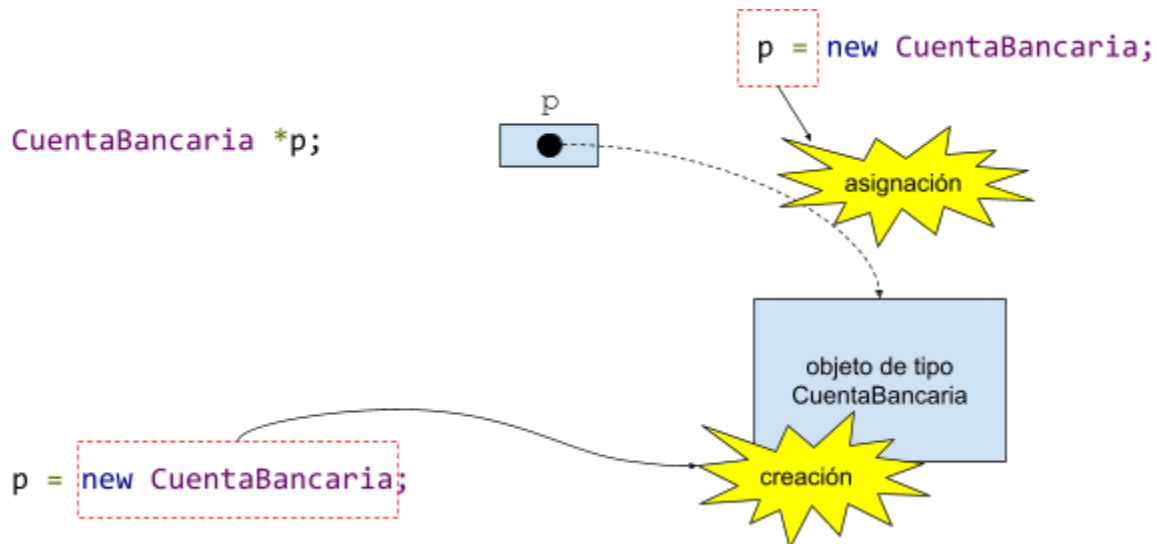
```
return &c;
```

se está devolviendo la dirección de un objeto que va a desaparecer.

Por suerte, podemos hacer que p apunte a una cuenta que nunca antes haya sido una variable y que sólo exista desde el momento que hayamos decidido crearla con el añadido de que no se destruirá al acabar la función, sino cuando el programador quiera. Observa el siguiente ejemplo:

```
CuentaBancaria *p;  
p = new CuentaBancaria;
```

La siguiente imagen trata de describir el comportamiento de las líneas anteriores



"new" crea un objeto de tipo CuentaBancaria (al igual que hace `CuentaBancaria unaCuenta;` del ejemplo anterior). `new` devuelve la dirección de memoria donde se ha creado ese objeto de tipo CuentaBancaria. Ese valor retornado se almacena en `p`. Por ello `p` ya apunta a un objeto de tipo cuenta. Si no hubiésemos ejecutado la línea `"p = new CuentaBancaria;"`, `p` no habría podido apuntar a ningún Objeto de Tipo Cuenta

Recuerda, si creas un objeto con `new` se devuelve un puntero

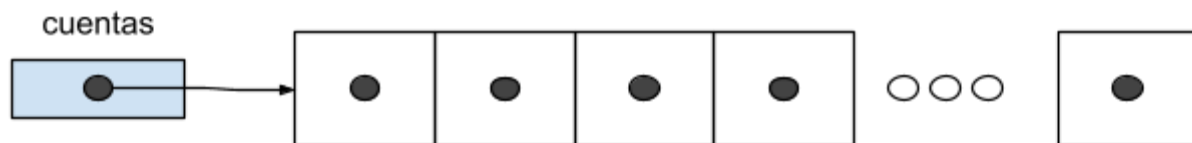
Recuerda, si creas un objeto con `new`, éste no se destruye hasta que tú no lo destruyas,

Recuerda, si creas un objeto con new, éste no se destruye hasta que tú no lo destruyas,

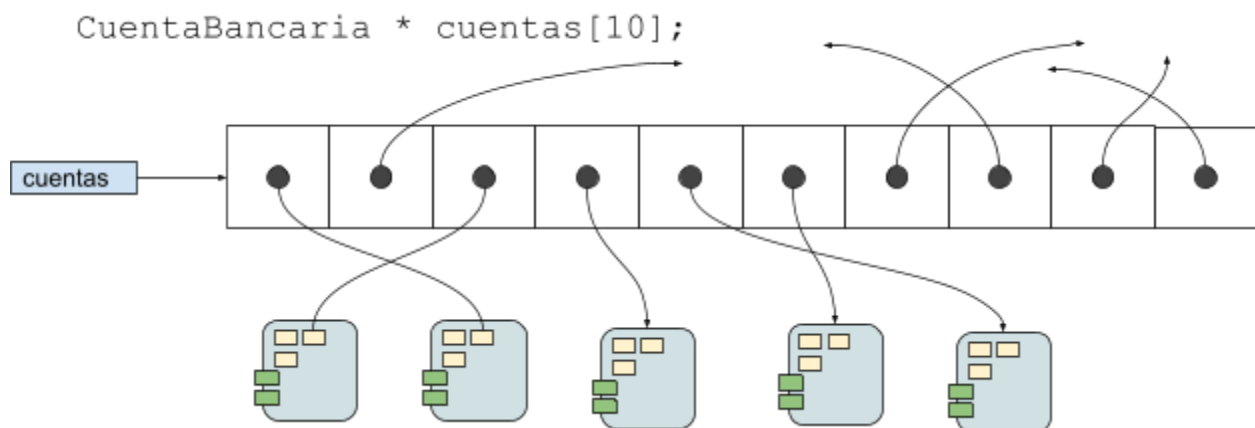
Vectores de punteros

La idea ya vista de tener sólo un puntero y crear el objeto en el momento que haga falta, puede ampliarse si concebimos *vectores de punteros*. Por ejemplo, en el problema del programa para la gestión de una oficina bancaria, se podría crear un vector de punteros (un vector en el que cada elemento es un puntero).

```
CuentaBancaria * cuentas[10];
```



De esta forma, al iniciar el proceso, no se crea ninguna cuenta, pero conforme se va ejecutando el programa, pueden ir creándose cuentas nuevas y éstas estarán referenciadas desde el vector. Llegado un momento, por ejemplo, nos podríamos encontrar con un vector algunos de cuyos elementos referencian a objetos estructura de tipo CuentaBancaria



El diagrama anterior refleja el estado del vector en algún momento de la ejecución. Aunque el vector tiene 10 elementos (desde el inicio del proceso), tan sólo hay en uso cinco cuentas, las cuales están referenciadas desde elementos del vector. Así no desperdiciamos memoria creando estructuras u objetos que no se usan al principio del programa

¿Cómo se llega hasta ahí? Veamos los pasos para tener esta situación o una similar-

Declaración del vector

```
CuentaBancarias * cuentas[10];
```

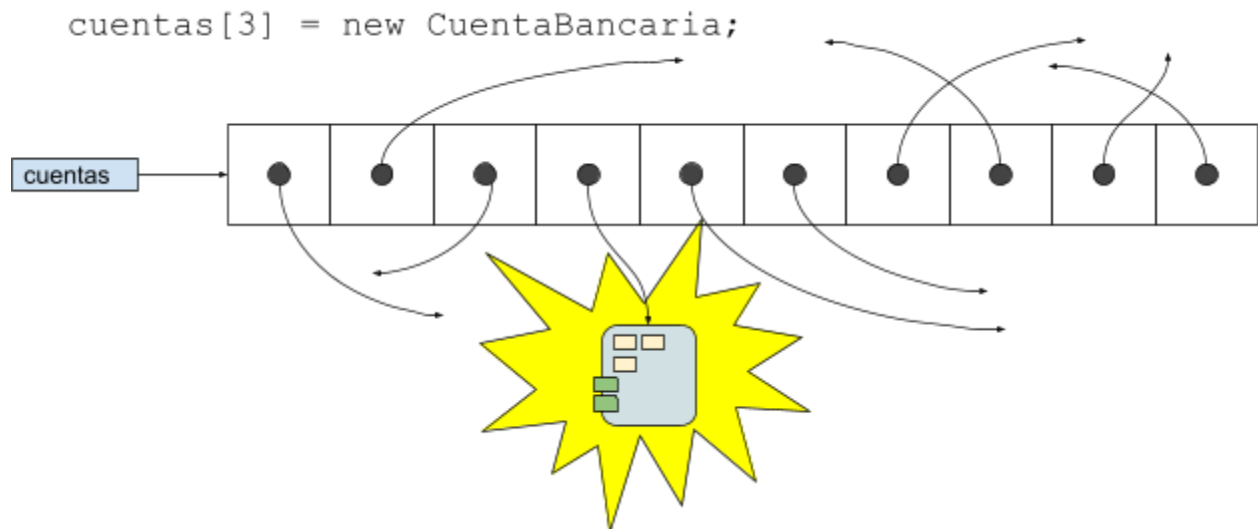
Esto es novedoso, cuentas es un vector porque al declararse lleva "[10]" a continuación. Esta es la manera de declarar un vector de tamaño conocido. El tipo de cada elemento es lo que aparece a la izquierda de la palabra cuentas. Esto es: "CuentaBancaria *". Se trata de un puntero a una estructura de tipo CuentaBancaria. Cada elemento es un puntero.

Es conveniente que incluso siendo punteros, los elementos del vector estén inicializados desde el primer momento. `NULL` es una dirección de memoria especial que indica un puntero nulo, inválido o cero. La inicialización puede hacerse así:

```
for (int i=0; i < 10 ; i++ )  
    cuentas[i] = NULL;
```

A continuación podemos empezar a crear cuentas y asignarlas a elementos del vector. Veámos una sola creación de una estructura u objeto y su asignación a un elemento cualquiera

```
cuentas[3] = new CuentaBancaria();
```



Si en algún momento, una cuenta deja de existir, se deberá borrar de la memoria... lo que también se llama "destruir" el objeto.

```
delete cuentas[3];  
cuentas[3] = NULL;
```

Visibilidad de los objetos y desperdicio de memoria

Hay ocasiones en las que se puede llegar a usar mal el operador `new` o la creación de memoria. Considera el siguiente código situado en medio de un programa o función.

```
new CuentaBancaria;
```

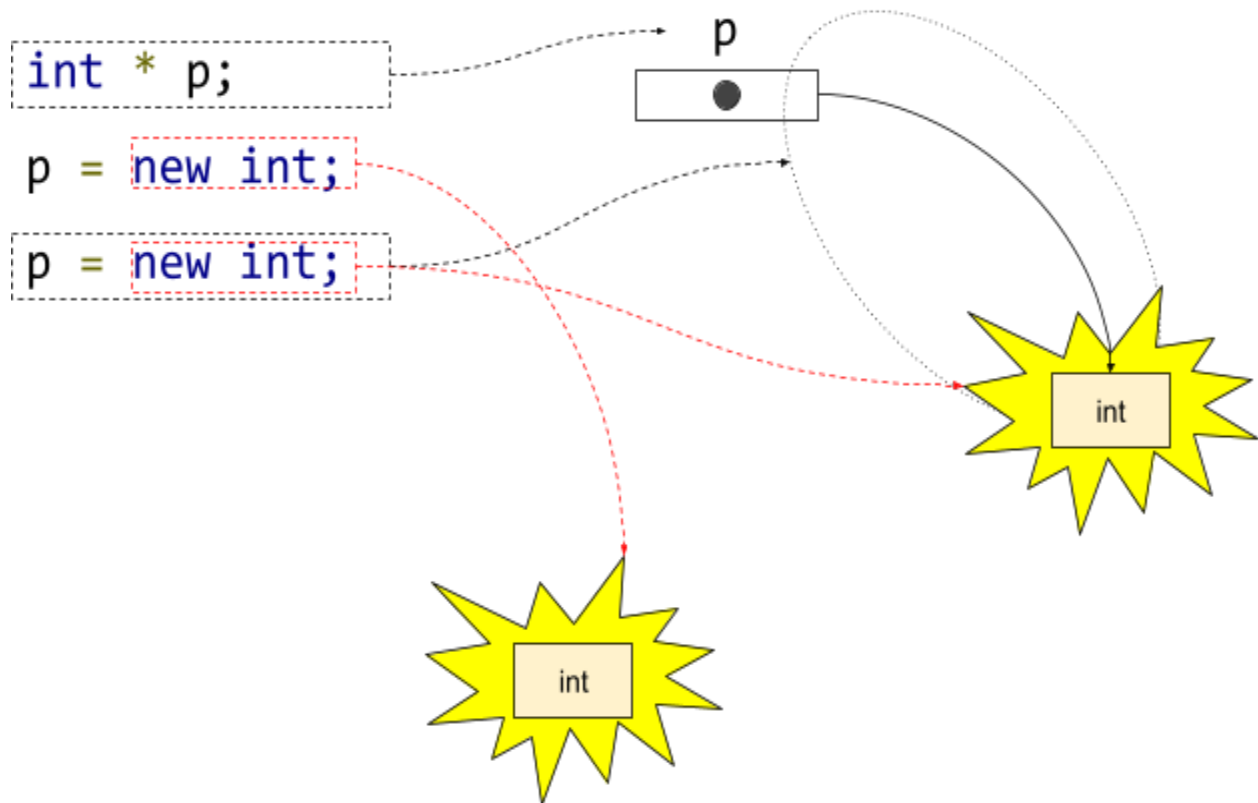
El efecto de esa línea es ninguno desde el punto de vista del programador o de la solución. Sin embargo, el efecto en la memoria del programa no es despreciable. En esa línea se crea un objeto nuevo, una nueva Cuenta Bancaria. Sin embargo, no es referenciada por ningún puntero o ninguna variable. No se puede acceder a ella. Es algo similar a declarar una variable que no se usa, solo qué:

- Esta cuenta no se puede usar, la variable podría llegar a usarse cambiando el programa
- Esta cuenta se ha creado y no se borrará ella sola. Una variable se borra cuando termina el bloque donde fue declarada (Por ejemplo, cuando termina la función donde está declarada)
- Esta cuenta no podrá ser borrada por el programador tampoco, porque no se conoce su dirección.

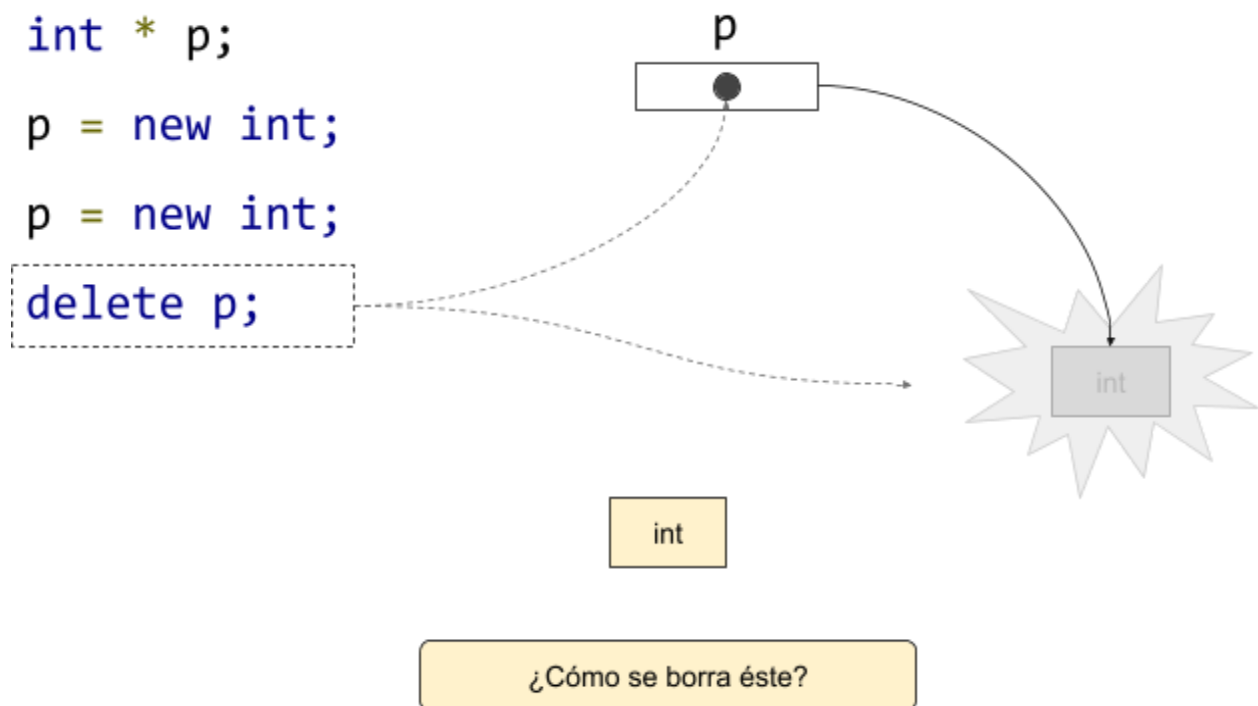
Algo similar ocurre en el siguiente ejemplo (esta vez con enteros):

```
int * p;  
p = new int;  
p = new int;  
  
delete p;
```

Se crean dos enteros y su dirección se almacena en `p`, pero el segundo de ellos hace que perdamos la dirección del primero



Ante esta situación, acceder al elemento no apuntado por `p` va a ser imposible y por ello no hay manera de borrar ese elemento



----- Lo que está en gris, , en construcción. -----

Referencias y punteros mezclados.

La explicación sobre referencias y punteros podría prolongarse. Poco a poco se podría ir viendo cosas más difíciles. Pero con lo aprendido hasta aquí, se puede seguir descubriendo poco a poco.

Continuará . . .

----- EL RESTO ES BASURA COPYPASTEABLE POR MI -----

El programa apenas tiene nada más interesante.

```
#include <stdio.h>
#include <iostream>

using namespace std;

int * max( int * uno, int * dos) {
    if (*uno > *dos) return uno;
    return dos;
}

int main (int argc, char *argv[] ) {

    int i,j;

    int &iAlternativa = i;
    int &jAlternativa = j;

    iAlternativa = 8;
    jAlternativa = 11;

    int * p;

    p = max(&iAlternativa,&jAlternativa);

    *p += 2 ;
    cout << "j está en " << &jAlternativa << " y j vale " <<jAlternativa <<
endl;
    cout << "p apunta a " << p<< " *p vale " << *p << endl;
```

```

    cout << "p está en " << &p << endl;

    cout << "i está en " << &i << " iAlternativa en " << &iAlternativa <<
endl;

}

```

El programa anterior es el mismo, todo lo que hemos podido hacer es usar los nuevos nombres de las mismas variables que antes

Paso por referencia

El verdadero poder de las referencias está en usarlas en el paso de parámetros. Cuando declaro

```

#include <stdio.h>
#include <iostream>

using namespace std;

int * max( int & uno, int & dos) {
    if (uno > dos) return & uno;
    return & dos;
}

int main (int argc, char *argv[] ) {

    int i,j;
    i = 8;
    j = 11;

    int * p;

    p = max(i,j);
    *p += 2 ;
    cout << "j está en " << &j << " y j vale " <<j << endl;
    cout << "p apunta a " << p<< " *p vale " << *p << endl;
    cout << "p está en " << &p << endl;
}

```

TOTAAAAAL

Haz dibujo, soperro


```
#include <stdio.h>
#include <iostream>

using namespace std;

int & max( int & uno, int & dos) {
    if (uno > dos) return  uno;
    return  dos;

}

int main (int argc, char *argv[] ) {

    int i,j;
    i = 8;
    j = 11;

    int * p;

    int & numMayor = max(i,j);

    p = & numMayor;
    *p += 2 ;
    cout << "j está en " << &j << " y j vale " << j << endl;
    cout << "p apunta a " << p<< " *p vale " << *p << endl;
    cout << "p está en " << &p << endl;
}
```

Fin del documento

Uso de matrices o vectores tridimensionales con creación dinámica del espacio en memoria

Las matrices o vectores tridimensionales son estructuras de datos de tipo vector en el que existen tres índices para acceder a un elemento.

Matrices o vectores 3D estáticos.

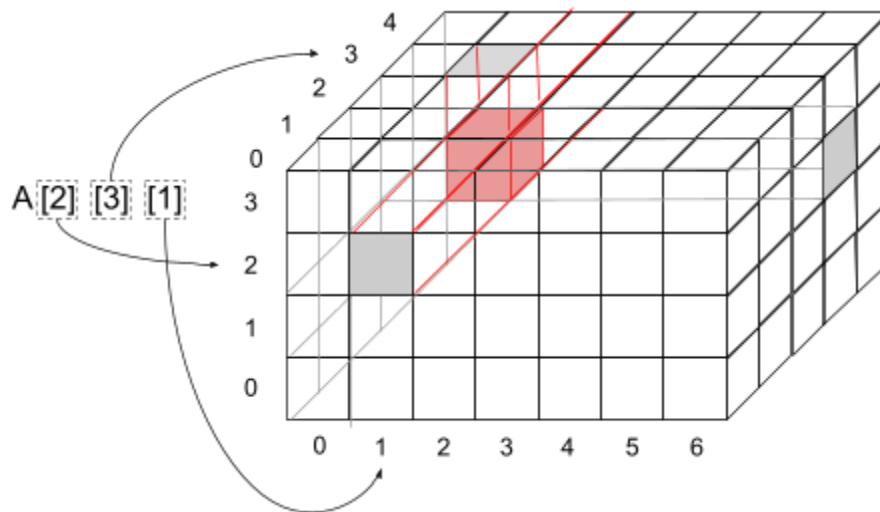
Las matrices o vectores 3D estáticos se caracterizan porque los datos o espacio en memoria se reserva al iniciar el programa como cualquier otra variable declarada en el código fuente. Es decir: el compilador sabe las dimensiones del vector y genera un programa que reserva la memoria necesaria para albergar todos los elementos del vector cuando se inicia el proceso. La declaración básica podría ser:

```
int A[4][7][5];
```

Una vez iniciado el proceso, acceder a los elementos del vector puede hacerse de la siguiente manera en ejemplos:

```
A[2][3][4] = 12;  
if ( A[2][2][3]++ < A[1][0][0] )
```

En el caso anterior "A" representa la matriz o vector 3D. Puede dibujarse una representación gráfica de dichos datos y también del elemento que está siendo asignado en ese ejemplo:

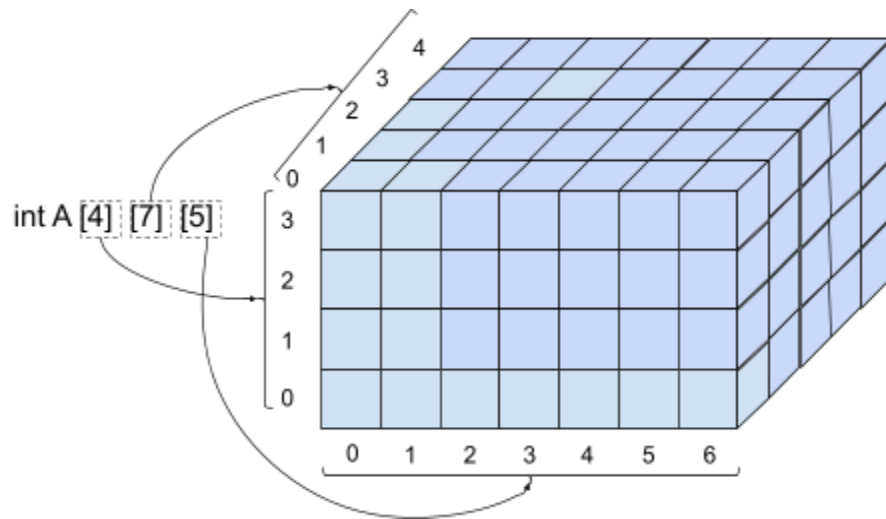


Con `A[2][3][1]` Estamos seleccionando un elemento entre un montón de elementos que se organizan por tres índices, cada uno de los cuales tendrá un límite diferente. La matriz o vector 3D no tiene por qué ser cúbica.

Por tanto la declaración siguiente:

```
int A[4][5][7];
```

genera un bloque de datos que podemos representar como sigue:



Como ejemplo, inicializar una matriz o vector 3d con valores aleatorios entre 0 y 999 se podría hacer

```
static const dimX = 4;
static const dimY = 7;
static const dimZ = 5;

int A[dimX][dimY][dimZ];

for (int i = 0; i < dimX; i++)
    for (int j; j < dimY; j++)
        for (int k ; k < dimZ; k++)
            A[i][j][k] = rand() % 1000 ;
```

Vector tridimensional dinámico.

De momento no supone mayor dificultad trabajar con vectores tridimensionales. El problema, sin embargo, viene cuando no sabemos las dimensiones de la matriz en el momento de hacer el programa y queremos que, durante su ejecución, determine cuáles van a ser cada una de las tres dimensiones. Ya no podemos recurrir a declarar la matriz como antes. Hay que crear la matriz en tiempo de ejecución, y obtendremos un objeto en memoria útil como el vector tridimensional anterior, pero diferente en cuanto a su declaración y número de dimensiones o elementos.

Este mismo problema se tiene en ocasiones con los vectores unidimensionales. Y la solución consiste en declarar el vector sin dimensiones y asignar o reservar un espacio de memoria durante la ejecución:

```

int datos[];
int numeroDatos ;

// inicializar numeroDatos a algún valor

// dos alternativas posibles
datos = malloc (numeroDatos * sizeof(int));
datos = new int[numeroDatos]

for (int i=0; i<numeroDatos; i++)
    datos[i] = ....

```

(Si no lo sabes, `int datos[];` declara realmente un puntero a `int`, al igual que `int * datos;`)

Para ampliar esta idea a una matriz o vector de tres dimensiones, una idea es pensar en la siguiente declaración que declara esa misma matriz pero sin indicar las dimensiones, con lo que es de esperar que A sea un puntero

```
int A[][][];
```

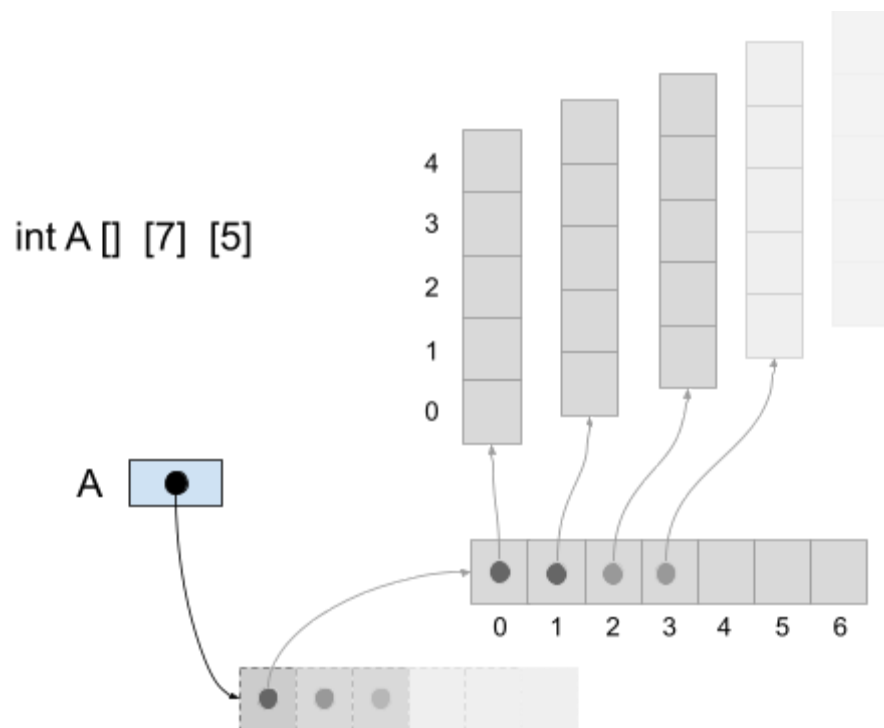
Esta declaración anterior es concebible en tu cabeza si has visto que los vectores unidimensionales sin tamaño predefinido pueden declararse así:

```
int A[];
int *A;
```

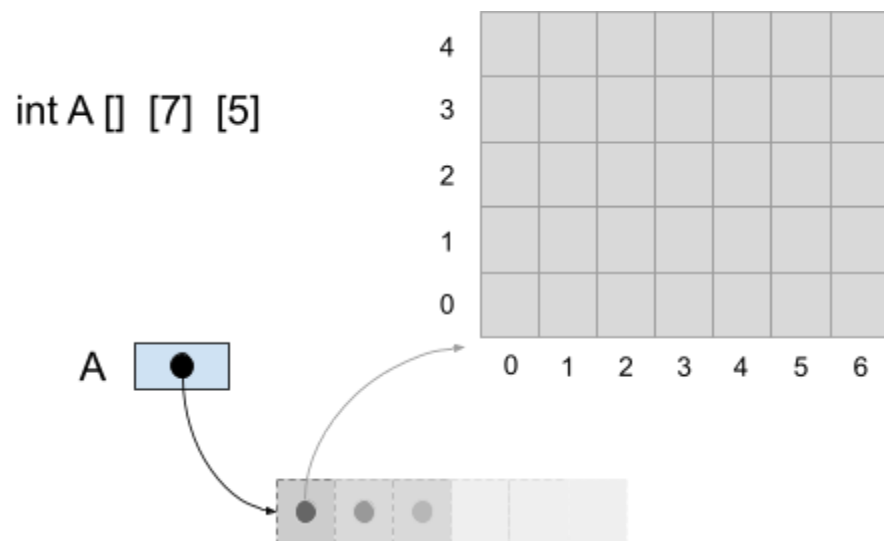
Lamentablemente, en la declaración `"int A[][][];"` el compilador protesta e indica que todas excepto la primera dimensión necesitan estar especificadas. Es decir, sí que podríamos hacer esta declaración así:

```
int A[][5][7];
```

De esta forma el compilador creará una variable llamada "A" que es un puntero a elementos del tipo `int[5][7]` sin que se cree realmente ningún otro elemento. En el siguiente diagrama se refleja lo que se crea realmente (en azul) y los tipos de las variables no creadas (en gris)



En el diagrama anterior se refleja algo importante. El vector A (apuntado por el puntero "A") no tiene tamaño, no sabemos cuántos elementos tendrá. Pero los otros dos vectores, que son apuntados desde elementos de A, sí. Estos vectores sólo pueden ser bidimensionales y de 7x5 elementos.



Es decir los elementos del vector A deben ser vectores bidimensionales de 5x7 de tamaño, y no otros.

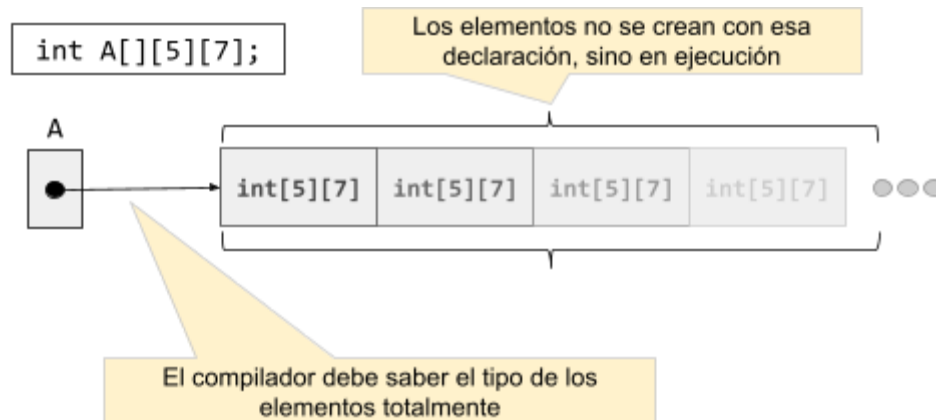
El problema es que estamos decidiendo el tamaño en tiempo de ejecución, por tanto esta aproximación no nos sirve porque aquí fijamos ya unas dimensiones que sólo sabremos

durante el tiempo de ejecución, aunque es cierto que aquí no se creaba ningún elemento o elementos del vector sino simplemente el puntero. En resumen, la definición es similar a :

```
int **A[];
```

Sólo que ahí, el vector es de números, pero no se crea ningún elemento, tan sólo un puntero "int *" llamado A.

En resumen podríamos decir, que a la hora de crear vectores de varias dimensiones, el compilador admite que se cree un vector vacío (lo que equivale a un puntero). Pero no admite desconocer el tamaño y tipo de los elementos del vector. El puntero debe apuntar a algo conocido durante la compilación, aunque no exista todavía.



"Hay que hacer un vector cuyos elementos son vectores cuyos elementos son vectores". En el ejemplo del diagrama anterior habría que hacer algo similar a estos pasos:

```
1 int ** matriz[4];    // se crean un vector de 4 elementos
2 int * vector2D[7];   // se crea un vector de 7 elementos
3 int vector[5];
```

En la línea 1 se crea un vector (llamado "matriz") de 4 elementos, cada uno de los cuales es un puntero a puntero a entero. Por tanto el vector matriz es una variable de tipo puntero a puntero a puntero a entero.

En la línea 2 se crea un vector cuyos elementos apuntan a un puntero a entero. Este vector (llamado "vector2D") tiene 7 elementos y resulta ser una variable de tipo puntero a puntero a entero.

Finalmente, en la línea 3, se crea un vector (llamado "vector") de cinco elementos que son enteros. Por ello, la variable "vector" es en el fondo un puntero a enteros.

¿Qué relación pueden tener las variables anteriores? Para entenderlo, veamos antes cuáles son sus definiciones sinónimas

declaración

```
1 int ** vector3D[4];
2 int * vector2D[7];
```

equivalencia

```
int *** vector3D;
int ** vector2D;
```

```
3 int vector[5];           int * vector;
```

Ahora podemos combinar esas definiciones . Una fila de números se podría construir dentro de la matriz así

```
vector2D[3] = vector;
```

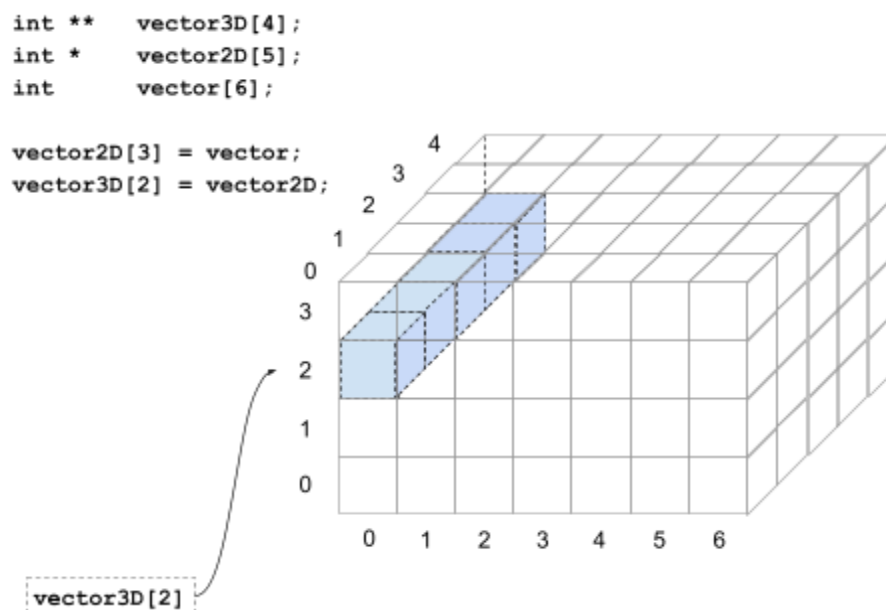
Ya que cada elemento de vector2D es un puntero a entero, o un vector unidimensional.

Un vector bidimensional perteneciente a la matriz 3D se podría construir así:

```
vector3D[2] = vector2D;
```

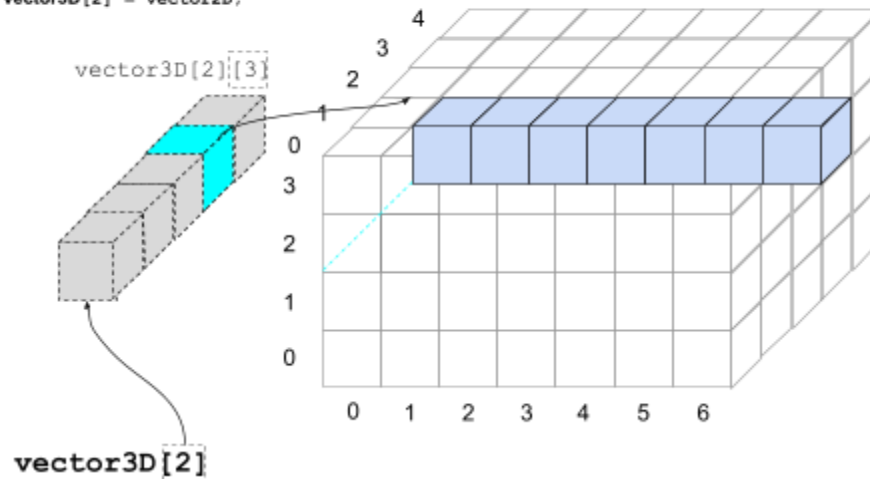
Ya que cada elemento de vector3D es un puntero a puntero a entero, o un vector de punteros a enteros, o un vector de vectores de entero.

Veamos una representación completa del anterior ejemplo, desde el final hacia el principio



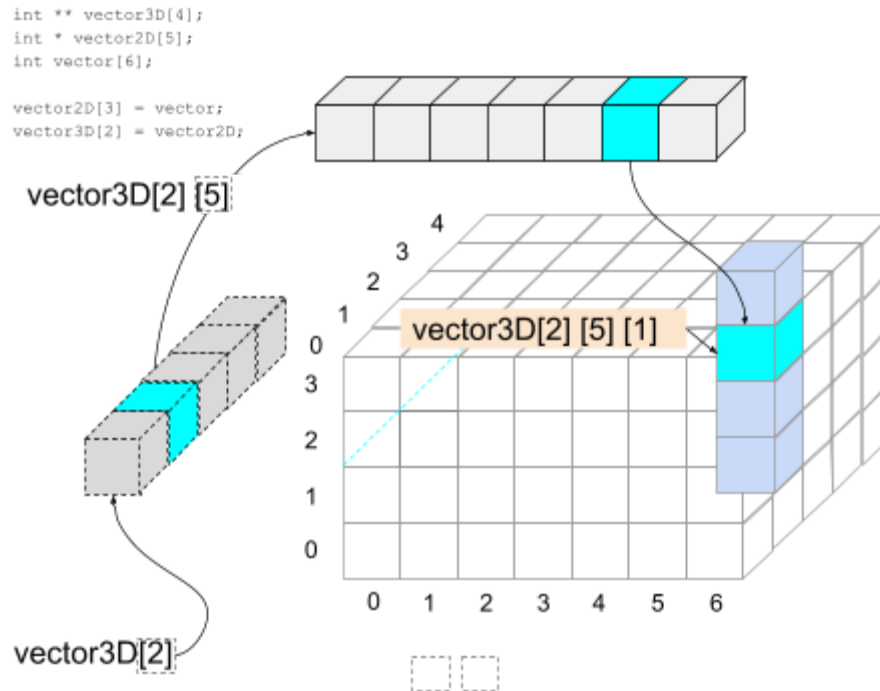
La matriz o vector2D es un puntero. Vamos a considerar que apunta a una fila de componentes como se ve en la imagen. Esta representación no es totalmente real, la usaremos como un punto de partida aproximado. Lo importante es que cada uno de esos elementos ahora también es un puntero que apunta a un vector. en la siguiente imagen se refleja esta situación

```
int ** vector3D[4];  
int * vector2D[5];  
int vector[6];  
  
vector2D[3] = vector;  
vector3D[2] = vector2D;
```



Este nuevo vector que aparece situado dentro de la matriz tiene elementos que sólo pueden ser accedidos a partir del vector "vector3D" mediante dos índices. El primer índice es el que nos ha llevado al primer vector y uno de sus elementos (resaltado en azul fosforito), mientras que el segundo índice seleccionaría alguno de los elementos del vector que está situado dentro de la matriz.

Sin embargo, esto no termina aquí, porque cada uno de esos elementos es a su vez un vector (o un puntero). Obsérvalo en la siguiente imagen



El último vector sí que tiene en sus elementos un número entero. De la misma manera que antes, para alcanzar uno de esos enteros, habremos de partir del vector "vector3D" e indicar 3 dimensiones. Es lo que nos permitirá llegar a un número concreto dentro de esa matriz.

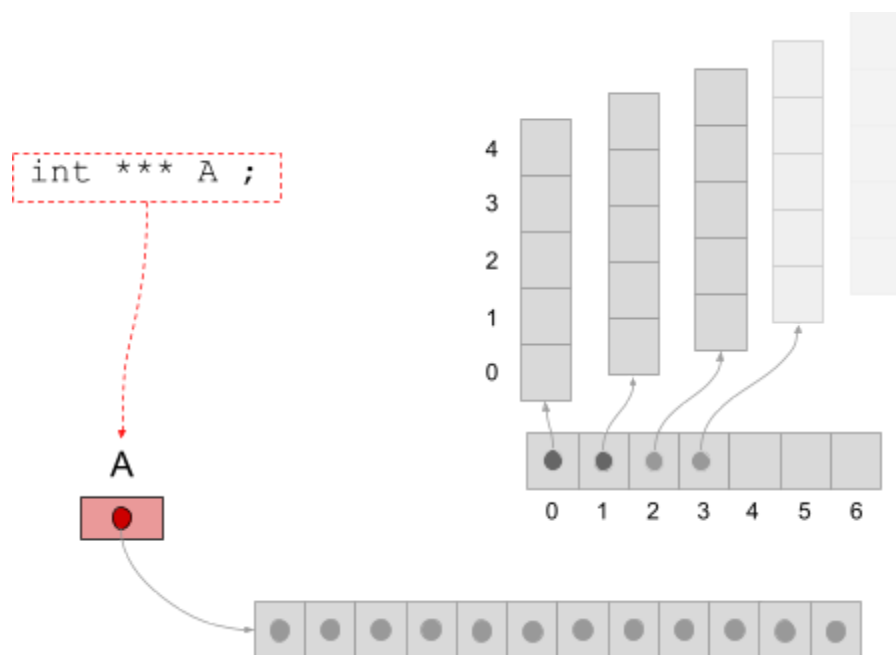
```
vector3D[2][5][1];
```

En el ejemplo anterior, cuyo código se ve en las figuras, se han creado vectores diferentes para componer la formación parcial de la matriz. Harían falta más vectores intermedios para hacerla completa. Sin embargo, como se ha dicho, no queremos vectores que se crean durante la compilación. Por tanto se han de crear vectores que serán referenciados por otros vectores de forma dinámica y tan sólo existirá una sola variable. A continuación se nos muestra cómo hacer esto

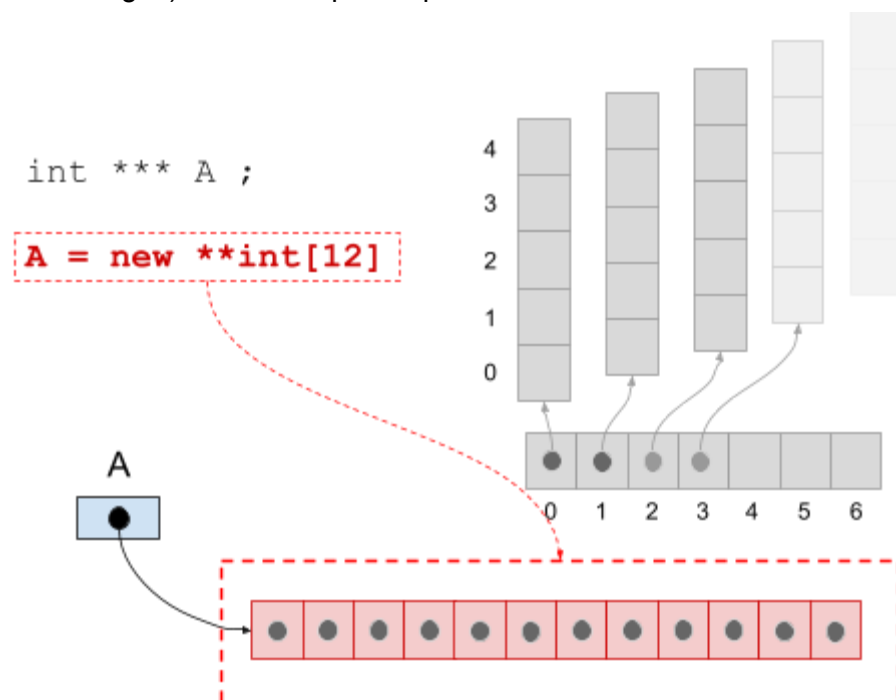
Reservando memoria

Así pues, vamos a construir ya la matriz tridimensional empezando por la definición inicial adecuada, que no crea ningún elemento (sólo el propio puntero, apuntando a nada en particular) (llamaremos "A" a la matriz que antes era vector3D para ahorrar espacio):

```
int ***A;
```

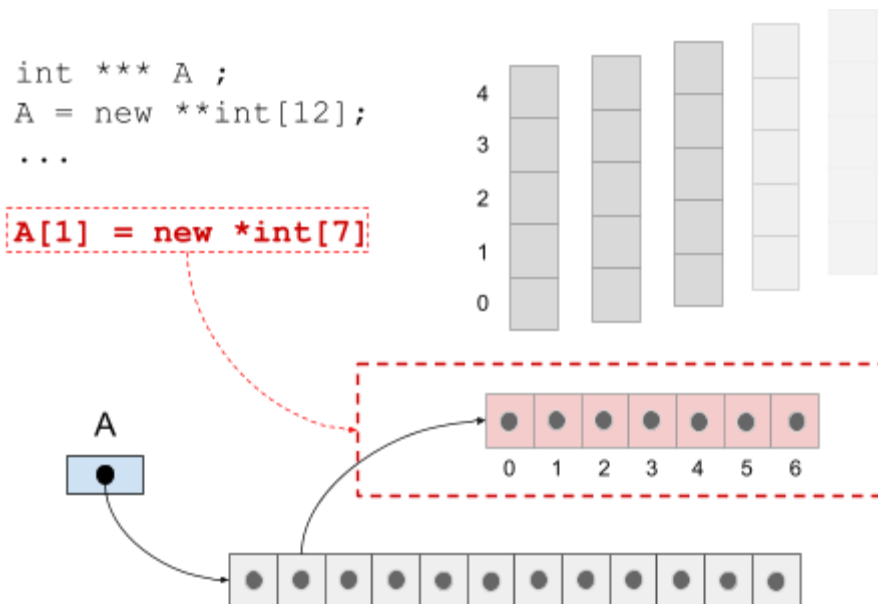


A continuación, el primer paso importante es crear el vector . En rojo se ve la creación en tiempo de ejecución de un vector de 12 elementos (12 para hacerlo diferente del ejemplo anterior), cada uno de los cuales apunta a un vector bidimensional que todavía no está creado (pero que aparece en gris). Este es el primer paso.

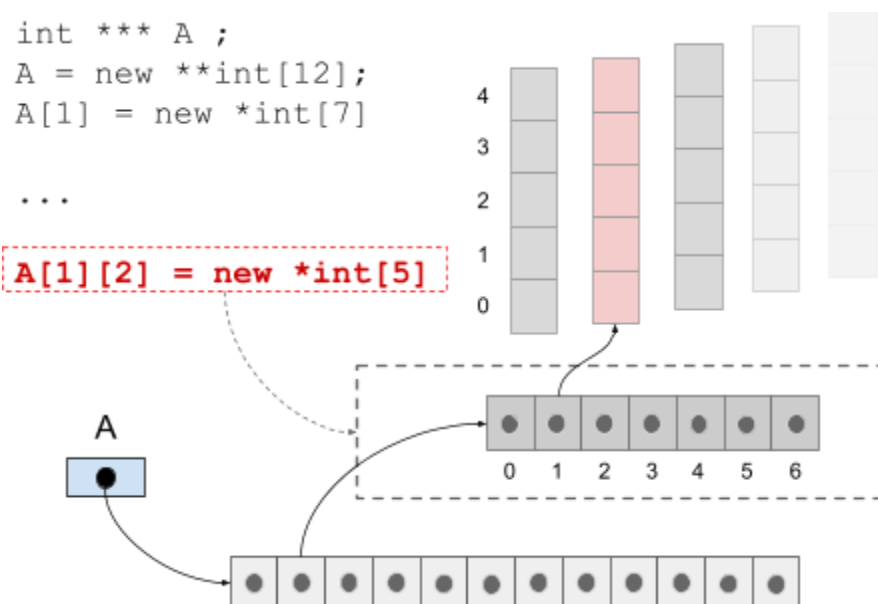


Como puedes suponer, para cada uno de los elementos del nuevo vector hay que crear después otro vector pudiendo elegir el tamaño (e incluso de tamaño diferente para cada uno

de ellos, cosa que daría muchos dolores de cabeza). El segundo paso por tanto toma el vector creado anteriormente y repite para cada elemento la creación de un nuevo vector. En la imagen sólo se ve la creación de uno de los 12 que harían falta (12 porque son las dimensiones del vector primero)



Finalmente para cada elemento de los nuevos vectores habría que asignar un vector entero creado también dinámicamente. En la siguiente imagen puedes ver uno de los tantos vectores que finalmente se crearán.



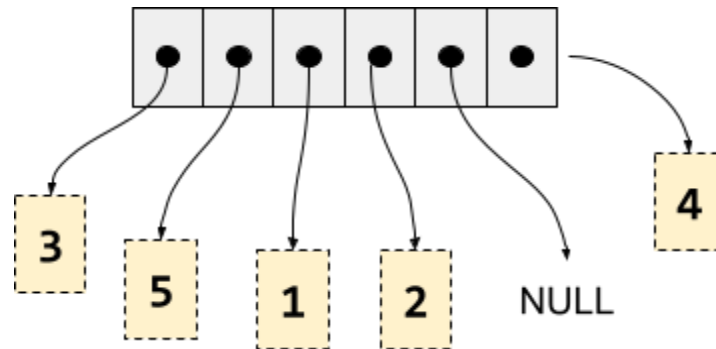
Lo que nos queda es repetir estas creaciones de vectores de forma sistemática y completa.

```
// Creación de un vector tridimensiona.
const unsigned x = 4, y = 7, z = 5;           // dimensiones
int*** A = new int**[x];                     // primer paso
for (unsigned i = 0; i < x; ++i) {
    A[i] = new int*[y];                       // segundo paso
    for (unsigned j = 0; j < y; ++j) {
        A[i][j] = new int[z];                // tercer paso
    }
}
// Ejemplos de acceso a elementos
A[2][3][4] = 12;
A[3][6][9] = -25;

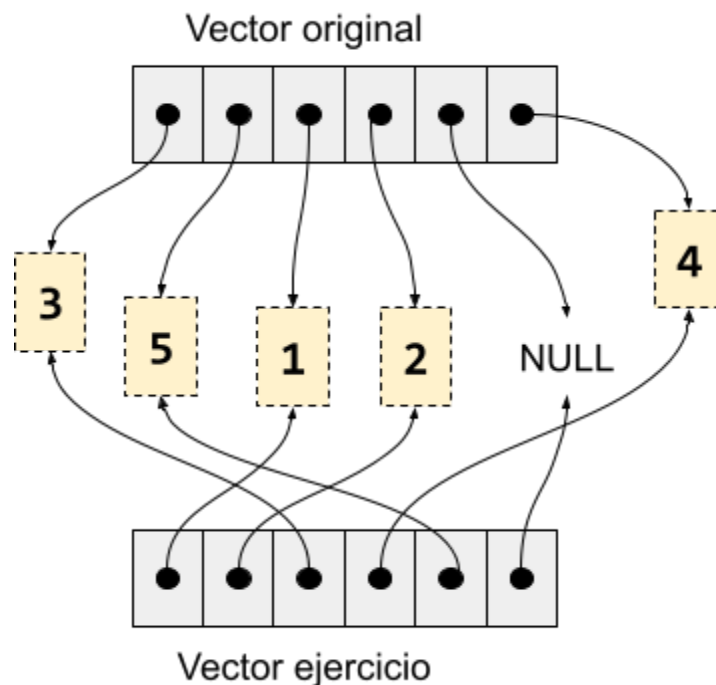
// Liberación de memoria.
for (unsigned i = 0; i < x; ++i) {
    for (unsigned j = 0; j < y; ++j) {
        delete[] A[i][j];
    }
    delete[] A[i];
}
delete[] A;
```

Ejercicio

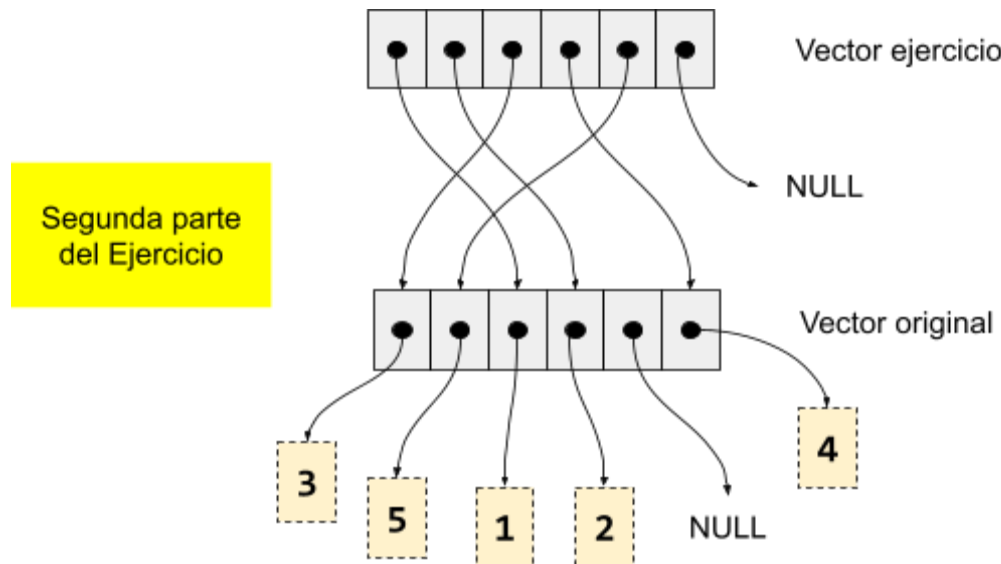
A partir de un vector de punteros a enteros no repetidos que puede estar parcialmente incompleto (algún elemento apunta a NULL), vamos a obtener otro vector que apunte a los mismos números pero sus elementos reflejarán el orden.



El otro vector a completar, estará apuntando a los mismo números pero de forma ordenada. Observa en la siguiente imagen el vector de la parte inferior. Verás que cada elemento apunta a un número superior al anterior. El vector a obtener está ordenado



Más tarde, como ampliación, se desea obtener otro vector para tener igualmente ordenados los números, pero en este caso el vector no apuntará a los números, sino a los elementos del otro vector. Otra vez este nuevo vector reflejará el orden en los elementos apuntados.

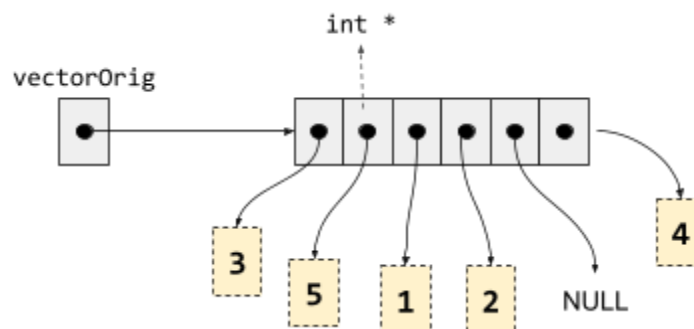


Solución.

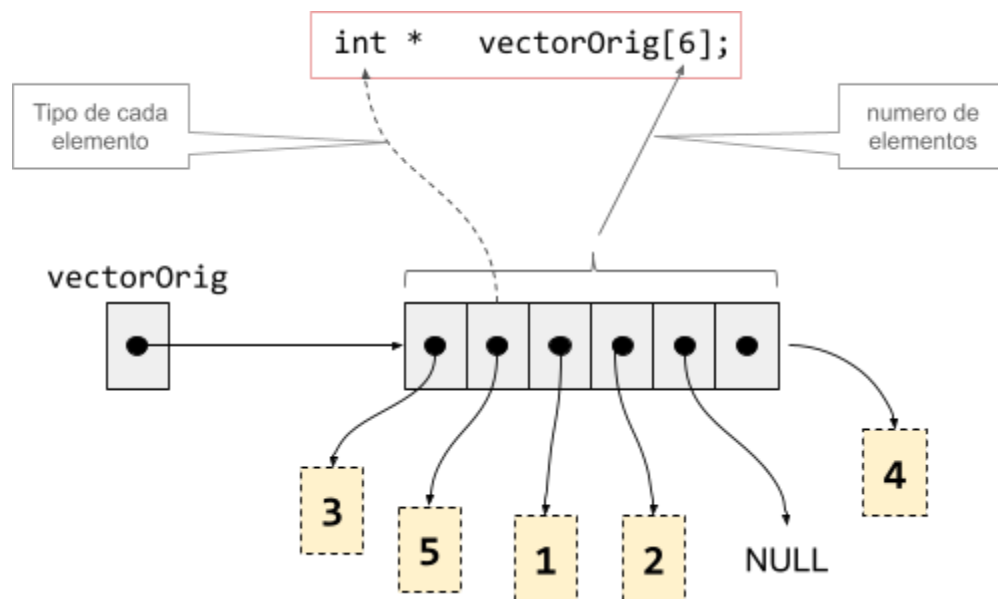
Vamos a realizar una solución, como siempre, que nos ayude a aprender, no que simplemente satisfaga el enunciado de la forma más directa posible y nos permita acabar rápido

Declaración del vector

Empezaremos por pensar en la declaración del vector



Observa que cada elemento del vector no es un entero, sino la dirección a un entero, esto es: "int *". Esto indica de qué tipo es cada elemento del vector y la declaración es entonces la siguiente:



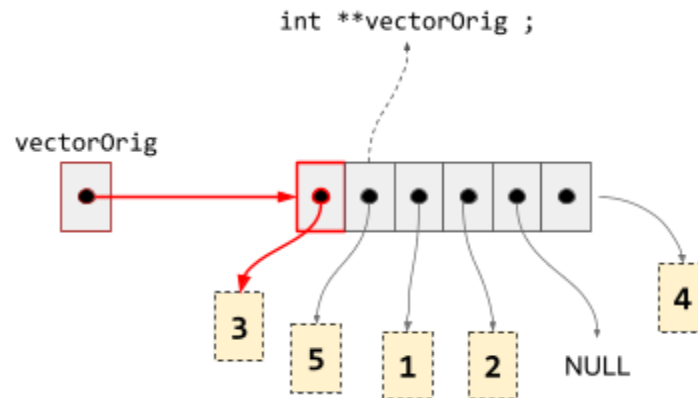
Esta es la declaración del vector:

```
int * vectorOrig[6];
```

Con ella, se crean 6 elementos (que en memoria están contiguos), y la variable que representa el vector es sólo un puntero al primero de ellos. Es decir "vectorOrig" es sólo un puntero -que resulta que apunta a un puntero a un entero-. Estamos tratando entonces con punteros a punteros.

Las dos declaraciones siguientes son iguales salvo porque la primera crea 6 elementos y la segunda no

```
int * vectorOrig[6];  
int ** vectorOrig;
```



La línea y marcos rojo expresan por qué `vectorOrig` es un `int **` (puntero a puntero)

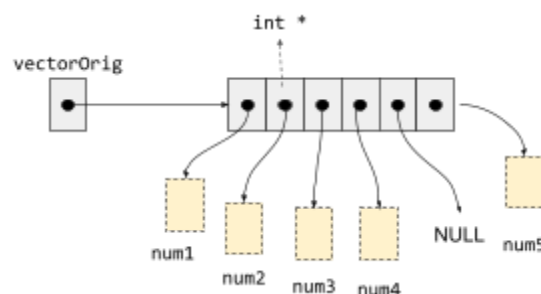
Creación de datos de ejemplo

Vamos a intentar crear datos y disponerlos tal cual están en el ejemplo gráfico que se viene siguiendo.

La creación de los enteros es dinámica, es decir . No creamos cinco variables de tipo `int` y después hacemos que apunten a ellas como aparece en el siguiente ejemplo

```
int num1 = 1;
int num2 = 2;
int num3 = 3;
int num4 = 4;
int num5 = 5;

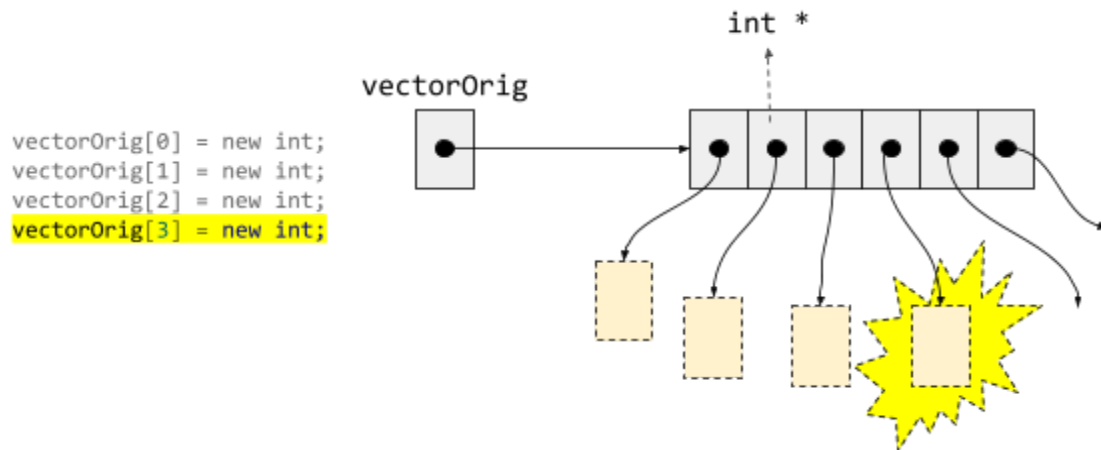
vectorOrig[0] = &num3;
vectorOrig[1] = &num5;
vectorOrig[2] = &num1;
vectorOrig[3] = &num2;
//vectorOrig[4] = NULL;
vectorOrig[5] = &num4;
```



De la siguiente forma es más correcto crear los números, ya que no dependemos de la existencia previa de unas variables y mientras no borremos los enteros creados, perdurarán en

la memoria. Date cuenta que estamos dejando todo el tiempo el elemento quinto (índice = 4) a NULL. Es como si el vector no estuviese totalmente ocupado.

```
vectorOrig[0] = new int;
vectorOrig[1] = new int;
vectorOrig[2] = new int;
vectorOrig[3] = new int;
vectorOrig[4] = NULL;
vectorOrig[5] = new int;
```



Ahora que ya existen unos enteros (y los elementos del vector apuntan a ellos), habrá que darles valores, aunque sólo sea para realizar pruebas y verificar que el ejercicio sale bien. Estamos en el caso de querer acceder al valor almacenado por un puntero.

```
vectorOrig[0] = 3;
vectorOrig[1] = 5;
vectorOrig[2] = 1;
vectorOrig[3] = 2;

vectorOrig[5] = 4;
```

```
for (int i = 0; i < 6 ; i++) {
    cout << "Posicion " << i << ": " << *vectorOrig[i] << endl;
}
```

Error:

```
mati@HiDi-Tec:~/Documentos/Interfaces$ g++ -o ordenacion repasopunteros.cpp
mati@HiDi-Tec:~/Documentos/Interfaces$ ./ordenacion
Posicion 0: 3
```

```
Posicion 1: 5
Posicion 2: 1
Posicion 3: 2
Violación de segmento (`core' generado)
mati@HiDi-Tec:~/Documentos/Interfaces$
```

¿Solución?

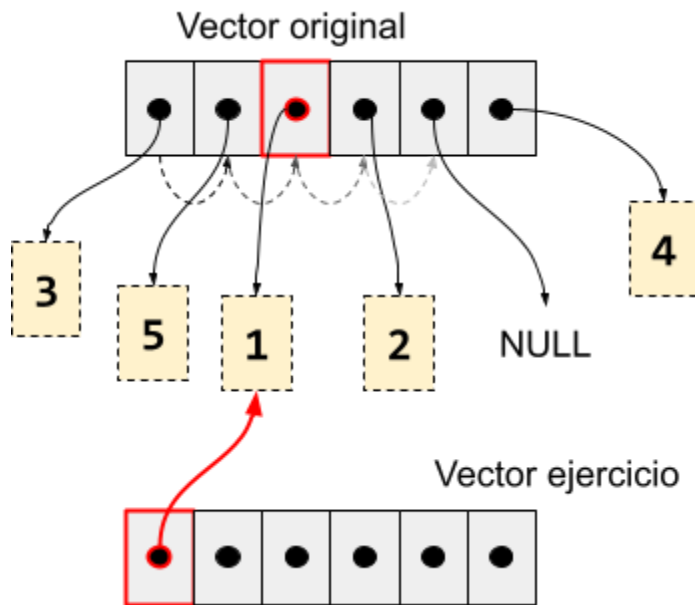
```
for (int i = 0; i < 6 ; i++) {
    if (*vectorOrig[i] != NULL)
        cout << "Posicion " << i << ": " << *vectorOrig[i] << endl;
}
```

Ahora si.

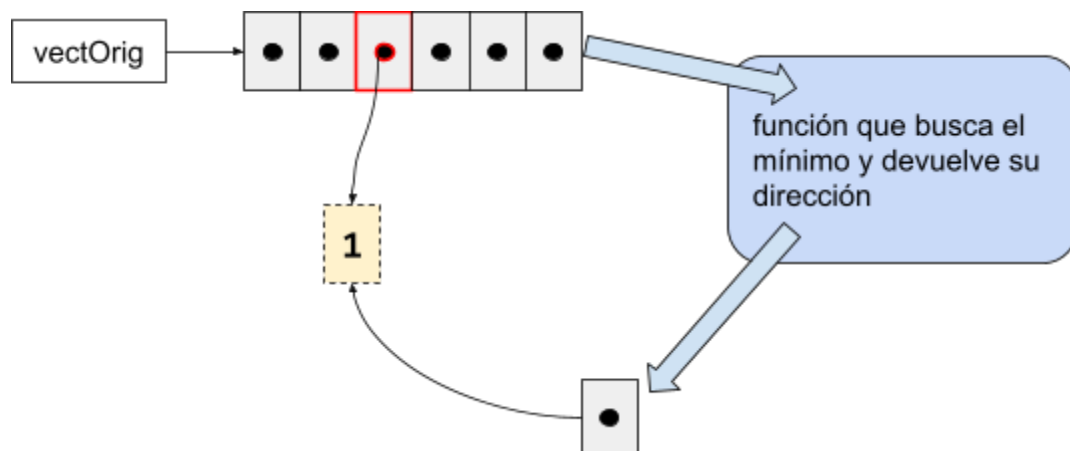
```
for (int i = 0; i < 6 ; i++) {
    if (vectorOrig[i] != NULL)
        cout << "Posicion " << i << ": " << *vectorOrig[i] << endl;
}
```

Declaración de datos del vector solución

Empecemos a ordenar



Vamos a encontrar ese número mínimo. Y ya que he de guardar la dirección de ese número en el primer elemento del vector, voy a crear una función a la que le paso el vector original y me devuelve el puntero al número mínimo



podré hacer algo así:

vectorOrig[0] = minimo(...)

¿Posible ?

```
int * min(int * numeros[6]) {
    int numAux = 100;
    for (int i = 0; i < 6 ; i++) {
        if (*numeros[i] < numAux)
```

```

        numAux = numeros[i];
    }
    return numAux; // o return &numAux;
}

```

lo único que recuerdas es que "en alguna posición, había un 1" y ese era el valor mínimo... pero no sabes la posición ya. Hay que recordarla usando otra variable en

```

int * min(int * numDesorden[6]) {
    int numAux = 100;
    int posicion;
    for (int i = 0; i < 6 ; i++) {
        if (*numDesorden[i] < numAux) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
    return numDesorden[posicion];
}

```

¿Violación de segmento? los elementos NULL no se pueden comparar, hay que evitarlo... y con cuidado de no acceder a él sin querer, observa la línea resaltada

```

int * min(int * numDesorden[6]) {
    int numAux = 100;
    int posicion;
    for (int i = 0; i < 6 ; i++) {
        if ((numDesorden[i] = NULL) && (*numDesorden[i] < numAux)) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
    return numDesorden[posicion];
}

```

Pero ésta función solo sirve para encontrar el mínimo... y encima lo hace mal. Vamos a tomar el valor inicial numAux real a partir del primer elemento del vector

```

int * min(int * numDesorden[6]) {
    int numAux = numDesorden[6];
    int posicion;
    for (int i = 0; i < 6 ; i++) {
        if ((numDesorden[i] = NULL) && (*numDesorden[i] < numAux)) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
}

```

```

    }
}
return numDesorden[posicion];
}

```

Ahora vamos a ampliar el uso de esta función para que nos devuelva el mínimo mayor que el que le pasamos, así podremos reusar la función para ir obteniendo nuevos mínimos... y practiquemos algo (recuerda que no pueden haber números repetidos en el vector tal como dice el enunciado).

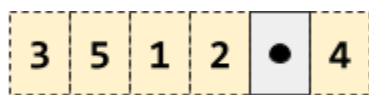
```

int * minSup(int * numDesorden[6], int numSup) {
    int numAux = *numDesorden[0];
    int posicion;
    for (int i = 0; i < 6 ; i++) {
        cout << "Nº:" << i << endl;
        if (
            numDesorden[i] != NULL &&
            *numDesorden[i] < numAux &&
            *numDesorden[i] > numSup) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
    return numDesorden[posicion];
}

...
int *p = minSup(vectorOrig,3);

```

Algo va mal, y es muy oscuro en esa llamada. Observa los valores del vector de forma simple:



numAux=3

numSup=3

int *p = minSup(vectorOrig,3);

```

if ( numDesorden[i] != NULL &&
    *numDesorden[i] < numAux &&
    *numDesorden[i] > numSup) {

```



Se pide un número mayor que 3 porque lo pasamos por argumento, pero menor que 3 porque se inicializa el numAux a ese valor. La solución es encontrar el máximo antes y proponer ese

máximo como mínimo a "batir", mientras el número probado sea superior a "numSup"... no se muestra el código, usaremos una aproximación cómoda pero errónea

```
int * minSup(int * numDesorden[6], int numSup) {
    int numAux = 10000000000000;
    int posicion;
    for (int i = 0; i < 6 ; i++) {
        cout << "Nº:" << i << endl;
        if (
            numDesorden[i] != NULL &&
            *numDesorden[i] < numAux &&
            *numDesorden[i] > numSup) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
    return numDesorden[posicion];
}
```

Ahora, en el cuerpo principal del programa vamos a utilizar esta función para mostrar, ordenados los elementos referenciados por el vector. La siguiente solución es bastante errónea porque la función devuelve un puntero y se asigna a un entero, pero se indica con bastante claridad la idea correcta para mostrar el

```
int numMin = 0;

for (int i = 0; i < 6 ; i++) {
    numMin = minSup(vectorOrig, numMin)
    cout << " " << numMin ;
}
cout << endl;
```

Si ahora nos aclaramos con los tipos, veremos que la función devuelve un puntero y por tanto hay que guardar en un puntero. Además, hemos de empezar por el valor mínimo real, es decir, hemos de pasarle un 0 a la función en la primera llamada. La siguiente propuesta parece reunirlos todo.

```
int * numMin;
*numMin = 0; // FALLO0000

for (int i = 0; i < 6 ; i++) {
    numMin = minSup(vectorOrig, numMin)
    cout << " " << numMin ;
}
cout << endl;
```

El fallo es asignar un valor a un numero que no existe, solo existe el puntero a una ubicación , pero allí no hay número. Hay que tener un número y una dirección, las dos cosas, y mantenerlas.

```
int * pMin;
int minEncontrado=0;

for (int i = 0; i < 6 ; i++) {
    pMin = minSup(vectorOrig, minEncontrado);
    minEncontrado = * pMin;
    cout << " " << *pMin ;
}
cout << endl;
```

Aunque esto funciona mayormente, sigue mostrando un error por culpa de que existen elementos del vector que apuntan a NULL y no los hemos previsto.

Primero, la función minSup debe devolver un puntero a un número, pero si no existe el número mínimo superior al que se pasa (porque este es ya muy elevado) ¿Qué devolvemos ? Devolvemos un NULL y de paso este sera el valor a devolver si no existen elementos:

```
int * minSup(int * numDesorden[6], int numSup) {
    int numAux = 10000000000000;
    int posicion=-1;
    for (int i = 0; i < 6 ; i++) {
        cout << "Nº:" << i << endl;
        if (
            numDesorden[i] != NULL &&
            *numDesorden[i] < numAux &&
            *numDesorden[i] > numSup) {
            numAux = *numDesorden[i];
            posicion = i;
        }
    }
    if (posicion != -1 ) return numDesorden[posicion];
    return NULL;
}
```

Modificamos el recorrido para contemplar la posibilidad de recibir NULL

```
int * pMin;
int minEncontrado=0;
```

```

for (int i = 0; i < 6 ; i++) {
    pMin = minSup(vectorOrig, minEncontrado);
    if (pMin != NULL ) {
        minEncontrado = * minSup;
        cout << " " << *pMin ;
    }
}

```

```

int * pMin;
int minEncontrado=0;

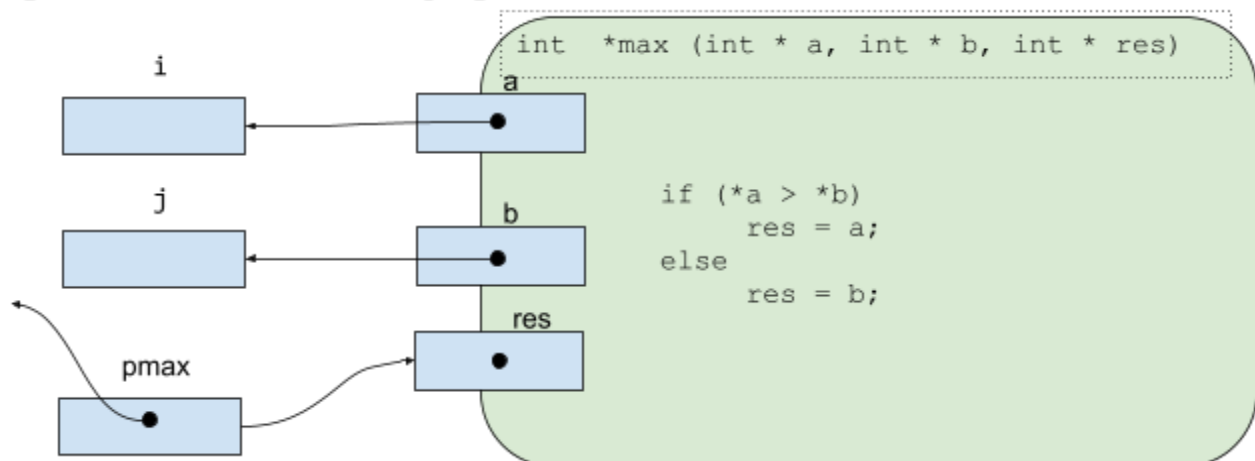
for (int i = 0; i < 6 ; i++) {
    pMin = minSup(vectorOrig, minEncontrado);
    if (pMin != NULL ) {
        minEncontrado = * minSup;
        cout << " " << *pMin ;
        vectorEjercicio[i] = pMin;
    }
}

```

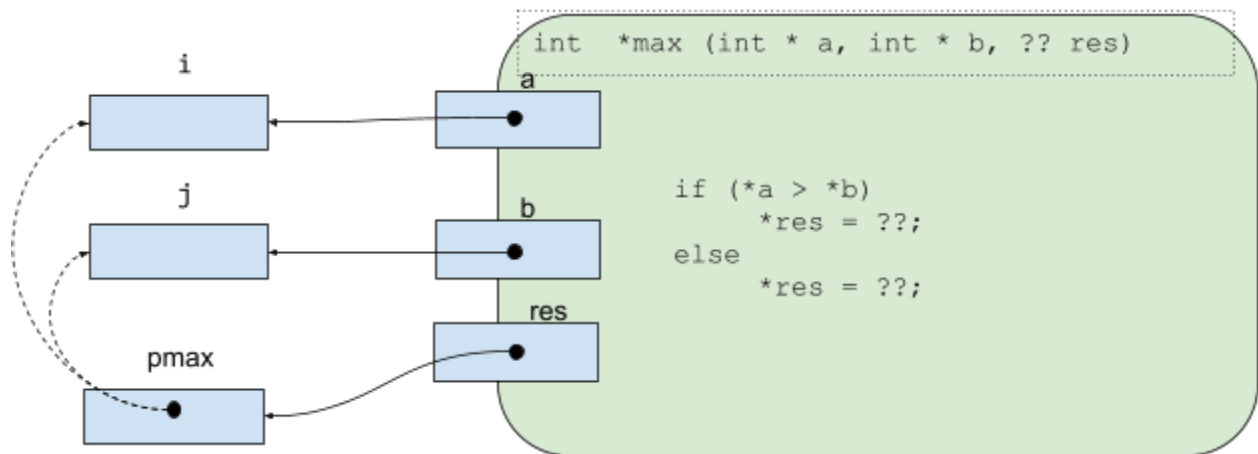
```

int * pmax;
pmax = max(&i, &j, pmax)

```




```
int * pmax;  
pmax = max(&i, &j, pmax)
```



```
int * pmax;  
pmax = max(&i, &j, &pmax)
```

