Punteros

Como ya vimos en la lección de las funciones (parámetros por copia y por referencia), las variables tienen dos “elementos”.

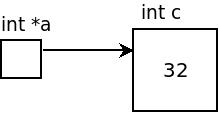
* Dirección de memoria en la que se almacena el valor
* El valor en sí

En c++ podemos trabajar con ambos elementos por separado declarando la variable como puntero. ¿Cómo declaramos un puntero?

|  |
| --- |
| int\* a; |

En este caso hemos declarado una variable *a* tipo puntero a entero, ¿qué quiere decir esto? que a apuntará a un entero. Como no le hemos dado ningún valor, inicialmente no está apuntando a nada. Fíjate en el siguiente código.

|  |
| --- |
| int c = 32;  int\* a = &c; |



Ahora el puntero *a* toma como valor la dirección de *c*. Si hicieramos

|  |  |
| --- | --- |
| 1  2  3  4 | int c = 32;  int\* a = &c;  cout << a; |

Contrariamente a lo que se podría esperar, no se imprimiría por pantalla 32, **porque la variable a es una dirección de memoria.** Entonces, ¿cómo accedemos al valor al que apunta *a*? A través del operador indirección.

|  |  |
| --- | --- |
| 1  2  3  4 | int c = 32;  int\* a = &c;  cout << \*a; //Muestra por pantalla 32. |

Pero, ¿cuál es la gracia de los punteros?, fíjate en el siguiente ejemplo de código:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int c = 32;  int\* a = &c;  cout << \*a; // Muestra por pantalla 32.  c = 25;  cout << \*a; // Muestra por pantalla 25.  \*a = 16;  cout << c; // Muestra por pantalla 16. |

Del mismo modo, a un puntero le podriamos asignar otro puntero.

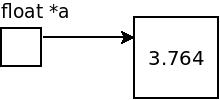
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | int c = 32;  int\* a = &c;  int\* b = a; //a y b apuntan al valor de c    cout << \*a; // Muestra por pantalla 32.  c = 25;  cout << \*a; // Muestra por pantalla 25.  \*a = 16;  cout << c; // Muestra por pantalla 16.  \*b = 22;  cout << c; // Muestra por pantalla 22.  cout << \*a; // Muestra por pantalla 22.  cout << \*b; // Muestra por pantalla 22. |

Inicializando un puntero

En el ejemplo anterior hemos visto que a un puntero le podemos asignar la dirección de otra variable del mismo tipo. Pero también podemos crear un espacio *nuevo de memoria*.

|  |  |
| --- | --- |
| 1  2  3 | float \*a = new float{3.764};    cout << \*a ; // muestra por pantalla 3.764 |

En este ejemplo creamos un puntero a, que apuntará a un número decimal, y además reservamos espacio de memoria para albergar ese número decimal. Inicializamos el puntero con el valor 3.764.



Podemos operar con el puntero del mismo modo que en los ejemplos anteriores:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | float \*a = new float{3.764};    cout << \*a ; // muestra por pantalla 3.764  float\* b = a; //a y b apuntan al valor de c  \*b = 22;  cout << \*a; // Muestra por pantalla 22.  cout << \*b; // Muestra por pantalla 22.    float c = \*a;  cout << c; // Muestra por pantalla 22. |

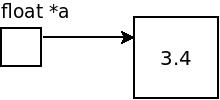
Importancia de la memoria reservada

Cuando reservamos espacio en memoria para un puntero esa memoria permanece *ocupada* hasta que explicitamente le digamos al programa que ya no la necesitamos.

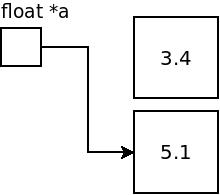
Un programa en el que se reserva continuamente memoria y no se libera corre el riesgo de quedarse sin memoria. Veamos qué pasa con el siguiente código.

|  |  |
| --- | --- |
| 1  2  3  4 | float \*a;  a = new float{3.4};  a = new float{5.1}; // perdemos la referencia a la memoria anterior |

En la primera línea creamos un puntero a un número decimal. En la segunda línea reservamos espacio en memoria y le asignamos el valor 3.4.



El problema viene en la tercera línea, volvermos a reservar espacio, y debido a ello perdemos la referencia a 3.4.



Además, ese espacio de memoria está reservado, por lo que no puede volver a utilizarse, y si esto sucede muy a menudo nuestro programa consumirá toda la memoria disponible del ordenador.

¿Qué pasaría en el siguiente programa?

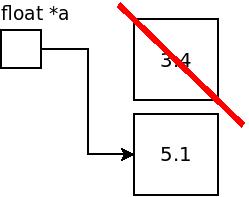
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | float \*a;  float \*b;  a = new float{3.4};  b = a;  a = new float{5.1}; // perdemos la referencia a la memoria anterior  cout << \* b; // muestra por pantalla 3.4  cout << \* a; // muestra por pantalla 5.1 |

Liberando la memoria reservada

El comando *new* reserva el espacio de memoria, el comando *delete* lo libera.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | float \*a;  a = new float{3.4};  cout << \* a; // muestra por pantalla 3.4  delete a; // borramos el espacio reservado para a  cout << \* a; // ERROR de ejecucion, a no apunta a ningun sitio  a = new float{5.1};  cout << \* a; // muestra por pantalla 5.1 |

Primero reservamos espacio para un número decimal, asignándole el valor 3.4. Cuando hacemos delete liberamos ese espacio de memoria, y por lo tanto **a no está apuntando en ese momento a ningún sitio.**. Por ello si queremos acceder a su valor con \*a nuestro programa dejará de funcionar (error de ejecución).



Esto puede ocasionar un error muy típico en los programas: **liberar un espacio de memoria al que apunta otro puntero**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | float \*a;  a = new float{3.4};  float \*b;  b = a;  cout << \* a; // muestra por pantalla 3.4  cout << \* b; // muestra por pantalla 3.4  delete a; // borramos el espacio reservado para a  cout << \* b; // ERROR. Mierda, hemos eliminado el espacio de memoria al que apunta b. |

