Apunte rápido sobre punteros

Punteros es un concepto bastante amplio y se extiende a lo largo de varios temas. En este apunte vamos a concentrarnos exclusivamente en su definición y su uso tanto en variables como en clases, esquivando varios detalles como interfaces y operadores. Aunque se adjunta un apunte más extendido para aquellos interesados. En C++ los punteros tienen particular uso, ya que es uno de los pocos lenguajes donde podemos definirlos explícitamente. En la mayoría de los lenguajes su uso se limita a su manipulación implícita (compiladores).

INTRODUCCIÓN

Cuando declaramos una variable, el compilador busca un espacio de memoria suficiente para almacenar la variable que estamos declarando y le asigna una etiqueta (nombre de la variable). En ese bloque de memoria se alojará el valor que le asignemos. Hasta el momento veníamos trabajando así. En C++, cuando no utilizamos punteros, la memoria utilizada para una variable no se libera hasta que el programa termina. Quizás con un par de variables no vale la pena ocuparse del ahorro de espacio, sin embargo cuando empezamos a utilizar clases, arreglos, matrices, imágenes, videos, etc, el tema de la memoria pasa a ser más delicado. Los punteros permiten reutilizar la memoria de las variables que ya no utilizamos. Un puntero es una variable que en su valor tiene una dirección de memoria. ¿Qué dirección de memoria? Aquella que fue reservada al crear el puntero. Cuando no ocupamos más el puntero, podemos borrarlo y de esta manera liberar esa memoria reservada.

OPERADOR * y &

Estos dos operadores permiten recuperar un valor de un puntero y una dirección de memoria de una variable, respectivamente. También sirven para iniciar los punteros. Vamos a tomar por ejemplo un entero.

```
int *t = new int(10);
```

El operador * sirve para crear el puntero, pero de ahora en adelante la variable **t** tendrá en su valor la dirección de memoria reservada, de la misma manera, si queremos ver el valor que hay en la memoria reservada, utilizamos el operador * para recuperar ese valor. La instrucción **new** permite inicializar y asignar la porción de memoria dinámica al puntero. Esta instrucción se utiliza para cualquier tipo de dato primitivo (int, float, etc), arreglos, clases, etc. En el ejemplo creamos un puntero que tiene en su memoria un entero con valor 10.

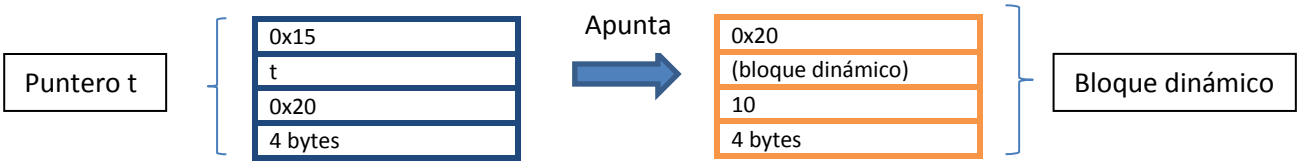
Ejemplo gráfico:

Vamos a usar el siguiente esquema para definir un bloque de datos:

Dirección del bloque (valor hexadecimal)
Nombre de la variable
Valor
Tamaño

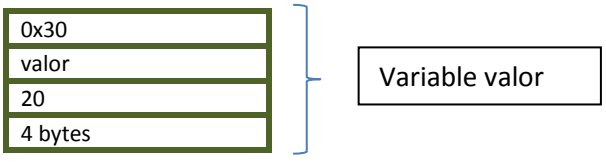
Iniciamos el puntero.

```
int *t = new int(10);
```



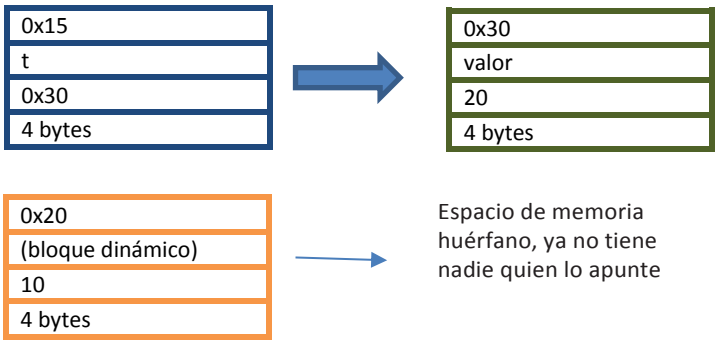
Iniciamos otra variable y le asignamos su valor:

```
int valor = 20;
```



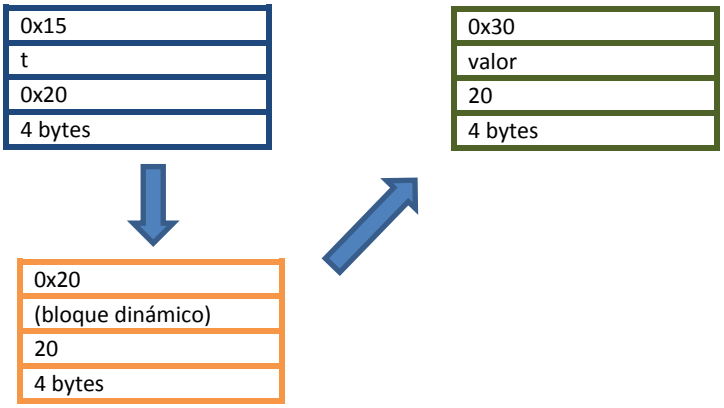
Reasignamos la dirección de la variable valor al valor de t.

`t = &valor;` (el operador & permite recuperar la dirección de memoria de una variable)



Si modificamos el valor del puntero (la dirección a la que apunta) perderemos la referencia del bloque dinámico, y luego ese valor ya no será accesible para ser modificado o eliminado. La forma correcta es modificar el valor del bloque dinámico, y no del puntero.

```
*t = valor;
```



En resumen, el operador * sirve para crear un puntero, una vez iniciado, el valor de esta variable será una dirección de memoria, mientras que su desreferencia (*) es un valor propiamente dicho.
En el siguiente ejemplo se ve como se utilizan todos estos operadores juntos:

```
// Creamos un puntero;
    int *t = new int(10);   // puntero t, asignamos 10 a su bloque de memoria
    cout<< t <<endl;  // Direccion de memoria del bloque dinanico
    cout<< *t <<endl;  // Valor alojado en el bloque (10)

    // Creamos una variable normal

    int valor = 20; // puntero t, asignamos 20 a su bloque de memoria
    cout<< &valor <<endl;  // Direccion de memoria de la variable
    cout<< valor <<endl;  // valor de la variable (20)

    // Le asignamos la dirección de memoria al valor del puntero

    t = &valor;
    cout<< *t <<endl;  // Valor alojado en el bloque (20)

    // si modificamos valor, podemos verlo reflejado desde el puntero

    valor = 30;
    cout<< *t <<endl;  // Valor alojado en el bloque (30)

    // El bloque dinamico queda huerfano y ya nadie tiene referencia a el
    // La forma correcta es modificar el valor del bloque dinamico

    *t = valor;
    cout<< *t <<endl;  // Valor alojado en el bloque (30)
```

INSTRUCCIONES NEW y DELETE

Así como la instrucción **new** permite inicializar un bloque dinámico, la instrucción **delete** permite borrarlo para que el compilador lo pueda volver a utilizar:

```
    // Creamos un puntero;
    int *t = new int(10);   // puntero t, asignamos 10 a su bloque de memoria
    cout<< t <<endl;  // Direccion de memoria del bloque dinanico
    cout<< *t <<endl;  // Valor alojado en el bloque (10)

    delete t;
    cout<< t <<endl; // Direccion de memoria del bloque dinanico (se mantiene)
    cout<<*t<<endl;  // Valor basura
```

PUNTEROS CON CLASE

La utilización de punteros en clases es idéntica a como venimos mostrando y es donde más fruto vamos a sacar. Supongamos una clase **Auto**, Vamos a suponer que tiene un constructor que recibe un entero equivalente a la cantidad de ruedas. Se declara de la siguiente manera:

```
Auto *miAuto = new Auto(10);
```

La iniciaclización es idéntica a cómo venimos usando: el tipo de la clase, el nombre de la variable (ahora le agregamos el operador *) y en vez de usar el constructor directamente a continuación del nombre, utilizamos la instrucción **new** con el constructor que queramos usar.

La otra diferencia la tenemos a la hora de llamar a los métodos y atributos, en vez de usar el operador punto (**.**), se utiliza una flecha construida por el menos y el mayor (**->**).

Supongamos que la clase Auto tiene un método que es **contarRuedas** y retorna la cantidad de ruedas:

```
int ruedas = miAuto->contarRuedas();
```

Y, al igual que cualquier variable, es recomendable borrar los bloques dinámicos de memoria cuando ya no usamos más la clase:

```
delete miAuto;
```