Clases en C++ para el que viene de Java

En C las clases se declaran de forma similar a Java. Durante el tutorial trabajaremos con un ejemplo en el que pretenderemos estar realizando un programa para gestionar una oficina bancaria. En ella, el principal concepto que se maneja es la "Cuenta bancaria". Ésta será nuestra primera y casi única clase:

Veamos su definición primitiva e incompleta (porque sólo tiene atributos)

```
class Cuenta {
public:
    string titular;
    string numCuenta;
    float saldo;
private:
    float interes;
};
```

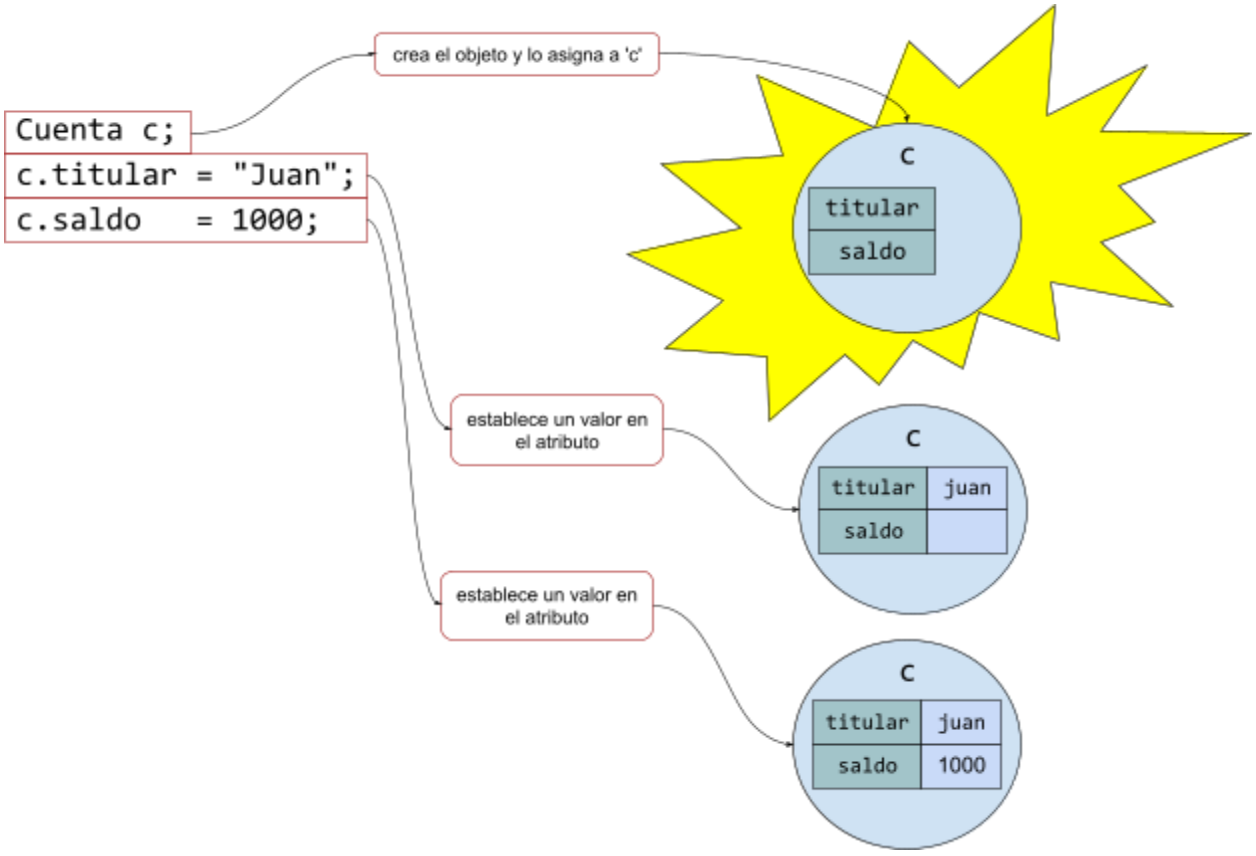
Fíjate en los siguientes elementos destacados de la declaración de la clase:

- Se escribe la palabra **class** y después el nombre de la clase que estás declarando
- La palabra "**public:**" (con dos puntos) se escribe antes de los elementos que van a ser públicos
- La palabra "**private:**" antes de los elementos privados
- ¡Atención! Se escribe un punto y coma ";" después de cerrar las llaves. Suele olvidarse

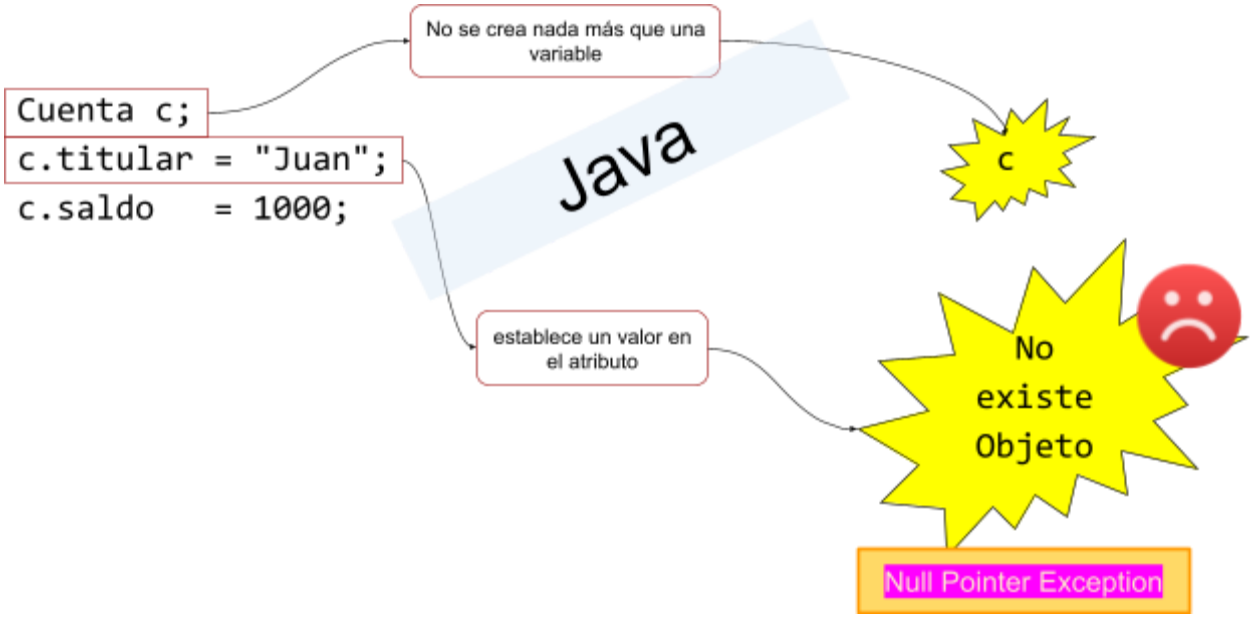
No se escribe "public" o "private" para cada método o atributo, aquí van agrupados. Recuerda el ";" (punto y coma) final

A partir de este momento se puede declarar objetos y variables de tipo "Cuenta". En el siguiente ejemplo se declara una variable de tipo cuenta que se bautiza "c". Además, se crea un objeto de tipo "Cuenta". En la segunda línea se asigna un valor a un atributo del objeto

```
Cuenta c;
c.titular = "Dolores Fuertes de Barriga";
c.saldo = 1000;
```

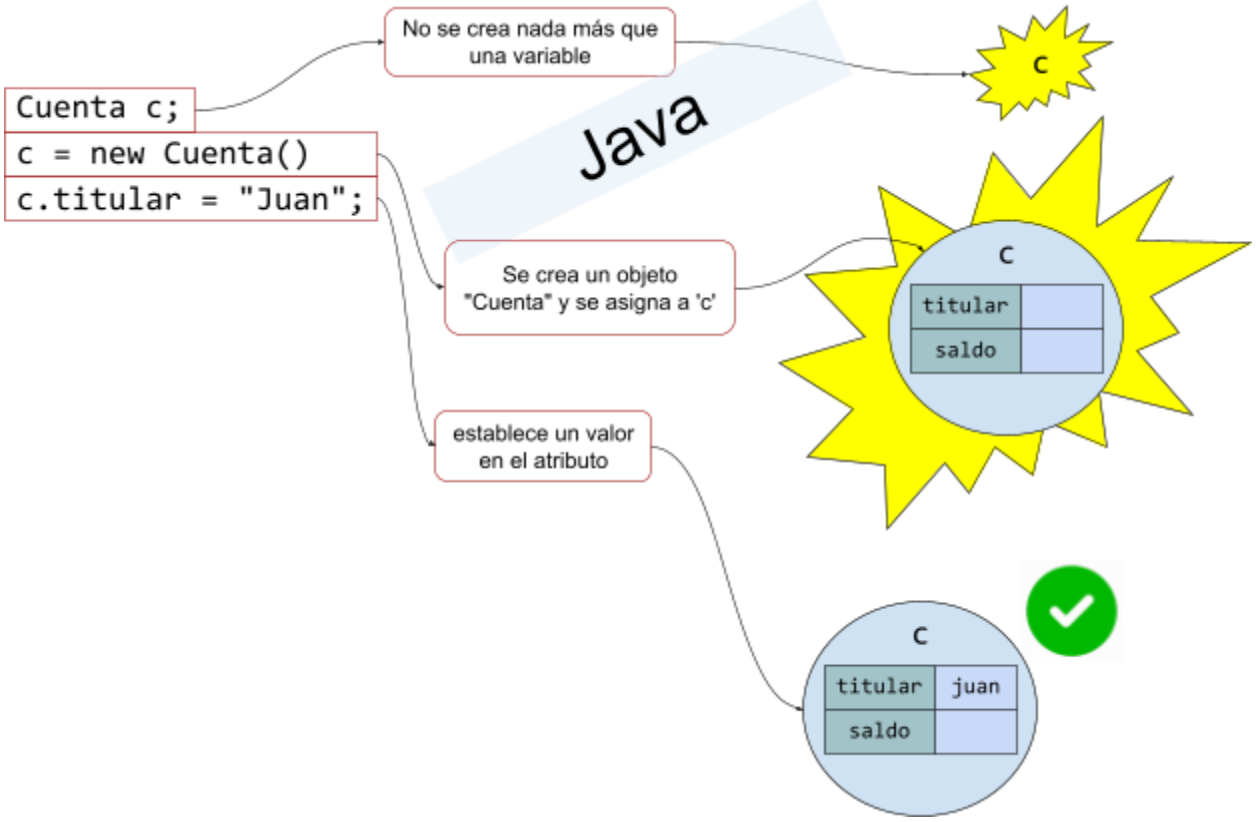


Fíjate que **en Java, el código daría un error de "Null pointer exception"**. En Java, en la primera línea no se crea nada, tan sólo se declara una variable "c".



Por ello en java creamos explícitamente el objeto con new Cuenta():

```
Cuenta c;
c = new Cuenta();
c.titular = "Dolores Fuertes de Barriga"
```



Podemos hacer nuestro primer programa que use objetos de tipo Cuenta.

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

class Cuenta {
public:
      string titular;
      string numCuenta;
      float saldo;

};

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

cout << "bienvenido al banco" <<endl;
```

```
      Cuenta c;
      c.titular = "pepe";
      c.numCuenta ="001";
      c.saldo = 10.04 ;

      cout << " la cuenta " << c.numCuenta << " de " << c.titular
      << " tiene " << c.saldo << " euros " <<endl;

}
```

La función "`int main(int argc, char *argv[])`" es la que inicia un programa en C. Es decir, al ejecutar un programa, se ejecuta la función ésta.

Recuerda: para compilar el programa y ejecutarlo:

1. Guarda el programa anterior en un fichero (por ejemplo llamado "fuente.c++")
2. Compila mediante con el siguiente comando

```
g++ -o programa fichero
```

3. Ejecuta invocando el programa ejecutable recién generado (llamado "programa")

```
$./programa
```

Primer método

En este ejemplo, la clase cuenta representa la idea de una cuenta bancaria, la cual es un ente que cambia de estado. En el caso de una cuenta bancaria, con ella se pueden hacer tres operaciones básicas:

1. Consultar el saldo
2. Ingresar dinero
3. Sacar dinero

Ingresar y sacar dinero logran alterar el saldo de la cuenta, cambiando con ello el estado de la misma. Cada una de las operaciones anteriores se materializa mediante un método distinto que en todos los casos se aplicará sobre la cuenta tratada.

Cada método se diseña o determina atendiendo a :

1. Nombre que recibe el método,
2. Datos que recoge
3. Resultado que devuelve.

Empezaremos por el método reintegrar, que está pensado para sacar dinero de una cuenta. Sus características son:

- Nombre: "reintegrar"
- Datos que recibe (argumentos): La cantidad de dinero a sacar
- Datos que devuelve: "cierto" o "falso" según haya o no dinero en la cuenta suficiente para lo solicitado.

Por todo ello su prototipo es:

```
bool reintegrar(float cantidad)
```

Y su implementación en la clase queda como sigue (incluye una prueba)

```
#include <iostream>

using namespace std;


class Cuenta {
public:
    string titular;
    int numCuenta;
    float saldo;

    bool reintegrar(float cantidad) {
       if (saldo < cantidad) return false;
       saldo -=cantidad;
       return true;
    }

};

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

    Cuenta c;
//    c = new Cuenta();
    c.saldo=100;

    cout << "El saldo de la cuenta es " << c.saldo << endl;
    if (    c.reintegrar(1000)  )
      cout << "PELIGRO! reintegro superior al saldo"<<endl;
    else
      cout << "sacar 1000 euros no produce cambios, saldo = "
       << c.saldo << "euros" << endl;

    if (c.reintegrar(10) )
```

```
        cout << "sacar 10 euros ha ido bien. el saldo es: " <<
              c.saldo<< endl;

}
```

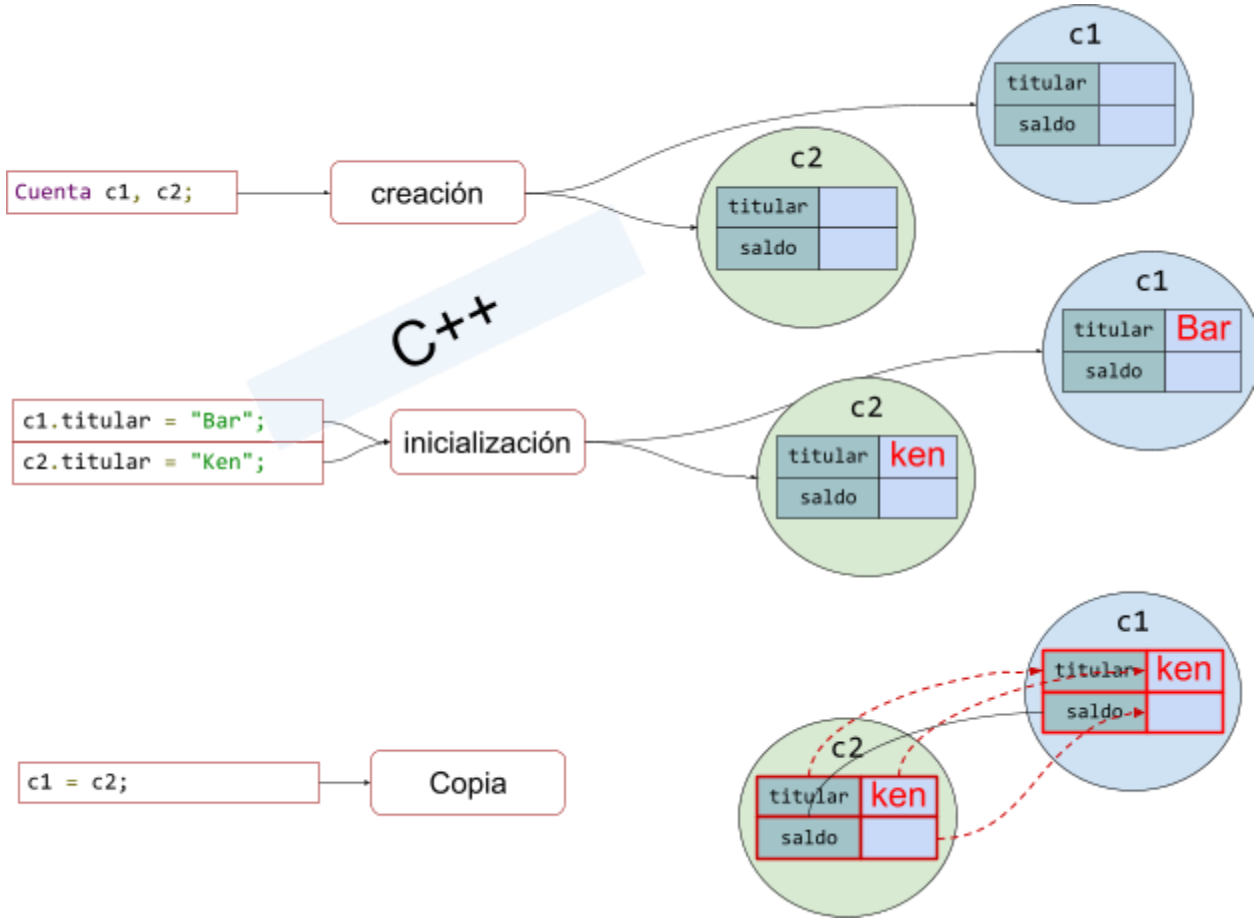
Copia de objetos.

En Java un objeto puede tener diferentes variables que lo referencien. En C, cada variable es un objeto distinto, incluso si se hace una asignación. Observa el código siguiente

```
Cuenta c1, c2;
/* en JAVA c1 = new Cuenta(); c2 = new Cuenta(); */

/* para hacer la prueba completa la asignación de valores
a todos los atributos. Haz una prueba sensata */
c1.titular = "Barbie";
c2.titular = "Ken";
c1 = c2;
```

En Java, después de la última línea haríamos algo destructivo, ya que la primera cuenta (la de "Barbie") dejaría de tener referencia a ella (tanto c1 como c2 son ahora la cuenta de Ken). La cuenta de "Barbie" dejaría de estar referenciada y no podríamos volver a acceder a ella. Ese es el efecto de la línea `c1 = c2`: La variable c1 referencia la misma cuenta que la variable c1. El objeto antes referenciado por c1 deja de existir

Pero en C++ esto no es así. En la última línea, **se hace una copia** del valor de los atributos, de la cuenta de Ken a la de Barbie. El valor de los atributos que había en la cuenta de Barbie se pierde, pero sigue habiendo dos cuentas. Las dos cuentas pasan a tener los mismos valores en todos sus atributos. Pero, y aquí está la diferencia, sigue habiendo dos cuentas diferentes.



El siguiente programa prueba lo anteriormente explicado.

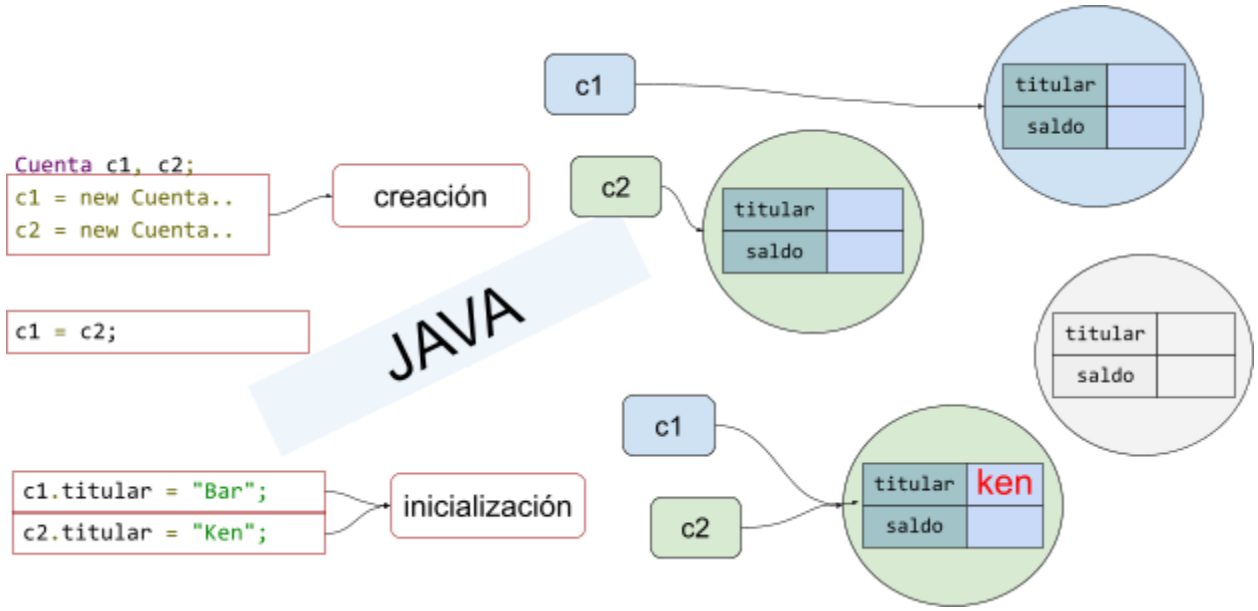
```
int main (int argc, char *argv[] ) {
    cout << "bienvenido al banco" <<endl;

   Cuenta c1, c2;
      /* JAVA añadiríamos c1 = new Cuenta(); c2 = new Cuenta(); */
      c1.numCuenta = "0001";
      c2.numCuenta = "0002";
      c1.saldo = 100000;
      c2.saldo = 200000;
      c1.titular = "Barbie";
      c2.titular = "Ken";

      c1 = c2;
      cout << " la cuenta " << c1.numCuenta << " de " << c1.titular
      << " tiene " << c1.saldo << " euros " <<endl;

      cout << " la cuenta " << c2.numCuenta << " de " << c2.titular
      << " tiene " << c2.saldo << " euros " <<endl;
```

```
    }
```

… En java es razonable, no vemos diferencia respecto a este lenguaje. Vamos a probar a colocar de forma diferente las asignaciones:

```
Cuenta c1, c2;
      /* JAVA añadiríamos c1 = new Cuenta(); c2 = new Cuenta(); */

      c1.titular = "Barbie";
      c2.titular = "Ken";
      c1 = c2;

      c1.numCuenta = "0001";
      c2.numCuenta = "0002";
      c1.saldo = 100000;
      c2.saldo = 200000;
```

¿Qué pasa si a continuación mostramos c1 y c2? La cosa cambia respecto a Java.



Objetos como argumentos de Funciones

En C pueden haber funciones "fuera" de las clases. Eso es: funciones solitarias que reciben argumentos y devuelven resultados, pero que no están ligados a ningún objeto.
Hagamos una función para mostrar los datos de la cuenta.

Este código va fuera de la función main() justamente antes de ella.

```
void mostrar(??? ) {

    cout << "Cuenta num" << ???  << " del fulano: " << ???
       << " tiene " << ??? << " leuros" << endl;

}
```

Ahora podemos probar todo esto dentro de main():

```
    if (c.reintegrar(10) )
       cout << "sacar 10 euros ha ido bien. el saldo es: " <<
             c.saldo<< endl;
      mostrar(c);
}
```

prueba incompleta. Resultado:

```
      Cuenta num1134987936 del fulano:  tiene 380 leuros
```

La prueba ha de ser correcta

```
      c.titular   = "Moncho";
      c.numCuenta = 0;
      c.saldo     = 100;
```

resultado

```
      Cuenta num: 0 del fulano: Moncho tiene 380 leuros
```

Con la siguiente función podremos fácilmente ampliar el programa con varias cuentas y mostrar cada una de ellas en el futuro. Observa la función y cómo se la invoca desde el programa principal

```
void mostrar(Cuenta cuentaMostrar){
      cout << " la cuenta " << cuentaMostrar.numCuenta <<
     " de " << cuentaMostrar.titular
```

```
      << " tiene " << cuentaMostrar.saldo <<
    " euros " <<endl;

}

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


    cout << "bienvenido al banco" <<endl;

   Cuenta c1, c2;
      /* JAVA añadiríamos c1 = new Cuenta(); c2 = new Cuenta(); */

      c1.titular = "Barbie";
      c2.titular = "Ken";
      c1 = c2;

      c1.numCuenta = "0001";
      c2.numCuenta = "0002";
      c1.saldo = 100000;
      c2.saldo = 200000;

      mostrar(c1);
      mostrar(c2);
```

Ahora, después de mostrar ambas cuentas, vamos a cambiar una para demostrar que siguen habiendo dos diferentes e independientes que simplemente se calcaron en un momento

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


 cout<<"-------------------------------------"<<endl;
    c2 = c1;

    mostrar(c1);
    mostrar(c2);

    c2.ingresar(1000);

    mostrar(c1);
    mostrar(c2);
```

Copia en paso de argumentos

Deseamos hacer una **función** que gestione el ingreso de saldo en una cuenta. Respecto al método, esta función tiene la misión de leer del teclado una cantidad e ingresarla… La lectura de la cantidad (del teclado) no es faena de la clase, por eso no lo hacemos en el método. Pero como vamos a repetir esta operación varias veces, lo hacemos en una función

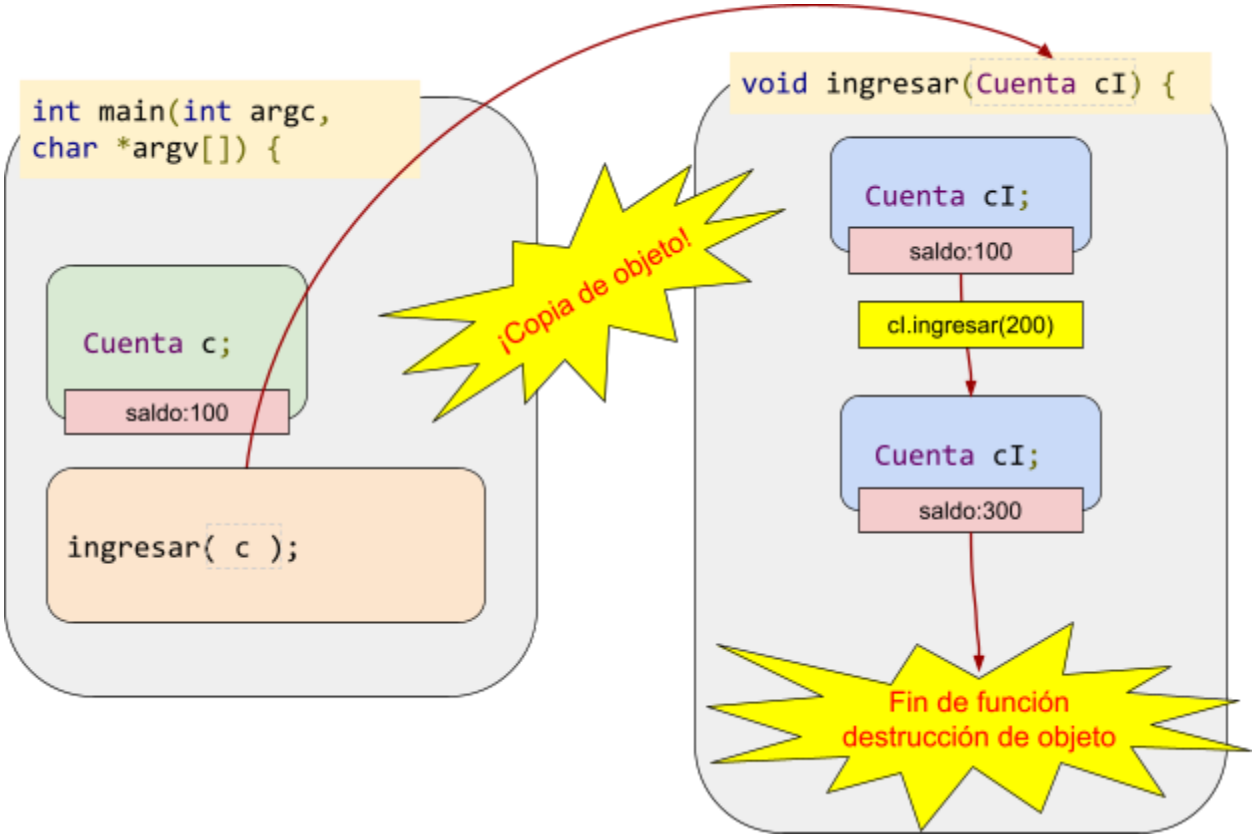
Preparemos la función así /(hemos vaciado main()) para que no se haga demasiado largo:

```
void ingresar(Cuenta c) {

    float cantidad;
    cout << "Escribe una cantidad: " ;
    cin >> cantidad;
    cout<< "Estás queriendo ingresar " << cantidad <<
       " euros en la cuenta "<< c.numCuenta << endl;
}

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

    Cuenta c;
    c.titular   = "Moncho";
    c.numCuenta = 100;
    c.saldo  = 0;

    ingresar(c);
    mostrar(c);
}
```

Al llamarlo para probarlo, el resultado es éste:

```
lliurex@HiDi-Tec:~/Documentos/Interfaces2016-2017/clasesI$ g++ -o programa
cuenta.cpp && ./programa
Escribe una cantidad: 33
Estás queriendo ingresar 33 euros en la cuenta 100
Cuenta num: 100 del fulano: Moncho tiene 0 leuros
lliurex@HiDi-Tec:~/Documentos/Interfaces2016-2017/clasesI$
```

Fíjate que el ingreso no se ha realizado , la cuenta sigue teniendo 0 euros después de haber intentado ingresar 5 euros.
¿Qué está ocurriendo? Se están copiando los valores entre la llamada y la función



Recuerda: C++ intenta copiar objetos, y por ello, en la llamada

```
ingresar(c);
```

se realiza una copia a otro objeto definido como parámetro de la función

```
void ingresar(Cuenta c) {
```

Toda la cuenta manejada dentro de `void ingresar(Cuenta c)` es una copia de la cuenta existente en la función principal main

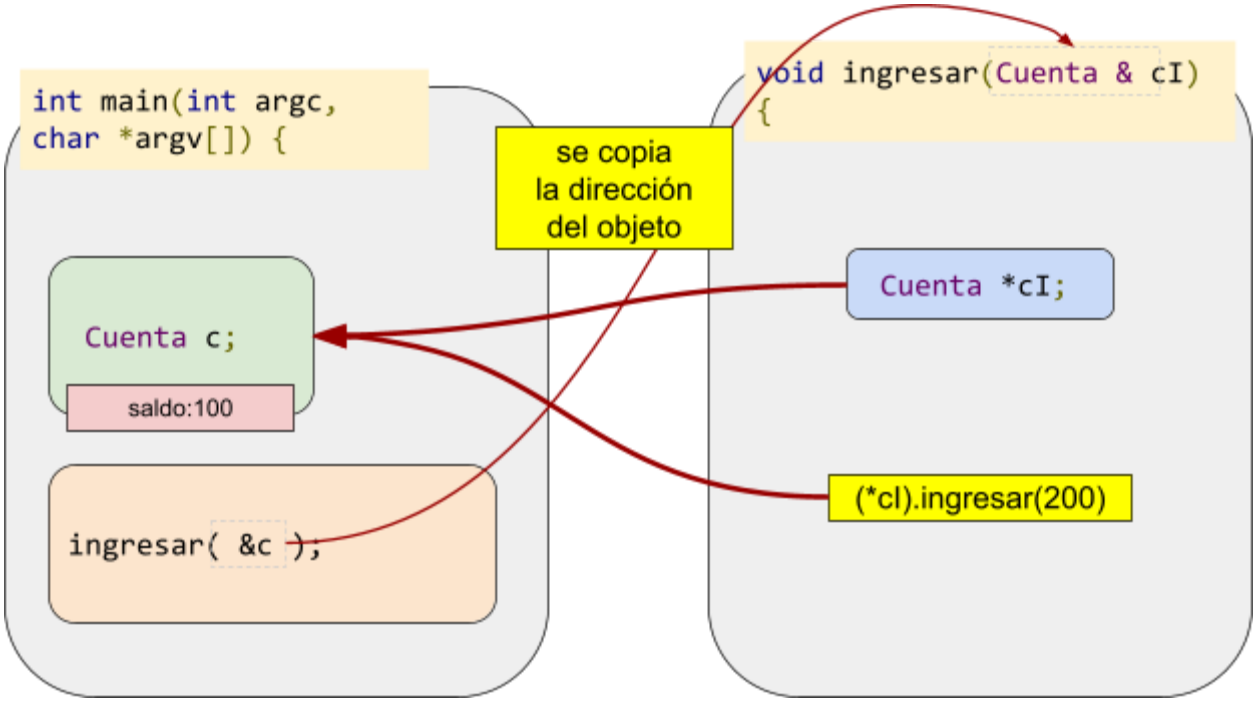
Paso de objetos por puntero.

Como se vió en el tema de punteros con valores enteros, esta manera de pasar datos a una función se denomina "por copia" porque se copian valores y nada que referencie al objeto original llega dentro de la función. Por ello, todo lo realizado dentro de la función ingresar se realiza con una copia
Vamos a pasar por puntero para solucionar el problema del paso por copia:

(En este punto debes saber algo sobre punteros, consulta los vídeos o la documentación sobre punteros para poder proseguir)

En un primer intento, cambiamos la función ingresar para que **recoja la dirección** de una cuenta (un puntero) y así poder acceder a un objeto externo a la función gracias a disponer de su dirección:

```
void mostrar(Cuenta cuentaMostrar){
     cout << "La cuenta " << cuentaMostrar.numCuenta <<
     " de " << cuentaMostrar.titular
    << " tiene " << cuentaMostrar.saldo <<
    " euros " <<endl;
}

void  ingresar(Cuenta * c) {

    float cantidad;
    cout << "DAme una cantidad que ingresar en la cuenta de "
         <<  *c.titular<< endl;  // ésta línea falla (1)
    cin >> cantidad;
    *c.saldo += cantidad;
}


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

      cout << "bienvenido al banco" <<endl;

       Cuenta c1, c2;
      /* JAVA añadiríamos c1 = new Cuenta(); c2 = new Cuenta(); */

      c1.titular = "Barbie";
      c2.titular = "Ken";
      c1 = c2;

      c1.numCuenta = "0001";
      c2.numCuenta = "0002";
      c1.saldo = 100000;
      c2.saldo = 200000;

      mostrar(c1);
      ingresar(&c1);
      mostrar(c1);

    }
```



Fíjate que la declaración del parámetro (`Cuenta * c`) es la declaración de un puntero. Dentro de la función debes **dereferenciar** el puntero para acceder a sus atributos o métodos

```
void  ingresar(Cuenta * c) {

    float cantidad;
    cout << "DAme una cantidad que ingresar en la cuenta de " <<
         (*c).titular<< endl; // ¡ BIEN !
    cin >> cantidad;
    (*c).saldo += cantidad;
}
```

```
lliurex@HiDi-Tec:~/Documentos/ejercicioC$ ./programa
bienvenido al banco
La cuenta 0001 de Ken tiene 100000 euros
DAme una cantidad que ingresar en la cuenta de Ken
12542
La cuenta 0001 de Ken tiene 112542 euros
lliurex@HiDi-Tec:~/Documentos/ejercicioC$
```

Dado que la forma siguiente de dereferenciar un objeto es algo incómoda:

```
(*c).titular
```

Se permite acceder a los métodos y atributos de un objeto del que se tiene un puntero, alternativamente con la flecha (símbolos "-" y ">") Observa el ejemplo

```
c->titular
```

Ambas formas son equivalentes.

Devolver un objeto como alternativa a usar punteros.

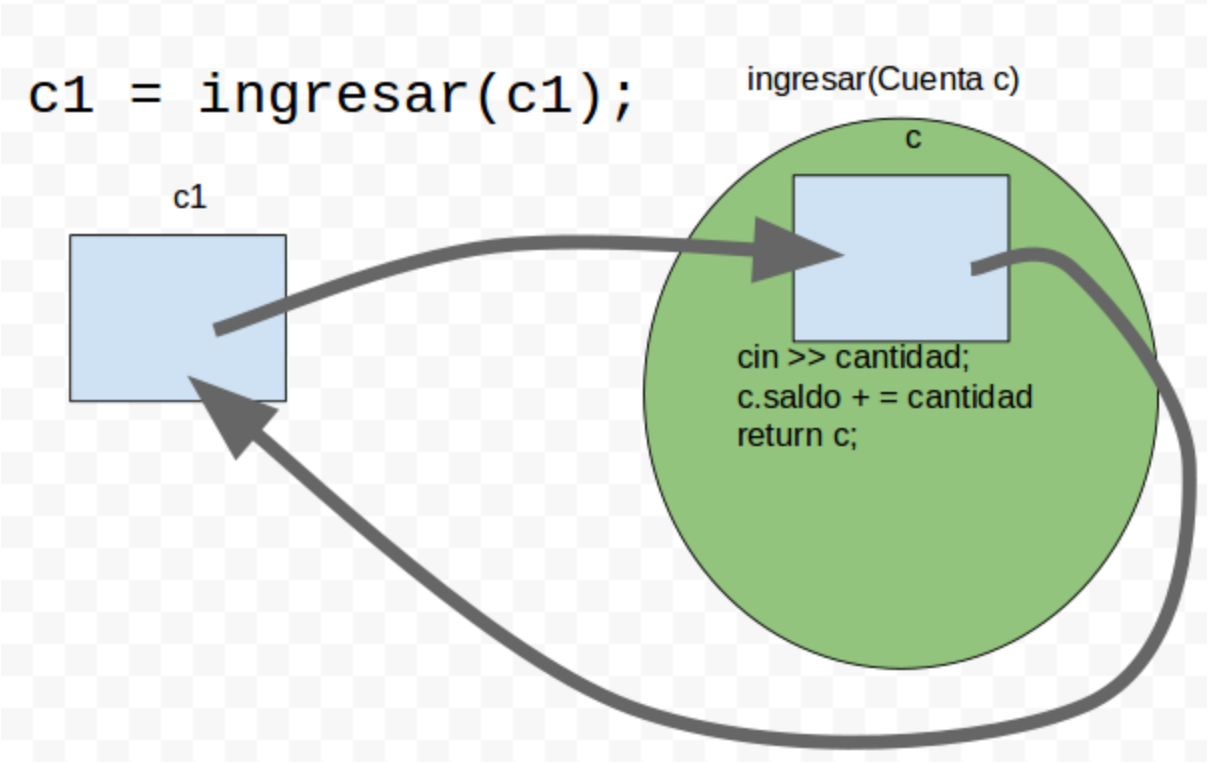
Existe, sin embargo, una alternativa para cumplir con el ejercicio sin tener que recurrir a los punteros. Aunque para ello, hay que cambiar la forma de llamar a la función y ser consciente de que la función no podrá alterar el objeto original ya que lo recibe por copia. Pero lo que sí puede hacerse es cambiar el objeto copiado a la función durante la llamada y devolver una copia a su vez de ese objeto cambiado. Después hay que usar la copia para cambiar el objeto original… ¡ Qué lío ! Mejor lo vemos implementado! Esta es la idea en la invocación:

```
c1 = ingresar(c1);
```

En esta línea se determina que la función ingresar va a devolver un objeto de tipo Cuenta. Se puede hacer esta afirmación porque es una igualdad, y lo que hay a los dos lados de la igualdad debe ser del mismo tipo. Además, también podemos afirmar que la función ingresar recibe como argumento un objeto de tipo Cuenta.

```
Cuenta ingresar(Cuenta c);
```

Esa sería la declaración adecuada a la llamada anterior. La implementaciòn y la descripción de lo que ocurre puede verse en el siguiente diagrama



La implementación es la siguiente y muestra la forma básica de devolver objetos:

```
Cuenta ingresar(Cuenta c) {

    float cantidad;
    cout << "DAme una cantidad que ingresar en la cuenta de " <<
         c.titular<< endl;
    cin >> cantidad;
    c.ingresar(cantidad);

    return c;
}
```

La forma correcta de realizar la llamada para materializar el cambio en una cuenta es

```
int main(...
    Cuenta c;

    c = ingresar(c);
}
```

Pasar los objetos por referencia

Si has aprendido bien los punteros y las referencias, sabrás que es posible usarlas para realizar el cometido de la función ingresar() sin usar punteros pero cumpliendo con la premisa de alterar el estado de la cuenta pasada (y no de una copia)

La declaración de la función quedaría

```
void  ingresar(Cuenta & c) {
```

y la llamada desde el cuerpo main()

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

      mostrar(c1);
      ingresar(c1);
      mostrar(c1);

    }
```

Observa cómo (por desgracia) la llamada es igual que por copia, pero ¡no hay copia porque usamos referencias!

La función quedaría así:

```
void  ingresar(Cuenta &c) {
    float cantidad;
    cout << "DAme una cantidad que ingresar en la cuenta de " <<
         c.titular<< endl;
    cin >> cantidad;
    c.ingresar(cantidad);
}
```

Ejercicio para hacer 2021-2022:

Implementa totalmente las funciones para ingresar y retirar dinero. Verifica su

```
correcto funcionamiento
```

Constructores

Un constructor es un método que se ejecuta cuando se crea un objeto.
Recordar ciertas reglas acerca de los constructores te eliminará muchas dudas en el futuro
Recuerda:

- Siempre que se crea un objeto se ejecuta un constructor.
- Puede haber varios constructores pero al crear un objeto se llamará a uno de ellos inicialmente
- Un constructor puede invocar a otros constructores (de la misma clase)
- El constructor suele utilizarse para inicializar los atributos de un objeto
- El constructor es un método más, excepto en que:
    - Se nombre igual que la clase. Clase y constructor coinciden en el nombre
    - No devuelve ningún tipo de dato, no se puede poner ni "void" al declararlo
- se puede escribir un constructor que no reciba argumentos y no haga nada. Este es el constructor vacío.

Hay un par de reglas más que pueden desconcertarte si las olvidas:

- Si el programador no escribe un constructor, C++ añade automáticamente un constructor vacío
- Si escribes un constructor, se quita el constructor vacío.

En el caso de las cuentas bancarias, un constructor podríamos usarlo para:

- Dar un saldo inicial de 0 a cualquier cuenta creada
- Exigir un número de cuenta y titular desde el principio.

Mi primer constructor vacío tendría la siguiente forma

```
class Cuenta {
public:
...
    Cuenta(){}
```

Pero si quiero un constructor para inicializar bien una clase, añadiré (o lo reemplazaré por, según interes):

```
class Cuenta {
public:
   ...
    Cuenta(string tit,string num) {
       titular = tit;
       numCuenta = num;
       saldo = 0;
    }
      Cuenta(){}
```

Ahora cuando creo un objeto de tipo cuenta puedo usar el nuevo constructor así:

```
    Cuenta c("pepe","001");
```

El programa entero queda así:

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

class Cuenta {
public:
    string titular;
    string numCuenta;

    Cuenta(string tit,string num) {
       titular = tit;
       numCuenta = num;
       saldo = 0;
    }
    Cuenta(){}

    void ingresar(float cantidad){
       saldo += cantidad;
    }

    bool reintegrar(float cantidad){
       if ( cantidad > saldo) return false;
       saldo -= cantidad;
       return true;
    }
    string info(void ) {
       std::stringstream s;
       s.str("");
       s << " la cuenta " << numCuenta << " de " <<                titular
<< " tiene " << saldo << " euros " <<endl;
       return s.str();
```

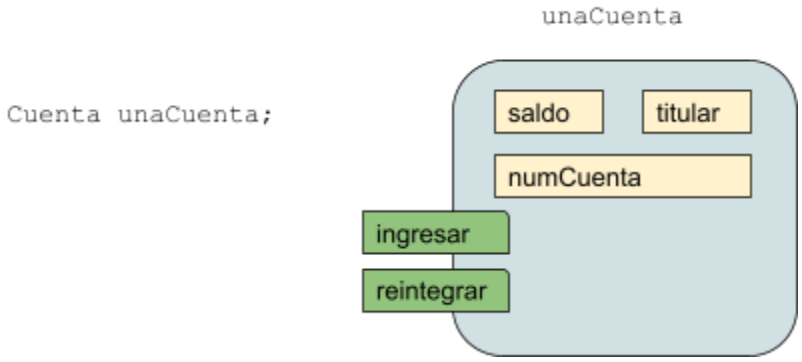
```
    }

private:
    float interes;
    float saldo;

};


void mostrar(Cuenta cuentaMostrar){
      cout << " la cuenta " << cuentaMostrar.numCuenta <<
     " de " << cuentaMostrar.titular
      << " tiene " << cuentaMostrar.saldo <<
    " euros " <<endl;

}

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

    Cuenta c("pepe","001");
    Cuenta c2;
    c.titular = "pepe";
    c.numCuenta ="001";

    c2 = c;
    cout << c.info()<< endl;

    c.ingresar(2000);
    if (c.reintegrar(1200))
       cout << "he podido sacar 1200 euros!!" << endl;
    else cout <<"No tengo suficiente para ir de botellón" << endl;

   mostrar(c);
}
```

Ejercicio para hacer 2021-2022:

Modifica el anterior programa eliminando el constructor vacío, observa los efectos

Arrays de objetos

En este tutorial, disponemos ya de la clase Cuenta,



Y hemos manipulado algunas cuentas mediante variables:

```
Cuenta c("pepe","001");
   Cuenta c2;
   c.titular = "pepe";
   c.numCuenta ="001";
```

En muchas ocasiones, un programa debe manipular una gran cantidad de objetos. Por ejemplo, nuestro programa de gestión de una oficina bancaria deberá almacenar y gestionar muchas diferentes cuentas.

Se plantea ahora la cuestión de ¿Qué ocurre cuando el número de cuentas a manipular es muy grande? No podemos crear cientos de variables. Hemos de recurrir a arrays o colecciones de objetos. Vamos a ver las distintas posibilidades, con sus ventajas y desventajas

Vector clásico de objetos

Como ya existía en C, podemos declarar un vector de objetos (llamémosle "cuentas") de la siguiente manera:

```
   Cuenta cuentas[10];
```

Esta declaración, crea un vector de 10 cuentas, que podemos representar de la siguiente manera

Con la declaración anterior, directamente disponemos de un vector de 10 cuentas, en las que ninguna de ellas ha sido inicializada a valores conocidos. Sin embargo, las cuentas existen y pueden ser manipuladas. La inicialización a valores conocidos debe hacerse a continuación

Mediante esta declaración disponemos de la posibilidad de manipular individualmente cualquier cuenta del vector. Si la variable cuentas es todo el vector, un elemento del vector sería

```
cuentas[7]
```

y dado que un elemento es un objeto (porque es un vector de objetos), podríamos manipular el objeto de la siguiente manera

ASignar un valor a un atributo

```
cuentas[4].titular = "Juanito";
```

Invocar un método sobre una cuenta del vector

```
cuentas[4].ingresar(100);
```

Acceder al valor de un atributo

```
float dinero = cuentas[3].saldo;
cout << "la cuenta es de " << cuentas[2].titular;
```

Asignación de objetos a y desde el vecto

Podemos recuperar todo un objeto del vector y asignarlo a otra variable

```
Cuenta miCuenta = cuentas[1] ;
```

Pero como es habitual, lo que logramos es hacer una copia. En el siguiente ejemplo, no estaríamos afectando para nada al elemento del vector que aparece, sino a la copia almacenada en la variable miCuenta

```
    Cuenta miCuenta = cuentas[1] ;
    miCuenta.ingresar(1000);
```

Igualmente podemos copiar un objeto encima de un elemento del vector. Arreglamos el ejemplo anteiror de la siguiente forma

```
    Cuenta miCuenta = cuentas[1] ;
    miCuenta.ingresar(1000);
    cuentas[1] = miCuenta;
```

Fíjate en el ejemplo anterior que hacemos una copia de cuentas[1] en la variable miCuenta, modificamos este objeto y finalmente copiamos todos los atributos otra vez del objeto miCuenta al elemento del vector

Problema con cuentas[10]

El problema con esta declaración es que desde el momento inicial tenemos 10 cuentas. Y quizá en nuestra oficina bancaria haya menos cuentas reales de las que hemos previsto en el vector. Y el estado del vector no refleja el estado de la oficina. Si en la oficina sólo hay 5 cuentas activas (por ejemplo) ¿ Por qué el vector tiene 10 cuentas creadas?

Por otra parte, el hecho de tener un vector de objetos puede hacer que en ocasiones sea difícil actualizar el estado de alguno de ellos, ya que lo que tenemos son copias. Por ejemplo. Supón la función realizaIngreso(Cuenta *cI) que vimos anteriormente y que recibe una cuenta por puntero (para poder modificarla)

```
void realizaIngreso(Cuenta * cI) {
    float cant;
    cout<<"Introduzca la cantidad a ingresar en la "
    <<"cuenta de "<< (*cI).titular <<":";
    cin >> cant;

    cout <<  "Vas a ingresar " << cant << " euros";

    //(*cI).ingresar(cant);
```

```
      cI->ingresar(cant);

      cout<<endl;
}
```

Para llamar convenientemente a esta función y realizar el ingreso sobre un elemento del vector deberíamos hacer

```
     realizaIngreso( * cuentas[4] );
```

Lo cual es un poco desconcertante en este momento. Lo siguiente no sirve

```
      Cuenta unaCuenta = cuentas[4];
      realizaIngreso( * unaCuenta) ;
```

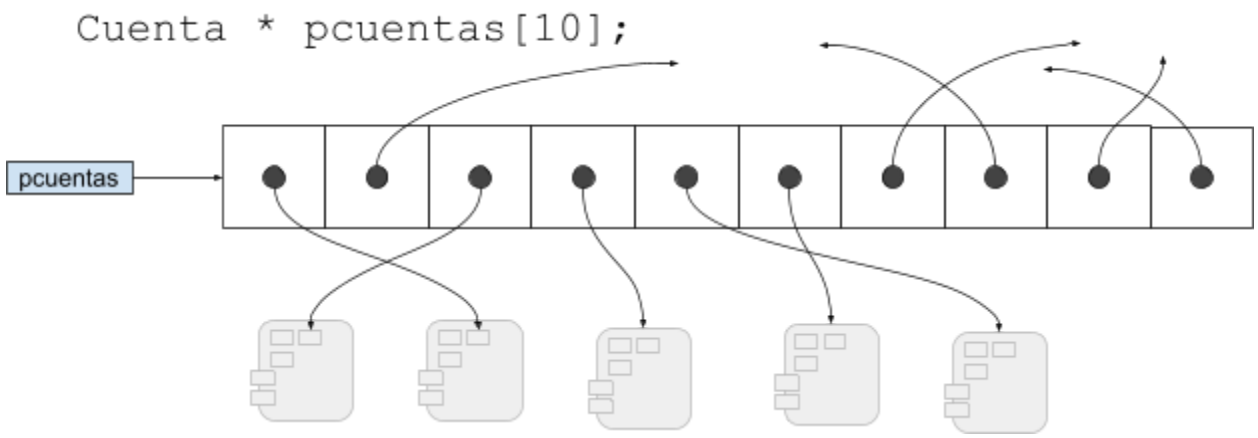
Otra vez estaríamos pasando una copia a la función realizaIngreso y no el objeto original

Vector de punteros

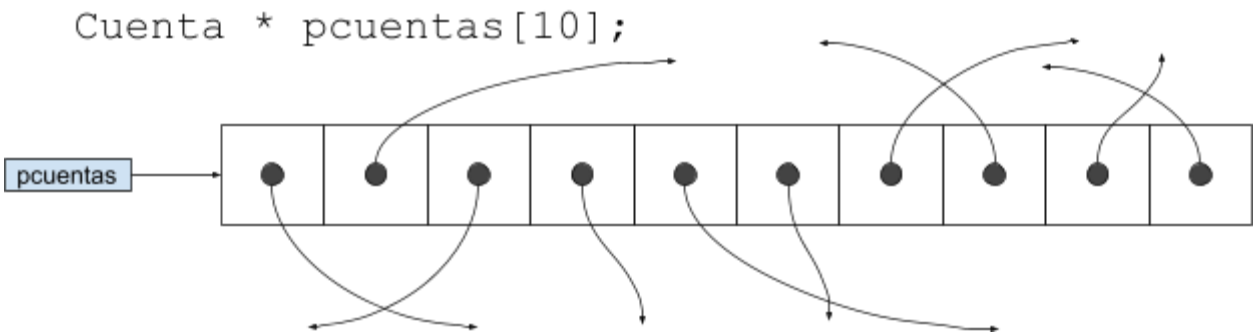
Por estos dos inconvenientes vamos a plantearnos la posibilidad de utilizar otro vector. En este caso, vamos a almacenar punteros a cuentas dentro del vector. Es decir, cada elemento del vector es un puntero. La declaración de un array de punteros es:

```
     Cuenta * pcuentas[10];
```

Ahora dibujemos qué es lo que hemos creado



Cada elemento del vector, contiene una dirección de memoria. Si esa dirección de memoria apunta a una cuenta real o al quinto cuerno, no lo sabemos…. bueno… sí lo sabemos. Nada más haber declarado el vector, cada elemento apunta al quinto cuerno.

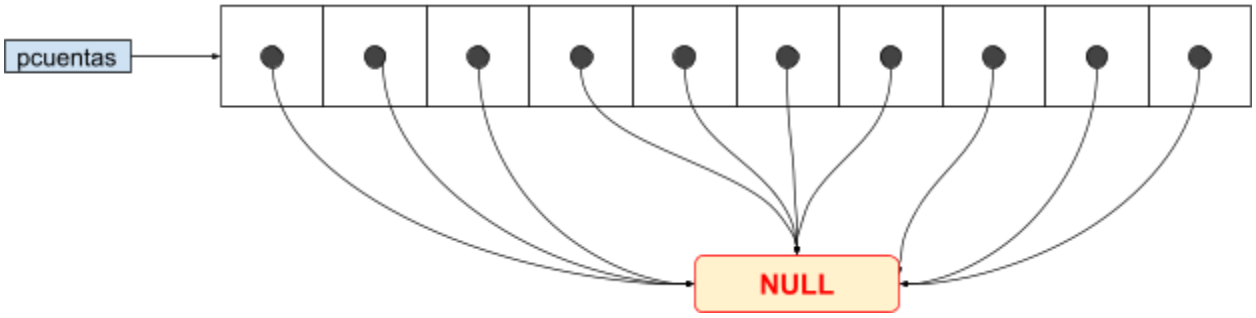


Lo primero que debemos hacer es lograr que todos los punteros apunten a la dirección especial NULL que indica que el vector no apunta a un objeto válido.

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

Ahora , todos los elementos apunta a NULL

```
Cuenta * pcuentas[10];
...
for (int i=0;...
   cuentas[i] = NULL
```



El hecho de inicializar todos los elementos a NULL permite que podamos en el futuro preguntar si un elemento del vector ha sido asignado a una cuenta o no:

```
if (pCuentas[3] != NULL )
   pCuentas[3]->ingresar(1000);
```

Crear cuentas

Con el vector anterior no tenemos ninguna cuenta creada al inicio. ¡Es lo que queríamos ! pero ahora deseamos ya crear una cuenta y poder usarla. Para ello hemos de crear la cuenta dinámicamente y referenciar esta cuenta desde el vector. Básicamente tenemos dos formas:

```
Cuenta nueva;
pcuentas[0] = * nueva;
```

o usando new

```
Cuenta * nueva = new Cuenta;
pcuentas[0] = nueva
```

o usando new directamente (lo mismo que arriba pero más compacto)

```
pcuentas[0] = new Cuenta;
```

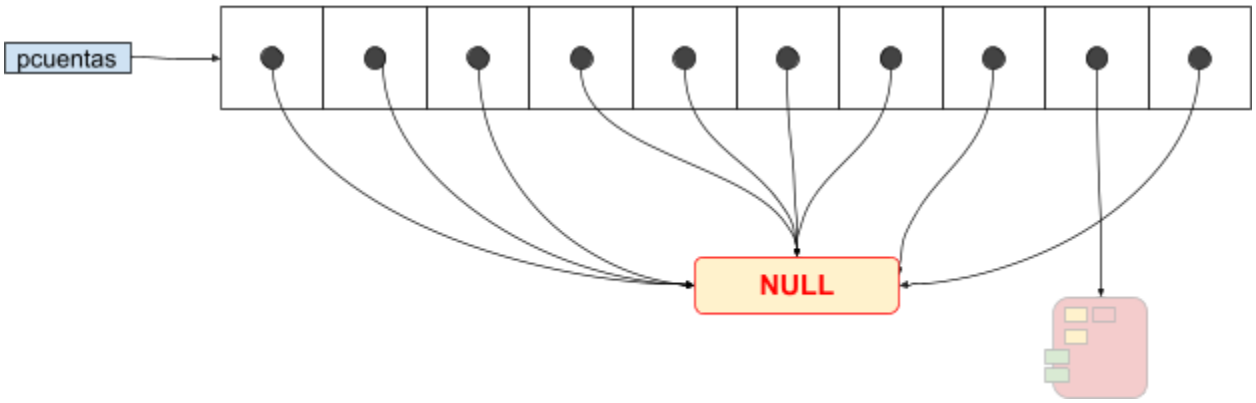
¿Cuál es la diferencia?

- Con la declaración de una variable normal, como en el primer ejemplo, el objeto creado sólo existe mientras exista la función donde se declaró. Si esto se hace en una función , al terminar la función la cuenta se destruye y el puntero almacenado en pcuentas[0] apunta a algo inválido
- No se puede repetir la línea primera, ya que la variable Cuenta nueva, sólo crea una única variable independientemente de las veces que se pase por ahí. Es decir, esa línea no podría estar dentro de un bucle
- De la segunda manera:
    - Se crea un objeto que no se va a destruir aunque se cree dentro de una función
    - Se puede repetir esa línea en un bucle y cada vez se crea una variable nueva:

Podríamos por ejemplo, crear 5 cuentas de la siguiente forma

```
for (int i= 0 ; i < 5 ; i++ )
      cuentas[i] = new Cuenta();
```

```
...
pcuentas[8] = new Cuenta
```



En el ejemplo anterior, qudarían 5 elementos por inicializar (a NULL si hemos hecho la preinicialización del vector anterior)

Manipular un vector de punteros.

Antes, cuando teníamos el vector de objetos declarado como:

```
Cuenta cuentas[10];
```

Accedíamos a un atributo de un elemento de la siguiente forma:

```
cuenas[1].saldo = 0;
```

Ahora, declarando el vector de la siguiente forma:

```
Cuenta * cuentas[10];
```

Cada elemento es un puntero, no un objeto. Para acceder al objeto hay que dereferenciar el puntero (es decir: acceder a lo apuntado por el objeto). Recuerda que cuando tienes un puntero, anteponer el asterisco * hace que accedas a lo apuntado.

```
( * (cuentas[1] )  .  saldo = 0;
```

Se han puesto paréntesis de sobra, pero hay alguno necesario. *cuentas[1].saldo no funcionaría.

la idea es la siguiente:

- `cuentas` es un array o vector
- `cuentas[1]` es un elemento del array, es decir, un puntero
  - `cuentas[1].saldo` no es nada porque no existe el operador punto "." para punteros no podemos acceder a un atributo de un puntero,
- `*cuentas[1]` es un objeto, ya que si cuentas[1] es un puntero, poniendo el asterisco delante obtenemos lo apuntado.
- `*cuentas[1].saldo` tiene un problema grave y es que el compilador resuelve antes el punto que el asterisco, es decir
  - La agrupación de operaciones en la expresión anterior que no lleva paréntesis desearíamos que fuese la siguiente

    ```
    (* ( cuentas [1] ) ). saldo
    ```

  - Pero realmente es así (si no escribimos paréntesis)

    ```
    * ( ( cuentas [1] ) . saldo)
    ```

  - Y como hemos visto antes, esto no funciona.

Existen dos soluciones,

- Como ya se ha visto, escribir paréntesis siempre permite controlar el orden en el que se ejecutan los operadores. Así escribiríamos

  ```
  (*cuentas [1]). saldo
  ```

- Utilizar una notación especial para este caso que por ser tan habitual ha requerido de esta notación

  ```
  cuentas[1]->saldo
  ```

A partir de aquí , manipular los objetos, mediante un vector de objetos no es más complejo que añadir un * o usar la flecha "->". Pero aún así hay que ir con cuidado con las copias. Por ejemplo, hacer un ingreso.

```
( * pCuentas[3]).ingresar(1000);
```

Usando la flecha es más bonito

```
pCuentas[3]->ingresar(1000);
```

Se puede usar un puntero intermedio. No problemo, funciona

```
Cuentas * unaCuenta = pCuentas[3];
unaCuenta->ingresar(1000);
```

Pero ojo con copiar objetos, en el siguiente ejemplo se haría una copia y el ingreso sobre una copia.

```
Cuentas  unaCuenta = *(pCuentas[3]);
unaCuenta. ingresar(1000);
```

Recorrido del vector.

Dado que el vector es de punteros **y hemos inicializado a NULL** todos sus elementos al principio, hemos de ir con cuidado cada vez que accedemos a un elemento, no vaya a ser que todavía apunte a null.

Por ejemplo para sumar todos los saldos de las cuentas. Lo siguiente sería mala idea:

```
int suma = 0 ;
for (int i=0; i< 10 ; i++)
```

```
    suma += pcuentas[i]->saldo;
```

Porque algunos pcuentas[i], pueden apuntar a NULL. Hay que verificar cada vez si el puntero es NULL o no

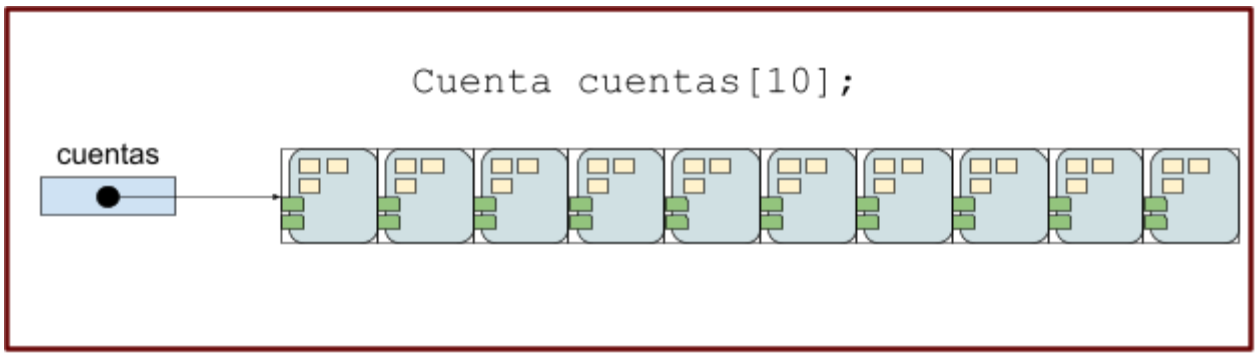
```
int suma = 0 ;
for (int i=0; i< 10 ; i++)
    if (pCuentas != NULL)
        suma += pcuentas[i]->saldo;
```

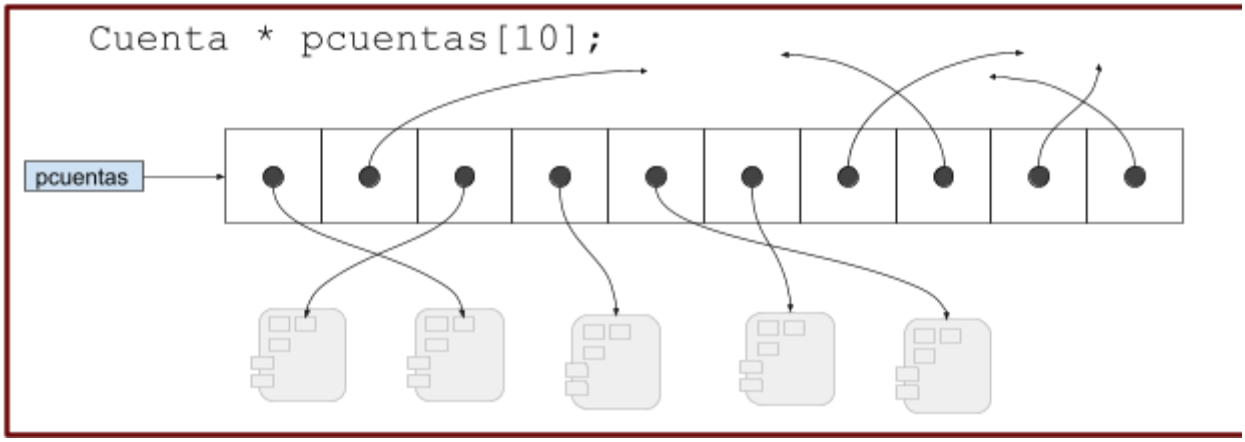
Inicialización del vector

Vamos a inicializar totalmente algunas cuentas dentro del vector.

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

     string nombres[5]={"juan", "perico","andres",
                                "fulano","vengano"};

    for (int i = 0; i< 5; i++ ) {
       /*formar un string con un num */
        stringstream ss;
                ss << i;
                string str = ss.str();

       pcuentas[i] = new Cuenta(nombres[i], str);

     }
```

Resumen:





Objetos como atributos de otros objetos.

Usando Templates de C++

Los vectores usados hasta ahora son incómodos y arcaicos. seguramente estés añorando el ArrayList de Java. C++ proporcionó una alternativa mucho más moderna a estos vectores arcaicos . Estas alternativas están explicadas en un documento aparte. que es muy extenso. Aquí tienes una versión reducida.

Para usar la clase vector Necesitamos realizar un #include más:

```
#include <vector>
```

std::vector de Cuentas

Ahora declaramos nuestro nuevo objeto que sustituye o reemplaza los vectores anteriores

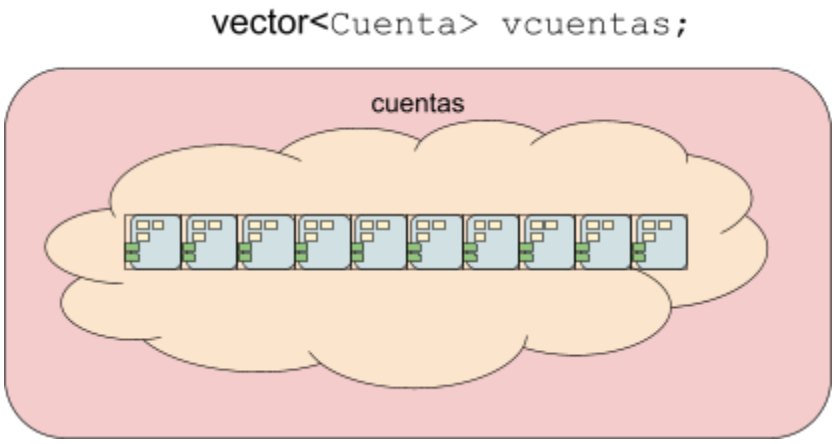
```
//    Cuenta cuentas[10];         // ANTES
//    Cuenta * pcuentas[10];      // ANTES
      vector<Cuenta> vcuentas;    // AHORA!!
```

Con esta declaración obtenemos **un objeto que se puede comportar como si fuese un vector pero sigue siendo un objeto**.

Características de esta nuevo objeto declarado como `vector<Cuenta> vcuentas;`:

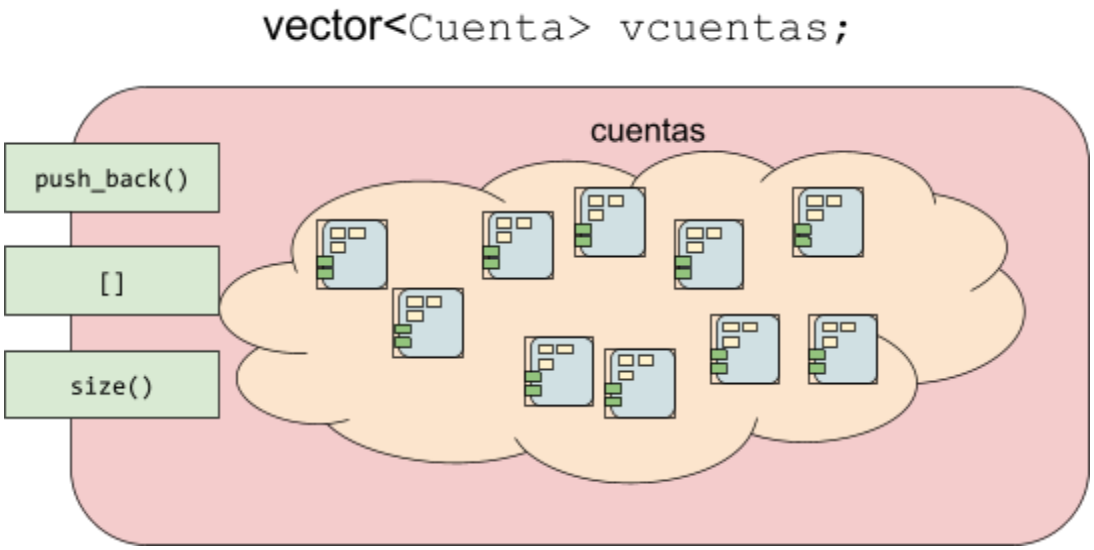
- El objeto existe totalmente desde esa declaración, pero no tiene Cuentas en su interior (el vector está vacío)
- Podemos manipular elementos (Cuentas) del vector individualmente como si de un vector real se tratase
- Podemos saber cuántos elementos tiene el vector.
- Podemos añadir nuevas Cuentas (de momento, siempre por el final)
- Podremos eliminar Cuentas del vector (aunque rara vez lo usaremos)
- Podremos pasar el objeto (vector) entero como copia, referencia o por puntero a funciones.
- Cada vez que añades un elemento, se crea una copia y lo que almacena en el interior es la copia del elemento.

Podríamos representar la variable vcuentas creada como un objeto que en su interior almacena de alguna forma algo que puede representar una colección ordenada de elementos, esto es: un vector



Aunque realmente para mejorar la representación gráfica del objeto creado, vamos a añadir dos ideas al dibujo:

- El objeto se maneja mediante métodos y operadores de la clase vector<>
- No sabemos cómo están dispuestas en memoria las cuentas que almacena el vector, vamos a dibujarlas de otra forma:



Añadir elementos,

por ejemplo en la función crearVCuentas(vcuentas).

```
void crearVCuentas(vector<Cuenta> & datos) {
  string nombres[5]={"juan", "perico","andres",
```

```
                                    "fulano","vengano"};
    int i=0;

    while ( i < 5) {
      Cuenta c;
      c.saldo= i * 100;
      c.titular = nombres[i];
      c.numCuenta = "por inicializar";
      datos.push_back(c);
       i++;
    }
}

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

    vector<Cuenta> vcuentas;
    crearVCuentas(vcuentas);
```

- El objeto vcuentas es pasado por referencia en la llamada a crearVCuentas. De lo contrario estaríamos inicializando una copia del objeto.
- La cuenta c no se almacena en el vector, sino una copia suya

Acceder a un elemento

Por ejemplo en la función mostrarVCuentas;

```
void mostrarVCuentas(vector<Cuenta> datos) {
    int i=0;

    while ( i < datos.size() ) { //.length()
            cout<< "titular: "<< datos.at(i).titular;
            cout<< "  saldo: "<< datos[i].saldo;
            cout << endl;
       i++;
    }
}
```

Observa cómo no hay diferencia (si así lo quieres), se usan los corchetes igual, pero podemos usar también algún método para acceder a un objeto, en este caso el método .at(i) que devuelve un elemento del vector… esto es… una Cuenta

Observa también cómo ahora podemos consultar la longitud o número de elementos del vector con

datos.size();

std::vector de punteros a Cuentas

El problema con la declaración anterior

```
vector<Cuenta> cuentas;
```

Es que las cuentas están en el interior del objeto cuentas, las posiciones de memoria se pueden conocer, pero ya no es tan correcto manipular esas direcciones porque el objeto cuentas es libre de cambiar internamente dónde almacena los objetos

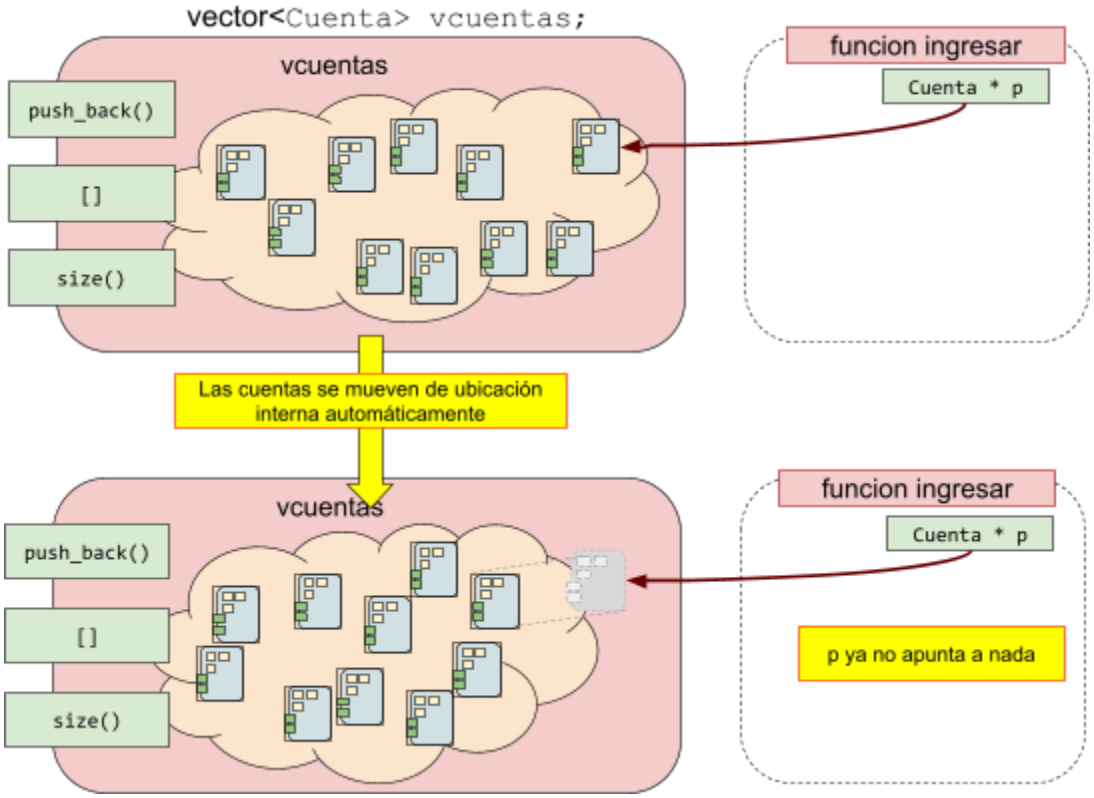
Observa el siguiente ejemplo, donde la función ingresar recibe un puntero a una Cuenta. ¿Cómo le pasamos ese puntero cuando tenemos un objeto de tipo vector<Cuenta> ?

```
void  ingresar(Cuenta * c) {
   ...
}

int main(...   {
      vector<Cuenta> cuentas;
      ...
      inicializarVCuentas(cuentas);

     Cuenta * pCuentaAIngresar = &(cuentas[3]);
      ingresar(pCuentaAIngresar);
      ...
     pCuentaAIngresar->titular...
     pCuentaAIngresar->saldo();
}
```

El problema es que la ubicación real donde está la cuenta es interna al objeto vector<Cuenta> y éste puede cambiarla cuando quiera, nadie asegura que la posición de memoria es siempre la misma..

Ejemplo vida real. A las 7 de la tarde dejas la mochila en habitación compartida . A las 12 has de ir a la mochila a coger el cepillo de dietnes, habiendo memorizado el camino porque no puedes encender la luz. Pero a las 8 de la tarde alguien ha cambido de sitio tu mochila y por la

noche entras caminas de memoria hasta el lugar donde crees que está tu mochila y allí ahora hay una mochila de otro, y coges el cepillo de dientes equivocado.



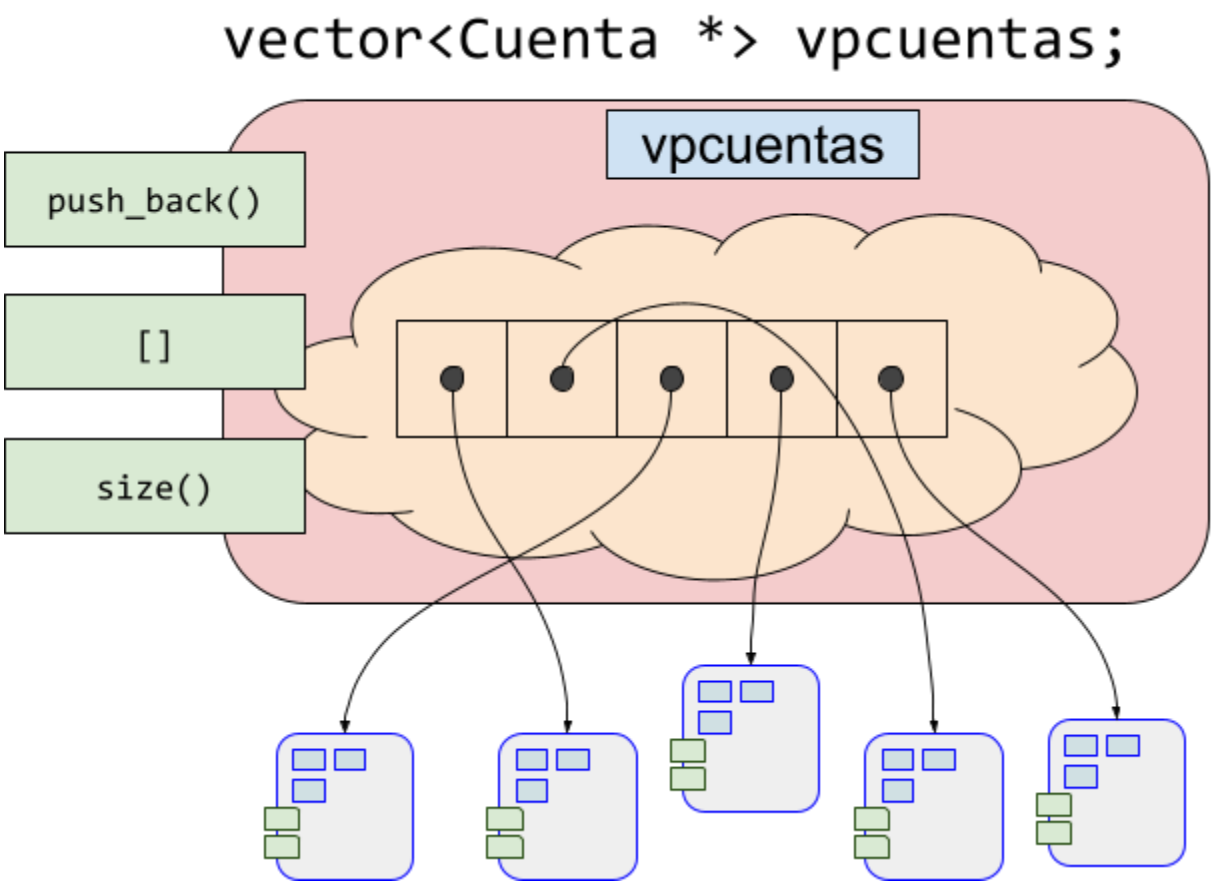
Estamos en una situación similar a cuando pasamos de

```
Cuenta cuentas[10];
```

a

```
Cuenta * cuentas[10];
```

Necesitamos tener los objetos de las cuentas "libres" por la memoria, sin estar dentro del vector, pero tener en el vector todas las direcciones de las cuentas. Esta situación y la declaración la muestra la siguiente imagen



Date cuenta de que el vector podría mover internamente los elementos del vector que son punteros, pero las cuentas no, están donde están fijas (ya que han sido creadas mediante "new Cuenta") y a los punteros del vector siempre podríamos acceder así:

```
vector<Cuenta *> cuentas;          // declaración del objeto vector
...
Cuenta *nueva = new Cuenta;        // creamos una cuenta
nueva->titular = "Joselito";       // configuramos la cuenta
cuentas.push_back(nueva);          // añadimos su dirección al vector
...
Cuenta *p = cuentas[2];            // así recuperamos la cuenta
```

Ahora, llamar a funciones a las que se les debe pasar un puntero o un objeto no resulta difícil

```
void  ingresar(Cuenta * c) {
    ...
}
```

```
void  ingresar(Cuenta & c) {
   ...
}
int main(...   {
      vector<Cuenta * > cuentas;
      ...
      inicializarVCuentas(cuentas);

     Cuenta pCuentaAIngresar = cuentas[3];
     ingresar(pCuentaAIngresar);
     retirar(*pCuentaAIngresar);
      ...
     pCuentaAIngresar->titular...
     cuentas[3]->saldo();

}
```

vvv