SEMINARIO C++

Introducción a la Programación Orientada a Objetos

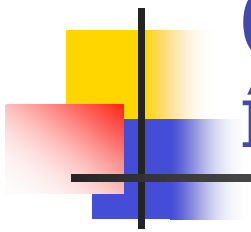
Parte I

v. 20070918

Cristina Cachero
Pedro J. Ponce de León

Depto. Lenguajes y Sistemas Informáticos - Universidad de Alicante

Universitat d'Alacant
Universidad de Alicante

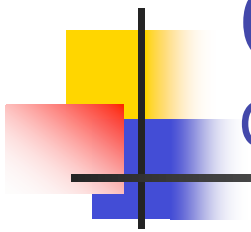


C++
ÍNDICE

C++
CLASES : ESTRUCTURA

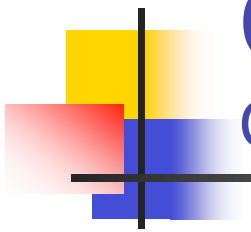
- Una clase es un tipo de dato (TAD) definido por el usuario.
- Constituye una plantilla a partir de la cual se instancian objetos.
- La clase define un conjunto de objetos que comparten las mismas propiedades: atributos, operaciones y roles.
  - **Atributos** *(datos miembro en C++)*: información que cada uno de los objetos instanciados a partir de ella desea guardar de sí mismo
  - **Relaciones/Roles**: referencias que cada uno de los objetos guarda acerca de los objetos con los que está relacionado
  - **Métodos** (*funciones miembro en C++*): instrumentos para la manipulación de atributos/relaciones

Pers
string nombre string dni
void setNom(string n) string getNom() bool anyadeAsig(Asig &a)



C++
CLASES : ESTRUCTURA

- **Instanciación de objetos**
  - Cuando tratamos con tipos predefinidos, siempre puede declararse una variable de ese tipo.
    - E.g. variable **x** del tipo int : `int x;`
  - Cuando tratamos con clases, no existe predefinición, por lo que antes de utilizarlas debemos "definirlas"
    - `class CL {/* defin. de la clase */ };`
  - Una vez definida, una clase se puede instanciar igual que cualquier tipo predefinido.
    - `CL c1; // declara objeto c1 de tipo CL.`

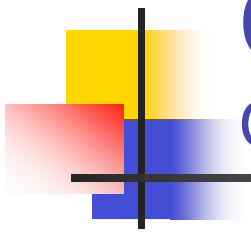


C++

CLASES : VISIBILIDAD (ÁMBITO)

- En la declaración de una clase para cada propiedad (atributo, operación, rol), debe especificarse mediante los **modificadores de acceso**, el ámbito desde el cual puede accederse a dicha propiedad.
- En C++ hay tres modificadores de acceso:
  - **Private (-)** : Sólo se permite su acceso desde las funciones miembro (métodos) de la clase.
  - **Public (+)**: Se permite su acceso desde cualquier punto que pueda usar la clase. Un dato público es accesible desde cualquier objeto de la clase.
  - **Protected (#)**: Se permite su uso en los métodos de la clase y en los de las clases derivadas mediante herencia.

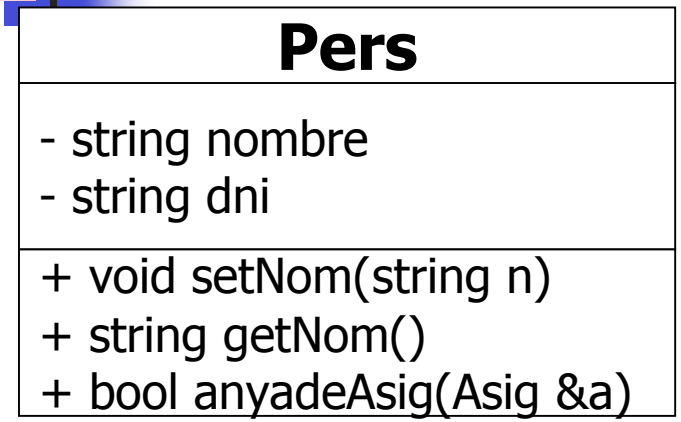
Pers
- string nombre - string dni
+ void setNom(string n) + string getNom() + bool anyadeAsig(Asig &a)



C++
CLASES : ESTRUCTURA, ÁMBITO

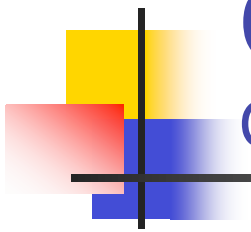
```
class <nombre_clase>
{
   public:
   ...
        Parte pública de la clase. Interfaz que ésta
   ofrece a los usuarios para que éstos manejen los
   datos.
   private:
   ...
        Parte privada de la clase. Incluye la
   declaración de los datos. Visibilidad por defecto de
   una clase (dif. respecto a struct).
};
```

C++
CLASES : ESTRUCTURA, ÁMBITO



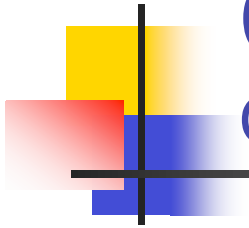
Fichero declaración (.h)

```
class Pers
{
   public:
        void setNom(string n);
        string getNom();
        bool anyadeAsig (Asig &a);
   private:
        string nombre;
        string dni;
};
```



C++
CLASES : ENCAPSULACIÓN

- En la POO rige el principio de ocultación de información, según la cual el usuario NO debería acceder nunca directamente a los datos de un objeto si no es a través de alguna operación (interfaz de manipulación) proporcionado por dicho objeto.
  - Eso implica que normalmente sólo la declaración de operaciones es pública, quedando oculta a los usuarios de la clase la forma en la que se han programado las distintas funciones y operadores miembro, así como los datos miembros y sus tipos.
- **Ejercicio**: modificad Pers.h para que el nombre sea un puntero a tipo char. ¿Qué otras cosas hay que modificar para que cualquier programa que utilice esa clase siga funcionando? ¿Qué pasaría si el atributo *nombre* fuera público?



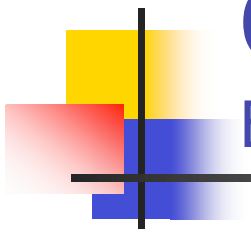
C++
CLASES : DEFINICIÓN EN C++

PARTES DE UN PROGRAMA:

- **Declaración de datos/funciones/operadores miembro**
  - **Datos miembro:** variable y su tipo que representan atributos y roles.
  - **Funciones/operadores miembro**: Como cualquier función C++. Consiste en especificar los prototipos de los métodos de la clase. En algunos casos, también se puede incluir su definición (funciones *inline*)
- **Definición de código asociado a cada función/operador miembro.**
  - Se copia el prototipo especificado en el .h y se añade el código correspondiente.
  - Las ***funciones y operadores miembro*** de una clase se definen anteponiendo a su nombre el nombre de la clase y el ***operador de resolución de ámbito (::)*** :

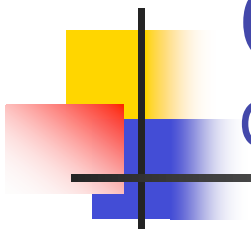
```
void Pers::setNom(string n)
{ //cuerpo de la función miembro }
```

- **Definición del punto de entrada al programa** int main(...){...};



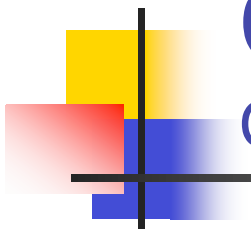
C++
ESTRUCTURA DE DIRECTORIOS

- **DISTRIBUCIÓN DE CÓDIGO EN C++**
  - Declaraciones en .h: directorio **include**
  - Código (definiciones) de clases y código objeto en directorio **lib**
  - Punto de entrada (main.cpp): directorio **src**
  - Documentación en directorio **doc**



C++
CLASES :

- **ORDEN DE LAS OPERACIONES EN LA DECLARACIÓN DE UNA CLASE**
  - Los rasgos más importantes deberían estar listados al inicio de la declaración de la clase.
    - Los constructores son uno de los aspectos más importantes de la definición de un objeto y por tanto deberían aparecer al ppio de la declaración
  - La declaración de métodos debería estar agrupada para facilitar la localización del cuerpo asociado con un determinado selector de mensaje (nombre de operación).
    - Orden alfabético
    - Agrupación en función de propósito
  - Los datos privados sólo son importantes para el desarrollador de la clase. Por tanto deberían estar listados hacia el final de la definición de dicha clase.



C++
CLASES : OPERACIONES INLINE

- Funciones *inline*: Funciones a las que no se llama realmente sino que el compilador inserta su código en el lugar de cada llamada
  - Ventajas: Mayor rapidez de ejecución
  - Inconvenientes:
    - Si es demasiado larga y se la llama demasiado a menudo, el programa aumentará su tamaño
    - El declarar una función inline no obliga al compilador a expandirla.
- Declaración de funciones *inline*
  - Cuerpo directamente en la declaración de la clase (.h)
  - ó palabra reservada ***inline*** delante de la definición de la función

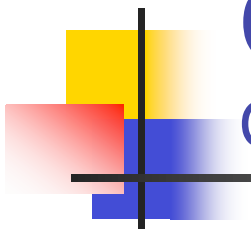
C++
CLASES : EJEMPLO

Mi primera clase

```
//rectangulo.h
class rectangulo {
public:
   void dimensiones(int,int);
   int area() // método inline
   {
        return base*altura;
   }
private:
   int base, altura;
};
```

```
//rectangulo.cpp
#include "rectangulo.h"
void rectangulo::dimensiones(int b,int h){
        base=b;
        altura=h;
}
```

```
//main.cpp
#include <iostream>
#include "rectangulo.h"
using namespace std;
/*o include iostream.h sin namespace*/
int main(){
        rectangulo r;     // declaro objeto
        r.dimensiones(3,5);// defino tamaño
        cout << "Area: " << r.area();
}
```



C++
CLASES : EJERCICIO

Ejercicio

1.- Ignorad la división en ficheros, y definid este código en un solo fichero llamado **todo.cpp**

Tendréis que eliminar directiva #include "rectangulo.h"

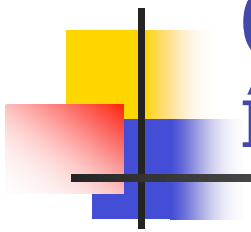
2.- Compilad el programa con **g++ –o rect todo.cpp**

3.- Dividid ahora ese código en los tres ficheros especificados (rectangulo.h, rectangulo.cpp y main.cpp)

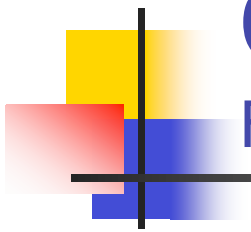
4.- Compilad el programa de nuevo con

g++ –o rect2 rectangulo.cpp main.cpp

5.- Probad a hacer lo mismo poniendo cada fichero en su directorio correspondiente (.h en *include,* .cpp en *lib* y main.cpp en *src*)



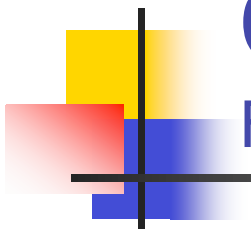
C++
ÍNDICE



C++
FUNCIONES O MÉTODOS SET/ GET / IS.

- Por el principio de encapsulación, casi nunca es conveniente acceder directamente a los atributos de una clase. Lo usual es definirlos como **private** y, para acceder a ellos, implementar funciones **set/get/is** (llamadas tb **ACCESORES**) (getDNI(), setEdad(), isVaron(),…)

- Los métodos **get** permiten representar la información dentro de la clase de una forma y proporcionarla de una forma distinta.
  - Por ejemplo, el DNI de una persona puede representarse internamente como un atributo de tipo long y otro de tipo char, pero el método getDNI() podría devolver el DNI como una cadena de caracteres.



C++
FUNCIONES O MÉTODOS SET/ GET / IS.

- Los métodos **set** permiten hacer algo más que simplemente asignar un valor a un atributo (por ejemplo, validar dicho valor) de forma transparente al usuario de la clase.

```
void setEdad(int laEdad) {
  if (laEdad<0) edad=0;
  else edad=laEdad;
}
```

- Por otro lado, para consultar si un atributo tiene o no tiene un determinado valor/rango de valores, por convenio utilizamos funciones de tipo **is**
  - P. ej. `isMatriculado();` /*nos dice si el alumno tiene alguna aignatura asociada o no*/
- Estas funciones debemos declararlas, si procede, como *inline.*

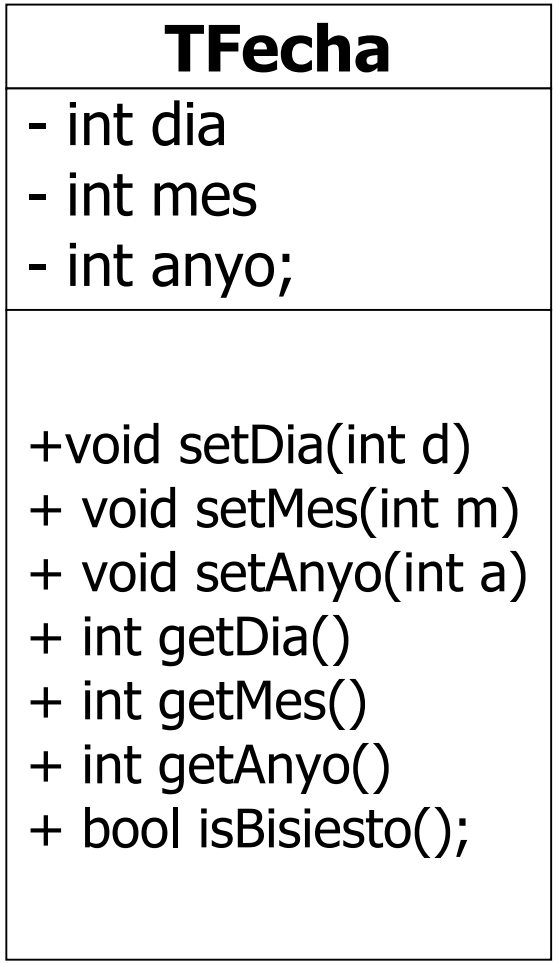
C++
FUNCIONES O MÉTODOS SET/ GET /IS.

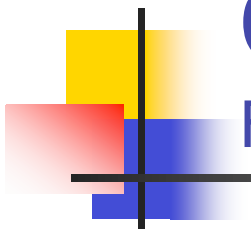
- **Ejemplo: clase TFecha mediante el uso de funciones inline.**

```
•g++ –o fecha TFecha.cpp main.cpp
```

- **Comprueba que funciona con el siguiente fichero:**

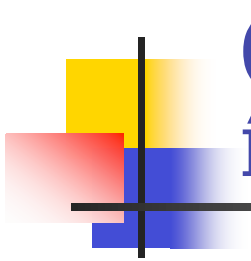
```
/* main.cpp */
#include <iostream.h>
#include "TFecha.h"
int main() {
 TFecha p;
 p.setDia(10);
 p.setMes(10);
 p.setAnyo(2007);
 cout << p.getDia( );
 cout << p.getMes( );
 cout << p.getAnyo( );
}
```



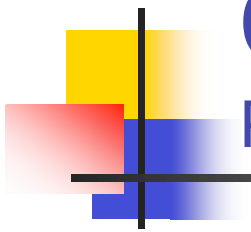


C++
FUNCIONES O MÉTODOS SET/ GET.

```
class TFecha {
public:
   void setDia(int d) {dia =d;}; //inline
   void setMes(int m) { mes = m; }   // inline
   void setAnyo(int a) { anyo = a; } // inline
   int getDia() { return dia; }      // inline
   int getMes() { return mes; }      // inline
   int getAnyo() { return anyo; }      // inline
   bool esBisiesto {return ((anyo%4)==0);}
   //inline
private:    // Parte privada de la clase
   int dia, mes, anyo;
};
```



C++
ÍNDICE



C++
REFERENCIA (&).

- Una **referencia** es simplemente otro nombre o alias de una variable. En esencia es equivalente a un puntero (contiene la dirección de un objeto), pero funciona de diferente modo, ya que **NO** se puede modificar la referencia en sí, pero sí el valor de la variable a la que está asociada.

Usando variable referencia

```
int i;
int &x=i;//x es un alias de i
x=40;// i vale 40
```

Usando punteros

```
int i;
int *p=&i;
*p=40;//i=40
```

C++
REFERENCIA (&).

PASO DE PARÁMETROS A UNA FUNCIÓN

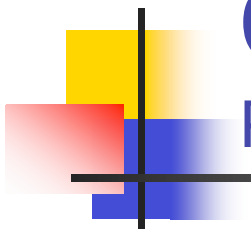
- **Paso por VALOR.** Al compilar la función y el código que llama a la función, ésta recibe una **COPIA** de los valores de los argumentos. Las variables reales no se pasan a la función, sólo copias de su valor.

- **Paso por REFERENCIA.** La usamos cuando la función debe modificar el valor de la variable pasada como parámetro y que esta modificación retorne a la función llamadora. El compilador NO pasa una copia del valor del argumento, sino una *referencia*, que le indica dónde existe la variable en memoria. La *referencia que una función recibe es la dirección de la variable .* En C++ todos los arrays son por **referencia**.
- **Ejemplo**

```
void demo(int& valor){
 valor=5;
 cout<<valor<<endl;
}
```

```
int main() {
 int n=10;
 cout<<n<<endl;
 demo(n);
 cout<<n<<endl;
}
```

¿Cuál será la salida del programa?

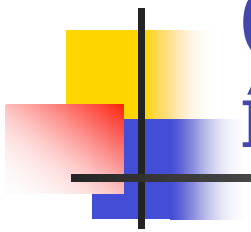


C++
REFERENCIA (&).

- **NO** se deben devolver direcciones de variables locales, ya que esta memoria se libera al salir del ámbito de la función.

Ejemplo

```
TFecha& Funcion2 (void)
{
    TFecha p;
    …
    return (p); // Incorrecto
}
```



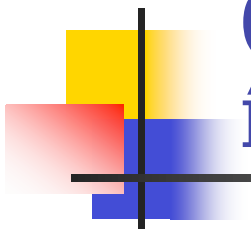
C++
ÍNDICE

C++
EL PUNTERO THIS

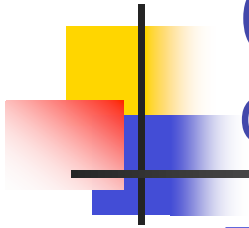
Asig
nombre creditos
void setCred(int c) void setNom(string n) bool asignaAlumno(Pers p)

- El puntero **`this`** es una pseudovariable
  - No se declara
  - No se puede modificar
- En C++ es un **argumento implícito** que reciben todas las funciones miembro
  - Eso excluye a las funciones amigas y a las funciones estáticas
- Apunta al objeto receptor del mensaje
- Suele omitirse para acceder a las propiedades del receptor del mensaje desde operaciones miembro
- Es necesario cuando
  - Queremos desambiguar nombre de parámetro y nombre de dato miembro
  - Queremos pasar como argumento a una función anidada el objeto receptor del mensaje original.

```
void setCred(int creditos){
     //creditos=creditos;->ERROR: ambiguo
     this->creditos=creditos;
     cout<< this->creditos <<endl;
}
```

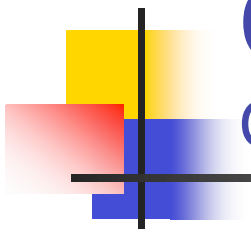


C++
ÍNDICE



C++
CONSTRUCTOR de una clase

- **Constructor**: función miembro de la clase cuyo objetivo es construir objetos e inicializarlos (asignando memoria dinámica si es necesario).
  - Tiene el mismo nombre que la clase.
  - NO devuelve valores (ni siquiera void).
  - Puede admitir parámetros como otra función.
  - Suele estar en la parte pública.
  - Al constructor se le llama cuando se crea el objeto implícitamente. **(excepto con jerarquías herencia).**
  - Si no se define, el compilador genera uno (a qué valores inicialice los datos miembro depende del compilador)
    - Ojo si hay que reservar memoria!
  - Es conveniente definir siempre un constructor sin parámetros (**CONSTRUCTOR POR DEFECTO)**

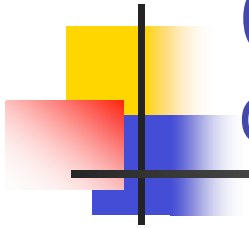


C++
CONSTRUCTOR POR DEFECTO

```
class TFecha
{
  public:// Parte pública de la clase
  TFecha( );  //Constructor por defecto
  TFecha(int,int,int ); //Constructor sobrecargado
  TFecha(TFecha&); //Constructor de copia
  ~TFecha( ); //Destructor
...
};

TFecha::TFecha()
{ // Inicializamos la fecha a 01/01/1900.
  dia = 1;      //También podríamos poner AsignarDia(1);
  mes = 1;      //También podríamos poner AsignarMes(1);
  anyo = 1900; //También AsignarAnyo(1900);
}
```

Constructor por defecto

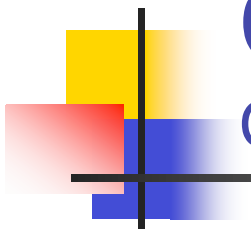


C++
CONSTRUCTOR SOBRECARGADO

```
TFecha::TFecha(int d, int m, int a)
{
  dia = d;    //O bien, AsignarDia(d);
  mes = m;    //O bien, AsignarMes(m);
  anyo = a;   //O bien, AsignarAnyo(a);
}
```

Constructor sobrecargado

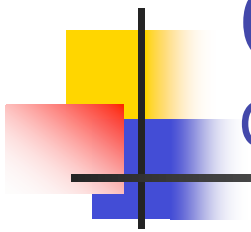
- ¿Qué haría falta para que el programa permitiera al usuario definir fechas sólo con el día, sólo con el día y mes?
  - TFecha f(10);
  - TFecha f(10,5);
- ¿Pueden convivir dentro de la misma clase estos dos constructores?
  - **`TFecha(){...};`**
  - **`TFecha(int d=1, int m=1, int a=1900){...}`**



C++
CONSTRUCTOR DE COPIA

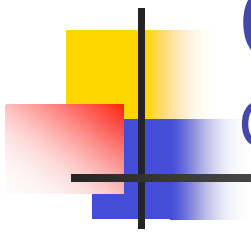
Constructor de Copia

- Crea un nuevo objeto a partir de otro del mismo tipo que ya existe.
  - Un solo argumento: una **referencia** constante a un objeto de la misma clase.
  - TFecha(const
    `TFecha c(d); TFecha c=d;`
- Tiene el mismo nombre que la clase.
- Si no se proporciona uno, el compilador genera uno por defecto que hace una copia bit a bit:
  - *No válido si el objeto original tiene atributos en memoria dinámica*
- Es similar a la asignación, pero no igual: ni la sustituye ni implementa.
  - `TFecha c=d;` ≠ `TFecha c; c=d;`



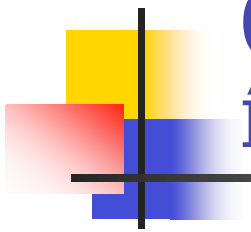
C++
CONSTRUCTOR DE COPIA

- Se invoca automáticamente:
  - Inicialización explícita de un objeto a partir de otro: TFecha **c (d); TFecha c=d;**
  - Al pasar un argumento por valor a una función: TFecha**&** Suma (**TFecha p**);
  - Al devolver una función un resultado **por valor** que es objeto de esa clase:
    - **TFecha** Suma() { TFecha **f**; ...return (**f**);}
- Es preferible que los argumentos de las funciones se pasen por referencia, para ahorrar tiempo y espacio. De otro modo se invoca al constructor de copia.
  - Ojo! Los arrays en C++ se pasan siempre por referencia.
- ¿Por qué en el ctor. de copia el parámetro se pasa por referencia?
- ¿Cómo impido que se modifiquen parámetros por referencia?

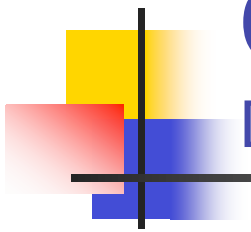


C++
CONSTRUCTOR DE COPIA.

```
TFecha::TFecha(const TFecha &tf)
{
  dia = tf.dia;       // O AsignarDia(tf.dia);
  mes = tf.mes        // O AsignarMes(tf.mes);
  anyo= tf.anyo       // O AsignarAnyo(tf.anyo);
}
```



C++
ÍNDICE



C++
DESTRUCTOR

- Realiza la operación opuesta de un constructor, limpiando el almacenamiento asignado a los objetos cuando se crean.
- Tiene el mismo nombre que la clase, pero precedido por el símbolo **~**.
- No recibe ningún argumento ni devuelve ningún tipo de dato (ni void)
- El compilador llama automáticamente a un destructor del objeto cuando el objeto sale fuera del ámbito.

```
int Suma() {TVector a; ...}
```

  - Ojo! Esto no ocurre si sólo tengo un puntero a objeto

```
int Suma() { TVector *a; …}
```

- También se invoca al destructor al hacer **delete**:

```
Int Suma() { TVector *a = new TVector();… delete a; }
```

- Se puede invocar explícitamente para destruir un objeto:

```
TVector a; a.~TVector();
```