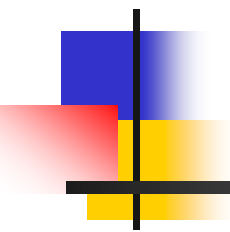
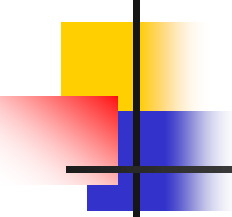


SEMINARIO C++

Introducción a la Programación Orientada a Objetos



Parte I



C++

ÍNDICE

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones Amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- 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*): 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*): instrumentos para la manipulación de atributos/relaciones

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

0..n	- alumno
1..10	- amatr

Asig	
string nombre string créditos	
void setCred(string c) void setNom(string n) bool asignaAlumno(Pers &p)	



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

■ 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 : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- 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)

0..n | - alumno

1..10 | - amatr

Asig

- string nombre - string créditos
+ void setCred(string c) + void setNom(string n) + bool asignaAlumno(Pers &p)



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

```
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 DE LAS VARIABLES

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

0..n - alumno

1..10 - amatr

Asig
- string nombre - string créditos
+ void setCred(string c) + void setNom(string n) + bool asignaAlumno(Pers &p)

class Asig; //forward definitionclass Pers
{private:

string nombre;

string dni;

Asig * amatr[10];

public:

void setNom(string n);

string getNom();

bool anyadeAsig (Asig &a);

};

Fichero
declaración (.h)

- **Ejercicio:** definid el fichero de declaración de la clase Asig



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- 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 Asig.h para que los créditos sean de tipo float. ¿Qué otras cosas hay que modificar para q cualquier programa q utilice esa clase siga funcionando? ¿Qué pasaría si la variable credits fuera pública?



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

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 *scope resolution operator* (::).
- **Definición del punto de entrada al programa** `int main(...){...};`



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- **DISTRIBUCIÓN DE CÓDIGO EN C++**
 - Definiciones en .h: directorio **include**
 - Código de clases y punto de entrada en .cpp: directorio **src**
 - Documentación en directorio **doc**
 - Código objeto en directorio **lib**



C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- **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 : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

- 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)
 - Palabra reservada inline delante de la definición de la función

C++

CLASES : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

Mi primera clase

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

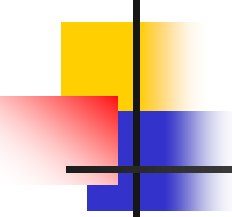
```
//rectangulo.cpp
#include "rectangulo.h"
void rectangulo::dimensiones(int b,int h){
    base=b;
    altura=h;
}
```

```
//main.cpp
#include <iostream>
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 : ESTRUCTURA, ÁMBITO DE LAS VARIABLES

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 dir include y .cpp en dir src)



C++ ÍNDICE

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.



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)
- Estas funciones debemos declararlas, si se puede, como *inline*. Es más profesional y no perdemos eficacia.
 - E.g. `getNombre(); setDNI();`
- 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**
 - E.g. `isMatriculado(); /*nos dice si el alumno tiene alguna asignatura asociada o no*/`



C++

FUNCIONES O MÉTODOS SET/ GET /IS.

- **Ejercicio: implementa la siguiente clase Tfecha mediante el uso de funciones inline.**

- `g++ -o fecha Tfecha.cpp main.cpp`

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

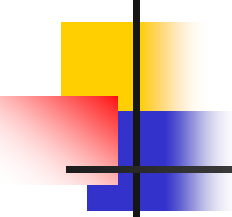
```
/* main.cpp */  
# include <iostream.h>  
int main() {  
    Tfecha p;  
    cout << p.getDia( );  
    cout << p.getMes( );  
    cout << p.getAnyo( );  
}
```

Tfecha
- int dia - int mes - int anyo; +void setDia(int d) + void setMes(int m) + void setAnyo(int a) + int getDia() + int getMes() + int getAnyo() + bool isBisiesto();

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

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.



C++

FUNCIONES AMIGAS

- La **clase** debe actuar como un subsistema cerrado dentro del contexto general del programa que la utiliza. Sólo interesa que sean accesibles determinados **datos y funciones**. Únicamente serán accesibles lo de la parte *public*. Así conseguimos “ocultar” todo lo que coloquemos en la *private*.
- Sentido? Las clases pueden venir empaquetadas en librerías que en muchas ocasiones son creadas por personas distintas del programador que las usa.
- La parte privada sólo es accesible por:
 - Métodos de la clase
 - **Funciones amigas (C++).**



C++

FUNCIONES AMIGAS

- **Funciones Amigas.** Funciones NO miembro de una clase, que puede tener acceso a las partes privadas de esa clase. Rompen el ppio de “encapsulación”.
- Se declara como amiga de la clase mediante la palabra reservada *“friend”*.
- Razones para usarlas:
 - Algunas funciones necesitan acceso privilegiado a más de una clase.
 - Algunas funciones resultan más legibles si pasamos todos sus argumentos a través de la lista de argumentos.
 - Algunas funciones tienen como primer operando un objeto distinto del que llama a la función.
- A veces, por comodidad se decide declarar TODA UNA CLASE como amiga de otra.
 - En este caso todas las funciones de la clase amiga pueden acceder a las partes privadas de la otra clase.

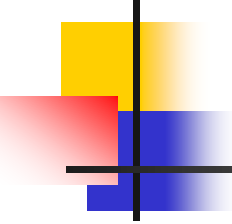


C++

FUNCIONES AMIGAS

■ Ejemplos Funciones Amigas

```
class Tfecha {  
    friend void Copiar(Tfecha*,Tfecha*); //legibilidad  
  
    //primer parámetro dto clase  
    friend ostream& operator<<(ostream&,Tfecha&);  
    friend istream& operator>>(istream&,Tfecha&);  
    ...  
}
```



C++

FUNCIONES AMIGAS

```
void Copiar( TFecha* p1, TFecha* p2 ) {  
    p2->dia = p1->dia;  
    p2->mes = p1->mes;  
    p2->anyo = p1->anyo;  
}
```

```
int main() {  
    TFecha p, q;  
    p.dia = 3;  
    p.AsignarDia(3);  
    Copiar(&p, &q);  
}
```

Preguntas:

- ¿Detectáis algún error en el main?
- ¿Es la función friend Copiar una función miembro de la clase Tfecha?
- ¿Se os ocurre algún otro modo de implementar el Copiar sin hacer uso de funciones friend?



C++

ÍNDICE

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.



C++

TIPO 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++

TIPO 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++

TIPO 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

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.

C++

CONSTRUCTOR: POR DEFECTO, DE COPIA.

- **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, DE COPIA.

```
class TFecha
{
friend ostream& operator<<(ostream&,Tfecha&); //F.amiga
friend istream& operator>>(istream&,Tfecha&); //F.amiga
friend void Copiar(Tfecha*,Tfecha*); //F.amiga
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: POR DEFECTO, DE COPIA.

```
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: POR DEFECTO, 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.
- 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:
problemas en estructuras enlazadas
- Es similar a la asignación, pero no igual: ni la sustituye ni implementa. TFecha **c (d); c=d;**



C++

CONSTRUCTOR: POR DEFECTO, 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 que es objeto de esa clase:
 - `TFecha Suma() {...return (TFecha);}`
- 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, independientemente de lo q especifiquemos en el prototipo de la función.
- ¿Por qué en Constructor de copia parámetro por referencia?
- ¿Cómo impido q se modifiquen parámetros por referencia?

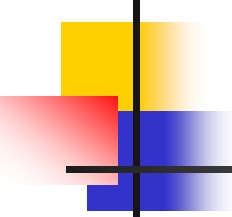
C++

CONSTRUCTOR: POR DEFECTO, DE COPIA.

```
TFecha::TFecha(const TFecha &tf) {  
    Copiar(*this, tf);  
} //Utilizando la func. amiga que ya tenemos
```

O bien,

```
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

1. Clases en C++: estructura, ámbito de variables.
2. Operaciones set/get/is.
3. Funciones amigas.
4. Tipo Referencia (&) .
5. Constructor: por defecto, de copia.
6. Destructor.



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
- Se puede invocar explícitamente para destruir un objeto:
`TVector a; a.~TVector();`