

Décimosegunda Sesión

Metodologías y Técnicas de Programación II

Programación Orientada a Objeto (POO) C++

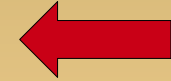
Repaso Primera Parte

Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

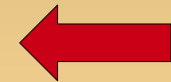
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores
Constantes e "inline"

Funciones Amigas
Sobrecarga de Funciones



Sobrecarga

De Operadores

Creación Dinámica de Objetos

Herencia.

Tipos de Visibilidad

Herencia Múltiple

Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

12.1 Repaso

¿Qué es programar?

Programar es tener en cuenta



- Un **problema**:
Quiero saber mi nota final en la asignatura.
- Un Conjunto de **Datos**

En las prácticas diarias tengo un	:	8
En la práctica final tengo un	:	9
En el exámen parcial un	:	6
En el exámen final un	:	7
- Unas **funciones** o algoritmos
$$\text{Nota Prácticas} = (20\% \text{ Prac.D}) + (80\% * \text{Prac.F})$$
$$\text{Nota Final} = 20\% \text{ Nota.Prac.} + 15\% \text{ Parcial} + 65\% \text{ Final}$$

Y entonces...

Aplicar las funciones para resolver el problema (7,21)

12.1 Repaso

¿Qué es programar?



Queremos por supuesto que nuestro programa:

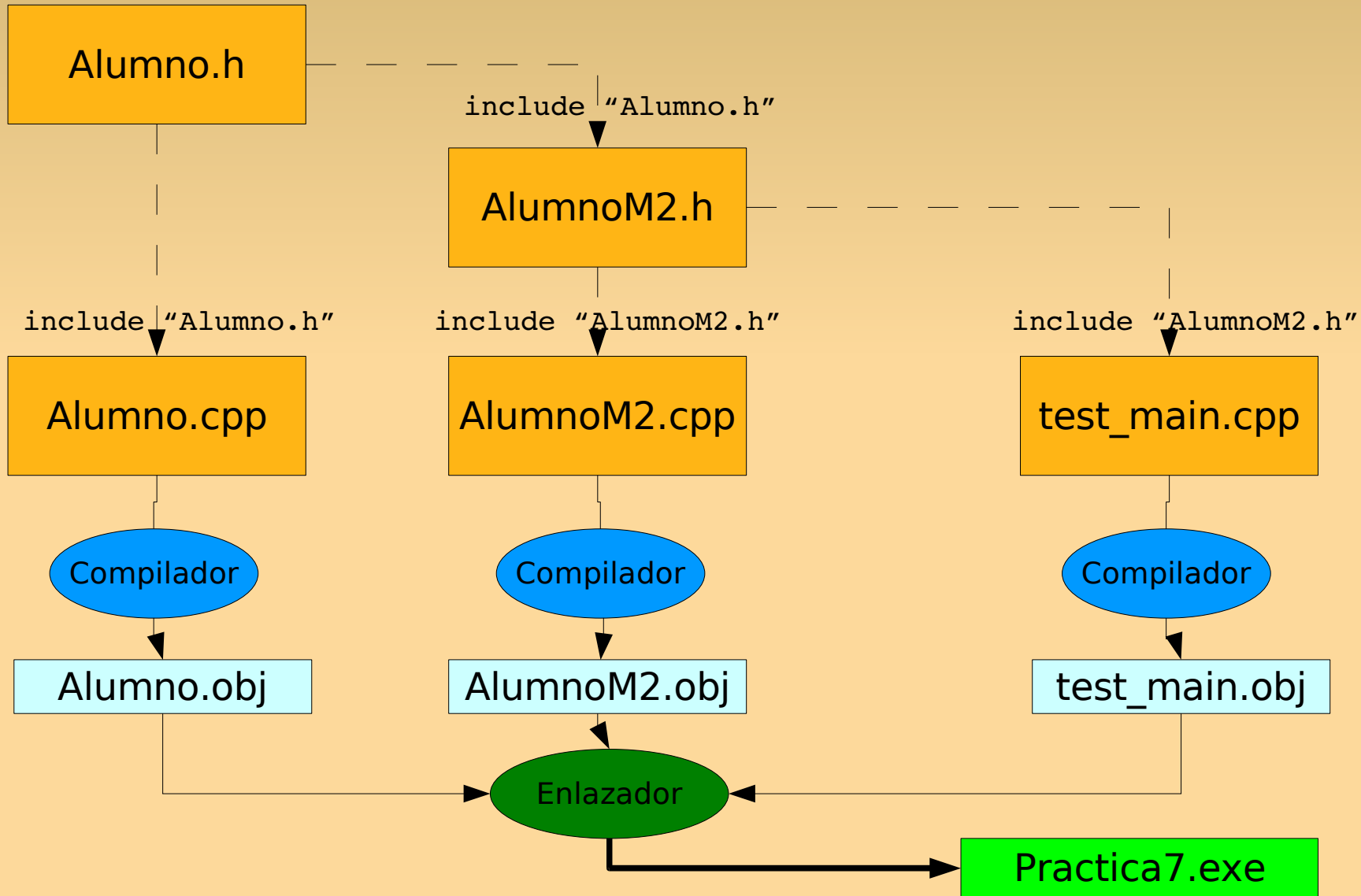
- Sea **Correcto**:
Funciona bien y de acuerdo a los requisitos.
El resultado se corresponde con unas especificaciones.

Pero que además:

- Sea **Eficaz**
Lo hace en un tiempo admisible.
Especificaciones No Funcionales o de Calidad.
- Sea **Adaptable o Flexible**
¿Qué pasa si cambian las especificaciones?
- Sea **Reutilizable**
Sería ideal que ante problemas parecidos, no tenga que volver a escribir todo el código y pueda aprovechar la mayor parte de él.

12.1 Repaso

Compilación y Enlazado por separado



12.1 Repaso

Buscamos la Reutilización Sistemática - (POO)

Queremos escribir sólo el código necesario, equivocarnos menos, poner y quitar funcionalidad de forma fácil.

Todo es un Objeto:

- Un objeto es como una variable mejorada.
- Un objeto es una instancia de una clase determinada.

Los objetos se comunican mediante mensajes:

- Los programas serán grupos de objetos enviándose mensajes entre sí.

Características:

- Abstracción
- Encapsulación
- Herencia
- Polimorfismo

C++:

- Language de Propósito General
- Lenguaje Orientado a Objeto.
- Eficiente y elegante
- 3.000.000 programadores (se usa)

Alumno
<pre>- nombre_ : string - apellido1_ : string - apellido2_ : string - dni_ : string - edad_ : int</pre>
<pre>+ nombre(string nom) + apellido1(string a1) + apellido2(string a2) + dni(string dni) + edad(int anios) + nombre() : string + apellido1() : string + apellido2() : string + dni() : string + edad(): int</pre>

12.1 Repaso

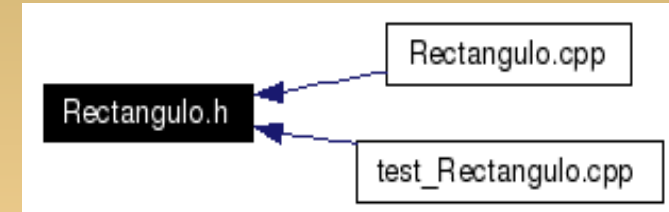
Estándares

Nombres de Ficheros:

Rectángulo.h (#ifndef NOMBRE_CLASE_H ...)

Rectangulo.cpp

test_Rectangulo.cpp (main)



Documentación de Cabeceras de Ficheros.

```
/**
 * @file test_Rectangulo.cpp
 * @brief Pruebas sobre la clase Rectangulo. Ejemplo Metodos II
 *
 * @author Jose Luis Marina <jlmarina@nebrija.es>
 * @date 19-FEB-2007
 */
```

Documentación Cabeceras de Clases.

Nombres de Clases, Objetos, atributos y métodos – Indexado y corchetes

Ayudan a: Reutilizar Código a Documentar (Código más comprensible)

12.1 Repaso

Programación Estructurada

Funciones

Encapsulan la complejidad

Declaración (.h) :

```
int translate(float x, float y, float z);  ó void f(void)
```

Implementación:

```
int translate(float x, float y, float z)    // Correcto
{
    x = y = z;
    return 33;
}
```

Control del Flujo

```
if (expresion)
{
    sentencias
}
else
{
    sentencias
}

while(guess != secret)
{
    cout << "Numero: ";
    cin >> apuesta;
}

//do {    } while ()

for(i = 0; i<128;i++)
{
    cout << i;
}

swicth (c)
{
    case 'a': ·g··
```

12.1 Repaso

Programación Estructurada

Operadores

Matemáticos	(+ , - , * , / , %)	Relacionales	(> , >= , < , <= , == , !=)
Lógicos	(&& ,)	Bits	(& , , ^ , >> , <<)
Unarios	(- , + , ++ , --)	Direcciones	(& , * , ->)
Ternario	a = --b ? b : (b = -99);		

Casting o Moldeado

Hay que utilizarlo con mucho cuidado y en pocas ocasiones

```
i = 1; // Posible perdida de digitos
i = f; // Posible perdida info,
i = static_cast<int>(1); // no warning
i = static_cast<int>(f); // no warning
char c = static_cast<char>(i);
```

Operador sizeof

Número de bytes utilizado por un tipo de datos o por una variable.

12.1 Repaso

Programación Estructurada

Tipos de Datos

Básicos:

Predefinidos : char, int, float, double)

Se combinan con : long y short signed y unsigned.

Sus límites están en ***limits.h*** y ***float.h***

```
# define SHRT_MIN (-32768)
```

```
# define SHRT_MAX 32767
```

Algunas combinaciones no valen (long float -- unsigned float)

Creados por el usuario (programador)

El compilador aprende a usarlos.

Utiliza las declaraciones (por ejemplo de la clases)

12.1 Repaso

Punteros

Las direcciones de memoria se pueden guardar dentro de otras variables para su uso posterior: Los Punteros.

Un puntero se define con el operador “*” y el tipo de variable a la que apunta.

```
int i;  
int *pi;
```

6684160

33

i

```
pi = &i; // Puntero a la dirección de i.  
i = 33; // Valor de i.
```

DIR pi

6684160

pi

```
*pi= 55; // Contenido de lo que apunta pi
```

```
// Atentos con:
```

```
int a, b, c; // Ok
```

```
int* pa, pb, pc ; // No OK. Solo pa es puntero.
```

```
int* p1;
```

```
int* p2; // Ok.
```

12.1 Repaso

Conceptos POO

```
int i;           ClaseEnteros mi_entero;  
i=33;          mi_entero.asignar(33);  
cout << i;     mi_entero.imprime_lo_que_vales();
```

Todo es un Objeto:

Un objeto es como una variable mejorada.

Cogemos un componente conceptual de nuestro problema y lo representamos como un objeto en nuestro programa (perro, edificio, tuerca, ventana,...)

Los objetos se comunican mediante mensajes:

Haremos programas que serán grupos de objetos enviando mensajes a otros para decirles qué hacer. Petición de invocación a una función que pertenece a un objeto en particular.

Datos Propios:

Podemos crear un nuevo tipo de objeto haciendo un paquete con otros objetos. Oculto la complejidad. Coche (ruedas, motor,etc)

Cada objeto es de un tipo:

Un clase define el tipo de un objeto. Clase Enteros -> un_entero.

Un objeto es una instancia de una clase determinada.

12.1 Repaso

Conceptos POO

Todos los Objetos de un tipo en particular pueden recibir los mismos mensajes:

Esto es muy potente. Nos abre un montón de posibilidades para REUTILIZAR CÓDIGO.

Clase Figura

Clase Círculo

Clase Cuadrado

Clase Triángulo

Un objeto del tipo o clase Círculo también es del tipo Figura. Está garantizado que un círculo recibirá los mensajes de Figura.

Si hacemos código que habla con objetos de tipo Figura, SIN TOCAR NADA, ese código también funciona con objetos de la clase Círculo.

```
mi_figura.dibujar_en_pantalla();
```

12.1 Repaso

Tipos de Acceso

public

private

protected

La declaración “friend” nos permite que otros accedan a nuestros datos protegidos

```
class X
{
private:
    int i;
public:
    void initialize();
    friend void g(X*, int);    // Global friend
    friend void Y::f(X*);    // Un método de la clase Y
    friend class Z;          // Toda Z es mi amiga
    friend void h();
};
```

12.1 Repaso

Relaciones de Amistad

Si A -> B y B -> C	NO A->C
Si A -> B	NO B->A
Si A -> B y b <<B	NO A->b (b es el hijo de B)

Podemos definir relaciones de amistades con una función, con un método de otra clase o con una clase entera.

Friend = “Puedes acceder a mis atributos privados como si fueras uno de mis métodos”

```
friend void g(X*, int); // Global friend
friend void Y::f(X*); // Un método de la clase Y
friend class Z; // Toda Z es mi amiga
friend void h();
```

12.1 Repaso

Constructores

Modo simplificado de inicialización:

```
int i(0);           Alumno      a("Pepe");
```

Inicializadores:

```
pareja::pareja(int a, int b)
{
    a_ = a;
    b_ = b;
}
```

// Es más seguro y eficiente así:

```
pareja::pareja(int a,int b) : a_(a), b_(b) {}
```

```
class Pareja
{
    ...
    // Constructores
    Pareja(int a, int b);
    Pareja() : a_(0), b_(0){};
    ...
}
```

```
class Pareja
{
    ...
    // Argumentos por defecto
    Pareja(int a=0,int b=0) :a_(0),b_(0){};
    ...
}
```

12.1 Repaso

Constructores por defecto

Es un constructor que puede ser invocado sin argumentos

```
class X
{
    int i_;
    char c_;
public:
    X(int i, char c): i_(i), c_(c) {}
};
X x1[10]; // El Compilador se va a quejar!!
X x3;
```

El constructor por defecto es tan importante que si (y sólo si) una clase no tiene constructor el compilador crea uno automáticamente (no hace nada).

```
class V {
    int i; // private
}; // No constructor

V v, v2[10]; // Esto funciona!!
```

12.1 Repaso

Referencias

1. Cuando una referencia se crea, se ha de inicializar. (Los punteros pueden inicializarse en cualquier momento.)
2. Una vez una referencia se inicializa ligándola a un objeto, no se puede ligar a otro objeto. (Los punteros se pueden apuntar a otro objeto en cualquier momento.)
3. No se pueden tener referencias con valor nulo. Siempre ha de suponer que una referencia está conectada a una trozo de memoria ya asignada.

```
int y;
int& r = y;          // La asignamos a algo que ya existe
const int& q = 12;  // También se puede hacer esto.
int x = 0;
int& a = x;         // x y a son lo mismo
int main()
{
    cout << "x = " << x << ", a = " << a << endl;
    a++;
    cout << "x = " << x << ", a = " << a << endl;
}
```

12.1 Repaso

El constructor Copia

Para evitar que el compilador haga una copia bit a bit, construimos nuestro función de copia.

Como estamos creando un objeto esta función tiene que ser un constructor, y el único argumento tiene que ver con el objeto del que partimos para crear el nuevo.

`X(X&)`

Si creamos un constructor copia el compilador no realizará una copia bit a bit cuando cree un nuevo objeto de otro existente.

Siempre se llamará al constructor copia que hemos creado.

12.1 Repaso

Utilización de Nombres en la sobrecarga de funciones

Podemos utilizar el mismo nombre siempre que la lista de argumentos sea diferente.

El compilador utiliza:

Nombre de la función

Ámbito

Lista de argumentos

Esto no es sobrecarga. Son funciones diferentes por el ámbito:

```
void f();
```

```
class X { void f(); };
```

No tenemos sobrecarga en el valor de retorno.

```
void f();
```

```
int f();
```

¿Por qué?.....

12.1 Repaso

Sobrecarga de Funciones

No deberíamos utilizar argumentos por defecto si hay que incluir una condición en el código (un “if”). En este caso es mejor tener varias funciones sobrecargadas.

Uso típico de argumentos por defecto es cuando empieza con una función con un conjunto de argumentos, y después de utilizarla por un tiempo se da cuenta de que necesita añadir más argumentos.

```
f (int a);    f(int a, int b=0); // Es compatible con lo anterior.
```

Tanto la sobrecarga de funciones como el uso de argumentos por defecto nos hacen más fáciles las llamadas a las funciones y además son más fáciles de leer.

12.1 Repaso

Inicialización y Limpieza de Objetos

El constructor es una función sin tipo de retorno y con el mismo nombre que la estructura. **Alumno()**

El destructor tiene la misma forma, salvo que el nombre va precedido el operador "~". **~Alumno()** Lo definimos cuando hay punteros

El constructor copia tiene este aspecto: **Alumno(const Alumno& a)**

Si creamos un constructor copia el compilador no realizará una copia bit a bit cuando cree un nuevo objeto de otro existente.

Las referencias necesitan un área de memoria que referenciar:

```
int& x = 1;      // Mal
```

```
const int& y = 1; // Bien
```

12.1 Repaso

Objetos y Métodos Constantes

```
class X
{
    int i_;
public:
    X(int i);
    int f() const;
};

X::X(int i) : i_(i) {}
int X::f() const { return i_; }
int main()
{
    X x1(10);
    const X x2(20);
    x1.f();
    x2.f();
}
```

La palabra `const` debe incluirse tanto en la declaración como en la definición del método.

Como `f()` es un método constante, si intenta modificar de alguna forma o llamar a otro método que no sea constante, el compilador informará de un error.

Un miembro constante puede llamarse tanto desde objetos constantes como desde no constantes de forma segura.

Un método que no modifica ningún atributo se debería escribir como constante y así se podría usar desde objetos constantes.

12.1 Repaso

Constantes

La palabra reservada para definir constantes es ***const***

const nos permite definir como constantes:

- Variables.
- Objetos.
- Argumentos de funciones.
- Punteros.
- Métodos.

Elimina el uso de constantes simbólicas con 'define' para sustitución de valores por el preprocesador sin perder sus ventajas, añadiendo comprobación de tipos y haciendo nuestro código más seguro.

El uso de la llamada “constancia exacta” (const correctness) es decir, el uso de const en todo lugar donde sea posible, puede ser un salvavidas para muchos proyectos.

12.1 Repaso

const

El motivo unicial para usar *const* fue para eliminar el uso de la directiva de compilación `#define` para sustitución de valores. Tiene ventajas sobre `#define`.

```
#define NOTA_ESPERADA 8
```

```
const float nota_esperada = 8;
```

```
cont int cantidad = 19;
```

```
Alumnos curso[cantidad];
```

Punteros constantes:

```
// "p es un puntero que apunta //  
a: un entero constante.
```

```
const int* p;
```

```
const int *p;
```

```
// "w es un puntero constante  
// que apunta a: un entero.
```

```
int i=3;
```

```
int* const w = &i;
```

```
*w = 2; // ¿Está bien o está mal?
```

En C++ lo mejor es utilizar una referencia constante en lugar de paso por valor.

Un método que no modifica ningún atributo se debería escribir como constante y así se podría usar desde objetos constantes.

12.1 Repaso

funciones inline

Sustituye la utilización de macros que es muy común en la programación en C.

Se pensó para dotar al lenguaje C++ de mayor eficiencia.

Es una sugerencia para el compilador.

Hay dos maneras de definir un método de una clase como “inline”. Es preferible la que se utiliza en la implementación y no en la declaración de la clase.

12.2 Repaso

Espacios de Nombres

```
#include <iostream>
namespace uno {
int x;
}
namespace dos {
int x;
}
using namespace uno;
int main() {
    x = 10;
    dos::x = 30;
    std::cout << x << ", " << dos::x << std::endl;
    std::cin.get();
    return 0;
}
```

11.3 Resumen

Static

Podemos controlar el almacenamiento de nuestros objetos y variables.

Podemos controlar el ámbito en el que se ven.

Espacios de Nombres

Utilizamos la palabra reservada ***“namespace”***.

Nos ayudan a dividir el espacio de nombres global de forma que podamos repetir nombres de funciones, clases y variables.

Es una declaración y se suele utilizar por tanto en los ficheros de cabecera.