



Universidad Antonio de Nebrija



Asignatura: “LS1168” - “Metodologías y Técnicas de Programación II”
Cuatrimestre: Segundo Examen: Parcial Convocatoria: Ordinaria
Grupo: 1IT2 / 1IT3 Curso: 2006/2007 Fecha: 21 de marzo de 2007

PREGUNTAS:

Dado el siguiente código de la clase Alumno :

```
// DECLARACIÓN
class Alumno
{
    private:
        int nota_;
        char* nombre_;

    public:
        Alumno(int nota);
        void poner_nota(int nota);
        void mostrar_alumno(void);

        void poner_nombre(const char* nombre);
        ~Alumno();
};

//-----
// IMPLEMENTACION INCOMPLETA
//-----
Alumno::Alumno(int nota)
{
    nota_ = nota;
    nombre_ = 0;
}

void Alumno::poner_nota(int nota)
{
    nota_ = nota;
}

void Alumno::mostrar_alumno(void)
{
    cout << nombre_ << "Nota:" << nota_ << endl;
}
```



Universidad Antonio de Nebrija

1.- Implementa el método `poner_nombre` de la clase `Alumno`, teniendo en cuenta que puede ser llamado varias veces para el mismo objeto.

```
void Alumno::poner_nombre(const char* nombre)
{
    // Eliminamos memoria pedida anteriormente.
    delete [] nombre_;

    // Nuevo espacio en que quepa la cadena y '\0'
    // nombre_ es sólo un puntero.
    nombre_ = new char[strlen(nombre)+1];

    // Copiamos lo recibido en nuestra atributo local.
    strcpy(nombre_, nombre);
}
```

2.- Codifica el destructor y argumenta por qué es necesario.

```
Alumno::~~Alumno()
{
    cout << ".....Destructor Alumno(): " << nombre_ << endl;
    delete nombre_; // Liberamos memoria
}
```

Es necesario porque al destruir el objeto debemos liberar la memoria que hayamos reservado para `nombre_`.

3.- Implementa el constructor por defecto de la clase `Alumno` que inicialice los atributos privados a valores consistentes. Además declara e implementa un nuevo constructor que reciba el nombre y la nota del alumno.

```
Alumno::Alumno()
{
    cout << "Constructor por defecto Alumno()" << endl;
    nota_ = 0;
    nombre_ = 0;
}

//-----

Alumno::Alumno(int nota, char* nombre)
{
    cout << "Constructor Alumno(int nota, char* nombre)" << endl;
    nota_ = nota;
}
```



Universidad Antonio de Nebrija

```
if (0 == nombre)
{
    nombre_ = 0;
}
else
{
    // Nuevo espacio en que quepa la cadena y '\0'
    nombre_ = new char[strlen(nombre)+1];
    // Copiamos lo recibido en nuestra atributo local.
    strcpy(nombre_, nombre);
}
}
```

4.- Cambia la implementación de los constructores de la clase `Alumno` para que utilicen listas de inicialización de forma que indiquemos al compilador en la implementación que trate de expandir su código.

```
//-----
De esta implementación.....
Alumno::Alumno()
{
    cout << "Constructor por defecto Alumno()" << endl;
    nota_   = 0;
    nombre_ = 0;
}

A esta:
inline Alumno::Alumno(): nota_(0), nombre_(0) // Lista de inicialización
{
    cout << "Constructor por defecto Alumno()" << endl;
}

//-----
De esta implementación.....
Alumno::Alumno(int nota)
{
    cout << "Constructor Alumno(int nota)" << endl;
    nota_   = nota;
    nombre_ = 0;
}

A esta:
inline Alumno::Alumno(int nota) : nota_(nota), nombre_(0)
{
    cout << "Constructor Alumno(int nota)" << endl;
}

//-----
```



Universidad Antonio de Nebrija

De esta implementación

```
Alumno::Alumno(int nota, char* nombre)
{
    cout << "Constructor Alumno(int nota, char* nombre)" << endl;
    nota_ = nota;

    if (0 == nombre)
    {
        nombre_ = 0;
    }
    else
    {
        // Nuevo espacio en que quepa la cadena y '\0'
        nombre_ = new char[strlen(nombre)+1];
        // Copiamos lo recibido en nuestra atributo local.
        strcpy(nombre_, nombre);
    }
}
```

A esta:

```
inline Alumno::Alumno(int nota, char* nombre) : nota_(nota), nombre_(0)
{
    cout << "Constructor Alumno(int nota, char* nombre)" << endl;
    if (0 == nombre)
    {
        nombre_ = 0;
    }
    else
    {
        // Nuevo espacio en que quepa la cadena y '\0'
        nombre_ = new char[strlen(nombre)+1];
        // Copiamos lo recibido en nuestra atributo local.
        strcpy(nombre_, nombre);
    }
}
```

5.- Implementar un nuevo constructor que nos permita eliminar todos los constructores actuales y que nos sirva para poder utilizar el siguiente código:

```
Alumno a1;
Alumno a2(8);
Alumno a3(8, "Juan Sanz");
```

Dejamos sólo este constructor:

```
inline Alumno::Alumno(int nota, char* nombre) : nota_(nota), nombre_(0)
{
    cout << "Constructor Alumno(int nota, char* nombre)" << endl;
    if (0 == nombre)
    {
        nombre_ = 0;
    }
    else
    {
        // Nuevo espacio en que quepa la cadena y '\0'
        nombre_ = new char[strlen(nombre)+1];
        // Copiamos lo recibido en nuestra atributo local.
        strcpy(nombre_, nombre);
    }
}
```



Universidad Antonio de Nebrija

Y cambiamos su declaración en la clase para que use parámetros por defecto:

```
class Alumno
{
private:
    int nota_;
    char* nombre_;

public:
    Alumno(int nota=0, char* nombre=0);
    void poner_nota(int nota);
    void poner_nombre(const char* nombre);
    void mostrar_alumno(void);
    ~Alumno();
};
```

6.- Explica por qué es necesario un constructor copia en este caso e impleméntalo de forma adecuada.

Es necesario porque si dejamos que se utilice el constructor copia que nos proporciona por defecto el compilador, se hará una copia bit a bit y tendremos que el puntero nombre_ del nuevo objeto apuntará a la misma zona de memoria que el puntero nombre del objeto pasado.

En nuestro constructor copia crearemos una nueva zona de memoria para albergar nuestro nombre y haremos que el puntero nombre_ apunte a esta zona y no a la del objeto copiado:

```
Alumno::Alumno(const Alumno& a)
{
    cout << "Constructor Copia : Alumno(const Alumno& a)" << endl;
    // Comprobamos si el objeto del que copio tiene nombre asignado o no.
    if (a.nombre_)
    {
        nombre_ = new char[strlen(a.nombre_)+1]; // Creamos y apuntamos a la nueva zona
        strcpy(nombre_,a.nombre_); // Copiamos el contenido.
    }
    else
    {
        nombre_ = 0;
    }
    nota_ = a.nota_;
}
```



Universidad Antonio de Nebrija

7.- Indica cuántas veces se está llamando al constructor copia en cada línea teniendo en cuenta el código de la función global `funcion3`.

```
Alumno funcion3(Alumno a)
{
    return a;
}
1) Alumno a1(5); // Número de Veces = 0
2) Alumno array_alumnos[3]; // Número de Veces = 0
3) Alumno a2(a1); // Número de Veces = 1
4) Alumno a3 = a2; // Número de Veces = 1
5) Alumno* a4 = new Alumno(a3); // Número de Veces = 1
6) Alumno* p= &a1; // Número de Veces = 0
7) Alumno z = funcion3(a2); // Número de Veces = 2
```

8.- ¿Qué métodos de la clase `Alumno` se pueden declarar como constantes? Codificar para ellos su nueva declaración utilizando constantes en todo lugar donde sea aplicable sin cambiar la funcionalidad de la clase:

```
1) El constructor único _____
2) void poner_nota(int nota); _____
3) void poner_nombre(char* nombre); _____
4) void mostrar_alumno(void); _____
```

`mostrar_alumno(void)`, se puede declarar como método constante porque no cambia ninguno de los atributos de la clase. En el resto no los podemos declarar como métodos constantes porque en su código cambian alguno de los atributos de la clase `Alumno`.

Sí podemos declarar como constantes los argumentos que reciben las funciones.

```
1) El constructor único Alumno(const int nota=0, const char* nombre=0)
2) void poner_nota(int nota); void poner_nota(const int nota);
3) void poner_nombre(char* nombre); void poner_nombre(const char* nombre)
4) void mostrar_alumno(void); void mostrar_alumno(void) const;
```

9.- Indica que líneas son incorrectas y por qué.

```
1) int e1 = 1;
2) const int ce2 = 2;
3) const char cc; // Incorrecta. Hay que inicializar las constantes.
4) e1 = ce2;
5) const int nota=3+4;
6) int calificacion = nota;
7) int* x = &calificacion;
8) int* y = &nota; // Incorrecto. Puntero no constante a zona constante.
9) int* z = (int*)&nota;
```



Universidad Antonio de Nebrija

10.- Indica si podemos definir dos métodos así y explica por qué si o por qué no:

- a) `int metodo1(char a);`
`int metodo1(char a, char b);`
- b) `void metodo2(float a);`
`void metodo2(float b);`
- c) `void metodo3(float a, float b=0);`
`void metodo3(float a);`
- d) `char metodo4(float a);`
`void metodo4(float a);`

a) Correcto. Podemos hacer llamadas a ambos métodos y distinguir entre ellas.

b) No podemos. Dos métodos que se llaman igual y reciben el mismo tipo y número de parámetros y en el mismo orden.

c) No podemos. No hay forma de distinguir a qué código llamar cuando nos encontramos con:

```
metodo3(7.5);
```

d) No podemos. Como en C++ podemos llamar a `metodo4()` sin esperar valor no podemos sobrecargar usando los retornos de las funciones.