

Quinta Sesión

Metodologías y Técnicas de Programación II Programación Orientada a Objeto (POO)

C++

Profesor:

José Luis Marina
jmarina@nebrija.es

Laboratorios:

José Luis Marina
jmarina@nebrija.es
Borland Builder 6.0

Puntuación:

Prácticas Laboratorio:	20%
Diarias	20%
Trabajo Laboratorio	80%

Exámen Parcial	15%
----------------	------------

Exámen Final	65%
--------------	------------

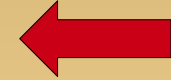
Exámen Final Extraordinario	70%
------------------------------------	------------

5.0 Estado del Programa

Introducción a la POO

Historia de la Programación
Conceptos de POO

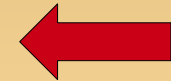
C++
Mi primera Clase



Repaso de Conceptos

Estándares de Programación
Punteros y Memoria

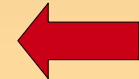
E/S
Control y Operadores



Clases y Objetos en C++

Uso y aplicación
Constructores

Funciones Amigas
Constantes e "inline"



Sobrecarga

De Operadores

De Funciones

Herencia.

Tipos de Visibilidad

Herencia Múltiple

Polimorfismo

Funciones Virtuales

Polimorfismo y Sobrecarga.

Plantillas

Contenedores

Iteradores

5.1 Repaso de Sesiones Anteriores

Estructuras y Clases

Las estructuras encapsulan datos. Las clases encapsulan además comportamiento.

Por defecto los miembros de una estructura son públicos (**public**).

Por defecto los miembros de una clase son privados (**private – protected**).

Las clases incorporan dos funciones especiales:

- Constructores: Alumno() ó Alumno(int edad)
- Destructores: ~Alumno() * No admite parámetros

```
Time sunset;           // Objeto de tipo Time
Time arrayOfTimes[ 5 ]; // Array
Time *pointerToTime;   // Puntero
Time &dinnerTime = sunset; // Referencia
```

5.1 Repaso de Sesiones Anteriores

Uso de Clases

```
class Time
{
public:
    Time();                // constructor
    void setTime( int, int, int ); // Pone hora, minutos, segundos.
    void print24H();       // Imprime la hora en formato 24H
    void print12H();       // Imprime la hora en formato 12 horas (pm)
private:
    int hour;             // 0 - 23
    int minute;          // 0 - 59
    int second;          // 0 - 59
    void chequea_valor(int i); // ¿Tiene sentido un método private?
};

void Time::setTime(int h, int m, int s)
{ .....
}
```

5.1 Repaso de Sesiones Anteriores

Ventajas hasta ahora

Acceso a datos controlado
(Funciones públicas)

Funciones de inicialización finalización incorporadas
(Constructores y Destrcutores)

No tengo que pasar un puntero a
la estructura en cada función

5.2 Ocultado la Implementación

Ocultando mediante tipos de datos enumerados.

```
class Time
{
public:
    enum
    {
        MIN_HORA = 0,
        MAX_HORA = 60
    };
    Time(); // constructor
    void setTime( int, int, int ); // Pone hora, minutos, segundos.
    .....
private:
    int hour; // 0 - 23
    .....
};
```

if (Time::MAX_HORA) Dentro de una función : MAX_MINUTOS

5.2 Ocultado la Implementación

Sistema de Protección

Ya sabemos que los miembros privados de una clase no son accesibles para funciones y clases exteriores a dicha clase.

Esto es un concepto de POO, el encapsulamiento hace que cada objeto se comporte de un modo autónomo y que lo que pase en su interior sea invisible para el resto de objetos. Cada objeto sólo responde a ciertos mensajes y proporciona determinadas salidas.

En ciertas ocasiones queremos “saltarnos” estas restricciones y acceder a miembros privados de un objeto de una clase desde objetos de clases diferentes.

5.2 Declaraciones Friend

Características de las relaciones de amistad

Declaro relaciones de amistad entre clases o funciones.

- **La amistad no puede transferirse:**

Si A es amigo de B, y B es amigo de C ->

A no tiene que ser amigo de C (y viceversa)

“Los amigos de mis amigos son mis amigos” NO APLICA.

- **La amistad no puede heredarse:**

Si A es amigo de B, y C deriva de B ->

A no tiene que ser amigo de C (y viceversa)

“Los hijos de mis amigos son mis amigos” NO APLICA.

- **La amistad no es simétrica:**

Si A es amigo de B -> B no tiene por qué ser amigo de A

“....”

5.2 Declaraciones Friend

Funciones amigas Externas

```
class ClaseA
{
public:
    ClaseA(int i=0) : {i = 0} // Funcion Constructor inline?
    void ver_a() { cout << a << endl; }
private:
    int a;
    friend void funcion_amiga_mia_ver(ClaseA); // Amiga de la clase A
};
void funcion_amiga_mia_ver(ClaseA Xa)
{
    // La función Ver puede acceder a miembros privados de la ClaseA
    cout << Xa.a << endl;
}
.....
ClaseA objeto1;
funcion_amiga_mia_ver(objeto1); // Ver el valor de Na.a
objeto1.ver_a(); // Equivalente a la anterior. Veremos su utilidad
```

5.2 Declaraciones Friend

Funciones amigas en otra clase

```
class A; // Declaración previa
class B
{
public:
    B(int i=0) : b(i) {}
    void Ver()
    {
        cout << b << endl;
    }
    // Compara b con a
    bool EsMayor(A Xa);
private:
    int b;
};
```

```
class A
{
public:
    A(int i=0) : a(i) {}
    void Ver()
    {
        cout << a << endl;
    }
private:
    // Amiga con acceso a A
    friend bool B::EsMayor(A Xa);
    int a;
};
```

5.2 Declaraciones Friend

Funciones amigas en otra Clase

```
bool B::EsMayor(A Xa)
{
    return b > Xa.a;
}
int main()
{
    A a(10);
    B b(12);
    a.Ver();
    b.Ver();
    if(b.EsMayor(a)) cout << "b es mayor que a" << endl;
    else cout << "b no es mayor que a" << endl;
    cin.get();
    return 0;
}
```

Estas "amistades" serán útiles cuando sobrecarguemos algunos operadores.

5.2 Declaraciones Friend

Clases amigas

```
class Elemento
{
    public:
        /// Constructor
        Elemento(int t);
        /// Obtener el tipo del elemento.
        /// @retval tipo del elemento.
        int Tipo()          // Funcion inline No abusemos de ellas...
        {
            return tipo;
        }
    private:
        /// Tipo
        int tipo;
        /// Siguiete elemento
        Elemento *sig;
        friend class Lista; // Declaro mi amistad con la lista (Clase Lista)
};
```

5.2 Declaraciones Friend

```
class Lista
{
public:
    /// Constructor
    Lista() : cabeza(NULL) {};
    /// Destructor.
    ~Lista() { liberar_lista(); };
    /// Insertar nuevo elemento.
    void Nuevo(int tipo);
    /// Obtener primer elemento
    Elemento *primero()
    {
        return cabeza;
    }
}
```

```
/// Siguiete elemento a p
Elemento *siguiete(Elemento *p)
{
    if(p)
        return p->sig;
    else
        return p;
};
/// lista está vacía?
bool esta_vacia()
{
    return cabeza == NULL;
}
private:
    // Puntero al primer elemento
    Elemento *cabeza;
    // Privada para borrar lista
    void liberar_lista();
};
```

5.2 Declaraciones Friend

```
Elemento::Elemento(int t) : tipo(t), sig(NULL) {}  
// -----  
void Lista::Nuevo(int tipo)  
{  
    Elemento *p;  
    p = new Elemento(tipo);    // Nuevo elemento.  
    p->sig = cabeza;          // Al principio de la lista  
    cabeza = p;  
}  
void Lista::liberar_lista()  
{  
    Elemento *p;  
    while(cabeza)  
    {  
        p = cabeza;  
        cabeza = p->sig;  
        delete p;  
    }  
}
```

5.3 Ejercicios

Clases y Funciones amigas

- 1.- Compilar el código fuente de Clases Amigas
- 2.- Quitar la declaración de friend
Mostrar lo que pasa en la pizarra virtual "MoonEdit"
- 3.- Compilar el código fuente de Clases_Amigas_Lista_Enlazada
- 5.- Identificar código que permite la declaración Friend
Mostrar lo que pasa en la pizarra virtual "MoonEdit"

5.2 Resumen

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();
};
```

5.2 Resumen

Otras formas de Funciones inline y declarando por adelantado

```
class A; // Declaración previa
// -----
class B
{
public:
    B(int i=0) : b(i) {}; // ??????? ¿no era B();?
    void Ver()
    {
        cout << b << endl;
    }
    // Compara b con a
    bool EsMayor(A Xa);
private:
    int b;
};
// -----
class A .....
```