



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Departamento de Electrónica

Funciones y Clases Amigas (Friend) Miembros Estáticos (Static)

ELO329: Diseño y Programación Orientados a Objetos

Departamento de Electrónica

Universidad Técnica Federico Santa María


Funciones y Clases Friend

- ❑ El calificador friend se puede aplicar a funciones globales y clases para otorgar acceso a miembros privados y protegidos de una clase.
- ❑ La función función global “friend” tendrá acceso a los miembros como si fuera un método de la clase.
- ❑ Una clase B es friend de otra A cuando sus métodos tiene acceso a los miembros privados y protegidos de la clase A que la ha declarado friend.

Ejemplo de función Friend:

```
class Course {  
public:  
    friend bool ValidateCourseData(const Course &C);  
private:  
    int nCredits;  
    //...  
};
```

Función global, No exclusiva de la clase!! Solo prototipo, su implementación no pertenece a la clase



Implementación de ValidateCourseData()

- ❑ El calificador friend no aparece en la implementación de la función global sólo en la clase que otorga el acceso.
- ❑ Notar el acceso a miembros privados (nCredits) de la clase. Esto es posible por ser función amiga.

```
bool ValidateCourseData(const Course & C) {  
    if( C.nCredits < 1 || C.nCredits > 5 )  
        return false;  
}  
return true;  
}
```

Una mejor forma de programar esto es:

```
return !(C.nCredits < 1 || C.nCredits > 5 );
```

Funciones Friend, otro ejemplo

```
class CVector {  
    private:  
        double x,y;  
    public:  
        CVector () {};  
        CVector (double , double);  
        CVector operator + (const CVector &) const;  
        friend CVector operator * (double factor, CVector v);  
        friend ostream & operator<< (ostream &, const CVector &);  
};
```

En este ejemplo, la función `operator*` tiene acceso a los miembros privados de `CVector`. Notar la sobrecarga de operaciones del tipo:
`v2 = 3*v1`; cosa que no podemos hacer como método de la clase.

```
#include "CVector.h"  
CVector::CVector (double a, double b) {  
    x = a;  
    y = b;  
}  
// sobre carga operador + dentro de clase  
CVector CVector::operator+ (const CVector &param) const {  
    CVector temp;  
    temp.x = x + param.x;  
    temp.y = y + param.y;  
    return temp;  
}  
// sobre carga operador * al operar double*CVector.  
CVector operator * (double factor, CVector v){  
    return CVector(factor*v.x, factor*v.y);  
}  
// sobre carga operador << como función global.  
ostream & operator << (ostream &os, const CVector &v) {  
    os << "(" << v.x << "," << v.y << " )";  
    return os;  
}
```

Ahora Clases Friend: clases amigas

// Example of a friend class

```
class YourClass {
```

```
    // .....
```

```
friend class YourOtherClass; // Declara una clase friend
```

```
private:
```

```
    int topSecret;
```

```
};
```

```
class YourOtherClass{
```

```
public:
```

```
    void change(YourClass & yc);
```

```
};
```

```
void YourOtherClass::change(YourClass & yc) {
```

```
    yc.topSecret++; // Puede acceder datos privados !
```

```
}
```

Una clase amiga (friend) es una clase cuyas funciones miembros son como funciones miembros de la clase que la hace amiga. Sus funciones miembros tienen acceso a los miembros privados y protegidos de la otra clase.

Classes Friend (cont.)

- ❑ La “Amistad” no es mutua. En el ejemplo previo, los métodos de YourClass no pueden acceder a miembros privados de YourOtherClass.
- ❑ La “Amistad” no se hereda; esto es, clases derivadas de YourOtherClass no pueden acceder a miembros privados de YourClass. Tampoco es transitiva; esto es clases que son “friends” de YourOtherClass no pueden acceder a miembros privados de YourClass.
- ❑ La “amistad” es importante en sobrecarga de operador <<, para escritura a pantalla pues en este caso no podemos agregar sobrecargas a clases estándares. Ver ejemplo [CVectorFriend](#)

Static: Miembros Estáticos

- ❑ No hay gran diferencia con Java. Difieren en su asignación.
- ❑ Estas variables tienen existencia desde que el programa se inicia hasta que termina.
- ❑ Atributos estáticos
 - El atributo es compartido por todas las instancias de la clase. Todas las instancias de la clase comparten el mismo valor del atributo estático. Igual que en Java.
- ❑ Métodos Estáticos
 - Estos métodos pueden ser invocados sobre la clase, no solo sobre una instancia en particular. Igual que en Java.
 - El método sólo puede acceder miembros estáticos de la clase
- ❑ Es posible pensar en miembros estáticos como atributos de la clase y no de objetos. Hasta aquí igual a Java.

Declaración de Datos Estáticos

- ❑ La palabra clave `static` debe ser usada.

```
class Student {  
    //...  
    private:  
        static int m_snCount;    // instance m_snCount  
};
```



Creación de un contador de instancias

- ❑ La inicialización de un dato estático no se efectúa en el constructor pues existe previo a la creación de cualquier objeto.
- ❑ En Java lo hacíamos en bloque de iniciación static
static { }
- ❑ La iniciación de atributos estáticos es una diferencia entre C++ y Java.

```
// student.cpp
```

```
int Student::m_snCount = 0;
```

Asigna memoria e inicia el valor de partida. Se ejecuta antes de ingresar al main.



Creación de un Contador de Instancias

- ❑ Usamos el constructor y destructor para incrementar y decrementar el contador:

```
Student::Student() {  
    m_snCount++;  
}  
  
Student::~~Student() {  
    m_snCount--;  
}
```

Métodos Estáticos

- ❑ Usamos métodos estáticos para permitir el acceso público a miembros de datos estáticos sin necesidad de instanciar la clase.

```
class Student {  
    public:  
        static int get_InstanceCount();  
    private:  
        static int m_snCount; // instance count  
};
```

Llamando a Métodos Estáticos

- ❑ Como en java, usamos ya sea el nombre de la clase o una instancia de la clase para acceder al método:

```
cout << Student::get_InstanceCount(); // 0
Student S1;
Student S2;
cout << Student::get_InstanceCount(); // 2
cout << S1.get_InstanceCount();      // 2
```