## Aspectos interesantes de C++

**Funciones en línea:**
Cuando en C++ definimos una función ( de la clase ) dentro de la clase, por ejemplo:

```
class puntoGordo
{
  private:
    int x,y;
public:
        puntoGordo( int x, int y ) { puntoGordo::x = x; puntoGordo::y = y; }
        void desplaza( int dx, int dy ) { x += dx; y += dy }
        void escribe() { cout << '(' << x << ',' << y << ')' <<'\n'; }
};
```

los objetos resultantes "traen" el código a cuestas, es decir cada instancia tiene su propia copia del código.

**Funciones fuera de línea:**
Por el contrario, cuando definimos solamente la interfaz, como en

```
class puntoFlaco
{
  private:
    int x,y;
public:
        puntoFlaco( int x, int y );
        void desplaza( int dx, int dy );
        void escribe();
};

        puntoFlaco::puntoFlaco( int x, int y ) { puntoFlaco::x = x; puntoFlaco::y = y; }
        void PuntoFlaco::desplaza( int dx, int dy ) { x += dx; y += dy }
        void PuntoFlaco::escribe() { cout << '(' << x << ',' << y << ')' <<'\n'; }
```

los objetos resultantes tienen solamente los atributos. Los métodos los comparten ( una sola copia para todos y el compilador resuelve reentrancia.

También es posible sobreseer la modalidad del código fuera de línea

```
        inline void PuntoFlaco::desplaza( int dx, int dy ) { x += dx; y += dy }
```

Argumentos con valores por default:

Es posible definir valores por default para los argumentos de las funciones:

```
  puntoFlaco::puntoFlaco( int x = 0, int y = 0 ) { puntoFlaco::x = x; puntoFlaco::y = y; }
```

esto es válido para todas las funciones que se pueden definir en C++, no solo métodos definidos en clases.
En este rubro, los valores por default no necesitan ser constantes, también pueden ser expresiones cuyos elementos estén al alcance de la definición de la función.

Las **struc** que se trabajaron anteriormente se convierten ahora ( sin más trámite ) en clases cuyos atributos y métodos son todos públicos, y como clases admiten métodos:

```
Struct Pto{
                    int x,y;
                    void Pto( int, int),
                    void desplaza( int, int );
                    };
```

En C++ es **possible "anidar" las declaraciones de clases**, sin embargo la clase anidada tiene un alcance a todo el módulo de código donde sea vista la clase anidadora, es decir equivale a que se hubiese declarado afuera.

```
class Triangulo2D
{
        class Punto2D
        {
                private:
                        int x,y;
                public:
                        Punto2D( int x = 0, int y = 0 )
                        { Punto2D::x = x; Punto2D::y = y;}
                        void escribe()
                        { cout << '(' << x << ',' << y << ')' << '\n'; }
        };

        private:
                Punto2D a,b,c;
        public:
                Triangulo( int ax, int ay, int bx,int by, int cx, int cy );
                Void desplaza( Punto2D d );
};
        .
        .
        .
```

**Variables globales:**

Cuando dentro de una función miembro de una clase deseamos hacer referencia a una variable global que tiene el mismo nombre que un atributo ( variable de instancia ) o que

una variable local al método, usamos el operador de alcance :: , sin embargo en este caso no escribimos nada antes del ::, lo dejamos "hueco" para referie¿rnos a la variable global:

```
void Punto::relocaliza()
{   Punto::x = ::x; }
```

en C++ es posible definir **"namespaces"**, espacios con nombre a los cuales podemos acceder por medio del operador de alcance:

```
// one.h
namespace one
{
        char func(char);
        class String { ... };
}

// somelib.h
namespace SomeLib
{
        class String { ... };
}
```

one::String y SomeLib::String, hacen referencia no ambigua a los respectivos símbolos o identificadores.

**Arreglos de Objetos:**

Si es posible instanciar objetos sin proporcionar argumentos específicos al constructor entonces se pueden definir arreglos de objetos de tal clase:

```
Punto3D a[10];        // 10 objetos de clase Punto3D
```

Asimismo es posible definir arreglos de un número de elementos que se conoce exactamente a la hora de la instanciación:

```
Punto3D b[m];        // al momento de ejecutar esta instrucción m debe tener valor (
```

i.e. no debe ser cobarde ).

**Miembros static:**

Es posible definir miembros de tipo "static" dentro de una clase. Existirá solamente un solo miembro para esta clase y lo comparten todos las instancias de tal clase.

```
static int cuantos = 0;
class CuentaHijos
{private:
```

```
  int  cual;

public:
        CuentaHijos()           { cual = ++cuantos; }
        static void diCuantos() { cout << "Van " << cuantos << " objetos\n"; }
        friend ostream& operator <<( ostream& s, CuentaHijos& x )
        { s << "Hijo:" << x.cual << '\n';   return s; }
};
void main(int argc, char* argv[])
{
        CuentaHijos::diCuantos();
        CuentaHijos x,y,z;
        CuentaHijos::diCuantos(); z.diCuantos();
        cout << x << y << z;
}
```

También es posible definir variables static dentro de una función, en tal caso se inicializan solamente la primera vez que se invocan y posteriormente no son re-inicializadas.

Respecto a la **herencia** señor...
Cuando derivamos una clase de otra, en ocasiones es útil la invocación al constructor de la clase padre como:

```
class Padre
{
        .
        .
        .
};

class Hija:Padre
{
        Hija( argumentos ):Padre( argumentos_de_inicialización_del_Padre )
        {
        }

};
```

Respecto a la **inclusión** ( o contención )...
Cuando un miembro de una clase es un objeto de otra clase, se le puede inicializar “al vuelo” en el constructor de la clase contenedora:

```
Triangulo::Triangulo( int ax, int ay, int bx, int by, int cx, int cy )
                        a(ax,ay),b(bx,by),c(cx,cy)
{

}
```