

# Introducción a la Programación Orientada a Objetos - Herramientas de C++

Informática II  
R2004 - 2021

# Programación Orientada a Objetos (POO)

La programación orientada a objetos surge como una necesidad de reutilizar código y hacer más eficientes y robustos los programas que venimos realizando.

Comúnmente, desarrollamos código pensando algoritmos que resuelvan una aplicación determinada. La POO, busca, sin embargo, desarrollar bloques (u objetos) de código que puedan luego ser reutilizados por diversas aplicaciones.

El foco, por lo tanto, estará puesto en el desarrollo de estos objetos, y posteriormente en buscar la forma de encapsularlos (de manera de que sean lo más robustos posibles), de darles métodos de comunicación con otros objetos (para vincularlos en diversas aplicaciones), y en permitirles crecer para adaptarse a proyectos de mayor envergadura.

El C++ surge como una “evolución” del C, que agrega características para facilitar la realización de programas orientados a objetos.

# C++ como “evolución” del C - Clases

El C++ aparece como un lenguaje en donde una de las características estructurales es la **encapsulación de datos**.

Para lograr esto posee algunas características distintas al C. La primera que veremos es la introducción de las CLASES:

```
class Persona {  
  
    public:  
  
        char *  getFullName();  
                setNombre(char *);  
                setApellido(char *);  
  
    private:  
  
        char Nombre[N];  
        char Apellido[N];  
  
}
```

Una clase contiene **PROPIEDADES** (variables) y **MÉTODOS** (funciones), que describen un subsistema dentro de nuestro programa. La **INSTANCIA** de una clase es un **OBJETO**.

Al momento de **INSTANCIARSE** el **OBJETO** se llama a un método **CONSTRUCTOR**, que típicamente es el método que inicializa mi objeto, dándole a sus **PROPIEDADES** los valores adecuados para su funcionamiento.

# C++ como “evolución” del C - Clases

Vamos viendo lo anterior, paso a paso:

1. **Una clase contiene PROPIEDADES y MÉTODOS.** Podemos pensar en una clase como una estructura, que además de variables tiene también funciones:

```
class Persona {  
    public:  
        char *  getFullName();  
        setNombre(char *);  
        setApellido(char *);  
  
    private:  
        char Nombre[N];  
        char Apellido[N];  
}
```



**MÉTODOS** de la clase: Son funciones que controlarán la forma en que se accede a las variables. De esta manera, no puedo cargar CUALQUIER VALOR en Nombre y Apellido, sino que debo llamar a la función SetNombre, que se encargará de controlar que el string que recibe por argumento sea válido (para mi programa)

**ATRIBUTOS** de la clase: Son las variables que se encuentran contenidas dentro de la clase. Al ser PRIVADAS, no podremos acceder directamente a ellas (lo veremos más adelante)

# C++ como “evolución” del C - Clases

Vamos viendo lo anterior, paso a paso:

**2. La INSTANCIA de una clase es un OBJETO.** Un objeto se crea cuando INSTANCIAMOS la clase. Para lo que veníamos haciendo, esto sería equivalente a decir que un objeto se crea cuando reservo memoria para una variable de ese tipo:

```
class Persona{
```

```
    public:
```

```
        ...
```

```
    private:
```

```
        ...
```

```
};
```

```
class Persona P1;
```

```
Persona P2,P3;
```



En este momento DISEÑO la clase, pero no instancio objetos.



En el momento en que reservo memoria (en forma local o global) para las variables de tipo Persona, se dice que estoy **INSTANCIANDO OBJETOS DE TIPO PERSONA.**

Notemos que en C++ podemos o no incluir la palabra reservada **class** cuando creamos los objetos.

# Paradigmas de programación: P00

En la programación orientada a OBJETOS vamos a focalizar la atención en **diseñar las clases** que sean necesarias para el funcionamiento de mi programa, cada una de ellas con las **interfaces** adecuadas, de manera de que el programa se reduzca a **instanciar** los objetos necesarios y **comunicarlos** de acuerdo a la lógica de mi sistema.

```
class Persona
```

```
class Estudiante
```

```
void main () {
```

```
    Persona P1;
```

```
    Estudiante E1;
```

```
    E1.setName( P1.getFullName() );
```

```
}
```

Si las clases están bien diseñadas, los objetos P1 y E1 tendrán métodos que permitan conocer el nombre de la persona P1 y cargar el nombre del estudiante E1, por lo que nuestra aplicación solo deberá invocar a dichos métodos

# Diferencias básicas entre C y C++

Una vez explicado el propósito de la creación de un nuevo lenguaje, veremos algunas características que fueron incluidas en esta “evolución” del lenguaje C.

Tengamos siempre presente que todo lo que veníamos haciendo en C (salvo algunas excepciones) puede realizarse en C++, pero además hay ciertos agregados y características que en algunos casos facilitan el uso, y en otros agregan ciertas modalidades especiales o restricciones a la forma en que acostumbramos programar.

Iremos viendo una a una estas nuevas características:



# Diferencias del lenguaje C++

- Los archivos fuente de C++ tienen la extensión **\*.cpp** (de C plus plus) en lugar de **\*.c** que conocemos y usamos en lenguaje C
- El compilador identifica el lenguaje de programación utilizado mediante la extensión de los archivos fuente

Inclusión de encabezadores en C

```
#include <stdio.h>
```

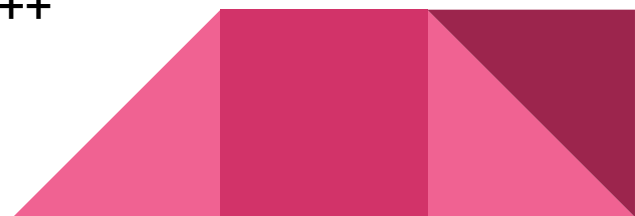
```
#include "complejos.h"
```

Inclusión de encabezadores en C++

```
#include <iostream>
```

```
#include "complejos.h"
```

En la mayoría de los casos  
los archivos de encabezado  
de librería no llevan la  
extensión .h






# Flexibilidad para declarar variables

En C++ las variables locales pueden ser declaradas en cualquier parte del bloque de código

```
for (double suma = 0.0, int i = 0; i<n; i++)  
    suma += a[i];
```



La variable suma se instancia (se crea en el stack) en el momento en que comienza el for y deja de existir (se libera del stack) una vez que termine el for.  
Se puede hacer en cualquier SCOPE (dentro de un if, de un while, de un for, etc.)

# Datos bool

- En C no existe un tipo de datos **bool** con valores verdadero o falso, hay que simularlo con variables char que adopten valores **"0"** o **"distinto de 0"** o **"negativo"** y **"positivo"**
- En C++ existen las variable tipo **bool** que pueden tomar los valores **TRUE** y **FALSE**

```
bool a = TRUE;
if ( a == FALSE ) {
    ...
}
```

TRUE y FALSE son palabras reservadas del lenguaje

# Constantes

C++ agrega las “variables constantes”. Les llamamos variables por la idea de que ocupan un lugar en memoria, pero su propósito es contener un valor constante. Se usan para reemplazar lo que en C hacíamos con la instrucción de preprocesador `#define`. La idea es que estas constantes pertenezcan siempre a una clase (o a otro ámbito, o namespace, que ya veremos más adelante) y de esta forma no sean propias de cada programa, sino que pertenezcan a un objeto. Por ejemplo:

```
class Persona{  
    public:  
        const char NONAME = 'N';  
    private:  
        ...  
};  
  
void main (void){  
    Persona P1;  
    P1.setNombre (P1.NONAME);|
```

**NONAME** es una variable (ocupa un lugar en la memoria), pero su valor no puede cambiar. Para los propósitos de los objetos de tipo Persona, **NONAME** representará que no tiene un nombre cargado.

Pero un objeto de otro tipo puede tener una constante **NONAME** que tenga OTRO valor.

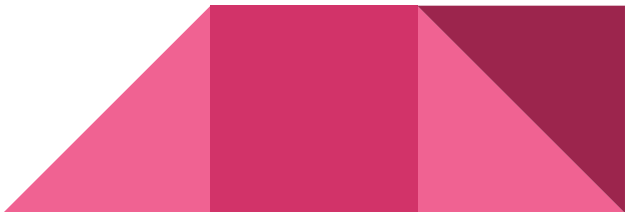
Cada constante se puede usar dentro de su ámbito de pertenencia (tengo que agregar la clase Persona para poder usar **NONAME**)

# Constantes

Constantes en C con el preprocesador **Dificultades:**

- Haciendo `#define PI 3,1416` las constantes quedan fuera del ámbito del compilador
- No se realiza comprobación de tipos y no se puede obtener la dirección de **PI**
- **PI** dura desde el momento en que es definida hasta el final del archivo.

Constantes en C++ (se declara con la palabra clave **const**) **Ventajas:**

- Es lo mismo que una variable pero su valor NO puede cambiar.
  - Las constantes tienen ámbito => se la puede 'esconder' dentro de una función => no afecta el resto del programa.
  - Al compilar, se verifica la comprobación de tipos.
- 

# Paso de valores por referencia

```
void main (void) {  
    int a, b = 4; // b vale 4  
    a = funcion (&b, 6);  
    //b vale posiblemente otro valor  
    -----  
}  
  
int funcion (int *x, int y) {  
    int z;  
    scanf ("%d", &z);  
    *x = z+y;  
    return z;  
}
```

Así es como veníamos haciendo pasajes por REFERENCIA en C. Notemos que no estamos pasando b por referencia a la función, sino que estamos pasando la **DIRECCIÓN DE b por VALOR**.

Esto nos ayuda a modificar el valor de b desde otra función que no tiene visibilidad de la variable, pero también significa que estamos creando una nueva variable de tipo puntero y tenemos que reservar memoria para esta y copiar su valor (es un pasaje por valor)


Notemos además, que si quisieramos pasar un PUNTERO por REFERENCIA, deberíamos pasar un puntero a puntero, ya que tenemos que pasar la dirección de una variable cuyo contenido también es una dirección (lo cual hace más engorrosos nuestros programas)

# VERDADERO Paso de valores por referencia

```
void main (void) {  
    int a, b = 4; // b vale 4  
    a = funcion (b, 6);  
    //b vale posiblemente otro valor  
    -----  
}  
  
int funcion (int &x, int y) {  
    int z;  
    scanf ("%d", &z);  
    x = z+y;  
    return z;  
}
```

x es un **ALIAS** de b

Para declarar una variable Alias se realiza con & en la declaración de la misma.



# Pasajes por referencia - diferencias con pasar un puntero por valor (en la declaración y en el uso)

```
int funcion (int *, int )
```

```
void main (void) {  
    int a, b = 4;  
    a = funcion (&b, 6);  
    -----  
}
```

```
int funcion (int *x, int y) {  
    int z;  
    scanf ("%d", &z);  
    *x = z+y;  
    return z;  
}
```

```
int funcion (int &, int y)
```

```
void main (void) {  
    int a, b = 4;  
    a = funcion (b, 6);  
    -----  
}
```

```
int funcion (int &x, int y) {  
    int z;  
    scanf ("%d", &z);  
    x = z+y;  
    return z;  
}
```

En el segundo caso, x es realmente un **ALIAS** de b. Es simplemente otra forma de llamar a la misma variable. No ocupa un lugar en la memoria, simplemente hace referencia al mismo lugar que b. No hace falta llamarla como "el contenido de" o "la dirección de". Se simboliza con el operador & en el lugar del argumento de la función

# Visibilidad o Scope

La **visibilidad** o **scope** de una variable es la parte del programa en la que dicha variable está definida y puede ser utilizada

```
int a;
```

```
void main (void) {
```

```
    int a;
```

```
    for (int a = 0 ; a < 10 ; a++ ) {
```

```
        vec[a] = a*2;
```

```
    }
```

```
    a = 5;
```

```
}
```

**SCOPE** de las variables:

**a** está definida en forma global

**a** está definida en forma local al main

**a** esté definida dentro del ciclo for

**Cuando haga referencia a la variable a, va a depender el scope desde donde lo haga para saber a que variable estoy haciendo referencia**



# Visibilidad o Scope - Operador ::

En C++ la visibilidad de una variable puede ser:

1. local
2. a nivel de archivo (global) o
3. a nivel de **clase**.

Al tener declaradas una **variable global** y otra **variable local** del mismo nombre, la **variable global** está oculta por la **variable local**.

scope resolution operator: ' :: '

Este operador, antepuesto al nombre de una **variable global** que está oculta por una **variable local** del mismo nombre, permite acceder al valor de la variable global



# Visibilidad o Scope - Operador ::

En el ejemplo anterior, podemos acceder a la variable **a** desde el main, o desde el for:

```
int a;
```

```
void main (void) {  
    int a;  
    ::a = 10;  
    a = 5;  
    for (int a = 0 ; a < 10 ; a++ ) {  
        vec[a] = a*2;  
        ::a=20  
    }  
}
```

**Cada vez que utilizo el operador ::  
"subo" en 1 el scope de la variable  
(del for paso al main, del main al  
scope global)**

# Operador :: para acceder a miembros de una clase

Los miembros de una clase se acceden mediante el operador “.”

```
Persona P1;  
P1.setEdad(20);
```

Sin embargo, algunos miembros que NO DEPENDEN de que haya objetos creados (típicamente, miembros constantes o ESTÁTICOS -ya los veremos más adelante-), pueden accederse con el operador de visibilidad:

```
char a = ( Persona :: NONAME );
```

Esto se lee como “El miembro NONAME dentro del scope de la clase **Persona**”

# Salida por pantalla y entrada por teclado

De la misma forma que veníamos utilizando `printf` y `scanf` (de ***stdio.h***) para la entrada de texto y salida por pantalla, la biblioteca ***iostream*** de c++ incorpora dos objetos que facilitan su utilización

Los objetos **`cin`** y **`cout`** nos permiten manejar los streams de pantalla y teclado facilmente:

- `cout << variable << "texto" << endl`
- `cin >> variable`

Imprime por pantalla variables (les da el formato automáticamente), cadenas de caracteres y caracteres especiales (`endl`, similar al `'\n'`, por ejemplo)

Nos abstraemos del formato de las variables y los datos a imprimir/leer.

*cin* es un objeto de la clase ***istream*** y *cout* un objeto de ***ostream***, ambas incluídas en la biblioteca ***iostream***

# namespace

Es un conjunto de nombres (*identificadores*), agrupados en un '**espacio**' en el cual todos ellos (variables, funciones, identificadores) **son únicos**

```
namespace espacio_2D {  
    struct Punto {  
        int x;  
        int y;  
    };  
}  
namespace espacio_3D {  
    struct Punto {  
        int x;  
        int y;  
        int z;  
    };  
}
```

Un espacio de nombres queda definido por el uso de la palabra clave **namespace** seguida de llaves de apertura y cierre

para utilizar las variables, funciones o identificadores se debe incluir la siguiente línea:

**using namespace espacio\_2D;**

Por ejemplo, **endl**, y otros identificadores que utilizamos para la realización de programas en una PC se encuentran en el **namespace std** (veremos incluido normalmente este namespace en nuestros programas)

# Memoria dinámica

Las funciones de librería malloc() y free() de ANSI C son reemplazadas por los operadores de C++ **new** y **delete**. Su utilización es la siguiente:

- new      **tipo\_de\_datos**      o      new      **tipo\_de\_datos** [tam\_vector]
- delete    **variable**                      o      delete [] **variable**

Genera un vector dinámico de i posiciones, cuya dirección de inicio se guarda en l (no hace falta hacer un cast)

Accedo al vector dinámico mediante su dirección de inicio (como lo veníamos haciendo)

Borra el vector dinámico completo (no hay que borrar posición por posición)

```
long * l, total = 0;
cout << "Cuantos números desea ingresar? ";
cin >> i;
l = new long[i];
if (l == NULL) exit (1);
for (n=0; n<i; n++) {
    cout << "Número: ";
    cin >> input;
    l[n]=atol (input); }
cout << "Usted ingreso: ";
for (n=0; n<i; n++) cout << l[n] << ", ";
delete[] l;
return 0; }
```

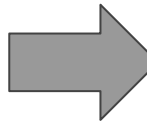
# Sobrecarga de funciones (Polimorfismo)

Otra de las ventajas que ofrece C++ es la posibilidad de sobrecargar una función. Esto es, varios métodos pueden compartir el mismo nombre pero diferenciarse según:

- El TIPO de parámetros que reciban
- La CANTIDAD de parámetros que reciban.

Sin embargo, **no puede** haber dos métodos con igual cantidad y tipo de parámetros, pero con distinto valor de retorno.

int	Suma	(int a, int b)
int	Suma	(int a, int b, int c)
float	Suma	(float a, float b)




3 funciones diferentes pueden tener el mismo nombre, y el compilador resolverá cuál de las 3 está siendo invocada por el tipo y el número de argumentos que recibe

# Ejemplos...

Ejemplo **Scope** - Aquí veremos como 3 variables distintas con el mismo nombre se diferencian a partir del scope donde se encuentran instanciadas, y como puedo utilizar el operador `::` para acceder a todas ellas.

Ejemplo **Referencia** - Aquí veremos como pasar valores por REFERENCIA, a diferencia de pasar un puntero con su dirección por VALOR.

Ejemplo **Sobrecarga** - Aquí veremos como 3 funciones con el mismo nombre se invocan en forma diferenciada según el tipo y la cantidad de parámetros que reciben.






# ¿Cómo estructuramos nuestro programa en C++?

A diferencia de la programación que venimos realizando hasta la POO requiere antes de ponerse a programar pensar las clases de nuestro programa y que métodos y atributos (funciones y variables) que van a tener. Debemos:

**Primero:** Pensamos las clases para resolver una aplicación y nos focalizamos en hacerlas lo más abarcativas (con todas las interfaces), seguras (con los datos debidamente protegidos) y encapsuladas (con todas las funcionalidades DENTRO de la clase) posibles.

**Luego:** Instanciamos las clases y usamos sus interfaces (creamos los objetos y llamamos a los métodos para resolver la aplicación)



# Clases en C++

Usando como ejemplo una aplicación que pida un número imaginario e imprima por pantalla su parte real y su parte imaginaria en forma separadas y su módulo.


**Creamos una clase "Complejo" que represente un número imaginario**

Siempre empieza con palabra **class** equivalente a la palabra **struct** en las estructuras

**private y public:** cómo se accede del exterior (profundizaremos más adelante)

función con el mismo nombre de la clase que se ejecuta al comienzo de instancia de la clase (se hacen las inicializaciones)

Los métodos que agregan funcionalidad a mi objeto

```
class Complejo {  
    private:  double real, imag;  
    public:  
        Complejo (double, double);  
  
        double getReal();  
        double getImag();  
        setReal (double);  
        setImag (double);  
        double getModulo();  
}
```

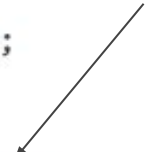
# Constructores

Las clases pueden tener un método **CONSTRUCTOR**. Este método es una función que será invocada en el momento en que se cree el objeto (o sea que se instancie la clase).

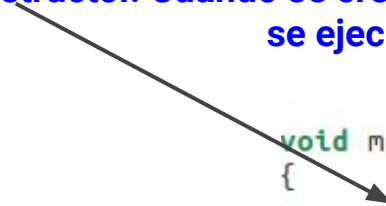
El constructor se diferencia del resto de los métodos por llamarse de igual manera que la clase.

```
class Punto {  
    private:  
    int x,y;  
  
    public:  
    Punto() {  
        x = 0;  
        y = 0;  
    }  
}
```

Se inicializan las variables ya que al igual que en las estructuras no se puede inicializar cuando creo la clase por no estoy creando una variable hasta que no la instancio.



Este es el momento cuando creo la variable y se ejecuta el constructor. Cuando se crea la variable Punto1 de la clase Punto se ejecuta el constructor.



```
void main ()  
{  
    Punto Punto1;  
    ...  
}
```

# Constructores (II)

El método **CONSTRUCTOR** también puede recibir parámetros para inicializar las variables de la clase. Para ello se reciben como parámetros al igual que en cualquier función.

```
class Punto {  
    private:  
    int x,y;  
  
    public:  
    Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
}
```

La función recibe los parámetros de las variables que se desea inicializar.



Se le pasan los parámetros cuando instancio el objeto de la clase.



```
void main ()  
{  
    Punto Punto1(3,5);  
  
    ...  
}
```

# Constructores (III)

El método **CONSTRUCTOR** también puede recibir como parámetro a otro objeto de la misma clase para al crear un nuevo objeto se cree como copia de otro. Se lo denomina constructor de copia.


```
class Punto
{
private:
    int x, y;
public:
    Punto(const Punto &P){
        x = P.x;
        y = P.y;
    }
}
```

Recibimos un objeto Punto. Se debe recibir como referencia para que no se vuelva a invocar al constructor.



```
int main()
{
    Punto c1(2, 3), c2(c1);
}
```

Se ejecuta el constructor de copia cuando instancio un objeto pasando otro objeto como parámetro.



# Destruectores

De la misma manera, el **destructor** es un método que se invoca al momento en que deja de existir el objeto (ya sea porque se libera la memoria con el operador delete o porque se abandona el ámbito de pertenencia del mismo)

Se lo reconoce porque se llama de igual manera que la clase, pero con el operador ~ antecediendo al nombre:

```
class Punto {  
    private:  
    int    x,y;  
  
    public:  
    Punto(int a, int b) {  
        x = a;  
        y = b;  
    }  
  
    ~Punto(){  
        cout << "Punto borrado: " << x << " ; " << y << endl;  
    }  
}
```



```
Punto *p;  
...  
p = new Punto(3,5);  
...  
delete p;  
...
```

**LOS CONSTRUCTORES Y DESTRUCTORES  
NO PUEDEN RETORNAR VALORES!!**

# Sobrecarga de funciones - constructores:

```
class Punto {
```

```
private:
```

```
double X, Y;
```

```
public:
```

```
Punto () {
```

```
    X = 0;
```

```
    Y = 0;
```

```
}
```



Constructor por defecto

```
Punto (double X1, double Y1) {
```

```
    X = X1;
```

```
    Y = Y1;
```

```
}
```



Constructor parametrizado

```
Punto (Punto &P1) {
```

```
    X = P1.X;
```

```
    Y = P1.Y;
```

```
}
```



Constructor "de copia"

```
~Punto() {
```

```
    cout << "Punto " << X << " ;" << Y << " Borrado";
```

```
}
```

Haciendo uso de la sobrecarga de funciones en las clases incluimos 3 constructores para que se puedan crear objetos que se inicialicen tanto con valores por defecto como con parámetros o creando una copia de otro objeto.

Punto P1 , P2(2,3) , P3(P2);

**Un constructor DE COPIA siempre debe recibir una referencia a un objeto (para que no se llame nuevamente a un constructor de copia dentro de otro constructor de copia)**

# Desarrollo del cuerpo de las funciones en las clases

Las funciones que son parte de una clase, como los constructores, pueden desarrollarse dentro de la clase cómo lo venimos viendo o como buena práctica de programación comenzaremos a desarrollarlas en otro archivo por separado dejando en la definición de la clase solo su prototipo. Esto se hará por medio del operador de visibilidad.

```
class Punto{  
private:  
    int x, y;  
  
public:  
    void Set_x_y(int a, int b);  
  
}
```

**En un archivo .hpp cómo la  
definición de una estructura**



**En un archivo .cpp todos los cuerpos de las  
funciones de la clase**



```
void Punto::Setx_y(int a, int b){  
    x = a;  
    y = b;  
}
```



# Ejemplo Constructores/destructores

En la siguiente carpeta del campus virtual:

<https://www.campusvirtual.frba.utn.edu.ar/especialidad/mod/folder/view.php?id=74096>

Pueden acceder al ejemplo de constructores y destructores donde observar el momento de ejecución del constructor y destructor.



# Lista inicializadora

La lista inicializadora simplifica la creación de constructores. Cuando veamos el tema de herencia será fundamental para algunas inicializaciones.

```
class punto {  
    int x_  
    int y_  
public:  
    punto (int,int);  
    punto (const punto&);  
    void set_xy (int, int);  
}
```

## Definición de un constructor parametrizado tradicional

```
punto::punto (int a, int b)  
{  
    x_ = a;  
    y_ = b;  
}
```

*//Definición del constructor usando lista de inicializadores*

```
punto::punto (int a, int b) : x_(a),y_(b){ }
```

Lo que se recibe en la variable **a** se carga en **x\_** y lo que se recibe en la variable **b** se carga en **y\_**

# miembros públicos y privados

Como observamos en los ejemplos que venimos mostrando se usaron especificadores de acceso, los mismos se utilizan para indicar cómo se podrá acceder a los miembros de la clase desde afuera de la misma. desde que se pone el especificador de acceso todos los miembros que están por debajo de él tendrán ese tipo de accesibilidad.

```
class nombre_clase
{
    especificador_de_acceso_1:
        int miembro_1;
        int miembro_2;
        ...
    especificador_de_acceso_2:
        int miembro_n;
        ...
};
```

## Los posibles especificadores son:

- **public:** estos miembros son accesibles cualquier parte de nuestro programa
- **private:** su acceso es restringido. Private es el estatus por defecto de acceso a los miembros de una clase
- **protected:** está vinculado al concepto de “herencia”. (lo veremos en detalle mas adelante)

# miembros públicos y privados

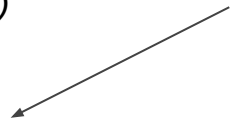
Dentro de la Programación Orientada a Objetos es fundamental poder proteger el acceso a los miembros de las clases que el exterior no debe modificar.

Fundamentalmente los **atributos** (variables) de las clases **SIEMPRE** serán **privados** en la clase, para que solo el programador de la misma, así solo los métodos (funciones) de la clase pueda acceder a los mismos.

```
class Punto{  
    private:  
        int x, y;  
  
    public:  
        set_x_y(int x, int y);  
}
```

```
int main(void)  
{  
    Punto P;  
  
    P.x = 5;  
  
    P.set_x_y(5,6)  
}
```

NO SE PUEDE x ES  
PRIVADA!



# buenas practicas de programacion - metodos set y get

Se recomienda en el diseño de las clases colocar las propiedades como privadas e incluir los siguientes métodos para acceso a ellas desde afuera de la clase:

- Una función **consultora** (get), es una función que retorna un valor desde su objeto, pero no cambia el objeto (sus atributos)
- Una función **modificadora** (set), es una función capaz de modificar el “estado” (sus atributos) de un objeto

```
class Punto
{
private:
    int x, y;

public:
    //Metodo de acceso
    void setX(double x);
    double getX() const;
    void setY(double y);
    double getY() const;
```

```
//Metodo para setear la parte x
void Punto::setX(double _x)
{
    x = _x;
}

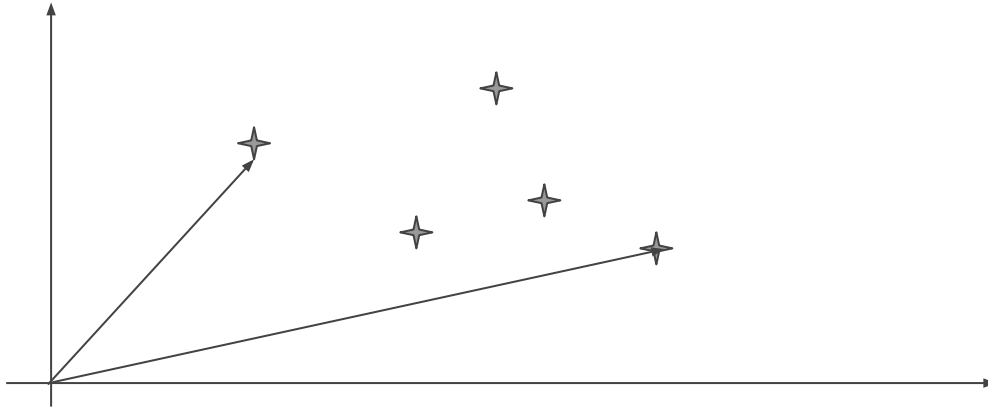
//Metodo para leer la parte x
double Punto::getX() const
{
    return x;
}

//Metodo para setear la parte yinaria
void Punto::setY(double _y)
{
    y = _y;
}

//Metodo para leer la parte yinaria
double Punto::getY() const
{
    return y;
}
```

# Pensamos el siguiente ejercicio en C++

1. Hacer una aplicación que pida  $N$  puntos en un plano e informe por pantalla el punto más cercano y más lejano del origen



(Para pensar este ejemplo hacemos  $N=5$ )

# ¿Cómo resolvemos en C++ el ejercicio?

1. Pensamos que clases necesitamos. Para este ejemplo Clase Punto.
2. Pensamos que atributos (variables) y métodos (funciones) debería de tener la clase.
3. Implementamos la clase.
4. Armamos el main resolviendo lo pedido con objetos de la clase creada.

