



# Introducción a c++ y la programación orientada a objetos

# Programación, diseño y complejidad.

- **¿cual es el objetivo del software, resolver un problema particular?**  
cálculo de problemas numéricos, mantenimiento de una base de datos organizada de información, búsqueda de nuevas partículas, análisis de imágenes...
- **En las ultimas décadas, el crecimiento del poder computacional nos ha permitido abordar problemas cada vez más complejos**
- **Como consecuencia, el software también se ha vuelto más poderoso y complejo.**  
Física no es la excepción: La colección de paquetes de software para la reconstrucción / análisis del experimento **BaBar** es ~ **6.4M líneas de C++**.  
“**CMSSW** is written in C++ and Python, and has several million lines of code”.
- **¿Cómo lidiamos con una complejidad tan creciente?**

# Filosofías de programación

- La **clave** para codificar con éxito sistemas complejos es descomponer el código en **módulos más pequeños** y **minimizar las dependencias** entre estos módulos.
- Los lenguajes de programación tradicionales (Fortran, Pascal, C) logran esto mediante los procedimientos **orientados**.

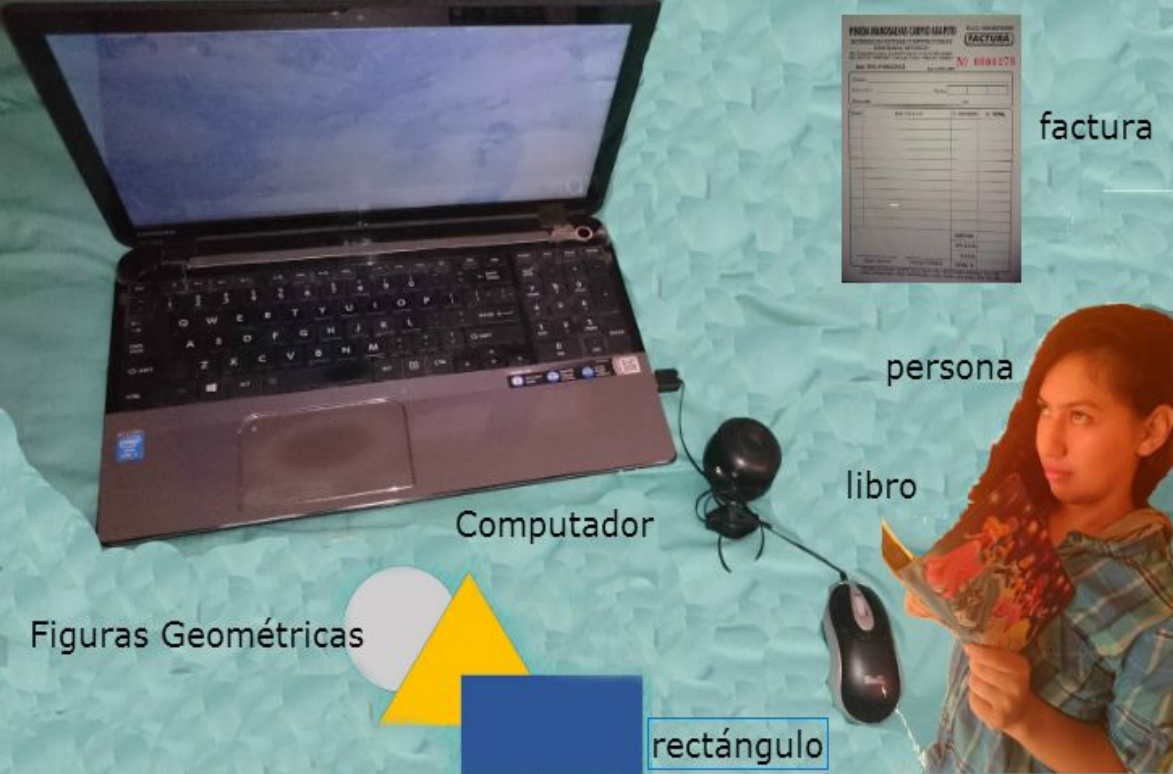
-La modularidad y la estructura del software giran en torno a algoritmos encapsulados en "**funciones**"

-Aunque las funciones son una herramienta importante en la estructuración de software, dejan algunos dolores de cabeza de diseño importantes

- Los lenguajes orientados a objetos (C++, Java, python ...) **llevan estos pasos más allá**

Agrupando datos y funciones asociadas en **objetos**.

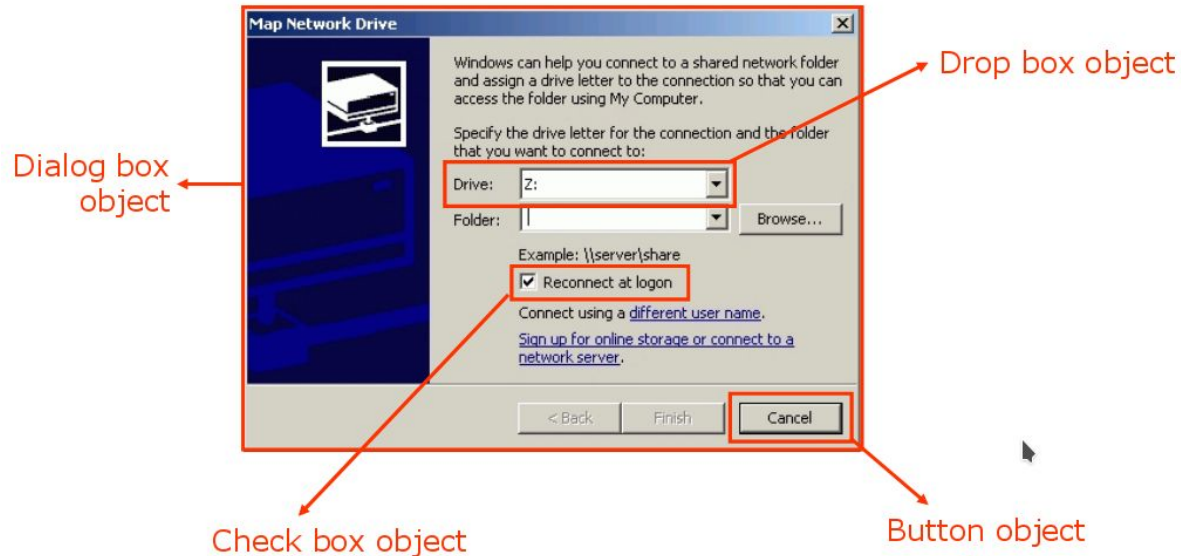




Identificar objetos concretos es relativamente sencillo, sin embargo identificar objetos abstractos requiere un poco de práctica e intuición como por ejemplo, transacción de una cuenta bancaria, detalle de una factura, que son objetos no tan evidentes, pero que son los que en la mayoría de casos serán los objetos que deberán ser implementados en programas, ya que interactuarán con los objetos más evidentes como la factura, o la cuenta bancaria en el caso de la transacción.

## ¿Qué son los objetos?

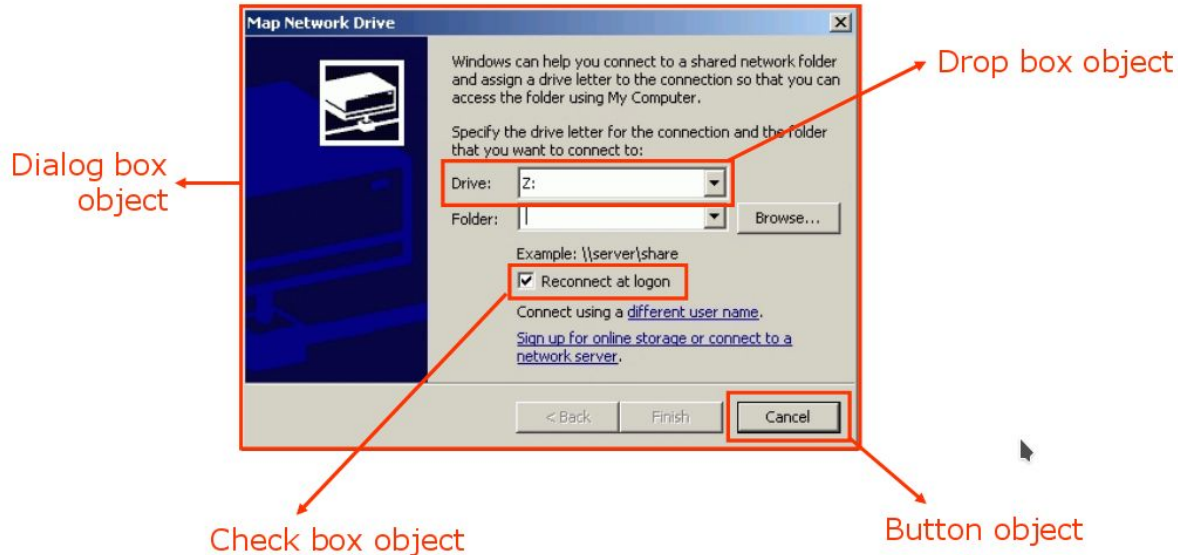
- **‘Software objects’** a menudo se encuentran de forma natural en problemas de la vida cotidiana.
- **Programación orientada a objetos (POO)** → Encontrar estos objetos y cual es su papel en su problema.





## ¿Qué son los objetos?

- Un objeto tiene:  
**Propiedades:** posición, forma, etiqueta de texto  
**Comportamiento:** si hace clic en el 'Cancel button', se produce una acción definida



El análisis y diseño orientado a objetos busca la relación entre objetos

- "Es-Un" Relación (un objeto "PushButton" es un objeto Clickable).
- 'Tiene-Un' Relación (a DialogBox Tiene Un CheckBox)

# Beneficios de la programación orientada a objetos

- Beneficios de la programación orientada a objetos
  - **Reutilización del código existente** - los objetos pueden representar problemas genéricos.
  - **Mantenimiento mejorado** - los objetos son más autónomos que las "subrutinas", por lo que el código está menos enredado.
  - **A menudo, una forma "natural" de describir un sistema** - podemos ver el ejemplo anterior del "Dialog box"
- Pero....
  - El modelado orientado a objetos no sustituye al pensamiento sólido.
  - La POO **no garantiza un alto rendimiento**, pero **tampoco se interpone en su camino**.
- **Sin embargo**
  - POO es actualmente la mejor forma en que sabemos describir sistemas complejos.

# Técnicas para lograr la abstracción

1. Modularidad 2. Encapsulación 3. Herencia 4. Polimorfismo.

1. Descomponga su problema de forma lógica en unidades independientes

- Minimizar dependencias entre unidades - Acoplamiento suelto
- Agrupar cosas que tengan una conexión lógica - Fuerte cohesión

```
long getBalance()  
void print()  
void calculateInterest()
```

```
char* ownersName  
long accountNumber  
long accountBalance
```

Account

2. Separar la interfaz y la implementación y proteja la implementación de los "usuarios" del objeto.

```
long getBalance()  
void print()  
void calculateInterest()
```

*interface*

```
char* ownersName  
long accountNumber  
long accountBalance
```

*implementation*  
(not visible from outside)

Account

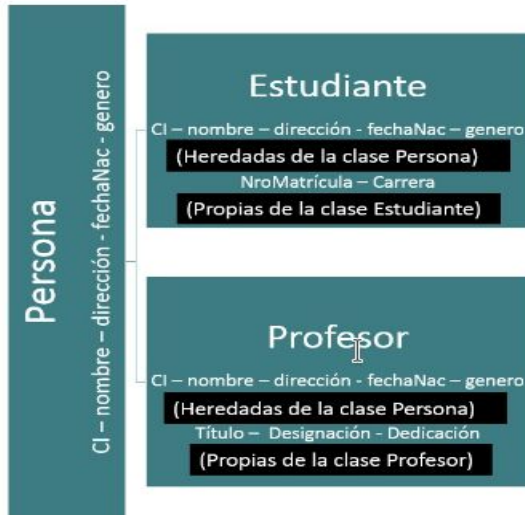


# Técnicas para lograr la abstracción

1. Modularidad 2. Encapsulación 3. Herencia 4. Polimorfismo.

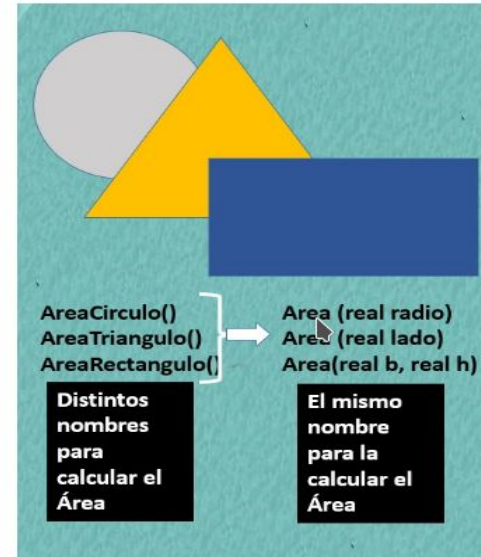
## 3. Herencia

Es el pilar más fuerte que asegura la reutilización de código, ya que a partir de esta característica es posible reutilizar (heredar) las características y comportamientos de una clase superior llamada clase padre, a sus clases hijas, denominadas clases derivadas.



## 4. Poliformismo

A través de esta característica es posible definir varios métodos o comportamientos de un objeto bajo un mismo nombre, de forma tal que es posible modificar los parámetros del método, o reescribir su funcionamiento, o incrementar más funcionalidades a un método.



# Introducción a C++

- Existe una amplia variedad de lenguajes POO: ¿por qué programar en C++?
  - Eso depende de para que lo necesitamos.
- Ventaja de C++ - **es un lenguaje compilado**
  - Cuando se usa correctamente, es el más rápido de todos los lenguajes POO
  - Debido a que las técnicas OO en C++ se resuelven e implementan en tiempo de compilación en lugar de tiempo de ejecución. Entonces:
    - **Maximiza el rendimiento en tiempo de ejecución**
    - **No pagas por lo que no usas**
- Desventaja de C++ - **sintaxis más compleja**
  - Además, darse cuenta de la ventaja de rendimiento no siempre es trivial.
- C++ se utiliza mejor para proyectos a gran escala donde el rendimiento es importante
  - C++ se convirtió rápidamente en el estándar en High Energy Physics para el procesamiento de datos convencional, la adquisición de datos en línea, etc.
  - **Sin embargo, si el código de su programa será O (100) líneas y el rendimiento no es crítico, C, Python, Java pueden ser más eficientes.**

# Bases de c++

Empezamos con un programa simple.

```
//Mi primer programa en C++  
#include<iostream>  
  
int main(){  
    std::cout<< "Bienvenidos a c++! \n";  
    return 0;  
}
```

# Bases de c++

Todas las líneas que comiencen con dos signos barra (//) se consideran comentarios y no tienen ningún efecto sobre el comportamiento del programa

Esta línea se corresponde con el comienzo de la definición de la función principal "**main**". La función principal es el punto por donde todos los programas inician su ejecución, independientemente de su ubicación dentro del código fuente.

```
//Mi primer programa en C++  
#include<iostream>  
  
int main(){  
    std::cout<< "Bienvenidos a c++! \n";  
    return 0;  
}
```

Las líneas que comienza con un símbolo de "#" son directivas para el preprocesador. Este archivo específico (**iostream**) incluye las declaraciones de la norma básica de entrada y salida de la biblioteca de C++.

Utilice los objetos de la biblioteca **iostream** para imprimir cadenas a la salida estándar. Los nombres **std :: cout** y **std :: endl** se declaran en el "archivo de encabezado".

Esta declaración hace que la función principal termine. Un código de retorno es 0, cuando la función principal interpreta de manera general que el programa trabajó como se esperaba, sin ningún error durante su ejecución.

# Variables y Tipos de datos

**Variables Locales:** Se definen solo en bloque en el que se vayan a ocupar, de esta manera evitamos tener variables definidas que luego no se utilizan.

**Variables Globales:** No son lo más recomendable, pues su existencia atenta contra la comprensión del código y su encapsulamiento.

**Variables estáticas:** Se tienen que inicializar en el momento en que se declaran, de manera obligatoria.

Los tipos de datos pueden ser predefinidos o abstractos. Un tipo de dato **predefinido** es intrínsecamente comprendido por el compilador. En contraste, un tipo de datos definido por el usuario es aquel que usted o cualquier otro programador crea como una clase, que comúnmente son llamados tipos de datos **abstractos**.

Los tipos de datos más comunes en C++ son:

TipodeDato	EspacioenMemoria	Rango
unsigned char	8 bits	0 a 255
char	8 bits	-128 a 127
short int	16 bits	-32,768 a 32,767
unsigned int	32 bits	0 a 4,294,967,295
int	32 bits	-2,147,483,648 a 2,147,483,647
unsigned long	32 bits	0 a 4,294,967,295
enum	16 bits	-2,147,483,648 a 2,147,483,647
long	32 bits	-2,147,483,648 a 2,147,483,647
float	32 bits	3.4 x 10 <sup>-38</sup> a 3.4 x 10 <sup>+38</sup> (6 dec)
double	64 bits	1.7 x 10 <sup>-308</sup> a 1.7*10 <sup>+308</sup> (15 dec)
long double	80 bits	3.4 x 10 <sup>-4932</sup> a 1.1 x 10 <sup>+4932</sup>
void	sin valor	

Decimal	Exponencial	cientifica
1625.0	1.625e3	1.625x10 <sup>3</sup>
0.00731	7.31e-3	7.31x10 <sup>-3</sup>

# Definición de objetos de datos - variables

**La definición de una variable se puede hacer de varias maneras.**

```
int main() {  
    int j ; // definición - valor inicial indefinido  
    int k = 0 ; // definición con valor inicial  
    int l(0) ; // definición con inicialización de constructor  
  
    int m = k + 1 ; // el inicializador puede ser cualquier expresión C++ válida  
  
    int a,b=0,c(b+5); // declaración múltiple  
  
    return 0;  
}
```

```
int main() {  
    const float pi = 3.14159268 ; // objeto de datos constantes  
    pi = 2 ; // ERROR – no se compila  
}
```



# Constantes en C++, const y #define

```
#include <iostream>
using namespace std;

#define PI 3.1416; //Definimos una constante llamada PI

int main()
{
    std::cout << "Mostrando el valor de PI: " << PI << std::endl;

    return 0;
}
```

## Error

Si intentamos ejecutar el código anterior obtendremos un error al haber usado el operador << justo después de PI, esto sucede porque PI no es tratado exactamente como una variable cualquiera sino como una expresión, así que realmente aunque podemos usar #define para declarar constantes no es la mejor opción.

bastante fácil y mejor aún ha sido mucho más intuitivo y sencillo. Se puede ver que la declaración es muy similar a la de una variable cualquiera y que ya no tenemos complicaciones al intentar añadir la instrucción *endl* para agregar el salto de línea

```
#include <iostream>

int main()
{
    const float PI = 3.1416;
    std::cout << "Mostrando el valor de PI: " << PI << std::endl;

    return 0;
}
```

# Aritmetica

**Operador modulo:  $7\%4=3$ ,  $17\%5 = 2$**

(muchas aplicaciones, entre otras saber si un numero es par o no)

## Reglas de precedencia


1. primero los parentesis
2.  $*$ ,  $/$ ,  $\%$  se aplican despues (izquierda derecha)
3. Suma y Resta a lo ultimo (izquierda derecha)

Algebra: 
$$m = \frac{a + b + c + d + e}{5}$$


C++: 
$$m = ( a + b + c + d + e ) / 5;$$

Algebra: 
$$z = pr \% q + w/x - y$$

C++: 
$$z = p * r \% q + w / x - y;$$



The diagram shows the sequence of operations for the C++ expression  $z = p * r \% q + w / x - y;$  using numbered circles 1 to 6. The sequence is: 1 (multiplication  $p * r$ ), 2 (modulo  $(p * r) \% q$ ), 4 (division  $w / x$ ), 3 (addition  $(p * r \% q) + (w / x)$ ), 5 (subtraction  $((p * r \% q) + (w / x)) - y$ ), and 6 (assignment  $z = \dots$ ).

$$y = a * x * x + b * x + c;$$


The diagram shows the sequence of operations for the C++ expression  $y = a * x * x + b * x + c;$  using numbered circles 1 to 6. The sequence is: 1 (multiplication  $a * x$ ), 2 (multiplication  $(a * x) * x$ ), 4 (multiplication  $b * x$ ), 3 (addition  $((a * x) * x) + (b * x)$ ), 5 (addition  $((a * x) * x) + (b * x) + c$ ), and 6 (assignment  $y = \dots$ ).

Operación en C++	Operador aritmético de C++	Expresión algebraica	Expresión en C++
Suma	+	$f + 7$	$f + 7$
Resta	-	$p - c$	$p - c$
Multiplicación	*	$bm$ o $b \cdot m$	$b * m$
División	/	$x/y$ o $\frac{x}{y}$ o $x \div y$	$x / y$
Residuo	%	$r \bmod s$	$r \% s$

# “using namespace std;”

El lenguaje C++ consta de un reducido número de instrucciones, pero ofrece un amplio repertorio de bibliotecas con herramientas que pueden ser importadas por los programas cuando son necesarias. Por este motivo, un programa suele comenzar por tantas líneas **#include** como bibliotecas se necesiten. Como se puede observar, en nuestro ejemplo se incluye la biblioteca **iostream**, necesaria cuando se van a efectuar operaciones de entrada (lectura de datos) o salida (escritura de datos). Para utilizar la biblioteca **iostream** es necesario utilizar el **espacio de nombres std**, éste es un concepto que estudiaremos luego. Por ahora nos basta con recordar que nuestros programas pueden contener algunas de las siguiente directivas:

```
using namespace std::cout;  
using namespace std::cin;  
using namespace std::endl;
```

E2. Código: Suma de enteros

o, simplemente

```
using namespace std;
```

# Operadores

Operador	Tipo de Operador	Asociatividad		
[] -> .	Binarios	Izq. a Dch.	valor * valor	Producto
! ~ - *	Unarios	Dch. a Izq.	valor / valor	División
* / %	Binarios	Izq. a Dch.	valor % valor	Módulo
+ -	Binarios	Izq. a Dch.	valor + valor	Suma
<< >>	Binarios	Izq. a Dch.	valor - valor	Resta
< <= > >=	Binarios	Izq. a Dch.		
== !=	Binarios	Izq. a Dch.	valor < valor	Comparación menor
&	Binario	Izq. a Dch.	valor <= valor	Comparación menor o igual
^	Binario	Izq. a Dch.	valor > valor	Comparación mayor
	Binario	Izq. a Dch.	valor >= valor	Comparación mayor o igual
&&	Binario	Izq. a Dch.	valor == valor	Comparación de igualdad
	Binario	Izq. a Dch.	valor != valor	Comparación de desigualdad
?:	Ternario	Dch. a Izq.		

# Cin y asignacion

```
#include<iostream>

using namespace std;

int main(){
int number1; // primer entero
int number2; // segundo entero

cout << "entre dos numeros enteros: ";
cin >> number1 >> number2;

if ( number1 == number2 )
    cout << number1 << " == " << number2 << endl;

if ( number1 != number2 )
    cout << number1 << " != " << number2 << endl;

return 0;
}
```

```
#include<iostream>

using namespace std;

const int MAXIMO = 15 ;

int main(){

int cnt ;

cnt = 30 * MAXIMO + 1 ; // asigna a cnt el valor 451
cnt = cnt + 10 ; // asigna a cnt el valor 461

    return 0;
}
```

# Acumulacion

Sentencia	Equivalencia
<code>++variable;</code>	<code>variable = variable + 1;</code>
<code>--variable;</code>	<code>variable = variable - 1;</code>
<code>variable++;</code>	<code>variable = variable + 1;</code>
<code>variable--;</code>	<code>variable = variable - 1;</code>
<code>variable += expresion;</code>	<code>variable = variable + (expresion);</code>
<code>variable -= expresion;</code>	<code>variable = variable - (expresion);</code>
<code>variable *= expresion;</code>	<code>variable = variable * (expresion);</code>
<code>variable /= expresion;</code>	<code>variable = variable / (expresion);</code>
<code>variable %= expresion;</code>	<code>variable = variable % (expresion);</code>
<code>variable &amp;= expresion;</code>	<code>variable = variable &amp; (expresion);</code>
<code>variable ^= expresion;</code>	<code>variable = variable ^ (expresion);</code>
<code>variable  = expresion;</code>	<code>variable = variable   (expresion);</code>
<code>variable &lt;&lt;= expresion;</code>	<code>variable = variable &lt;&lt; (expresion);</code>
<code>variable &gt;&gt;= expresion;</code>	<code>variable = variable &gt;&gt; (expresion);</code>

```
int sum = 0;  
sum = sum +1;  
sum = sum +95;
```





# Conteo

**variable = variable + numero fijo;**

`i = i + 1; n = n + 2; m = m + 22;`

## El caso especial

**`i = i + 1; => i++ o ++i`**

```
int main(){
    int ii = 0;

    cout << "contador = " << ii << endl;
    i++;
    cout << "contador = " << ii << endl;
}
```

**`k=++n; => n=n+1; k=n;`**

**`k=n++; => k=n; n=n+1`**

# Coerción

el valor de la expresión en el lado derecho del operador de asignación será convertido en el tipo de datos de la variable a la izquierda del operador de asignación.

`int a = 25.9; // realmente en a se almacena 25`

`int b; float c;`  
`int d= b*c;`

en el momento del calculo “b” y “c” son **double**, pero mantendrán su valor asignado (**int** y **float**) despues de eso. Ademas, aunque “b\*c” es **double** “a” sera **int**.

# Formato

```
#include <iostream>
using namespace std;

int main()
{
    cout << 6 << endl
         << 18 << endl
         << 124 << endl
         << "---\n"
         << (6+18+124) << endl;

    return 0;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << setw(3) << 6 << endl
         << setw(3) << 18 << endl
         << setw(3) << 124 << endl
         << "---\n"
         << (6+18+124) << endl;

    return 0;
}
```

```
cout << "|" << setw(10) << fixed << setprecision(3) << 25.67 << "|";
```

**Tarea: Revisar con detalle la seccion 3.2 del texto guia**

# FUNCIONES MATEMÁTICAS

Nombre de la función	Descripción	Valor devuelto
<code>abs(a)</code>	valor absoluto	mismo tipo de datos que el argumento
<code>pow(a1, a2)</code>	a1 elevado a la potencia a2	tipo de datos del argumento a1
<code>sqrt(a)</code>	raíz cuadrada de un número real	precisión doble
<code>sin(a)</code>	seno de a (a en radianes)	doble
<code>cos(a)</code>	coseno de a (d en radianes)	doble
<code>tan(a)</code>	tangente de a (d en radianes)	doble
<code>log(a)</code>	logaritmo natural de a	doble
<code>log10(a)</code>	logaritmo común (base 10) de a	doble
<code>exp(a)</code>	e elevado a la potencia a	doble

```
4*sqrt(3.8*10.9-8.2)-7.6
```

```
sqrt(cos(abs(θ)))
```

```
#include<iostream>
#include<iomanip>
#include<cmath>
```

## E3. Código: matfuncion

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
//define PI 3.1416
int main()
{
```

```
    const float PI = 3.1416;
    float ag, ar, s,c,t;
    cout<< "Ingrese el angulo en grados"<< endl;
    cin>>ag;
    ar = ag * PI /180;
    s = sin(ar);
    c = cos(ar);
    t = tan(ar);
    cout<<"El seno es "<<s<<endl;
    cout<<"El coseno es "<<c<<endl;
    cout<<"La tangente es "<<t<<endl;
```

```
    cout<<" "<<endl;
    cout<<"funciones log and exp "<<endl;
    cout<<" "<<endl;
```

```
    cout<<"exp(1.0) "<<exp(1.0)<<endl;
    cout<<"log(10.0) "<<log(10.0)<<endl;
    cout<<"log(exp(1.0)) "<<log(exp(1.0))<<endl;
    cout<<"exp(2.30259) "<<exp(2.30259)<<endl;
```

```
    //return 0;
}
```

# Introducción a las clases

```
class nombreClase
{
    public:
        lista_de_funciones_miembro // pueden ser los prototipos o implementación de la función

    private:
        lista_datos_miembro; // son las variables donde defines el tipo de dato nombre variable y ;
};
```

Vamos a empezar con un ejemplo que consiste en la clase “LibroCalificar” , la cual representa un libro de calificaciones que un profesor puede utilizar para mantener las calificaciones de los exámenes de sus estudiantes.

Primero vamos a describir cómo definir una clase y una **función miembro**. Después explicaremos cómo se **crea un objeto**, y cómo llamar a una función miembro de éste.

# Introducción a las clases

```
#include <iostream>
using namespace std;
```

```
class LibroCalificar
```

```
{
public:
    void displayMessage()
    {
        cout << "Bienvenido al libro de calificaciones!" << endl;
    }
};
```

```
int main()
```

```
{
    LibroCalificar myLibroCalificar;
    myLibroCalificar.displayMessage();

    return 0;
}
```

**define la clase.**

convención: letra mayúscula  
inicio del nombre

etiqueta del  
especificador de acceso  
**public:**

**función miembro.**  
tenga cuidado es tipo  
“**void**”. tipo de valor  
de retorno.

**NO olvide este “;”**

**crea un objeto de la clase**

llamada a la función miembro.  
**operador punto “.”**

# Introducción a las clases

## E. Codigo Cursoname03\_03

#include string

getline, ¿porque no “**cin**”? (**LINEA FANTASMA**)

Cuando se usa “cin”, este lee hasta el primer espacio en blanco.

Como buena práctica de programación se acostumbra no usar los mismos nombres del parámetro que se pasan a la funcion y en la definicion de la funcion.

## E. Codigo Cursoname03\_05

### Miembros de datos, funciones establecer y funciones obtener:

una clase consiste en una o más funciones miembro que manipulan los atributos pertenecientes a un objeto específico de la clase.

### Public y Private:

Como regla empírica, los miembros de datos deben declararse como **private** y las funciones miembro deben declararse como **public**.



# Introducción a las clases

E. Codigo Cursoname03\_07

## Constructores

Una importante diferencia entre los constructores y las otras funciones es que los primeros no pueden devolver valores, por lo cual **no pueden especificar un tipo de valor de retorno** (ni siquiera void ).

Si una clase no incluye un constructor en forma explícita, el compilador proporciona **un constructor predeterminado**, es decir, un constructor sin parámetros.

para los miembros de datos que son objetos de otras clases, el constructor llama de manera implícita al constructor predeterminado de cada miembro de datos, para asegurar que ese miembro de datos se inicialice en forma apropiada

E. Codigo Cursoname03\_10

**archivo de encabezado #include "name.h"**  
note las comillas ""

**¿El código cliente necesita saber cómo se implementan las funciones de la clase?**



# Introducción a las clases

## Archivo de código fuente

### E. Codigo Cursoname03\_13

#### Prototipos de funciones

nombre de la función, su tipo de valor de retorno y los tipos de sus parámetros.

```
class LibroCalificar
{
public:
    LibroCalificar( string );
    void setCourseName( string );
    string getCourseName();
    void displayMessage();
private:
    string courseName;
};
```

los nombres de los parámetros  
son opcionales en los prototipos

```
#include "Cursoname.h"

LibroCalificar::LibroCalificar( string name )
{
    setCourseName( name );
}

void LibroCalificar::setCourseName( string name )
{
    courseName = name;
}

string LibroCalificar::getCourseName()
{
    return courseName;
}

void LibroCalificar::displayMessage()
{
    cout << "Bienvenido al libro de calificaciones para\n" <<
    getCourseName()
        << "!" << endl;
}
```

“**LibroCalificar::**” → “enlaza” a cada función con la clase  
“**::**” operador binario de resolución de ámbito (o alcance)