



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

Física Computacional II  
Jhovanny Andres Mejia Guisao  
UNIVERSIDAD DE ANTIOQUIA, COLOMBIA

# 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 últimas 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**.

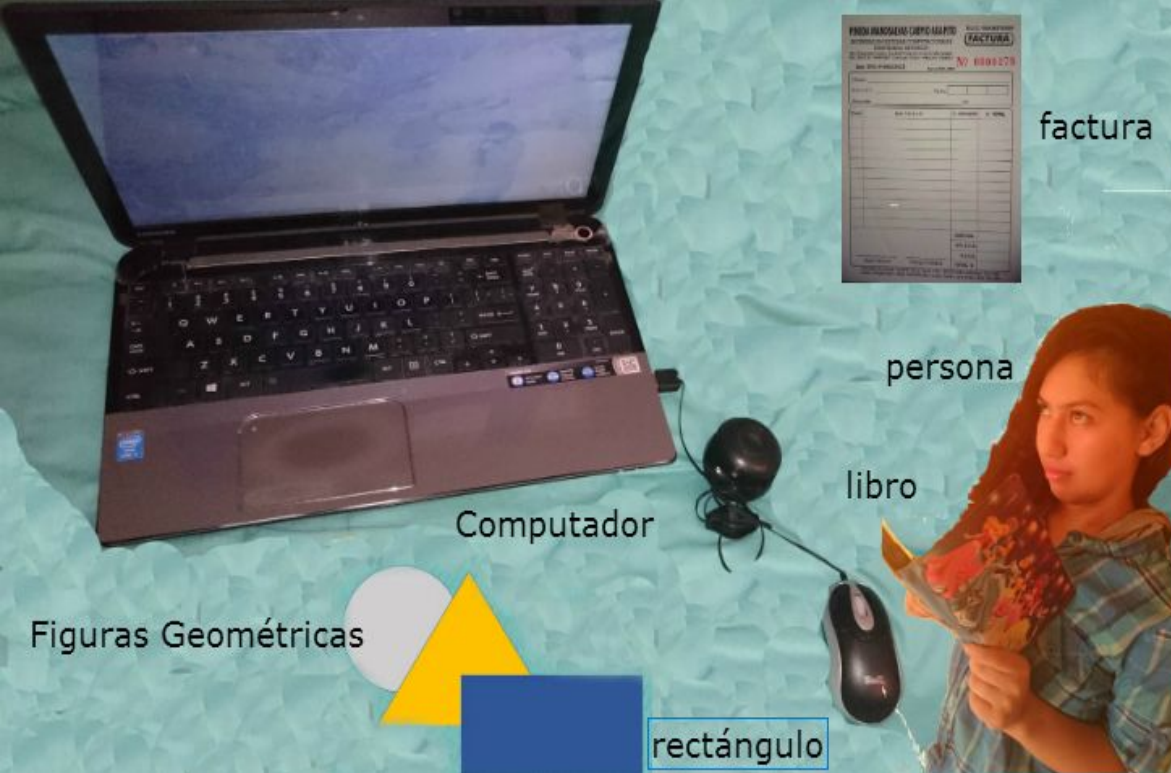
-La modularidad y la estructura del software giran en torno a algoritmos encapsulados en "**funciones**"

-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**.

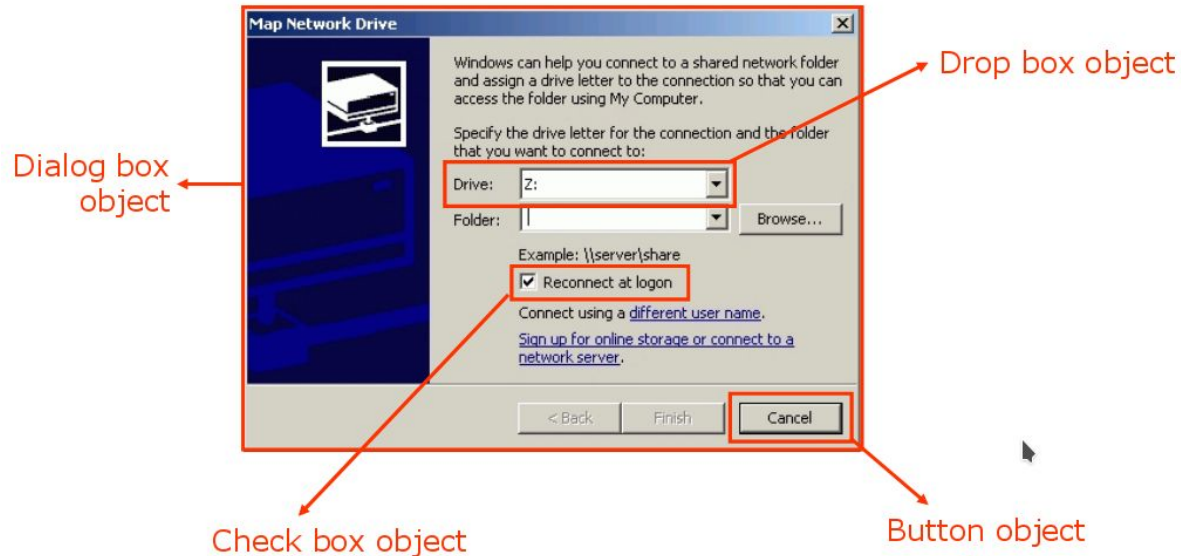




Identificar objetos concretos es relativamente sencillo, sin embargo identificar objetos abstractos requiere un poco de práctica e intuición como por ejemplo, transacción de una cuenta bancaria, detalle de una factura, que son objetos no tan evidentes, pero que son los que en la mayoría de casos serán 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?

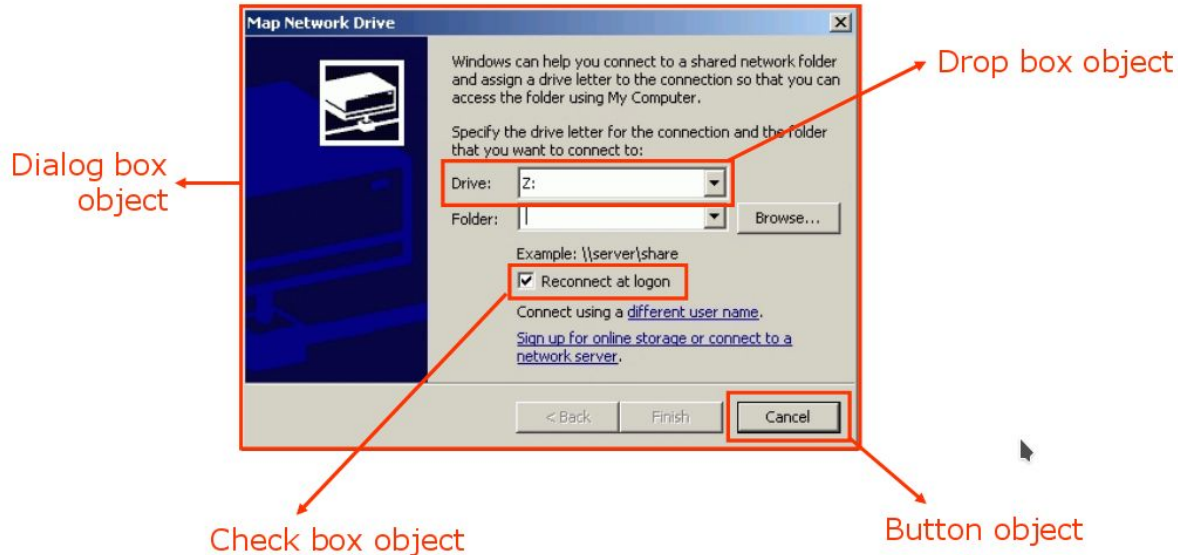
- **‘Software objects’** a menudo se encuentran de forma natural en problemas de la vida cotidiana.
- **Programación orientada a objetos (POO)** → Encontrar estos objetos y cual es su papel en su problema.





## ¿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 (un 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

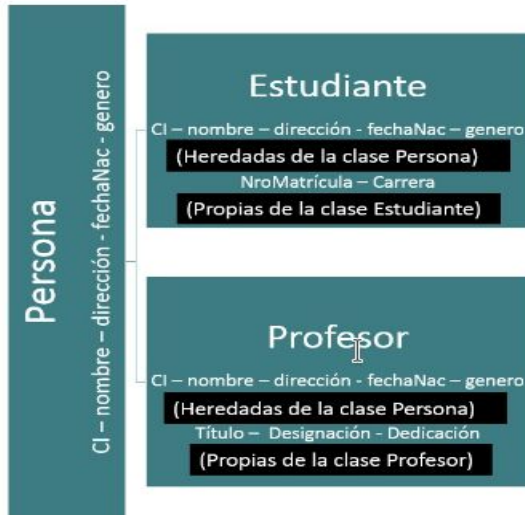


# Técnicas para lograr la abstracción

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

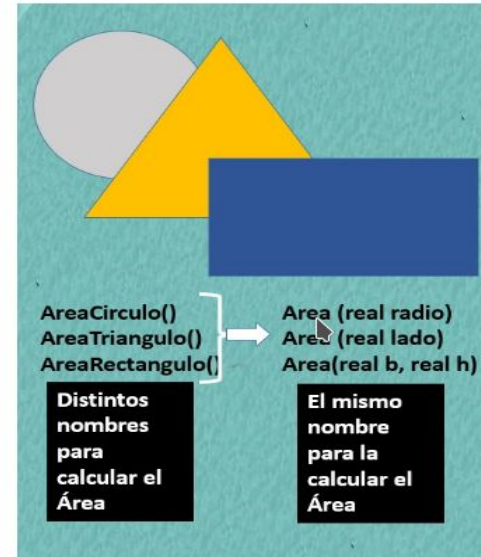
## 3. Herencia

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:

<i>TipodeDato</i>	<i>EspacioenMemoria</i>	<i>Rango</i>
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	$3.4 \times 10^{-38}$ a $3.4 \times 10^{+38}$ (6 dec)
double	64 bits	$1.7 \times 10^{-308}$ a $1.7 \times 10^{+308}$ (15 dec)
long double	80 bits	$3.4 \times 10^{-4932}$ a $1.1 \times 10^{+4932}$
void	sin valor	

Decimal	Exponencial	cientifica
1625.0	1.625e3	$1.625 \times 10^3$
0.00731	7.31e-3	$7.31 \times 10^{-3}$

# 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 L{0} ; // List initialization (introducido en C++11)
```

```
    int m = k + l ; // el inicializador puede ser cualquier expresión C++ válida
```

```
    int a,b=0,c(b+5); // declaración múltiple
```

```
    return 0;  
}
```

```
g++ -std=c++11 sumaenteros.C -o myco  
g++ -std=c++14 sumaenteros.C -o myco  
g++ sumaenteros.C
```

```
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 número es par o no)

## Reglas de precedencia

1. primero los parentesis
2.  $*$ ,  $/$ ,  $\%$  se aplican después (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++: 
$$z = p * r \% q + w / x - y;$$

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

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

# Cin y asignación

```
#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;
}
```

E3. Código: Compare enteros  
E3.1. compare2.cpp

# 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;
    ii++;
    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** “d” 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>
```

## E4. 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;
}

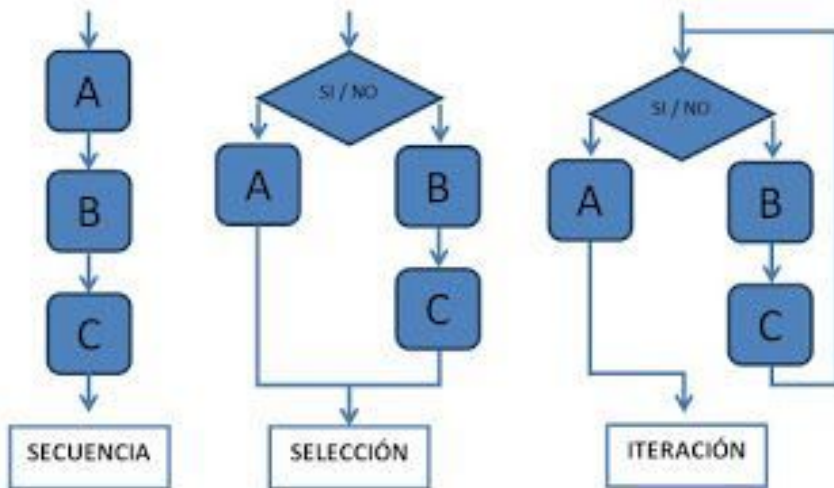
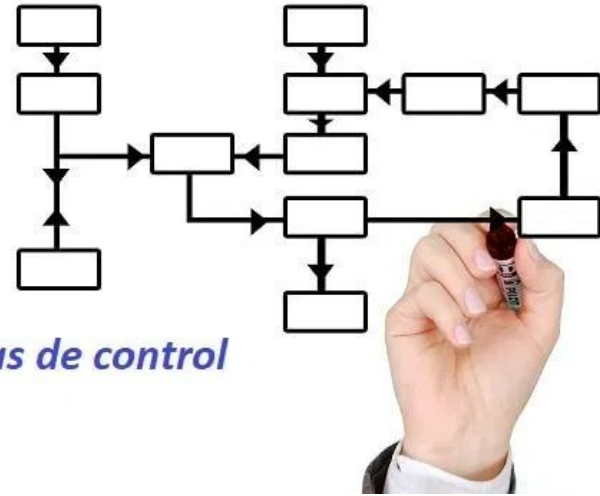
void LibroCalificar::displayMessage()
{
    cout << "Bienvenido al libro de calificaciones para\n" <<
    getCourseName()
        << "!" << endl;
}
```

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

# Estructuras de control

- Estructuras de secuencia en cpp
- Instrucciones de selección en cpp
- Instrucciones de repetición en cpp

*Estructuras de control  
en C++*



## Ésta es la esencia de la simpleza

Cualquier programa de C++ que se desee crear puede construirse a partir de sólo siete tipos distintos de instrucciones de control (secuencia, if, if...else, switch, while, do...while y for), combinadas en sólo dos formas (apilamiento de instrucciones de control y anidamiento de instrucciones de control).

# Criterios de selección. if-else

**Si** la calificación del estudiante es mayor o igual a 3.0  
Imprimir "Aprobado"

**De lo contrario**  
Imprimir "Rajado"

Si la condición produce cualquier valor numérico positivo o negativo diferente de cero, la condición es considerada como una condición "**verdadera**"

Condicion

```
if(calificacion>=3)
cout << "Aprobado";
else
cout << "Rajado";
```

instrucción ejecutada si la condición es verdadera

instrucción ejecutada si la condición es falsa;

Opcional

```
cout << ( calificacionEstudiante >= 3 ? "Aprobado" : "Reprobado" );
```

# Un paréntesis útil (1)

Expresión	Valor	Interpretación
'A' > 'C'	0	falso
'D' <= 'Z'	1	verdadero
'E' == 'F'	0	falso
'g' >= 'm'	0	falso
'b' != 'c'	1	verdadero
'a' == 'A'	0	falso
'B' < 'a'	1	verdadero
'b' > 'Z'	1	verdadero

“‘Hola” > ”hola” (Falsa??)

“Bejuco” > ”Beato” (verdadero??)

“Planta” > “Planeta” (verdadera??)

```
cout << "El valor de 3 < 4 es " << (3 < 4) << endl;  
cout << "El valor de 2.0 > 3.0 es " << (2.0 > 3.3) << endl;  
cout << "El valor de verdadero es " << verdadero << endl;  
cout << "El valor de falso es " << falso << endl;
```

# Un paréntesis util (2)

a = 12.0, b = 2.0, i = 15, j = 30 y completo = 0.0

Expresión	Valor	Interpretación
a > b	1	completo
(i == j)    (a < b)    completo	0	falso
(a/b > 5) && (i <= 20)	1	verdadero

**Operadores relacionales:**

>, <, =, >=, !=

**Operadores logicos:**

&&, ||, !

**mayor precedencia** de los operadores relacionales en relación con los operadores lógicos

**Un problema de exactitud numérica**

tenga cuidado con los “números” float y double

**(a==b)?**



**abs(a-b)<0.0000?**



## Palabras clave de C++

### *Palabras clave comunes para los lenguajes de programación C y C++*

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

### *Palabras clave sólo de C++*

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

# Criterios de selección. if-else

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

```
int main(){
    int n1, porc;
```

```
    cout<< "Programa para determinar naturaleza par o impar de un numero"<< endl;
    cout<<"Introduzca un numero entero: "<<endl;
```

```
    cin>> n1;
```

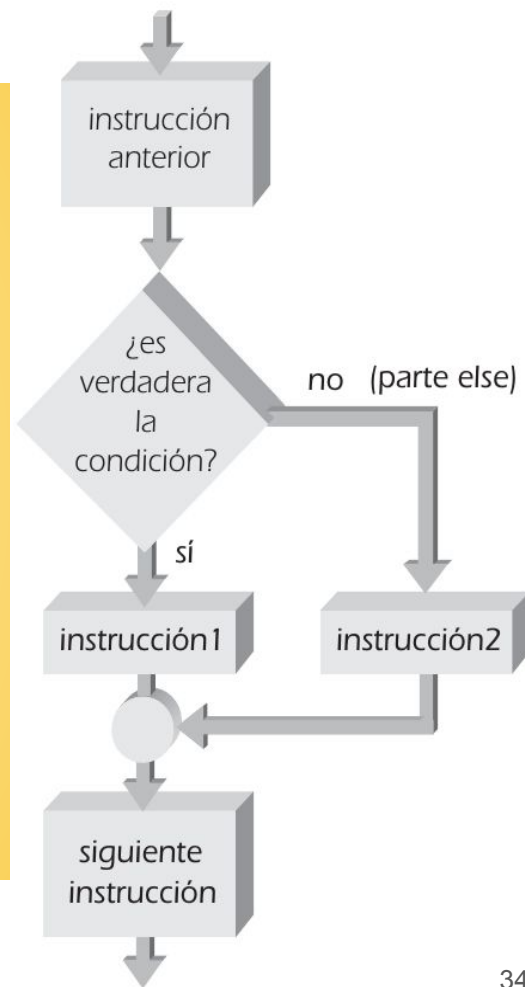
```
    porc= n1%2;
```

```
    if(porc==0)
        cout<< "El numero es par\n";
    else
        cout<< "El numero es impar\n";
```

```
    return 0;
}
```

E. Código par.cpp (compuestos y anidados)  
¿¿(if m=0){cout<<.....}?? tenga cuidado con el “==”

“Alcance de un bloque”



# Criterios de selección. La instrucción switch

```
switch(opción) //donde opción es la variable a comparar
{
case valor1:
    Bloque de instrucciones 1;
    break;
case valor2:
    Bloque de instrucciones 2;
    break;
case valor3:
    Bloque de instrucciones 3;
    break;
//Nótese que valor 1 2 y 3 son los valores que puede tomar la opción
//la instrucción break es necesaria, para no ejecutar todos los casos.
default:
    Bloque de instrucciones por defecto;
//default, es el bloque que se ejecuta en caso de que no se de ningún caso
}
```

**note:**

**case, break, default, switch**

**case:** punto de partida

**break:** punto de terminacion

**default:** caso opcional

# Instrucción de repetición while

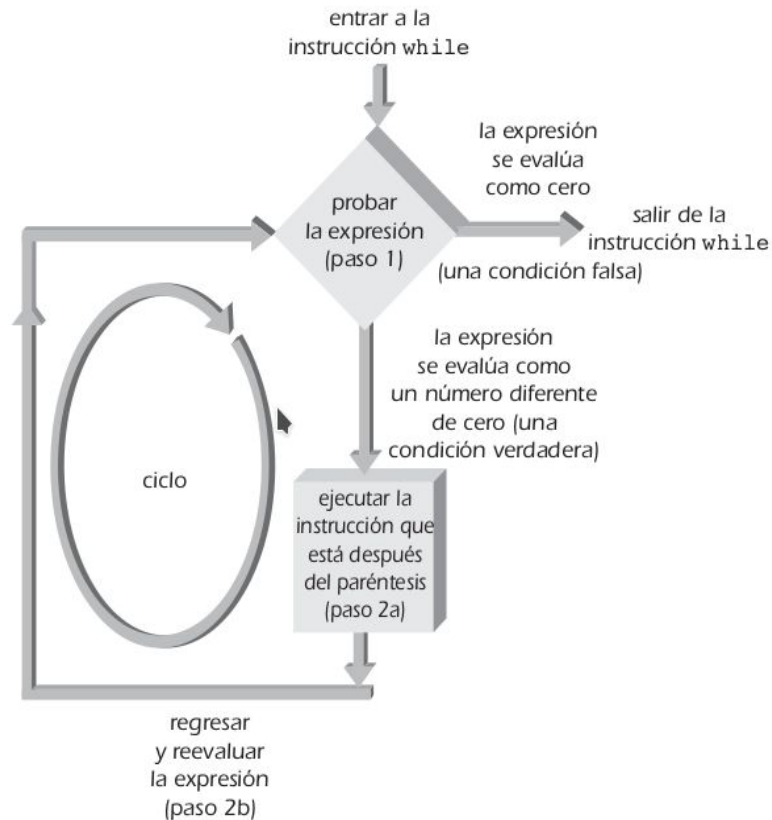
```
while(condición)
{
grupo cierto de instrucciones;
instrucción(es) para salir del ciclo;
}
```

```
#include <iostream>
using namespace std;
int main(){

    int producto = 3;
    while ( producto <= 100 ){
        producto = 3 * producto;
        cout << " el valor de producto es " << producto << endl;
    }

    return 0;
}
```

E. Código testwhile



Tenga en cuenta que tenemos Ciclos de cuenta variable y **Ciclos de cuenta fija.**

# Repetición controlada por un centinela

```
int main(){  
  
    int n1, val;  
    char ch;  
  
    while (val != 1)  
    {  
        cout<< "entre un letra: ";  
        cin >> ch;  
  
        cout<< "entre un numero: ";  
        cin >> n1;  
  
        cout<< "¿desea entra mas letras y numeros? (NO = 1, SI = 2)";  
        cin >> val;  
  
        cout << " su letra y numero son: "<< ch << " y " << n1 << endl;  
    }  
  
    return 0;  
}
```

tenga cuidado con el tipo de datos

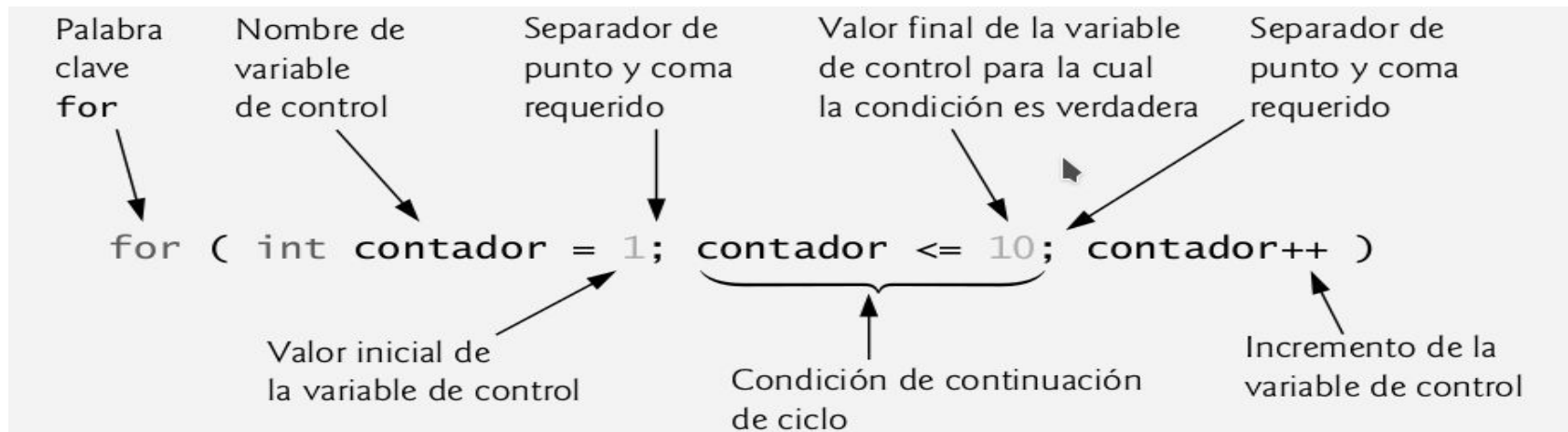
nota bene.

tenga cuidado con las  
expresiones  
“break” y “continue”

E. Codigo example{1,2}\_promediocali

**Task 1: Exercises 4.5, point8  
(Heat transfer).**

# Instrucción de repetición for



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

```
int main(){
```

```
    for ( int j = 1; j <= 10; ++ ){
        cout << j << endl;
    }
```

```
    return 0;
}
```

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

```
for ( int i = 100; i >= 1; i-- ) // 100 a 1 incrementos de -1
```

```
for ( int i = 7; i <= 77; i += 7 ) // de 7 a 77 en incrementos de 7
```

```
for ( int i = 99; i >= 0; i -= 11 )// ¿ que hace esto?
```

# Ciclos anidados

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

int main()
{
    int i=1,j;

    for(i; i <= 5;i++)
    {
        j=1;
        for (j; j <= i; j++ )
        {
            cout <<j;
        }
        cout << endl;
    }

    return 0;
}
```

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

int main()
{
    int i=1,j;
    while (i <= 5)
    {
        j=1;
        while (j <= i )
        {
            cout <<j;
            j++;
        }
        cout << endl;
        i++;
    }

    return 0;
}
```

E. example1\_switch

# Ciclo do\_while

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


int main()
{
    int contador = 1;

    do
    {
        cout << contador << " ";
        contador++;
    } while ( contador <= 10 );

    cout << endl;

    return 0;
}
```

do  
    *instrucción*  
while ( *condición* );



# Generación de números aleatorios

**rand()**

produce una serie de números aleatorios

**srand():**

proporciona un valor “semilla” inicial para rand()

**Números aleatorios entre 0 y 50**

`rand()%51`

**En general:**

`num = Lim_inf + rand()%(Lim_sup+1-Lim_inf)`

**`srand(time(NULL))`**

hace que los aleatorios sean siempre diferentes

**Note:**

**`double(rand())/RAND_MAX`**

aleatorios de precisión double entre 0.0 y 1.0

E. Código borracho

```
#include <stdlib.h>
#include <time.h>
#include <iostream>

using namespace std;

int main()
{
    srand(time(NULL));

    for (int j=0; j<=10; j++){

        cout << rand()%51<< endl;// 0-50

        //cout << 1+rand()%(101-1)<< endl;//1-100

        //cout << 250+rand()%(421-250)<< endl;//250-420

    }

    return 0;
}
```



```
#include <stdlib.h>
#include <time.h>
#include <iostream>
```

```
using namespace std;
```

```
int main()
{
```

```
    srand(time(NULL));
```

```
    for ( int contador = 1; contador <= 20; contador++ )
    {
        cout << setw( 10 ) << ( 1 + rand() % 6 );

        if ( contador % 5 == 0 )
            cout << endl;
    }
```

```
    return 0;
}
```

# son números aleatorios?

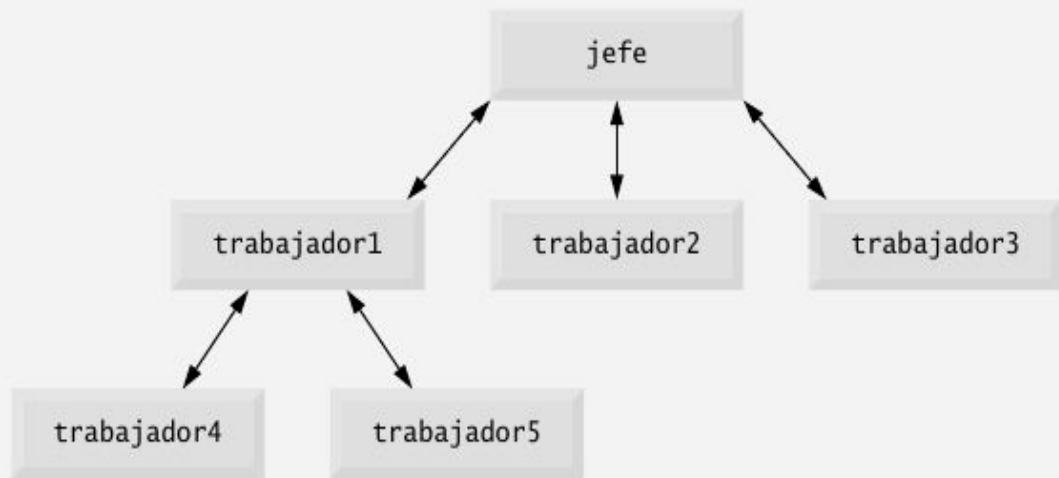
¿los números que produce la función rand() ocurren con una probabilidad aproximadamente igual?

si simulamos 6,000,000 de tiros de un dado, ¿cuántas veces debería aparecer cada entero de 1 a 6?



¿Jugamos “craps”?

# Continuamos con las funciones



## divide y vencerás

Para promover la reutilización de software, toda función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar esa tarea con efectividad. Dichas funciones facilitan la escritura, prueba, depuración y mantenimiento de los programas.

Una pequeña función que realiza una tarea es más fácil de probar y depurar que una función más grande que realiza muchas tareas.

Si no puede elegir un nombre conciso que exprese la tarea de una función, tal vez ésta esté tratando de realizar demasiadas tareas diversas. Por lo general, es mejor descomponer dicha función en varias funciones más pequeñas.

E. example1

# Continuamos con las funciones

```
#include<iostream>

using namespace std;

//int VolumenCaja(int , int , int );
int VolumenCaja(int = 1, int = 1, int =1 );

int main()
{

    cout << "el volumen predeterminado es " << VolumenCaja()<<
endl;
    cout << endl;

    cout << "el volumen que le damos es " << VolumenCaja(2,2,2)<<
endl;

    return 0;
}

int VolumenCaja(int L, int A , int P ){

    return L*A*P;

}
```

## Reglas Argumentos por omisión

**los valores por omisión deberían asignarse en el prototipo de función.**

si a cualquier parámetro se le da un valor por omisión en el prototipo de función, a todos los parámetros que siguen también deben asignarles valores por omisión.

si un argumento se omite en la llamada a la función real, entonces todos los argumentos a su derecha también deben omitirse.

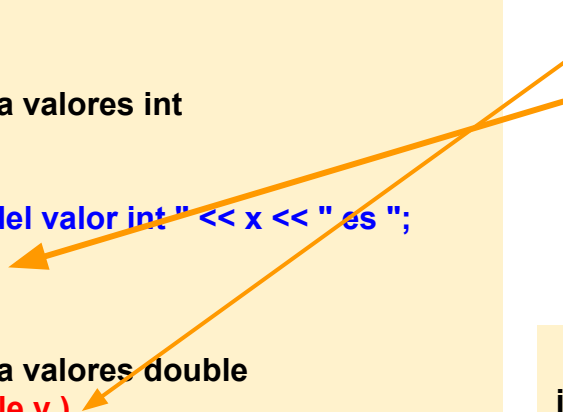
# Continuamos con las funciones

```
#include<iostream>

using namespace std;

// función cuadrado para valores int
int cuadrado( int x )
{
    cout << "el cuadrado del valor int " << x << " es ";
    return x * x;
}

// función cuadrado para valores double
double cuadrado( double y )
{
    cout << "el cuadrado del valor double " << y << " es ";
    return y * y;
}
```



**Note: no tenemos prototipo**

## Funciones sobrecargadas

Las funciones sobrecargadas se diferencian mediante sus firmas. Una firma es una combinación del nombre de una función y los tipos de sus parámetros (en orden)

```
int main()
{
    cout << cuadrado( 7 ); // llama a la versión int
    cout << endl;

    cout << cuadrado( 7.5 ); // llama a la versión double
    cout << endl;

    return 0;
}
```

# Continuamos con las funciones

```
#include<iostream>

using namespace std;

inline double cubo( const double lado )
{
    return lado * lado * lado; // calcula el cubo
}

int main()
{
    double valorLado;
    cout << "Escriba la longitud del lado de su cubo: ";
    cin >> valorLado;

    // calcula el cubo de valorLado y muestra el resultado
    cout << "El volumen del cubo con un lado de "
        << valorLado << " es " << cubo( valorLado ) << endl;

    return 0;
}
```

Locales: auto, static, y register  
Globales: static o extern

## Funciones Inline

**“Hace copias de la función donde se llame”**

solo para funciones muy pequeñas de uso frecuente.

paréntesis: Variables **“static”**

```
void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    count++;
}

int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

# Pila de llamadas a funciones y los registros de activación



Cuando se coloca un plato en la pila, por lo general se coloca en la parte superior (lo que se conoce como meter el plato en la pila). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como sacar el plato de la pila). Las pilas se denominan estructuras de datos “último en entrar, primero en salir” (UEPS); el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.

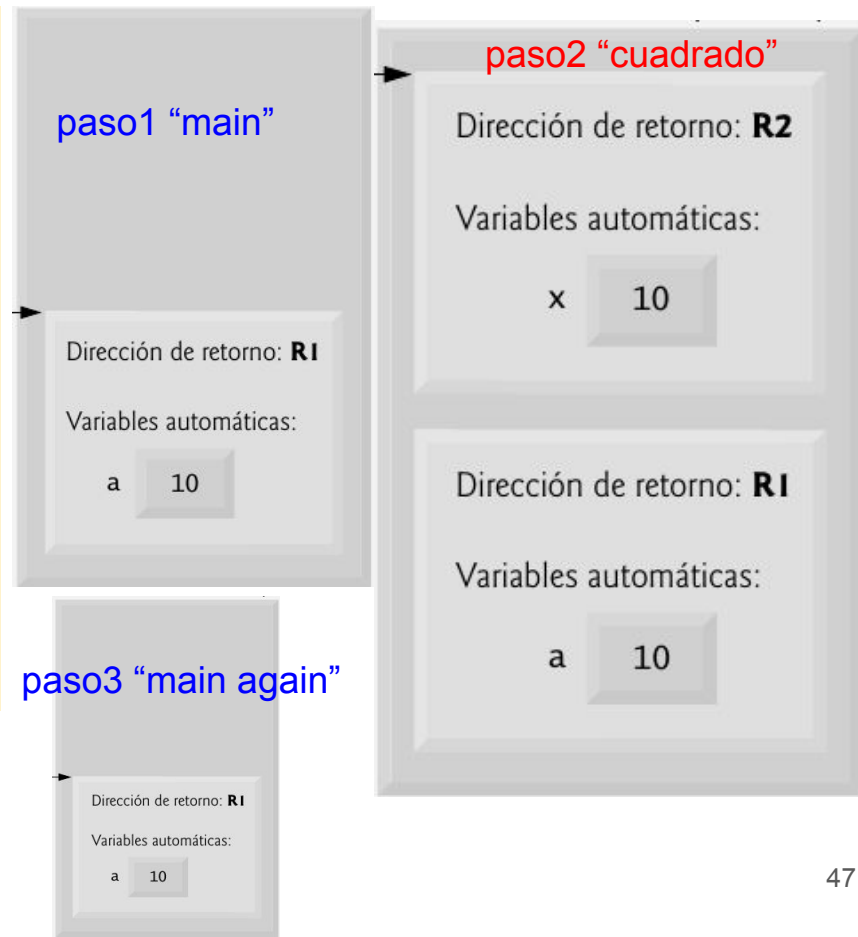


# Pila de llamadas a funciones y los registros de activación

```
int cuadrado( int );
int main()
{
    int a = 10;
    //valor para cuadrado (variable local automática en main)

    cout << a << " al cuadrado: " << cuadrado( a ) << endl;
    return 0;
}

int cuadrado( int x ) // x es una variable local
{
    return x * x; // calcula el cuadrado y devuelve el resultado
}
```



# Plantillas de funciones “template”

```
template < class T > // o template< typename T >
T maximo( T valor1, T valor2, T valor3 )
{
    T valorMaximo = valor1; // asume que valor1 es maximo

    // determina si valor2 es mayor que valorMaximo
    if ( valor2 > valorMaximo ){valorMaximo = valor2;}

    // determina si valor3 es mayor que valorMaximo
    if ( valor3 > valorMaximo ){valorMaximo = valor3;}

    return valorMaximo;
} // fin de la plantilla de función maximo
```

E. panti.h  
and myplanti

Si no se coloca la palabra clave "class" o "typename" (que son sinónimos) antes de cada parámetro de tipo formal de una plantilla de función (por ejemplo, escribir <class S, T> en vez de <class S, class T> ), se produce un error de sintaxis.



```
int cuadradoPorValor( int );  
void cuadradoPorReferencia( int & );
```

```
int main()  
{  
    int x = 2;  
    int z = 4;  
  
    // demuestra cuadradoPorValor  
    cout << "x = " << x << " antes de cuadradoPorValor\n";  
    cout << "Valor devuelto por cuadradoPorValor: "  
        << cuadradoPorValor( x ) << endl;  
    cout << "x = " << x << " despues de cuadradoPorValor\n" << endl;  
  
    // demuestra cuadradoPorReferencia  
    cout << "z = " << z << " antes de cuadradoPorReferencia" << endl;  
    cuadradoPorReferencia( z );  
    cout << "z = " << z << " despues de cuadradoPorReferencia" << endl;  
    return 0;  
}
```

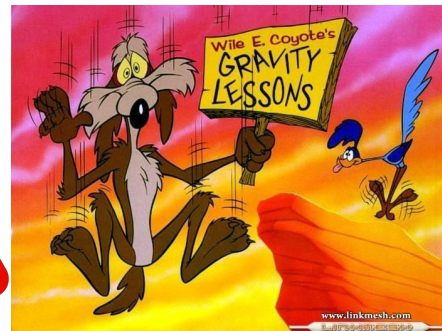
```
int cuadradoPorValor( int numero )  
{  
    return numero *= numero; // no se modificó el argumento de la función que hizo la llamada  
}
```

```
void cuadradoPorReferencia( int &refNumero )  
{  
    refNumero *= refNumero; // se modificó el argumento  
}
```

## Parametros por referencia

Una desventaja del paso por valor es que, si se va a pasar un elemento de datos extenso, el proceso de copiar esos datos puede requerir una cantidad considerable de tiempo de ejecución y espacio en memoria.

El paso por referencia es bueno por cuestiones de rendimiento, ya que puede eliminar la sobrecarga de copiar grandes cantidades de datos en el paso por valor.



El paso por referencia puede debilitar la seguridad, ya que la función a la que se llamó puede corromper los datos de la función que hizo la llamada.

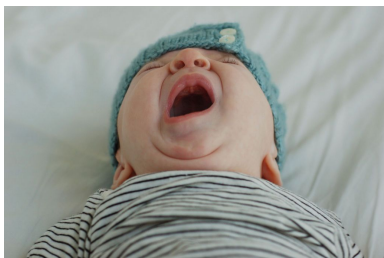
Sabemos la solución de un caso simple y usamos la "recursividad" para solucionar uno mas difícil.

se divide el problema en dos pasos, uno que si sabe resolver y otro que no ==> la funcion llama al problema que si sabe para resolver al que no sabe ==> esto se hace tantas veces como sea necesario.

**Veamos un ejemplo con el factorial**

$n! = n(n-1)(n-2).....1$   
 $n! = n (n-1)!$

$5! = 5*4*3*2*1$   
 $5! = 5*(4*3*2*1)$   
 $5! = 5*4!$



**Una visión "corta" de recursividad**

