

INTRODUCCIÓN A LA PROGRAMACIÓN

---

CURSO 2003/2004



**INGENIERIA TECNICA**  
**EN**  
**INFORMATICA DE SISTEMAS**

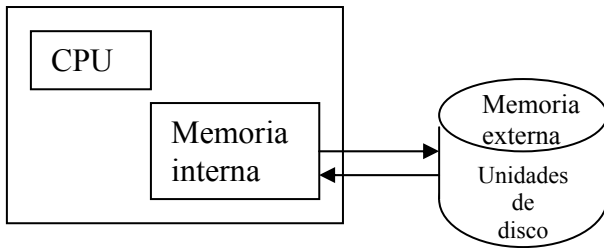
*Por: Juan Luis Romero Dueñas*

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

### DIAGRAMA GENERAL DE UN SISTEMA COMPUTADOR.



### DEFINICION DE LENGUAJE DE PROGRAMACION

Un lenguaje de programación, es el medio de comunicación con un sistema computador, diciéndole por medio de instrucciones entendibles por este, lo que queremos que haga.

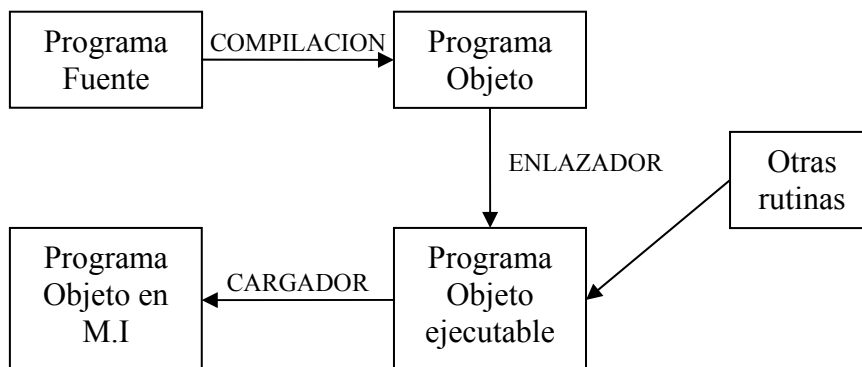
### ORDEN DE APARICION DE LOS DISTINTOS TIPOS DE LENGUAJES DE PROGRAMACION

DÉCADA	HECHO OCURRIDO
70	Aparece el concepto de programación estructurada.
80	Surge la programación concurrente.
90	Aparece la programación visual.

### PARADIGMAS DE PROGRAMACION

- Lenguajes Imperativos.
- Lenguajes Declarativos.

### PROCESO DE TRADUCCION EN UN SISTEMA COMPUTADOR



# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

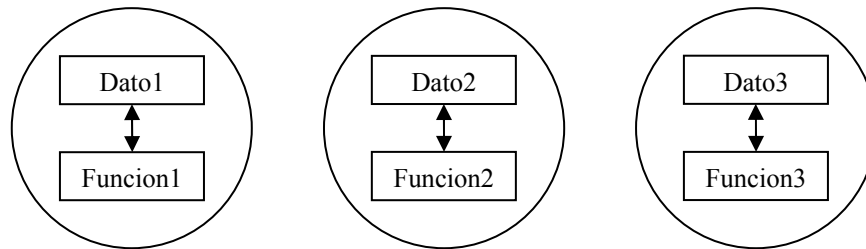
### PROGRAMACION ORIENTADA A OBJETOS

Antes de que se introdujera el concepto de programación orientada a objetos, se introdujo el concepto de programación estructurada, la cual emplea las siguientes sentencias:

- Secuencias.
- Alternativas.
- Iterativas.

Cualquier función que se diseñada en programación estructurada, puede modificar el valor de un dato.

En la programación orientada a objetos, cada función es específica para cada dato, de este modo se crea un único grupo de dato y función.



La idea fundamental de la programación orientada a objetos es la de unir datos y las funciones que están relacionadas con esos datos.

Ejemplo:

Vehículo:

Datos:

Modelo, color, nº bastidor, peso, nº plazas,....

Funciones:

Arrancar, frenar, repostar, acelerar,....

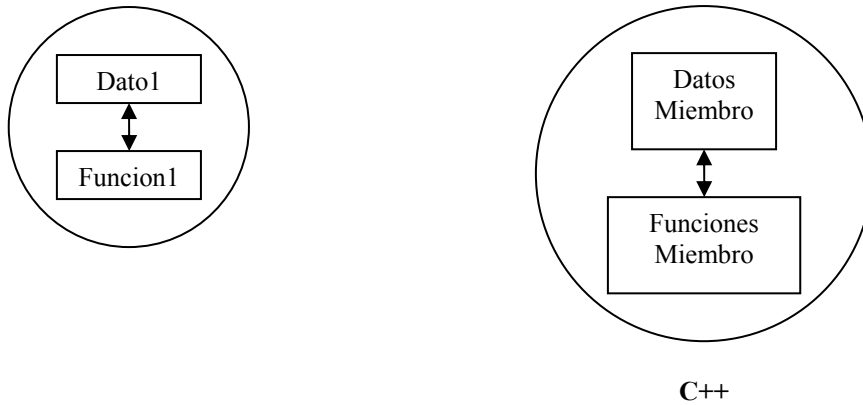
**El vehículo es una clase, modelo o molde.**

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

### CLASES Y OBJETOS



La representación grafica normalizada de la declaración de una clase, seria del siguiente modo:

Nombre de la clase (Objeto)
Datos miembro
Funciones miembro

Ejemplo:

Vehículo
Modelo, color, nº bastidor, nº de plazas, peso, ...
Arrancar, frenar, repostar, acelerar, cambiar de velocidad,...

Cuenta corriente
Nº de titulares, DNI del titular_X, direccion, saldo,...
Ingresar dinero, sacar dinero, cambiar titular, consultar cuenta,...

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

### DECLARACION DE CLASES EN C++

En el lenguaje de programación C++, las clases se declaran del siguiente modo:

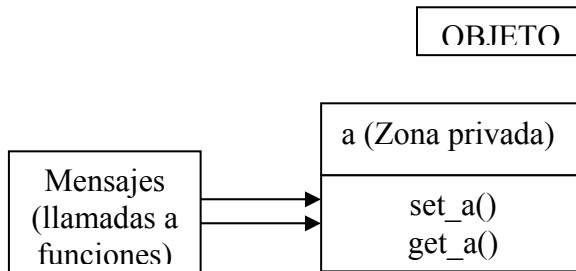
```
class nombre_de_la_clase
{
    Variables y funciones privadas de clase

public:
    Variables y funciones publicas de la clase
};
```

Ejemplo:

```
class mi clase
{
    int a ; //variable 'a' de tipo entero
public:
    ...set_a();
    //funciones publicas.
    ...get_a();
};
```

### *REPRESENTACIÓN GRAFICA DE LA CLASE:*



La definición de objetos a partir de una clase definida de modo previo en C++, se realiza del siguiente modo:

Ejemplo:

```
mi clase ob1, ob2;
```

Así de esta manera, se reservara espacio en memoria para los objetos **ob1** y **ob2** de tipo **mi clase**.

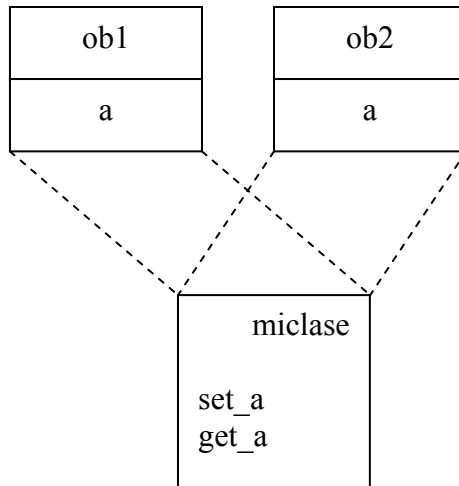
La clase, es el modelo del objeto, y los objetos son elementos reales de la clase.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

La disposición en memoria de los objetos de una clase, es del siguiente modo:



Cada objeto de una clase tiene su propia copia de los datos miembro y solo existe una única copia de las funciones miembro para todos los objetos de esa misma clase.

La identidad expresa que aunque dos objetos del mismo tipo tengan los mismos atributos, como por ejemplo **ob1** y **ob2**, son distintos entre sí ya que el valor de **a** es diferente.

Las llamadas a funciones miembro, se realizan mediante lo que se denominan **mensajes**, y su formato en C++ es del siguiente modo:

```
nombre_objeto.funcion_miembro
```

Ejemplo:

```
class miclase
{
    int a;
    public:
        ...set_a();
        ...get_a();
};
//vendrían detalles de las funciones miembro set_a y get_a.

void main()
{
    miclase ob1, ob2; //declaración de los objetos.
    .....
    ob1.set_a();
    ob2.set_a(); //llamadas a funciones miembro.
    ob1.get_a();
    ob2.get_a();
};
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

Podemos considerar tres características fundamentales que incluye todo programa, de programación orientado a objetos:

- Encapsulación.
- Polimorfismo.
- Herencia.

La **encapsulación** es el mecanismo que agrupa el código y los datos que maneja y a los que mantiene protegidos frente a cualquier interferencia y mal uso por parte del exterior.

El **polimorfismo** es la cualidad que permite que un nombre se utilice para dos o más propósitos relacionados, pero técnicamente diferentes.

- En C para calcular el valor absoluto de un número tenemos las funciones `abs()` y `fabs()`, para un número entero y para un número real respectivamente.

Ejemplo:

```
abs(-3)
fabs(-3.2)
```

- En C++ se incorpora polimorfismo para calcular el valor absoluto con una sola función `abs()`.

Ejemplo:

```
abs(-3)
abs(-3.2)
```

A esto se le conoce con el nombre de **sobrecarga de funciones**.

- Otro tipo de polimorfismo es el conocido con el nombre de **sobrecarga de operadores**. El siguiente ejemplo nos indica a que nos referimos con este término.

Ejemplo:

El operador `+`, puede sumar tanto números enteros como cualquier otro tipo de números.

```
(3) + (8)
(3.2) + (8.5)
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

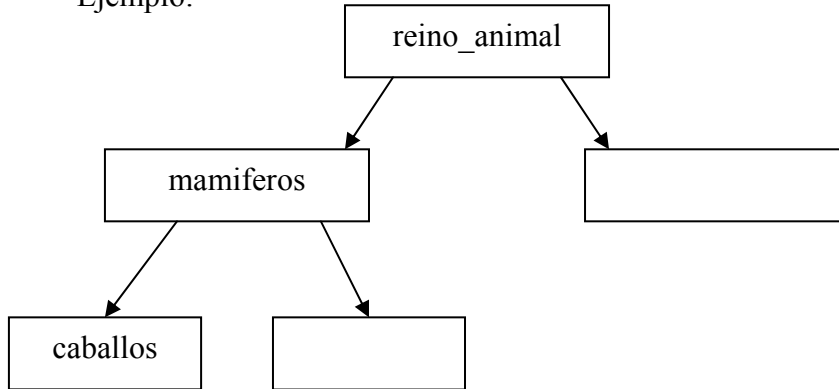
## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

La **herencia** es el proceso mediante el cual un objeto puede adquirir (heredar) las propiedades de otro.

Con la herencia se puede describir un objeto mediante una clase que reúna las características generales y después heredar otra clase que contenga características particulares de ese objeto en particular (clasificación de jerárquica de clases).

Ejemplo:



```
class reino_animal
{
    .....
};
class mamíferos:public reino_animal
{
    .....
};
```

Hemos de reiterar una vez más que **LOS DATOS MIEMBRO SON DE TIPO PRIVADO**.

### RESUMEN:

<b>POO</b>	<b>C++</b>
CLASE	CLASE
OBJETO	OBJETO
VARIABLES ASOCIADAS	DATOS MIEMBRO
METODO	FUNCIONES MIEMBRO
MENSAJE	LLAMADAS A FUNCIONES MIEMBRO
SUBCLASE	CLASE DERIVADA
HERENCIA	DERIVACION

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA I: LOS LENGUAJES DE PROGRAMACIÓN

---

### EJERCICIOS RESUELTOS:

1.- Ponga el esquema general de cómo sería el diseño de la clase en caso de tener que controlar números fraccionarios.

Ejemplo:

$$\frac{5}{6}$$

*SOLUCION:*

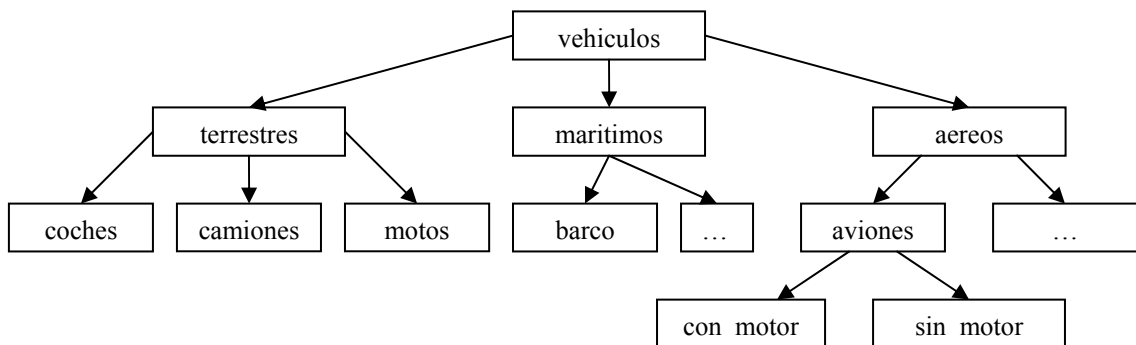
```
class fraccionarios
{
    int numerador, denominador;

    public:
        ...leer_datos();
        ...mostrar_datos();
        ...suma();
        ...resta();
        ...multiplica();
        ...divide();
        ...potencia();
        .....
};
```

2.- Ponga el esquema general de cómo sería el diseño de las clases en el caso de tener que controlar en el programa coches, camiones, motos, barcos, aviones con motor, aviones sin motor.

*SOLUCION:*

Partiremos del siguiente diseño jerárquico.



```
class vehiculos{
    ...};
class terrestres:public vehiculos{
    ...}; //igual con todas las demás clases derivadas
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

---

CURSO 2003/2004

**Tema 1.-** Lenguajes de programación.

**Tema 2.-** Algoritmos. Tipos de datos, operadores y expresiones.

**Tema 3.-** Estructuras de control.

**Tema 4.-** Tipos de datos estructurados.

**Tema 5.-** Diseño descendente.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### CONCEPTO DE ALGORITMO. ESTRUCTURA GENERAL

Un algoritmo, es la descripción precisa de una sucesión de instrucciones que permiten llevar a cabo un trabajo en un número finito de pasos.

La estructura general de un algoritmo, consta de los siguientes elementos:

- **Cabecera:** en ella se da nombre al algoritmo, indicando así el comienzo del mismo. En el lenguaje C++ la cabecera es:

```
void main()
{
```

- **Declaración de variables:** es el punto donde se reserva espacio en memoria para las variables indicadas, dándoles un nombre que identifique a las mismas. Una declaración de variables hecha en C podría ser del siguiente modo:

```
int a,b;
```

- **Cuerpo del algoritmo:** es la parte donde se encuentran las instrucciones que componen el algoritmo.

Ejemplo:

```
a = 1;
b = a + 1;
```

Basándonos en todo lo expuesto hasta este punto, nuestro algoritmo de ejemplo quedaría del siguiente modo:

```
Void main() //cabecera
{
    int a,b; //Declaración de variables

    a = 1;    //Cuerpo del algoritmo
    b = a + 1;
} //Fin del algoritmo
```

También diremos que las fases que debemos de seguir para resolver un problema (creación del algoritmo), serán las siguientes:

- **Análisis.**
- **Diseño.**
- **Programación.**

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### PALABRAS RESERVADAS, IDENTIFICADORES, CONSTANTES, VARIABLES Y COMENTARIOS

Las **palabras reservadas** de un lenguaje de programación, son aquellas que tienen un significado especial para el lenguaje.

Ejemplo: **class**, **private**, **public**, etc...

Los **identificadores** son nombres que se les dan a las constantes, variables, etc... para diferenciarlos entre ellos. Están formados por una secuencia de caracteres, siendo en el caso de C++ el primero obligatoriamente un carácter alfabético o el carácter guión bajo, pudiendo ser el resto de caracteres, dígitos, letras, no pudiendo existir de ningún modo blancos intermedios en cualquier caso. De igual modo, no serán identificadores validos las palabras reservadas del lenguaje de programación. Hemos de indicar, que en C++ se distinguen las mayúsculas y las minúsculas, de modo que un identificador escrito el primer carácter en mayúscula, no será el mismo aunque la secuencia de caracteres sea exactamente igual, pero cambiando el primer carácter por un carácter minúsculo.

Ejemplo:

Suma  $\neq$  suma

A continuación ilustramos una lista con algunos identificadores validos y otros no validos.

SON VALIDOS	NO SON VALIDOS
A1876	Cantidad total
X W W 1	2numeros
Cinco_numeros	new

Las **constantes** son datos cuyo valor no se puede alterar, durante la ejecución del programa.

La declaración de constantes en el lenguaje C presenta dos formas de poderlas declarar, las cuales son:

```
const tipo_dato identificador = valor; // Forma 1  
#define identificador valor // Forma 2
```

Cualquiera de las dos formas aquí expuestas es igualmente valida para declarar una constante en C++, la elección de cada una de las formas, será a gusto del programador.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

A continuación mostramos sendos ejemplos de declaración de constantes:

Ejemplo:

```
// Declaración de una constante según la forma 1  
const float PI=3.14;
```

```
// Declaración de una constante según la forma 2
```

```
#define PI 3.14
```

Las **variables** son datos cuyo valor se puede alterar durante el desarrollo del algoritmo.

La declaración de variables en C++, es del siguiente modo:

```
tipo_dato1 identificador1,identificador2,.....,identificadorN;  
.....
```

A continuación mostramos el siguiente ejemplo, que nos ilustrara mejor como declarar variables en C/C++.

Ejemplo:

```
int a,b,c;  
float renta;
```

También, hemos de indicar que en el lenguaje de programación C/C++, una vez declarada la variable, se puede al mismo tiempo inicializarla a un valor. A continuación se muestra un ejemplo de lo aquí expuesto.

Ejemplo:

```
// Declaración de 3 variables de tipo entero con inicialización de  
// las mismas en su declaración
```

```
int a=10,b=5,c=8;
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

Los **comentarios** son la documentación interna de nuestro algoritmo, la cual se incluye dentro del código del mismo, facilitando así la comprensión de este.

En el lenguaje de programación C/C++, los comentarios se pueden realizar de dos maneras:

- En una sola línea: Se escribirá el comentario, a continuación de los caracteres // y terminando este mismo al final de la línea en cuestión, tal y como muestra este ejemplo.

Ejemplo:

```
void main() // esto es un comentario de una sola línea
```

- En varias líneas: Se englobará el comentario entre /\* ...\*/, y así de ese modo, el comentario podrá extenderse a varias líneas, tal y como vemos en el siguiente ejemplo.

Ejemplo:

```
void main() /* esto es un comentario que continua en  
dos líneas del código del programa */
```

Ya para finalizar, mostraremos a continuación la sintaxis empleada en C/C++ para definir tipos de datos. La manera de declarar estos será del siguiente modo:

```
typedef tipo_dato identificador;
```

A continuación mostramos un ejemplo de definición de un tipo de dato.

Ejemplo:

```
typedef int entero;
```

```
.....
```

```
entero num; // Declaración de una variables num  
// de tipo entero.
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### TIPOS DE DATOS. CLASIFICACION

Simple:	Entero Real Carácter
Definidos por el usuario:	Enumerados
Estructurados:	Tabla Registros Dinámicos

### TIPOS DE DATOS SIMPLES

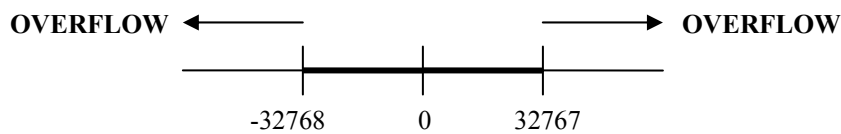
#### **Tipo Entero:**

Partiremos del siguiente ejemplo para ver y comprender mejor este tipo de dato.

Si tenemos 16 bits (2 Bytes) para almacenar los valores de tipo entero, en binario se representaría de la siguiente forma.

0000 0000 0000 0000 → 0  
.....  
1111 1111 1111 1111 → 65534

Su rango de valores se estará comprendido entre:



La manera de expresar una variable entera es:

```
int a; //declaración de una variable entera normal  
short int b; // Declaración de una variable entera corta  
long n; // Declaración de una variable entera larga
```

*/\* Los enteros largos, tienen un rango comprendido entre -2147483648 y 2147483647, lo cual hace una ocupación en memoria de 4 bytes \*/*

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

Las operaciones permitidas para los tipos enteros, son:

Suma.....+  
Resta.....-  
Producto.....\*  
División...../  
Modulo (resto de la división entera).....%

Como ultimo dato, diremos que si conocemos de antemano si las variables de tipo entero que vamos a manejar no van a ser números negativos, podemos declararlas del siguiente modo:

Ejemplo:

```
unsigned int a;  
/* variable "a" de tipo entero sin signo, rango de  
representación de 0 a 65534 */
```

**Tipo Real:**

Numero = mantisa \* base<sup>exponente</sup>

La mantisa no tiene números decimales

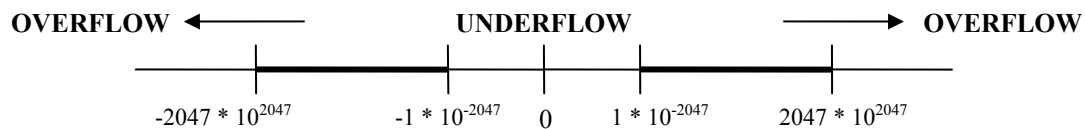
Ejemplo: 3.14 → 0.314 \* 10<sup>1</sup>

Basándonos en lo aquí expuesto, vamos a decir que, si una maquina tiene 24 bits para almacenar números reales, dejando 12 bits para la mantisa y 12 bits para el exponente, expresada en base 10, tendremos que:

Exponente:  $2^{12} = 4096 \rightarrow -2047, \dots, 0, \dots, 2047$

Mantisa:  $2^{12} = 4096 \rightarrow -2097, \dots, 0, \dots, 2047$

Con lo cual tendremos el siguiente rango de representación :



Las operaciones que podemos realizar serán las mismas que para el tipo de datos entero, a excepción del modulo.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

Atendiendo a que rango de números reales queremos atender, las variables reales en C/C++, serán de dos tipos:

- **float**: están comprendidas en un rango desde  $-3.4*10^{38}$  hasta  $1.17*10^{37}$  y desde  $1.17*10^{37}$  hasta  $3.4*10^{38}$ .
- **double**: es una variable real de precisión doble. Su rango esta comprendido desde  $-1.79*10^{308}$  hasta  $2.22*10^{307}$  y desde  $2.22*10^{307}$  hasta  $1.79*10^{308}$ .

Hemos de hacer una mención importante, y es que para asignar un valor de inicialización a una variable real, debemos de tener en cuenta el dígito siguiente después de la coma decimal.

Ejemplo:

```
float a=0.0;  
float b=2.0;
```

### *Tipo Carácter:*

Los valores de tipo carácter, están definidos en un conjunto finito de caracteres. Su ocupación en memoria es de 1 Byte, y su rango de representación esta comprendido entre 0 y 255 valores. Las variables de este tipo solo guardan un único carácter. Su declaración será del siguiente modo:

```
char a;
```

Los valores de tipo carácter serán escritos entre comillas simples, con el fin de diferenciarlos de los identificadores.

Ejemplo:

```
char a='z';
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### **Tipo Enumerado:**

Se define explícitamente citando un conjunto finito de identificadores, que designaran los valores posibles de los datos pertenecientes a dicho tipo.

Su formato de declaración en C/C++, es el siguiente:

```
enum identificador
    {id1, id2,...,idN};
```

Ejemplo:

```
enum dia
    {lunes,martes,miércoles,jueves,viernes,sabado,domingo};

dia hoy; //declaración de la variable "hoy" de tipo enumerado

hoy=jueves;
```

Internamente los datos de tipo enumerado, se almacenan como valores enteros. A cada valor del tipo se le asocia un entero consecutivo comenzando por cero, que indica la posición dentro del conjunto de valores.

### **Tipo Lógico o booleano:**

Este tipo de dato, solo puede tomar dos valores, los cuales son *verdadero* o *falso*.

Este tipo, no se encuentra implementado en las primeras versiones del compilador de C, siendo añadido al mismo, en las versiones mas recientes de este.

Una manera practica y sencilla de simular el tipo booleano, es tal y como se muestra a continuación:

```
typedef int booleano;
const int true=1;
const int false=0;
```

Para terminar con los tipos de datos simples, se muestra a continuación un cuadro con los cualificadotes permitidos para los datos estudiados hasta aquí.

	signed/unsigned	short	long
int	SI	SI	SI
char	SI	NO	NO
float	NO	NO	NO
double	NO	NO	SI
enum	NO	NO	NO

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### **TIPOS DE DATOS ESTRUCTURADOS**

Estos datos, están formados por tipos de datos simples, con una cierta relación entre ellos.

Los tipos de datos estructurados, pueden atender a ciertas características las cuales exponemos y definiremos a continuación:

- Homogénea: se dice que una estructura de datos es homogénea, cuando todos los datos de esta, son del mismo tipo; en caso contrario se dice que la estructura es *heterogénea*.
- Estática: una estructura de datos es estática, cuando el tamaño que ocupa en memoria no puede ser modificado, durante la ejecución del programa; en caso contrario, hablaremos de estructuras *dinámicas*.

A continuación citamos algunos tipos de datos estructurados:

- Tablas
- Registros
- Listas
- Árboles.
- Grafos.

#### ***Tipo de dato estructurado TABLA:***

Una tabla o ARRAY, es un conjunto de datos homogéneos, los cuales ocupan posiciones sucesivas de memoria. Hemos de destacar, que se trata de un tipo de dato estático.

La forma de declarar una tabla en el lenguaje de programación C/C++, es la siguiente:

```
tipo_dato identificador[expresión];
```

Ejemplo:

```
int lista[40]; /* Declaración de una tabla  
unidimensional para guardar de 0 a 39 enteros */
```

De igual forma que hemos hecho en el ejemplo anterior, para declarar una tabla que guardara una lista de números enteros comprendidos entre sus posiciones 0 y 39, la declaración de una lista de caracteres o cadena de caracteres, sería del siguiente modo:

```
char identificador[expresión];
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

Diremos también, que una cadena de caracteres se debe terminar siempre con el carácter `\0` (**CODIGO ASCII 0**) como delimitador de la cadena.

Y para finalizar hemos de indicar que para inicializar una tabla, no es necesario el especificar el tamaño. La inicialización será de la siguiente manera:

```
tipo_dato identificador[]=valor_inicial;
```

Ejemplo:

```
char nombre[]="federico";
```

### **FUNCIONES QUE FACILITAN EL MANEJO CON LAS CADENAS DE CARACTERES**

El lenguaje de programación de C que nosotros vamos a emplear en el laboratorio, trae implementada en una librería llamada *string.h*, una serie de funciones las cuales nos harán mas sencillo y mas cómodo el manejo de cadenas de caracteres.

La llamada a esta biblioteca, se realiza mediante una directiva del compilador expresada de la siguiente forma:

```
#include<string.h>
```

A continuación se muestra las funciones que contiene dicha librería, así como una breve explicación de su uso. Estas funciones son:

- **strcmp** (cad1,cad2): Compara las dos cadenas, e indica si son iguales o en caso contrario nos indica cual es la que va primera.

Datos devueltos por la función y su significado:

<0 → La cadena1 (cad1) va antes que la cadena2(cad2).

=0 → cad1 y cad2 son iguales.

>0 → cad2 va antes que cad1.

- **strcpy** (cad1,cad2): Copia la cadena2 (cad2) en la cadena1(cad1).
- **strcat** (cad1,cad2): Concatena a la cadena1 (cad1), cad2.
- **strlen** (cad1): Devuelve la longitud de cad1.
- **strupr** (cad1): Convierte cad1 todo a mayúsculas.
- **strlwr** (cad1): Convierte cad1 todo a minúsculas.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### OPERADORES DE ASIGNACION, ARITMETICOS, RELACIONALES Y LOGICOS

#### *Operador de asignación:*

El operador de asignación, sirve para poner un valor a una variable.  
El formato de una instrucción de asignación en C es el siguiente:

`identificador=expresion;`

Diremos también que la operación de asignación, es realizada en dos pasos bien diferenciados, los cuales son:

- 1º- Calculo del valor de la expresión.
- 2º- Almacenamiento del valor en la variable cuyo identificador aparece a la izquierda.

Ejemplo:

```
int a;
```

```
a=3+9;
```

Hemos de advertir, que la operación de asignación es una operación destructiva, y queremos decir con esto que una vez asignado un valor a una variable, el valor que esta tenia anteriormente, habrá sido eliminado por el nuevo valor asignado. También diremos que es un error muy común el emplear una variable sin inicializarla, y por eso se recomienda encarecidamente **INICIALIZAR TODAS LAS VARIABLES ANTES DE USARLAS.**

A continuación, mostraremos algunas curiosidades en cuanto a modos de asignar, las mas destacadas son:

```
x++ → x=x+1 → ++x  
x-- → x=x-1 → --x
```

```
x+=2 → x=x+2  
x-=2 → x=x-2
```

```
x*=2 → x=x*2  
x/=2 → x=x/2
```

Hemos de indicar que las instrucciones `x++` y `++x` aunque son iguales, no funcionan de igual modo, a continuación se muestra como realiza la asignación ambas:

$$y = x++ \rightarrow \begin{cases} y = x \\ x = x + 1 \end{cases}$$
$$y = ++x \rightarrow \begin{cases} x = x + 1 \\ y = x \end{cases}$$

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

También, para terminar diremos que se pueden hacer asignaciones en una misma línea, de acuerdo y tal como expresa el siguiente ejemplo.

Ejemplo:

```
aux=v1, v1=v2, v2=aux;
```

### ***Operadores aritméticos:***

Los operadores aritméticos, son aquellos que intervienen en relaciones numéricas, relaciones tales como la suma, resta, división, etc...

Los operadores aritméticos de C son los siguientes:

```
+ → Suma  
- → Resta  
* → Multiplicación  
/ → división  
% → Modulo de la división entera (resto)
```

Hemos de indicar, que los operandos pueden ser números reales o enteros, tan solo para el caso de la división de dos números enteros, se ha de advertir, que en su resultado se producirá un truncamiento en el caso de que existan decimales.

### ***Operadores relacionales:***

Son los que nos permiten realizar operaciones de comparación entre datos, siendo su resultado de salida *verdadero* o *falso*.

Los operadores relacionales incluidos en C son:

```
== → Igual que...  
!= → Distinto a...  
<, >, <=, >=
```

Al realizar operaciones de comparación, los operandos podrán ser enteros o reales, siendo el resultado de salida de la comparación 1 si se cumple la condición (*verdadero*) y 0 en caso contrario (*falso*).

Ejemplo:

```
int p,q;  
  
p=15, q=5;  
  
p<q; // resultado de salida 0 (falso)  
p>q; // resultado de salida 1 (verdadero)
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### *Operadores lógicos:*

Son los que se emplean para la realización de operaciones de tipo lógico.  
Los operadores lógicos de C son:

&& → operación AND  
|| → operación OR  
! → operación NOT

<b>&amp;&amp;</b>		
<b>operando1</b>	<b>operando2</b>	<b>Resultado</b>
Falso	Falso	Falso
Falso	Verdadero	Falso
Verdadero	Falso	Falso
Verdadero	Verdadero	Verdadero

<b>  </b>		
<b>operando1</b>	<b>operando2</b>	<b>Resultado</b>
Falso	Falso	Falso
Falso	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Verdadero	Verdadero	Verdadero

<b>!</b>	
<b>operando</b>	<b>Resultado</b>
Falso	Verdadero
Verdadero	Falso

Para concluir con este tipo de operadores, diremos que los operandos pueden ser enteros o reales, y al igual que sucedía con los operadores relacionales, el resultado de la operación lógica, devolverá 1 si la operación es verdadera o 0 en caso contrario.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### **EXPRESIONES Y ORDEN DE PRECEDENCIA**

#### *Reglas de precedencia y asociatividad:*

Como sabemos, una expresión que se encuentre entre paréntesis, siempre tendrá la mayor prioridad, frente a cualquier otra. En el caso de que en una expresión no existan paréntesis, los cuales marquen la prioridad de operaciones en la misma, la prioridad de operadores en C/C++, será la siguiente:

Prioridad	Operador/es	Asociatividad
Mayor ↓ Menor	!, ++, --, -(unario), new, delete	D→I
	*, /, %	I→D
	+, -(binario)	I→D
	<, <=, >, >=	I→D
	=, !=	I→D
	&&	I→D
		I→D

Ejemplo:

$$6/3*2 \rightarrow (6/3)*2 = 2*2 = 4$$

$$6/(3*2) \rightarrow 6/6 = 1$$

Se aconseja para el uso de expresiones, el empleo del paréntesis para marcar el orden de prioridad de las operaciones, y diremos a modo de que no se olvide que, **SE ACONSEJA ENCARECIDAMENTE EL USO DE PARENTESIS EN LAS EXPRESIONES, SIENDO MEJOR EL HECHO DE QUE SOBREN PARENTESIS, ANTES DE QUE FALTEN.**

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### *Conversiones de tipos en las expresiones:*

- Si un operando es de tipo **double**, el otro operando se convierte a **double**.
- Si un operando es de tipo **float**, el otro operando se convertirá a **float**.
- Cualquier operando de tipo **char** o **short** es convertido a **int**, si este tipo puede representar un valor del tipo original, si no se convierte a **unsigned short**.
- Si un operando es de tipo **long**, el otro operando se convertirá a **long**.
- Si un operando es de tipo **unsigned int**, el otro operando se convierte a **unsigned int**.
- En cualquier otro caso, ambos son de tipo **int**.

Como norma general, los operandos que formen la expresión, han de ser convertidos al tipo de dato de mayor precisión.

### *Coherencia de tipos. Moldeado de tipos:*

Se puede realizar una conversión explícita de tipos de datos (moldeado de tipos), de una expresión mediante una construcción **cast**, de la siguiente forma.

<pre>(tipo_dato)expresión; tipo_dato(expresion);</pre>
--

Por ejemplo, la función **sqrt** la cual calcula la raíz cuadrada, espera como argumento un tipo **double**, y por tanto podemos hacer lo siguiente:

Ejemplo:

```
int n=7;  
sqrt((double)(n+2));
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### OPERACIONES BASICAS DE ENTRADA-SALIDA

Para poder comunicarnos con el programa, hemos de emplear unas operaciones llamadas de entrada-salida, y para poder hacer uso de estas operaciones en C/C++, deberemos de emplear la librería *iostream.h*. Para poder hacer uso de esta librería, hemos de referenciarla, mediante una directiva de compilación de la siguiente forma:

```
#include"iostream.h"
ó
#include <iostream.h>
```

Para las operaciones de entrada, emplearemos la siguiente instrucción:

```
cin>>expresión
```

También, diremos que se pueden realizar operaciones de lectura múltiple, tal y como se expresa a continuación:

```
cin>>expresión1 >>expresion2 >>expresionN;
ó
cin >>expresion1
  >>expresion2
  >>expresión;
```

Hemos de mencionar, que la instrucción **cin**, termina cuando se pulsa **INTRO**, a esto es lo que se conoce como **entrada con buffer de línea**. También, cuando la instrucción encuentra un blanco, deja de almacenar en la variable. Si queremos almacenar varias palabras, debería ser de la siguiente forma:

Ejemplo:

```
char var1[80],var2[80],var3[80];
cin >>var1 >>var2 >>var3;
```

.....

```
Juan Luis Romero
var1=Juan
var2=Luis
var3=Romero
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

Llegados hasta este punto, tan solo hemos hablado de las operaciones de lectura, pero otra operación muy importante, es la de salida (mostrar por pantalla), la instrucción de salida que emplearemos será:

```
cout<<expresión;
```

Como todos sabemos, es muy importante que nuestro programa se legible desde la pantalla de nuestro ordenador, y para poder ordenar los datos que mostramos por la misma, la mayoría requieren uno o varios saltos de línea, con el fin de organizar en la pantalla, la salida de nuestro programa. Si la expresión que queremos mostrar por pantalla, queremos que disponga de un salto de línea, nuestra instrucción deberá quedar del siguiente modo:

```
cout<<"expresion \n"
```

Aquí vemos que para conseguir el salto de línea, hemos empleado el carácter de escape ASCII `\n`, el cual nos provee de un salto de línea.

Ejemplo:

```
cout <<"La cantidad es: \n" <<var1;
```

Hemos de recordar e insistir en que **NO SE PUEDE LEER DESDE TECLADO NI ESCRIBIR VALORES EN PANTALLA DIRECTAMENTE DE VARIABLES DE TIPO ENUMERADO.**

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### - APENDICE A-

#### INTRODUCCIÓN A LAS FUNCIONES SIN PARAMETROS

Ya como podemos saber llegados hasta este punto, una *función* es una colección de variables y sentencias.

Ejemplo:

```
void main()
{
    //declaracion de variables y sentencias
}
```

En el ejemplo aquí expuesto, vemos la función cuyo nombre ó identificador es **main**, como vemos, a continuación del nombre de la función aparecen sendos paréntesis, los cuales al estar vacíos expresan que en dicha función no existen parámetros de entrada, al igual que tal y como vemos delante del nombre que nos aparece la palabra reservada **void** la cual expresa que la función no devuelve nada. A continuación mostramos ejemplos de otras funciones sin parámetros de entrada.

Ejemplo:

```
#include <iostream.h>
void main()
{
    int a;
    cout<<"Introduzca un numero por teclado: ";
    cin>>a;
}

#include <iostream.h>
int suma() //la función suma devuelve un entero
{
    int dato1,dato2,dato3;
    cin>>dato1>>dato2;
    dato3=dato1+dato2;

    return dato3;
}
```

Para terminar, diremos que a la línea de la cabecera de la función se la conoce con el nombre de **función prototipo**.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### - APENDICE A -

#### DECLARACIONES DE CLASES CON FUNCIONES

Tal y como sabemos, la sintaxis de declaracion de una función es:

```
tipo_dato identificador()
{
    Sentencias_y_variables
}
```

Y también sabemos, que el formato de declaracion de una clase es:

```
class identificador
{
    declaración de variables y funciones privadas

    public:
        declaración de variables y funciones publicas

};
```

Pues basándonos en lo aquí expuesto, diremos que para definir funciones de una clase determinada, el formato deberá de ser el siguiente:

```
tipo_dato_devuelto identificador_clase::nombre_funcion()
{
    variables y sentencias
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

A continuación se puede ver el siguiente ejemplo que no ilustrara mejor lo que aquí acabamos de exponer. Vamos a definir una clase llamada rectángulo, con algunas operaciones matemáticas con las que se puede definir un rectángulo.

Ejemplo:

```
#include <iostream.h>
class rectángulo
{
    int base,altura;

    public:
        void inicia();
        void lee_base_altura();
        int area();
        void cambia_base_por_altura();
};

void rectángulo::inicia()
{
    base=0; altura=0;
};

void rectángulo::carga()
{
    cin>>base>>altura;
};

int rectangulo::area()
{
    return base*altura;
};

void rectangulo::cambia()
{
    int aux;

    aux=base;
    base=altura;
    altura=aux;
};

/*aquí a continuación vendrían las sentencias del programa que utilice la clase aquí declarada*/
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### - APENDICE A -

#### ASIGNACIÓN ENTRE OBJETOS

Tal y como hemos visto en el caso de la asignación de una variable, diremos que un objeto se puede asignar a otro, siempre con la única condición de que ese otro sea otro objeto de la misma clase.

A continuación vemos un ejemplo a fin de ilustrar lo aquí expuesto.

Ejemplo:

```
class miClase
{
    int a,b;

    public:
        void set();
};

void miClase::set()
{
    a=5; b=5;
};

void main()
{
    miClase obj1,obj2;

    obj1.set();

    obj2=obj1;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

### EJERCICIOS RESUELTOS:

1.- ¿Para que sirve la directiva <iostream.h>?

*SOLUCION:*

Nos permite realizar operaciones de entrada salida y controlar el flujo de datos de un conjunto de caracteres.

2.- Dado el siguiente programa determinar los errores que se producen:

```
#define c 5
#define n -9
enum altura{
bajo, mediano, alto};
void main() {
int numero;
char media;
altura altos;
booleano b;
media='a';
altos=55;
numero=33 / c;
}
```

*SOLUCION:*

```
#define c 5
#define n -9
enum altura{
bajo, mediano, alto};
void main() {
int numero;
char media;
altura altos;
booleano b; //TIPO BOOLEANO NO DEFINIDO
media='a';
altos=55; //EL VALOR 55 NO PERTENECE AL TIPO ENUMERADO
numero=33 / c;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

3.- Diseñar una función que convierta una cantidad positiva de segundos leída desde teclado a su equivalente en horas, minutos y segundos, expresando estos valores por pantalla.

*SOLUCION:*

```
#include <iostream.h>
#include <conio.h>

void main()
{
    int segtot;

    clrscr();
    cout<<"Indique una cantidad positiva de segundos y presione INTRO: ";
    cin>>segtot;
    cout<<"\nSu resultado es: "
        <<(segtot/3600)<<':'
        <<((segtot/60)%60)<<':'
        <<(segtot/60)
    }
}
```

4.- Colocar a cada uno de los siguientes terminos el numero de la definición correcta. Hay solo una respuesta correcta para cada uno.

- Programa
- Algoritmo
- Compilador
- Identificador
- Fase de traducción
- Fase de ejecución
- Variable
- Constante
- Memoria

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

1. Un lugar en memoria donde puede almacenarse durante la ejecución de un programa un valor.
2. Un lugar en memoria donde se almacena un valor y no puede cambiarse durante la ejecución del programa.
3. La parte del ordenador que almacena programas y datos.
4. Un dispositivo de entrada a un ordenador.
5. El tiempo transcurrido en la planificación de un programa.
6. Reglas gramaticales.
7. Una estructura iterativa.
8. Significado.
9. Un programa que traduce instrucciones en lenguaje ensamblador a código máquina.
10. Cuando la versión en código máquina de un programa se está ejecutando.
11. Nombres simbólicos formados de letras, dígitos y que comienzan con una letra o subrayado.
12. Cuando se traduce un programa en un lenguaje de alto nivel a código máquina.
13. Un programa que toma un programa escrito en lenguaje de alto nivel y lo traduce a código máquina completo.
14. Descripción detallada de los pasos a realizar para resolver un problema en un número finito de pasos.
15. Implementación de un algoritmo en un lenguaje de programación concreto.

*SOLUCION:*

- 15 Programa
- 14 Algoritmo
- 13 Compilador
- 11 Identificador
- 12 Fase de traducción
- 10 Fase de ejecución
- 1 Variable
- 2 Constante
- 3 Memoria

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

5.- ¿Qué sacaría por pantalla la ejecución del siguiente programa?

```
#include <iostream.h>
void main() {
int primero, segundo, tercero;
cout << primero << '\n' << segundo << '\n' << tercero;
}
```

*SOLUCION:*

Mostraría el valor que contenga la posición de memoria que corresponda a primero con un salto de línea, el valor que contenga la posición de memoria que corresponda al segundo con un salto de línea y el valor contenido en la posición de memoria que corresponda al tercero. Los valores contenidos en la memoria seran valores extraños por no haber sido inicializada las variables.

Ejemplo:

```
-3240
61890
932
```

6.- Escriba un programa que almacene en cinco variables los valores 1.000, 1.414, 1.732, 2.000, 2.236, sacando en pantalla la siguiente información de las variables:

N	Raiz cuadrada
1	1.000
2	1.414
3	1.732
4	2.000
5	2.236

*SOLUCION:*

```
#include <iostream.h>

void main()
{
    float n1=1.000,n2=1.414,n3=1.732,n4=2.000,n5=2.236;

    cout<<"N\tRaiz Cuadrada\n"
        <<int((n1*n1)+1)<<'\t'<<n1<<'\n'
        <<int((n2*n2)+1)<<'\t'<<n2<<'\n'
        <<int((n3*n3)+1)<<'\t'<<n3<<'\n'
        <<int((n4*n4)+1)<<'\t'<<n4<<'\n'
        <<int((n5*n5)+1)<<'\t'<<n5;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA II: LOS LENGUAJES DE PROGRAMACIÓN

---

7.- Determinar los errores que hay en el siguiente programa:

```
#define n 12;
typedef int b;
void main() { ;
int a, c;
cin>>n;
cin>>a;
b=3;
cin>>c;
a*b*c;
cout<<a;
}
```

*SOLUCION:*

```
#define n 12; //sobra el punto y coma despues del 12
typedef int b;
void main() { ; //sobra el punto y coma después de la llave
int a, c;
cin>>n; //no se puede modificar el valor de una constante
cin>>a;
b=3; //no se le puede asignar un valor a un tipo
cin>>c;
a*b*c; //no se puede realizar la operación, b se trata de un tipo de dato
cout<<a;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

### INTRODUCCIÓN

A lo largo de este tema, vamos a ver las diversas estructuras de control con las cuales desarrollaremos nuestros programas, la mas simple de estas estructuras es la secuencial en la que las instrucciones se ejecutan una tras otra, pero la verdad es que desarrollar un programa tan solo con estructura secuencial es realmente complejo por no decir casi imposible, por tanto se recurre a otras estructuras además de la secuencial, y eso es lo que veremos a lo largo de este tema.

### ESTRUCTURA SECUENCIAL

Tal y como hemos expuesto en la introducción anterior, la estructura secuencial es la mas simple de todas, en ellas las instrucciones se ejecutan una tras otra hasta llegar a la ultima. Las instrucciones escritas secuencialmente en C/C++, han de terminar en punto y coma.

El formato de este tipo de estructura es el siguiente:

```
.....  
Sentencia1;  
Sentencia2;  
.....  
SentenciaN;  
.....
```

Un ejemplo de este tipo de estructura, podría ser como el siguiente:

Ejemplo:

```
void main()  
{  
    int a,b,c;  
  
    cin>>a>>b;  
    cout<<c=a+b;  
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

También, hemos de decir que las instrucciones también se pueden agrupar en lo que se conoce con el nombre de **bloque de sentencias** el cual se define introduciendo las sentencias que correspondan entre llaves. A continuación se muestra un ejemplo a modo de ilustrar lo aquí expuesto.

Ejemplo:

```
//BLOQUE DE SENTENCIAS  
.....  
{  
    sentencia1;  
    sentencia2;  
    .....  
    sentenciaN;  
    .....  
} //FIN DEL BLOQUE DE SENTENCIAS
```

### ESTRUCTURA CONDICIONAL (ESTRUCTURA ALTERNATIVA)

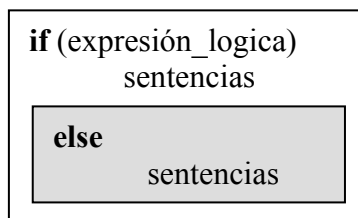
Como hemos visto en el caso de las estructuras secuenciales, en el cual todas las instrucciones se ejecutaban una tras otra hasta llegar a la última sin la posibilidad de que en nuestro algoritmo se pueda tomar la decisión de ir por un camino u otro del mismo.

Para suplir la falta de la toma de decisiones que presenta la estructura secuencial, surge el concepto de estructura condicional. Las estructuras condicionales lo que nos permiten es elegir una u otra sentencia en función a una serie de condiciones previamente definidas, según nuestro requerimiento.

A continuación veremos una serie de estructuras condicionales presentes en el lenguaje de programación C/C++.

#### ***Estructura condicional IF:***

El formato de la sintaxis de la estructura IF es el siguiente:



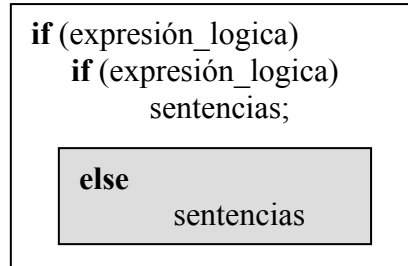
En esta estructura, las sentencias que están a continuación de la expresión lógica que se evalúa a continuación de la palabra reservada IF se ejecutarán siempre que la expresión lógica que se impone como condición cumple.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

También diremos que las estructuras IF también se pueden agrupar tal y como sucedía con los bloques de sentencias, por tanto nos aparece lo que se conoce con el nombre de **estructura condicional IF anidada**, cuyo formato es el siguiente:

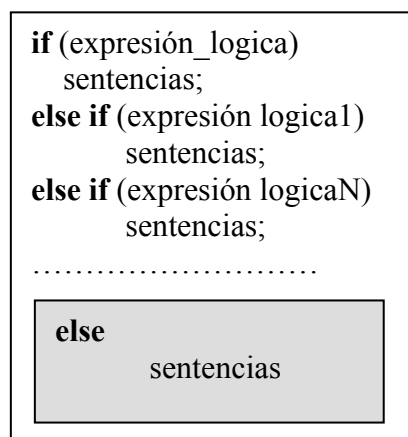


En este tipo de estructura, la sentencia **else** siempre se refiere al IF precedente mas próximo que no tenga ya asociada una sentencia IF. A continuación mostraremos un ejemplo de este tipo de estructura.

Ejemplo:

```
.....
if (x==y)
  if (z==t)
    cout<<"Primer texto";
  else //else correspondiente a if (z==t)
    cout<<"Segundo texto";
```

La última variante de la estructura IF, es la estructura condicional IF-ELSE IF la cual se podría definir como una mezcla de las dos estructuras anteriormente expuestas. El formato de esta estructura es como sigue a continuación:



# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

### **Estructura condicional SWITCH:**

Esta instrucción posee el siguiente formato:

```
switch (expresión_logica)
{
    case valor1:
        sentencias
        break;
    case valorN:
        sentencias
        break;
    .....
    default: //esta clausula es opcional
        sentencias
};
```

Esta instrucción es muy útil en programación para el uso de menús en los diversos programas que vayamos desarrollando a lo largo de nuestro curso. A continuación se muestra un ejemplo de lo que aquí acabamos de exponer.

Ejemplo:

```
.....
cout<<"Seleccione una opción: ";
cin>>opc;
switch(opc)
{
    case 1:
        cout<<"Opcion1";
        break;
    case 2:
        cout<<"Opcion2";
        break;
    .....
    default:
        cout<<"La opción seleccionada no es valida";
};
```

En esta instrucción las sentencias **break** se emplean para finalizar las secuencias de sentencias asociadas con el **case** correspondiente. Si se omite el **break**, la ejecución continúa hasta que alcance un **break** o hasta que llegue al final de la estructura **switch**.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

### ***Estructura condicional ?:***

El operador condicional “?”, sirve para escribir de una sentencia alternativa.

Su sintaxis es del siguiente modo:

`expresion1? expresion2:expresion3`

En este tipo de estructura, su funcionamiento es del siguiente modo. Se evalúa en un primer lugar, la expresión 1 y si esta es cierta (*diferente de cero*) entonces se evalúa la expresión 2 en cuyo caso de ser cierta, no se evaluaría la expresión 3; si por el contrario la expresión 2 fuera falsa, se evaluaría la expresión 3.

La equivalencia de esta estructura expresada con una estructura IF, sería:

```
if (expresion1)
    expresion2;
else
    expresion3;
```

A continuación mostramos un ejemplo comparativo de estructura condicional con IF y otra con ?.

Con IF	Con ?
<code>x = 5;</code> <code>if (x&lt;3)</code> <code>  cout&lt;&lt;20;</code> <code>else cout&lt;&lt;25;</code>	<code>x = 5;</code> <code>x&lt;3 ? cout&lt;&lt;20:cout&lt;&lt;25;</code>

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

### **ESTRUCTURA ITERATIVA**

Este tipo de estructura nos sirve para poder repetir un número de veces una parte de nuestro código. Imaginemos que por ejemplo queremos que nuestro programa le pida al usuario un número de veces una serie de datos, ello sería imposible con las estructuras estudiadas hasta ahora, por eso surgen las estructuras iterativas (estructuras repetitivas o bucles). Las estructuras iterativas tal y como dijimos al principio de este enunciado, permiten repetir una parte de código una serie de veces. En este apartado estudiaremos una serie de estructuras iterativas de las cuales nos provee el lenguaje de programación C/C++.

#### ***Estructura iterativa WHILE:***

El formato de esta estructura es el siguiente:

```
while (condición)
    sentencia_o_bloque_de_sentencias;
```

Esta estructura funciona del siguiente modo. En primer lugar se evalúa la condición para ver si cumple o no. En el caso de que la condición cumpla, se ejecuta el bloque de sentencias o la sentencia que se encuentra en el cuerpo del bucle. Si la condición no cumple, pasará a ejecutarse la siguiente instrucción tras el bucle, por tanto, esta instrucción podrá o no llegar a ejecutarse a lo largo de la vida de nuestro programa.

A continuación se muestra un ejemplo de la estructura aquí expuesta.

Ejemplo:

```
.....
a=1;
while(a==1) cout<<"Hola";
```

#### ***Estructura iterativa DO-WHILE:***

Es una variante de la estructura WHILE que vimos anteriormente. A muchas personas que hayan estudiado el lenguaje de programación PASCAL, esta estructura se asemeja al conocido REPEAT de PASCAL, con ello queremos decir que esta estructura se ejecutará al menos una vez en nuestro programa.

El formato de esta instrucción es el siguiente:

```
do
    sentencias
    .....
while (condición);
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

En esta instrucción primero se ejecutara el bloque de sentencias que se encuentren a continuación del **do** y una vez ejecutada la ultima instrucción y lleguemos al **while** la condición será evaluada, en el caso de que la condición sea cierta las sentencias que se encuentren el interior del cuerpo del bucle serán nuevamente ejecutadas. Si la condición que se encuentra en el **while** es falsa, entonces se continuara con las sentencias que se encuentren inmediatamente a continuación del **while**.

Tal y como hemos visto en esta explicación, esta instrucción se ejecutara al menos una vez aunque la condición no cumpla en un principio.

### *Estructura iterativa FOR:*

Esta instrucción se emplea para casos en los que de antemano conocemos el numero de veces que ha de iterar el programa en un punto.

Su formato es del siguiente modo:

```
for(inicializacion;condicion;incremento)
    sentencias
    .....
```

A continuación vamos a explicar brevemente que son cada elemento de los que esta compuesta esta instrucción.

La inicialización, es una asignación en la cual se especifica el valor inicial de la iteración.

La condición, es la evaluación para ver cuantas veces se ejecuta el bucle.

El incremento, es el valor que define como cambia la variable de control cada vez que itera el bucle.

A continuación mostraremos un ejemplo de esta estructura.

Ejemplo:

```
.....
suma=0;
for(i=1;i<=5;i++)
{
    cout<<"\nIndique un numero: ";
    cin>>num;
    suma=suma+num;
};
.....
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA III: ESTRUCTURAS DE CONTROL

---

### MACROS DE SENTENCIAS

Tal y como ya sabemos, la instrucción *#define* es usada para definir constantes dentro de un programa en C. Ya como sabemos, esta instrucción sustituye en nuestro programa antes del proceso de compilación todas las constantes simbólicas, reemplazando su texto equivalente, por su valor. Esto que acabamos de exponer, es realizado por el preprocesador de C, el cual es el encargado de realizar la primera etapa del proceso de compilación de un programa escrito en C.

Sin embargo, la instrucción *#define* puede ser empleada para definir algo mas que constantes simbólicas. Esta se puede usar para definir lo que se conoce con el nombre de **macros**; esto no es otra cosa que identificadores simples que son equivalentes a expresiones, instrucciones completas o a bloques de instrucciones.

A continuación se muestra un ejemplo de lo que aquí se acaba de decir.

Ejemplo:

```
#include <iostream.h>

#define area base * altura
void main()
{
    int base, altura;
    cout<<"Base: ";
    cin>>base;
    cout<<"Altura: ";
    cin>>altura;

    //se sustituye area por la expresión del define
    cout<<"\nArea = "<<area;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

### INTRODUCCION

Tal y como vimos en el tema 2 al hacer la clasificación de los tipos de datos, vimos que teníamos unos conocidos como tipos de datos estructurados, estos eran las TABLAS, REGISTROS, LISTAS, etc....

En este capítulo, veremos los REGISTROS y las TABLAS, centrándonos mayoritariamente en estas últimas, ya que será el tipo de estructuras que manejemos mayoritariamente en esta asignatura.

### REGISTROS

Un registro, es un tipo de dato estructurado, estático y heterogéneo.

Decimos que un registro es heterogéneo, porque estará formado por datos de distinto tipo.

Para definir un registro en C/C++, la sintaxis será tal que así:

```
struct nombre
{
    tipo_dato id_campo1;
    ....
    tipo_dato id_campoN;
};
```

A continuación explicaremos cada parte de la sintaxis:

- **nombre**: es el identificado que le daremos a nuestro registro.
- **tipo\_dato**: es el tipo de dato simple del campo del registro.
- **id\_campoX**: es el identificador del campo X del registro.

Un ejemplo de registro en C/C++ sería del siguiente modo.

Ejemplo:

```
.....
struct personas
{
    int DNI;
    int edad;
    ....
    double peso;
};
.....
personas persona;
//declaracion de una variable persona de tipo personas de tipo registro
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

### *Acceso a campo/s de un registro:*

La manera de acceder a un campo de un registro se hará primeramente mediante el nombre de la variable registro con un punto y a continuación, con el identificador del campo del registro.

```
nombre_vbl.nombre_campo;
```

A continuación mostramos un ejemplo que nos ilustrara mucho mejor lo que acabamos de exponer.

Ejemplo:

```
.....  
cout<<persona.edad;  
.....
```

En el ejemplo aquí expuesto, accedemos al campo edad del registro persona, que es de tipo personas, y mostramos el valor contenido en el campo por pantalla.

Las operaciones permitidas con un registro, serán aquellas que permitan realizarse con el tipo de dato de cada campo.

Ejemplo:

```
struct fechas  
{  
    int dia;  
    int mes;  
    int agno;  
};  
.....  
fechas fecha={31, 4, 1934}; //Inicializacion del registro  
  
fecha.dia=12; //Asignación
```

Otra manera de definir un tipo registro sera:

```
typedef  
    struct  
    {  
        Tipo_dato id_campo1;  
        .....  
        Tipo_dato id_campoN;  
    }nombre_registro;
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

Otra operación permitida con registros, es la copia directa mediante una simple asignación. El unico requisito que se ha de cumplir para poder copiar dos registros, es que ambos sean del mismo tipo.

Ejemplo:

```
.....  
Tipo_registro origen, destino;  
.....  
destino=origen;  
.....
```

En un registro, tambien se pueden hacer operaciones de lectura desde teclado, de igual modo que haciamos con una variable de un tipo de dato simple. La manera de leer un registro desde teclado es:

```
cin>>nombre_registro.nombre_campo;
```

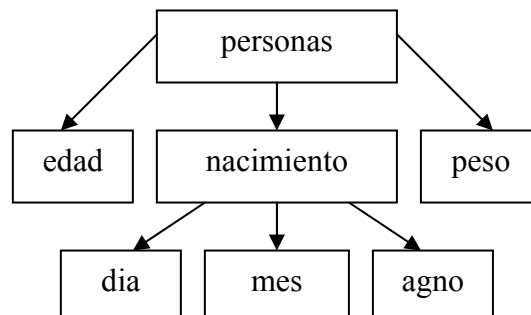
## REGISTROS JERARQUICOS

Un registro jerarquico, es aquel en cuyo interior contiene otro registro.

Ejemplo:

```
.....  
struct fechas  
{  
    int dia, mes, agno;  
};  
  
struct personas  
{  
    int edad;  
    fechas nacimiento;  
    float peso;  
};
```

El registro personas, visto de manera grafica, seria del siguiente modo:



# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

El acceso de un registro jerárquico será:

```
id_registro1.id_campoR1.id_campoR2;
```

Las operaciones que se pueden realizar son las mismas que para un registro simple.

## VECTORES Y MATRICES

### *Tablas:*

Una tabla es un tipo de dato estructurado, estático y homogéneo.  
Los elementos podrán ser de cualquier tipo.  
Los índices serán de tipo ordinal.

Ejemplo:

```
tipo id_tipo[num];  
double iva [100];//desde iva [0] hasta iva [99]
```

El acceso a un elemento de la tabla, será con el nombre de la variable y especificando la posición.

Ejemplo:

```
cout<< iva [3];
```

Las tablas pueden tener más de una dimensión y es lo que se conoce como tablas multidimensionales.

Su declaración será:

```
tipo_dato; nombre [a] [b] [c]...[N];
```

Ejemplo:

```
int tablero [10] [20];
```

Las tablas también podrán contener registros almacenados en sus elementos.

Ejemplo:

```
fechas calendario [100];  
.....  
cout<<calendario[5].dia;  
.....
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

La inicialización de una tabla, se puede realizar como en el caso de los registros.

```
tipo id_variable [a] [b].....[N]={0, 0,.....,0};
```

Ejemplo:

```
int tab2 [2] [3]={1,2,3,4,5,6};
```

De una manera gráfica, nuestra tabla sería así:

<b>tab2</b>			
	<b>0</b>	<b>1</b>	<b>2</b>
<b>0</b>	1	2	3
<b>1</b>	4	5	6

El relleno de datos se hace por filas.

### **Vectores:**

#### *Cadenas:*

- No se puede asignar cadenas.

Ejemplo: `cad1=cad2;` (\*NO ES VÁLIDO\*)

Se debe usar la función `strcpy`.

- No se puede comparar cadenas.

Ejemplo: `cad1==cad2` (\*NO ES VÁLIDO\*)

Se debe usar la función `strcmp`.

- No se puede asignar una cadena completa.

Salvo en la declaración.

Ejemplo: `cad1="Hola Mundo"`

Se debe usar la función `strcpy`.

#### *Dimensión de un vector:*

Son el número de índices que contiene.

Ejemplo:    Unidimensional → `vect [N];`  
              Bidimensional → `vect [N] [M];`  
              Multidimensional → `vect [N] [M] [X]...[..];`

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA IV: TIPOS DE DATOS ESTRUCTURADOS

---

*Tamaño de un vector:*

Es el número de elementos que puede tener la estructura.

Ejemplo:      tab [N] [M];  
                  Tamaño= N\*M;

### **BUCLES DE BÚSQUEDA Y RECORRIDO**

*Esquema de recorrido:*

Siempre que haya que pasar por todos los elementos obligatoriamente.

```
/* ESQUEMA DE RECORRIDO*/  
  
    declaración de variables;  
    for (i=0; i=N; i ++)  
        tratar elemento-i;  
    tto final
```

Ejemplo:

```
/*Recorrido para tablas de más de una dimensión*/  
int i, j, tabla [10] [15];  
for (i=0; i<10; i++)  
for(j=0; j<15; j++)  
    tto elemento-1;  
tto final
```

*Esquema de búsqueda:*

Cuando no hay que tratar todos los elementos.

```
declaración de variables;  
acceder primer elemento; /*i=0*/  
no encontrado; /*encontrado=0*/  
while ((i<N) & (encontrado= =0))  
    .....  
    if (elemento-i encontrado) encontrado = 1;  
    else acceder siguiente elemento; /*i++*/  
    .....  
if (encontrado= =1) tto elementos;  
tto elementos no encontrados;
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### INTRODUCCION

La metodología del diseño descendente, pretende subdividir un problema complejo, en otros problemas menores mas sencillos de abordar.

Por tanto, se podría definir el diseño descendente como el proceso mediante el cual un problema es descompuesto en una serie de subproblemas y estos a su vez en otros subproblemas mas simples de resolver. Al conjunto de todos los subproblemas parciales resueltos, constituirían la solución del problema global.

La programación estructurada, aborda este tipo de metodología de diseño para el desarrollo de programas complejos.

### PROGRAMACION ESTRUCTURADA

La programación estructurada, esta basada en la metodología del diseño descendente, que tal y como hemos visto simplemente se trata de subdividir un problema complejo en otros mas simples de resolver, con el fin de que el conjunto global de todas estas pequeñas soluciones, de una solución concreta al problema inicial.

A continuación vamos a mostrar un ejemplo a fin de ilustrar mejor lo aquí expuesto.

Ejemplo:

```
/*Programa sin uso de programación estructurada*/
#include<iostream.h>
int tab1[10],tab2[10];

void main()
{
    int i, encontrado;
    //Problema1: Cargar Tab1
    for(i=0;i<10;i++) cin>>tab1[i];
    //Problema2: Cargar Tab2
    for(i=0;i<20;i++) cin>>tab2[i];
    i=0; //Problema 3: Buscar si las tablas son distintas
    encontrado=0;
    while((i<10)&&(encontrado==0))
        if(tab1[i]!=tab2[i]) encontrado=1;
        else i++;
    if(encontrado==0) cout<<"Los vectores son distintos...";
    else cout<<"Los vectores son iguales...";
}
```

Tal y como podemos ver, este programa pudo ser subdividido en otra serie de subproblemas menores, tales como cargar las tablas o buscar si los dos vectores son distintos.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

Un programa escrito en C/C++, usando la programación estructurada, tendría la siguiente forma:

Prototipo de funciones //Cabeceras
Programa Principal //Llamadas a funciones
Declaracion de funciones //Desarrollo de las funciones

Mediante la metodología de la programación estructurada, se consigue expresar el programa principal, como una corta secuencia de sentencias que hacen llamada a subprogramas o funciones.

## FUNCIONES

Una función es un conjunto de instrucciones que realizan una tarea concreta y que puede ser llamada por el *main()* o por otras funciones.

La sintaxis de declaración de una función es:

```
Tipo_devuelto nombre_funcion()
{
    Declaraciones
    Sentencias
    .....
    return valor_devuelto;
}
```

La palabra **return** se usa para forzar una salida inmediata de la función.

### **OJO:**

**TODAS LAS SENTENCIAS DESPUES DE `return` NO SE EJECUTAN**

El valor devuelto por el valor de retorno, aparece sustituyendo al nombre de la función.

Las llamadas a funciones, se harán mediante el nombre de la función.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

El control del flujo de un programa cuando se produce una llamada a una función, se cede desde el *main* a la función llamada, de modo que se ejecutara las instrucciones agrupadas dentro de la definición de la función. Cuando se termine su ejecución, el control es devuelto al *main* en la siguiente instrucción tras la llamada a la función.

El orden en que sean declaradas las funciones es indiferente, siempre que no se llame a una función que no halla sido declarada previamente.

### **DECLARACIONES GLOBALES Y DECLARACIONES LOCALES. AMBITO DE VISIBILIDAD**

Una **declaracion local**, es aquella que se encuentra presente y declarada dentro de una función, bloque o clase.

Una **declaracion global**, es aquella que se encuentra declarada fuera de una función o de un bloque.

Por tanto, una **variable global** se define fuera de cualquier bloque, función o clase. Las variables locales son las que se definen dentro de una función o bloque.

En el caso de una **variable local**, se reserva espacio en memoria para esta variable, cuando comienza la ejecución de la función o bloque que la declara y desaparece cuando termina la ejecución de dicha función o bloque.

Entonces, diremos que se conoce como **ambito** a la zona de actuación de una declaración. Se dira que una declaración es visible desde una serie de lugares si se reconoce en esos lugares.

Las funciones tienen visibles sus variables locales y las globales.

El **ambito de una variable local** es exclusivamente dentro de la función o bloque que la declara.

El **ambito de una variable global**, es en todo el programa y en todas las funciones y bloques, excepto si dentro de una función o bloque existe una variable local con el mismo identificador que el de la variable global, en cuyo caso no estaría accesible la global.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### FUNCIONES Y METODOS: INTERFACES

El formato de una función genérica, ya sabemos como es, ya que ha sido expuesto anteriormente.

En C++, el centro del lenguaje, no son las funciones, sino los objetos. Y como ya sabemos de capítulos anteriores, el formato de un método es:

```
Tipo_devuelto nombre_clase::nombre_metodo()
{
    Declaraciones
    Sentencias
    .....
    return valor_devuelto;
}
```

Si nos fijamos, el formato de declaración de un método es el mismo que para el de una función.

En C++, una variable puede tener tres ámbitos:

- Local
- Global
- De Clase

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### **FUNCIONES inline**

Las funciones **inline** son un tipo de funciones especiales solo exclusivas de C++.

Este tipo de funciones son interesantes, en el caso de funciones muy cortas.

La sintaxis de una funcion **inline** es de dos formas:

```
//Forma1
inline dato_devuelto id_funcion()
{ return ....;
}

//Forma2
dato_devuelto id_funcion(); //Declaracion del prototipo
.....

inline dato_devuelto id_funcion() //Definición de la funcion
{
.....
}
```

A continuación vemos un ejemplo que muestra las dos formas de declarar una funcion *inline*:

Ejemplo:

```
//Forma1
inline int cuadrado()
{ return ...;
}
```

```
//Forma2
int cuadrado();
.....
inline int cuadrado()
{
.....
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### **METODOS inline**

A modo de ilustrar mejor los metodos *inline* emplearemos el siguiente ejemplo:

Ejemplo:

```
#include<iostream.h>

int dos();
int factorial(); //Prototipo de funciones

class complejo
{
    int real, imag;
public:
    void inicializa2(){real=2;imag=2;}//Metodo inline
    void inicializa1();
    void mostrar(){cout<<real<<"+"<<imag<<"i";}
};

inline void complejo::inicializa1()
{
    real=1; imag=1;
}
inline int dos(){return 2;}
int factorial()
{
    int fact=1;
    int n,i;
    cout<<"Indique el n° al que calcularemos el factorial: ";
    cin>>n;
    for(i=1;i<=n;i++) fact=fact * i;

    return fact;
}

void main()
{
    .....
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### PARAMETRIZACION. PARAMETROS POR VALOR Y POR REFERENCIA

El formato de función estudiado hasta ahora es el formato genérico, en el cual no existían los parámetros. Hasta este punto, solo hemos diseñado funciones genéricas sin parámetros, por una función podrá tener o no parámetros.

Los parámetros que aparecen en la declaración de una función se conocen con el nombre de **parámetros formales** y los que aparecen en la llamada a la función, son los **parámetros reales**. Para entender un poco mejor este concepto, lo ilustraremos con un ejemplo.

Ejemplo:

```
#include<iostream.h>
int lee_num(int n) {cin>>n;}  //(int n) parametro formal
.....
void main()
{
.....
int nume;
.....
lee_num(nume); //nume es el parametro real
.....
}
```

Por tanto, como hemos visto en el ejemplo, el formato de declaración de una función con parámetros será:

```
Tipo_devuelto id_funcion(tipo par1, tipo par2,...,tipo parN);
```

El valor de retorno de la función, aparecerá sustituyendo al nombre de la función, en el lugar donde se halla hecho la llamada a la función, una vez terminada la ejecución de la función.

Por tanto, y en base a lo anteriormente expuesto, el formato de declaración de una función miembro de una clase (método), será:

```
Tipo_devuelto nombre_clase::id_funcion(parámetros formales);
```

Los parámetros, no son otra cosa que tan solo valores que se le pasa a la función al ser llamada. Por tanto, los parámetros son el medio de comunicación existente entre el universo externo a la función y la función en sí.

Cada parámetro funciona dentro de una función, como una variable local, creándose cuando se entra en la función y destruyéndose cuando se sale de la misma.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

La relacion existente entre los parámetros formales y los reales, vendra dada por su orden de aparicion. Los parámetros formales son sustituidos por los reales, estos ademas han de coincidir en numero y tipo con los formales. (*NO SIEMPRE, HAY UNA EXCEPCION*).

A continuación, mostramos un ejemplo de funciones con parámetros.

Ejemplo:

```
#include<iostream.h>
class complejo
{
    int real, imag;
public:
    void inicializa(int re, int im){real=re;imag=im;}
    void inicializa0(){real=0;imag=0;}
    void incrementa(int re, int im){real=real+re;imag=imag+im;}
    void mostrar(){cout<<real<<'+'<<imag<<'i';}
};

void main()
{
    complejo ob,ob1;
    ob.inicializa(1,2);
    ob1.inicializa0();
    ob.incrementa(4,2);
    ob1.incrementa(1,2);
    ob.mostrar();
    ob1.mostrar();
}
```

Inicialmente y por defecto, todos los parámetros son pasados por **valor**, los parámetros que son pasados por **referencia** llevan antepuesto el símbolo **&**.

Si el parámetro es pasado por **valor**, se copia el valor del parámetro real en el correspondiente parámetro formal de la funcion.

Si el parámetro es pasado por **referencia**, se transfiere la propia variable al parámetro formal. Por tanto:

- Los **parámetros por valor ó copia** son aquellos para los que los cambios en el parámetro formal no afectan al real y por tanto los cambios dentro de la funcion no afectan a la variable real.
- Los **parámetros por referencia ó variable** son aquellos para los cuales los cambios en el parámetro formal, si afectan al real que esta fuera de la funcion.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

Mientras que el parámetro real correspondiente a un parámetro formal por valor, puede ser una constante, una variable o expresión del tipo especificado, el parámetro por referencia debe ser forzosamente una variable.

### REFERENCIAS:

```
int i=2;
int &irof=i; //irof es una referencia a i
irof=6; /*Al modificar el valor de irof y asignarle 6, lo que hacemos es
asignar el valor 6 a la variable i a la cual apunta la referencia*/
```

Como dato curioso y para concluir diremos que, una función puede tener como valor de retorno una variable de tipo referencia.

### Ejemplo:

```
int &maxref(int &a,int &b)
{
    if(a>=b) return a;
    else return b;
}
```

.....

```
maxref(i,j)=0; /*ES VALIDO. Asignaremos el valor 0 a la
variable la cual cumpla la condicion de retorno de la funcion*/
```

### Consejo para un buen diseño:

- Una función tendrá acceso solo a variables locales y a sus parámetros formales.
- Si es un método, también podrá acceder a sus datos miembro (públicos y privados).
- No se permitirá el acceso a variables externas directamente, aunque estas sean visibles.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### CONSTRUCTORES Y DESTRUCTORES

Los constructores son unas funciones miembro especiales que son llamadas automáticamente siempre que se crea un objeto de una clase. De esta forma cualquier inicialización que sea necesaria en un objeto, se hará en la función constructora.

Un destructor es llamado cuando un objeto va a dejar de existir por haber llegado al final de su vida y para realizar algunas acciones cuando el objeto se destruye. Su nombre es el mismo que el de la clase, pero precedido del carácter ~.

Ejemplo:

```
class lista
{
    .....
    public:
        lista(int a); //Constructor
        ~lista(); //Destructor
};
```

Para el caso de los objetos locales, el constructor se invoca cada vez que se cree el objeto, o sea, cada vez que el flujo del programa alcance su declaración, y su destructor se invoca cada vez que se destruye el objeto o dicho de otra forma, al terminar la función en la cual está declarada el objeto.

Para los objetos globales, el constructor es invocado una sola vez al inicio del programa antes de la ejecución del **main()** y su destructor, justo antes de que termine el programa.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

A fin de ilustrar un poco mejor lo aquí expuesto, mostraremos el siguiente ejemplo.

Ejemplo:

```
class complejo
{
    .....
    public:
        complejo(); //Constructor
        ~complejo(); //Destructor
        .....
};
void complejo::complejo()
{
    .....//Codigo de inicializacion
}
complejo a; //Declaracion de objeto global
void proceso()
{
    .....
    complejo b ; //Declaracion de objeto local
    .....
}
void complejo::~complejo()
{
    .....
    //Codigo del destructor
}
void main()
{
    complejo c; //Objeto local c
    proceso();
    complejo d; //Objeto local d
}
```

Si nos fijamos en el ejemplo expuesto, el flujo del programa sería del siguiente modo:

- 1- Constructor A
- 2- Destructor C
- 3- Constructor B
- 4- Destructor B
- 5- Constructor D
- 6- Destructor D
- 7- Destructor C
- 8- Destructor A

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### CONSTRUCTORES CON PARAMETROS

Los constructores pueden tener parámetros al igual que cualquier otra función. Esto permite inicializar el objeto con los valores que deseemos. Para ello debemos indicar los parámetros como argumentos en el momento de la declaración del objeto.

Ejemplo:

```
class complejo
{
    double real,imag;
    public:
        complejo(double re,double im){real=re;imag=im;}
};

void main()
{
    int n=4;
    complejo x(1,5), y(n+2,-3);
    .....
}
```

A diferencia del constructor, el destructor **no tiene argumentos en ningún caso**.

### INICIALIZADORES

Tal y como hemos visto, la idea del constructor es la de inicializar variables, pero C++ permite una forma distinta a la que hemos visto hasta ahora y mediante la instrucción de asignación ....=....

C++ permite inicializar variables miembro fuera del cuerpo del constructor, de la siguiente forma.

Ejemplo:

```
class complejo
{
    double real,imag;
    public:
        complejo(double re,double im);
        .....
};

complejo::complejo(double re,double im):real(re),imag(im)
{ //El cuerpo del constructor esta vacio }
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

Los inicializadores, son introducidos tras el símbolo (:) y separados por comas, justo antes de abrir las llaves del cuerpo del constructor. Los inicializadores, constan del nombre de la variable miembro seguido del argumento entre paréntesis.

### PASO DE ESTRUCTURAS DE DATOS COMPLEJOS A FUNCIONES Y METODOS

Los datos complejos tales como tablas, registros, objetos, etc... se pueden pasar como argumento a funciones, por defecto todos estos datos se pasan por valor, a excepción de las tablas que se pasan por referencia.

A continuación mostraremos un ejemplo para comprender lo aquí expuesto.

Ejemplo:

```
class objeto
{
    int n;
    public:
        objeto(int i){n=i;} //Funcion In-Line
        void set(int i){n=i;}
        int get(){return n;}
};

void cambia(objeto a, int valor)
{
    a.set(valor);
    cout<<"El valor A vale: "<<a.get()<<endl;
}

void main()
{
    objeto a(10);
    cambia(a,7);
    cout<<"Ahora vale: "<<a.get()<<endl;
}
```

Si quisieramos pasar el objeto se pase por referencia, la funcion quedaria escrita del siguiente modo:

```
void cambia(objeto &a, int valor)
{
    a.set(valor);
    cout<<"El parámetro A vale: "<<a.get()<<endl;
}
```

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

El formato de una función con un objeto pasador por referencia es el siguiente:

```
Tipo_devuelto id_funcion(vbl_objeto &id,.....)
{
    //Instrucciones de la funcion
}
```

Las cadenas y las matrices en C/C++ se pasan siempre por defecto por referencia, aunque no se le ponga el símbolo &.

Si pasamos un elemento de la tabla, este se puede pasar por valor, pero si es la tabla completa lo que pasamos, esta se pasara por referencia siempre.

Ejemplo:

```
class objeto
{
    int n;
    public:
        objeto(){n=0;}
        void set(int i){n=i;}
        int get(){return n;}
};

void main()
{
    objeto x[5];
    .....
}
```

A continuación mostramos otro ejemplo de paso de tablas de objetos como parámetro de una función.

Ejemplo:

```
void incrementa(objeto a[], int n)
{
    for(int i=0;i<5;i++) a[i].set(a[i].get()+n)
}
```

También hemos de decir que cuando pasamos un objeto por valor, no se llama a la función constructora, sin embargo cuando se termina la función, se destruye el objeto mediante una función destructora, pero por el contrario si pasamos el objeto por referencia, no se hace copia de este y cuando la función termina no se llamaría al destructor tal y como ocurría en el caso del paso de este por valor.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### **SOBRECARGA DE METODOS Y OPERADORES**

#### ***SOBRECARGA DE METODOS Y FUNCIONES***

En el lenguaje de programación C++, se pueden definir varias funciones con el mismo nombre, siempre y cuando varíe el número y el tipo de los argumentos. Es a esto a lo que se conoce con el nombre de *funciones sobrecargadas* (overload functions).

Es el compilador el que se encarga en tiempo de ejecución de seleccionar la función correcta, en base al número y tipo de los argumento/s pasados en la llamada de la función.

A continuación, vamos a mostrar un ejemplo de sobrecarga de funciones.

Ejemplo:

```
#include<iostream.h>
int abs(int n);
long abs(long n);
double abs(double n);
.....
{
.....
}
int abs(int n)
{
    return (n<0?-n:n);
}
long abs(long n)
{
    return(n<0?-n:n);
}
.....//Igual con el resto de funciones
```

Además de diferir en el tipo de datos, también hemos dicho que puede diferir en el número de sus argumentos.

Si solo difiere en el valor devuelto, el compilador no podría determinar qué función sobrecargada debería emplear, generando un error en este debido a un problema de *ambigüedad*.

C++, permite sobrecargar la función constructora de una clase, pero **UN DESTRUCTOR NUNCA PODRÁ SER SOBRECARGADO.**

Como dijimos anteriormente, cuando en un programa existen varias funciones sobrecargadas, se suele presentar el problema de la ambigüedad.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

Para que podamos compilar un programa, primero deberemos resolver los errores debido a la ambigüedad.

Una de las causas mas normales para que se produzca ambigüedad en un programa, es la conversión automática de tipos.

### **SOBRECARGA DE OPERADORES**

Los operadores en C++, a excepcion de (. :: .\* ?) podran ser sobrecargados del mismo modo que sucedia con las funciones y los metodos. Es decir, podremos redefinir para que actuen de una manera determinada para el tipo de dato para el cual este sobrecargado.

La sintaxis para definir un operador sobrecargado es la siguiente:

```
tipo_devuelto nombre_clase::operator#(argumentos)
{
    //Operaciones de la funcion
}

//El símbolo # indica el símbolo del operador a sobrecargar.
```

Cuando se sobrecarga un operador binario con una funcion operadora miembro, el operador izquierdo se pasa implícitamente a la funcion y el operando derecho se pasa como argumento.

Cuando se sobrecarga un operador unario, la funcion operadora miembro no tiene parámetro ya que el unico operando que hay se pasa implícitamente a la funcion.

A continuacion indicaremos una serie de cosas a tener en consideración a la hora de sobrecargar un operador.

#### ***Consideraciones:***

- No se puede cambiar su precedencia y asociatividad.
- No se puede modificar el numero de operandos.
- Es necesario que al menos un operando sea un objeto de la clase en la que se ha definido el operador subrayado.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### ARGUMENTOS IMPLICITOS

En C++, podemos definir valores por defecto para los argumentos formales, que serán tomados en caso que dicho argumento se encuentre ausente en la llamada a la función, de este modo con una misma función la cual tuviese argumentos implícitos, podríamos tener dos funciones en una. A continuación mostraremos un ejemplo de lo aquí expuesto, con el fin de facilitar su comprensión.

Ejemplo:

```
double potencia(int base, int x=1, int y=0)
.....
void main()
{
    potencia(10,2,3);
    potencia(10,7);
    potencia(10);
    .....
}
```

A continuación expresaremos la sintaxis de una función con parámetros implícitos.

```
tipo_devuelto id_funcion(tipo_datos vbl=valor_implicito)
{
    .....
}
```

### **Consideraciones:**

- Los argumentos implícitos deben estar al final de la lista de argumentos.
- En la llamada a la función, pueden omitirse los argumentos implícitos. Si se omite un argumento, deberán omitirse todos aquellos que este detrás.
- Los valores dados a los argumentos implícitos deben ser constantes o variables globales. No pudiendo ser en ningún caso variables globales u otros parámetros.
- Los argumentos implícitos de una función, solo deben especificarse una vez, bien en la definición de la función o bien en su prototipo, pero nunca en ambos.

# INTRODUCCIÓN A LA PROGRAMACIÓN

## TEMA V: DISEÑO DESCENDENTE

---

### PARAMETROS POR REFERENCIA CON “CONST”

Si pasamos un parámetro que es una constante por referencia, no se realizara modificacion alguna en dicho parámetro.

Si por ejemplo, paso una tabla dado que en C/C++ las tablas se pasan por referencia por defecto, y si modificamos algun dato contenido dentro de una tabla dentro de dicha funcion, cuando termine la ejecución de la funcion nos devolveria la tabla modificada, pues es por ello que usaremos **const** para evitar que los datos se vean modificados.

La sintaxis para una funcion según lo aquí expuesto, seria del siguiente modo:

```
tipo_devuelto id_funcion(tipo_dato id1, const tipo_dato id2[])  
{  
    .....  
}
```