Fundamentos de Programación

Curso 2015/2016

Apuntes confeccionados por
Juan Carlos Cubero.
Universidad de Granada.

El color marrón se utilizará para los títulos de las secciones, apartados, etc

El color azul se usará para los términos cuya definición aparece por primera vez. En primer lugar aparecerá el término en español y entre paréntesis la traducción al inglés.

El color rojo se usará para destacar partes especialmente importantes

Algunos símbolos usados:

Principio de Programación.

denota algo especialmente importante.

denota código bien diseñado que nos ha de servir de modelo en otras construcciones.

denota código o prácticas de programación que pueden producir errores lógicos graves

denota código que nos da escalofríos de sólo verlo.

denota código que se está desarrollando y por tanto tiene problemas de diseño.

denota un consejo de programación.

denota contenido de ampliación. No entra como materia en el examen.

Reseña histórica.

denota contenido que el alumno debe estudiar por su cuenta. Entra como materia en el examen.

Contenidos

Tema I

Introducción a la Programación

***Objetivos*:**

- ▷ **Introducir los conceptos básicos de programación, para poder construir los primeros programas.**
- ▷ **Introducir los principales tipos de datos disponibles en C++ para representar información del mundo real.**
- ▷ **Enfatizar, desde un principio, la necesidad de seguir buenos hábitos de programación.**

Autor: Juan Carlos Cubero.

Sugerencias: por favor, enviar un e-mail a JC.Cubero@decsai.ugr.es

I.1. El ordenador, algoritmos y programas

I.1.1. El Ordenador: Conceptos Básicos

"Los ordenadores son inútiles. Sólo pueden darte respuestas".
Pablo Picasso

- ▷ ***Hardware***
- ▷ ***Software***
- ▷ ***Usuario (User)***
- ▷ ***Programador (Programmer)***

I.1.2. Datos y Algoritmos

Algoritmo (Algorithm) **: es una secuencia ordenada de instrucciones que resuelve un problema concreto, atendiendo a las siguientes características:**

- ***Características básicas:***
  - **Corrección (sin errores).**
  - **Precisión (no puede haber ambigüedad).**
  - **Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).**
- ***Características esenciales:***
  - **Finitud (termina en algún momento). Número finito de órdenes no implica finitud.**
  - **Validez (resuelve el problema pedido)**
  - **Eficiencia (lo hace en un tiempo aceptable)**

Un *dato (data)* es una representación simbólica de una característica o propiedad de una entidad.

Los algoritmos operan sobre los datos. Usualmente, reciben unos *datos de entrada* con los que operan, y a veces, calculan unos nuevos *datos de salida*.

Ejemplo. **Algoritmo de la media aritmética de N valores.**

- ▷ ***Datos de entrada*****: valor1, valor2, ..., valorN**
- ▷ ***Datos de salida*****: media**
- ▷ ***Instrucciones en lenguaje natural*****:**
  Sumar los N valores y dividir el resultado por N

Ejemplo. **Algoritmo para la resolución de una ecuación de primer grado** $ax + b = 0$

- ▷ ***Datos de entrada*****:** a, b
- ▷ ***Datos de salida*****:** x
- ▷ ***Instrucciones en lenguaje natural*****:**
  Calcular x **como el resultado de la división** $-b/a$

Podría mejorarse el algoritmo contemplando el caso de ecuaciones degeneradas, es decir, con a **o** b **igual a cero**

Ejemplo. **Algoritmo para el cálculo de la hipotenusa de un triángulo rectángulo.**

- ▷ ***Datos de entrada*****: lado1, lado2**
- ▷ ***Datos de salida*****: hipotenusa**
- ▷ ***Instrucciones en lenguaje natural*****:**

$$\text{hipotenusa} = \sqrt{\text{lado1}^2 + \text{lado2}^2}$$

Ejemplo. **Algoritmo para ordenar un vector (lista) de valores numéricos.**

$$(9, 8, 1, 6, 10, 4) \longrightarrow (1, 4, 6, 8, 9, 10)$$

- ***Datos de entrada*****: el vector**
- ***Datos de salida*****: el mismo vector**
- ***Instrucciones en lenguaje natural*****:**
  - **Calcular el mínimo valor de todo el vector**
  - **Intercambiarlo con la primera posición**
  - **Volver a hacer lo mismo con el vector formado por todas las componentes menos la primera.**

$(9, 8, 1, 6, 10, 4) \rightarrow$

$(1, 8, 9, 6, 10, 4) \rightarrow$

$(X, 8, 9, 6, 10, 4) \rightarrow$

$(X, 4, 9, 6, 10, 8) \rightarrow$

$(X, X, 9, 6, 10, 8) \rightarrow$

$\ldots$

Instrucciones no válidas en un algoritmo:

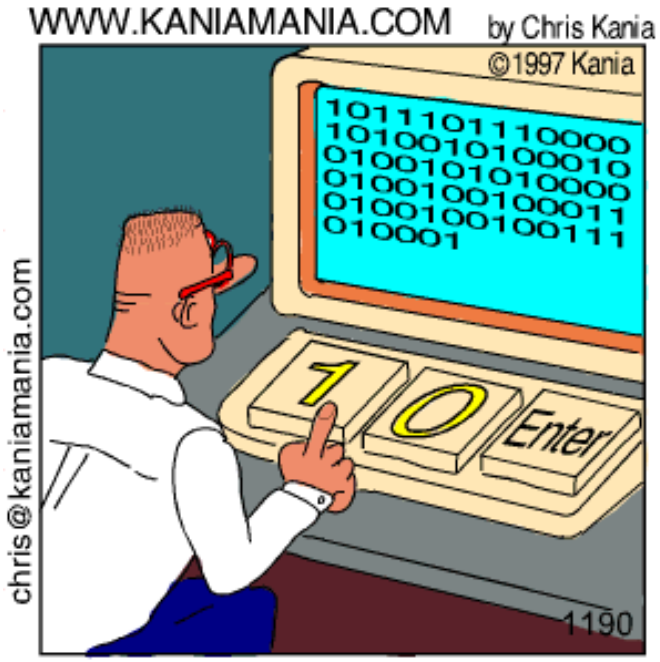
- **Calcular un valor *bastante* pequeño en todo el vector**
- **Intercambiarlo con el que está en una posición *adecuada***
- **Volver a hacer lo mismo con el vector formado por *la mayor parte* de las componentes.**

Una vez diseñado el algoritmo, debemos escribir las órdenes que lo constituyen en un lenguaje que entienda el ordenador.

"First, solve the problem. Then, write the code".

I.1.3. Lenguajes de programación

Código binario (Binary code) :



Real programmers code in binary.

"There are 10 types of people in the world, those who can read binary, and those who can't".

Lenguaje de programación (Programming language) **: Lenguaje formal utilizado para comunicarnos con un ordenador e imponerle la ejecución de un conjunto de órdenes.**

- ***Lenguaje ensamblador (Assembly language)*** **. Depende del microprocesador (Intel 8086, Motorola 88000, etc) Se usa para programar drivers, microcontroladores (que son circuitos integrados que agrupan microprocesador, memoria y periféricos), compiladores, etc. Se ve en otras asignaturas.**

```
.model small
.stack
.data
   Cadena1 DB 'Hola Mundo.$'
.code
   mov ax, @data
   mov ds, ax
   mov dx, offset Cadena1
   mov ah, 9
   int 21h
end
```

- ***Lenguajes de alto nivel (High level language)*** **(C, C++, Java, Lisp, Prolog, Perl, Visual Basic, C#, Go ...) En esta asignatura usaremos C++1114 (ISO C++).**

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

int main(){
   cout << "Hola Mundo";
}
```

Reseña histórica del lenguaje C++:

1967 Martin Richards: BCPL para escribir S.O.

1970 Ken Thompson: B para escribir UNIX (inicial)

1972 Dennis Ritchie: C

1983 Comité Técnico X3J11: ANSI C

1983 Bjarne Stroustrup: C++

1989 Comité técnico X3J16: ANSI C++

1990 Internacional Standarization Organization http://www.iso.org

Comité técnico JTC1: Information Technology

Subcomité SC-22: Programming languages, their environments and system software interfaces.

Working Group 21: C++

http://www.open-std.org/jtc1/sc22/wg21/

2011 Revisión del estándar con importantes cambios.

2014 Última revisión del estándar con cambios menores.

2017? Actualmente en desarrollo la siguiente versión C++ 17.

¿Qué programas se han hecho en C++?

Google, Amazon, sistema de reservas aéreas (Amadeus), omnipresente en la industria automovilística y aérea, sistemas de telecomunicaciones, el explorador Mars Rovers, el proyecto de secuenciación del genoma humano, videojuegos como Doom, Warcraft, Age of Empires, Halo, la mayor parte del software de Microsoft y una gran parte del de Apple, la máquina virtual Java, Photoshop, Thunderbird y Firefox, MySQL, OpenOffice, etc.

Implementación de un algoritmo (Algorithm implementation) **: Transcripción de un algoritmo a un lenguaje de programación.**

Ejemplo. **Implementación del algoritmo para el cálculo de la media de 4 valores en C++:**

```
suma = valor1 + valor2 + valor3 + valor4;
media = suma / 4;
```

Ejemplo. **Implementación del algoritmo para el cálculo de la hipotenusa:**

```
hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
```

Ejemplo. **Implementación del algoritmo para la ordenación de un vector.**

```
for (izda = 1; izda < total_utilizados; izda++){
   a_desplazar = vector[izda];

   for (i = izda; i>0 && a_desplazar < vector[i-1]; i--)
      vector[i] = vector[i-1];

   vector[i] = a_desplazar;
}
```

Estas instrucciones se escriben en un fichero de texto normal. Al código escrito en un lenguaje concreto se le denomina *código fuente (source code)* . En C++ llevan la extensión `.cpp`.

Para que las instrucciones anteriores puedan ejecutarse correctamente, debemos especificar dentro del código fuente los datos con los que vamos a trabajar, incluir ciertos recursos externos, etc. Todo ello constituye un programa:

Un *programa (program)* es un conjunto de instrucciones especificadas en un lenguaje de programación concreto, que pueden ejecutarse en un ordenador.

Ejemplo. Programa para calcular la hipotenusa de un triángulo rectángulo.

Pitagoras.cpp

```
/*
   Programa simple para el cálculo de la hipotenusa
   de un triángulo rectángulo, aplicando el teorema de Pitágoras
*/

#include <iostream>   // Inclusión de recursos de E/S
#include <cmath>      // Inclusión de recursos matemáticos
using namespace std;

int main(){               // Programa Principal
   double lado1;          // Declara variables para guardar
   double lado2;          // los dos lados y la hipotenusa
   double hipotenusa;

   cout << "Introduzca la longitud del primer cateto: ";
   cin >> lado1;
   cout << "Introduzca la longitud del segundo cateto: ";
   cin >> lado2;

   hipotenusa = sqrt(lado1*lado1 + lado2*lado2);

   cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

http://decsai.ugr.es/~carlos/FP/I_Pitagoras.cpp

La *programación (programming)* es el proceso de diseñar, codificar (implementar), depurar y mantener un programa.

Un programa incluirá la implementación de uno o más algoritmos.

***Ejemplo.* Programa para dibujar planos de pisos.**
Utilizará algoritmos para dibujar cuadrados, de medias aritméticas, salidas gráficas en plotter, etc.

Muchos de los programas que se verán en FP implementarán un único algoritmo.

I.1.4. Compilación

Para obtener el programa ejecutable (el fichero en binario que puede ejecutarse en un ordenador) se utiliza un *compilador (compiler)* :

La extensión en Windows de los programas ejecutables es `.exe`

Código Fuente: `Pitagoras.cpp`

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

int main() {
 double lado1;
 ..........
 cout << "Introduzca la longitud ...
 cin >> lado1;
 ..........
}
```

→ **Compilador** →

Programa Ejecutable: `Pitagoras.exe`

```
10011000010000
10010000111101
00110100000001
11110001011110
11100001111100
11100101011000
00001101000111
00011000111100
```

I.2. Especificación de programas

En este apartado se introducen los conceptos básicos involucrados en la construcción de un programa. Se introducen términos que posteriormente se verán con más detalle.

I.2.1. Organización de un programa

- **Los programas en C++ pueden dividirse en varios ficheros aunque por ahora vamos a suponer que cada programa está escrito en un único fichero (`Pitagoras.cpp`).**
- **Se pueden incluir comentarios en lenguaje natural.**

```
/* Comentario partido en
   varias líneas */
// Comentario en una sola línea
```

El texto de un comentario no es procesado por el compilador.

- **Al principio del fichero se indica que vamos a usar una serie de recursos definidos en un fichero externo o *biblioteca (library)***

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

También aparece

```
using namespace std;
```

La finalidad de esta declaración se verá posteriormente.

- **A continuación aparece `int main(){` que indica que comienza el programa principal. Éste se extiende desde la llave abierta `{`, hasta encontrar la correspondiente llave cerrada `}`.**
- **Dentro del programa principal van las sentencias. Una *sentencia***

(sentence/statement) **es una parte del código fuente que el compilador traduce en una instrucción en código binario. Ésta van obligatoriamente separadas por punto y coma** `;` **y se van ejecutando secuencialmente de arriba abajo. En el tema II se verá como realizar *saltos*, es decir, interrumpir la estructura secuencial.**

- **Cuando llega a la llave cerrada** `}` **correspondiente a** `main()`**, y si no han aparecido problemas, el programa termina de ejecutarse y el Sistema Operativo libera los recursos asignados a dicho programa.**

Veamos algunos tipos de sentencias usuales:

- ***Sentencias de declaración de datos***

  Un *dato (data)* es una unidad de información que representamos en el ordenador (longitud del lado de un triángulo rectángulo, longitud de la hipotenusa, nombre de una persona, número de habitantes, el número π, etc)

  El compilador ofrece distintos *tipos de datos (data types)* , como enteros (`int`), reales (`double`), caracteres (`char`), etc. En una sentencia de *declaración (declaration)* , el programador indica el nombre o *identificador (identifier)* que usará para referirse a un dato concreto y establece su tipo de dato, el cual no se podrá cambiar posteriormente.

  Cada dato que se desee usar en un programa debe declararse previamente. Por ahora, lo haremos al principio (después de `main`**),**

```
double lado1;
double lado2;
double hipotenusa;
```

  Declara tres datos de tipo real que el programador puede usar en el programa.

▷ *Sentencias de asignación*

A los datos se les asigna un *valor (value)* a través del denominado *operador de asignación (assignment operator)* = (no confundir con la igualdad en Matemáticas)

```
lado1 = 7;
lado2 = 5;
hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
```

Asigna `7` a `lado1`, `5` a `lado2` y asigna al dato `hipotenusa` el resultado de evaluar lo que aparece a la derecha de la asignación. Se ha usado el *operador (operator)* de multiplicación (`*`), el operador de suma (`+`) y la *función (function)* raíz cuadrada (`sqrt`). Posteriormente se verá con más detalle el uso de operadores y funciones.

Podremos cambiar el valor de los datos tantas veces como queramos.

```
lado1 = 7;   // lado1 contiene 7
lado1 = 8;   // lado1 contiene 8. Se pierde el antiguo valor (7)
```

▷ *Sentencias de entrada de datos*

¿Y si queremos asignarle a `lado1` un valor introducido por el usuario del programa?

Las sentencias de *entrada de datos (data input)* permiten leer valores desde el dispositivo de entrada establecido por defecto. Por ahora, será el teclado (también podrá ser un fichero, por ejemplo). Se construyen usando `cin`, que es un recurso externo incluido en la biblioteca `iostream`.

```
cin >> dato;
```

Por ejemplo, al ejecutarse la sentencia

```
cin >> lado1;
```

el programa espera a que el usuario introduzca un valor real (`double`) desde el teclado (dispositivo de entrada) y, cuando se pul-

sa la tecla `Intro`, lo almacena en el dato `lado1` (si la entrada es desde un fichero, no hay que introducir `Intro`). Conforme se va escribiendo el valor, éste se muestra en pantalla, incluyendo el salto de línea.

> ***La lectura de datos con `cin` puede considerarse como una asignación en tiempo de ejecución***

▷ *Sentencias de salida de datos*

Por otra parte, las sentencias de *salida de datos (data output)* permiten escribir mensajes y los valores de los datos en el dispositivo de salida establecido por defecto. Por ahora, será la pantalla (podrá ser también un fichero). Se construyen usando `cout`, que es un recurso externo incluido en la biblioteca `iostream`.

```
cout << "Este texto se muestra tal cual " << dato;
```

- **Lo que haya dentro de un par de comillas dobles se muestra tal cual, excepto los caracteres precedidos de `\`. Por ejemplo, `\n` hace que el cursor salte al principio de la línea siguiente.**

```
cout << "Bienvenido. Salto a la siguiente linea ->\n";
cout << "\n<- Empiezo en una nueva linea.";
```

- **Los números se escriben tal cual (decimales con punto)**

```
cout << 3.1415927;
```

- **Si ponemos un dato, se imprime su contenido.**

```
cout << hipotenusa;
```

Podemos usar una única sentencia:

```
cout << "\nLa hipotenusa vale " << hipotenusa;
```

Realmente, hay que anteponer el *espacio de nombres (namespace)* `std` en la llamada a `cout` y `cin`:

```
std::cout << variable;
```

Al haber incluido `using namespace std;` al inicio del programa, ya no es necesario. Los namespaces sirven para organizar los recursos (funciones, clases, etc) ofrecidos por el compilador o construidos por nosotros. La idea es similar a la estructura en carpetas de los ficheros de un sistema operativo. En FP no los usaremos.

Estructura básica de un programa (los corchetes delimitan secciones opcionales):

[/* Breve descripción en lenguaje natural
de lo que hace el programa */]
[*Inclusión de recursos externos*]
[`using namespace std;` **]**
`int main(){`
[*Declaración de datos*]
[*Sentencias del programa separadas por ;*]
`}`

I.2.2. Elementos básicos de un lenguaje de programación

I.2.2.1. Tokens y reglas sintácticas

A la hora de escribir un programa, cada lenguaje de programación tiene una sintaxis propia que debe respetarse. Ésta queda definida por:

a) **Los *componentes léxicos (tokens)* . Formados por caracteres alfanuméricos y/o simbólicos. Representan la unidad léxica mínima que el lenguaje entiende.**

`main` `;` `(` `==` `=` `hipotenusa` `*` `/*`

Pero por ejemplo, ni `¡` **ni** `((*` **ni** `/ *` **son tokens válidos.**

b) ***Reglas sintácticas (Syntactic rules)* : determinan cómo han de combinarse los tokens para formar sentencias. Algunas se especifican con tokens especiales (formados usualmente por símbolos):**

- **Separador de sentencias** `;`
- **Para agrupar varias sentencias se usa** `{ }`
  Se verá su uso en el tema II. Por ahora, sirve para agrupar las sentencias que hay en el programa principal
- **Para agrupar expresiones (fórmulas) se usa** `( )`

```
sqrt( (lado1*lado1) + (lado2*lado2) );
```

I.2.2.2. Palabras reservadas

Suelen ser tokens formados por caracteres alfabéticos.
Tienen un significado específico para el compilador, y por tanto, el programador no puede definir datos con el mismo identificador.

Algunos usos:

- ▷ `main` **(formalmente,** `main` **no es una palabra reservada, pero a efectos prácticos, así lo consideraremos)**
- ▷ **Para definir tipos de datos como por ejemplo** `double`
- ▷ **Para establecer el *flujo de control (control flow)* , es decir, para especificar el orden en el que se han de ejecutar las sentencias, como** `if`, `while`, `for` **etc.**
  Estos se verán en el tema II.

Palabras reservadas comunes a C (C89) 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 reservadas adicionales 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

I.2.3. Tipos de errores en la programación

▷ ***Errores en tiempo de compilación (Compilation error)***

Ocasionados por un fallo de sintaxis en el código fuente.

No se genera el programa ejecutable.

```
/*  CONTIENE ERRORES  */
#include <iostre am>

USING namespace std;

int main{}(
   double main;
   double lado1:
   double lado 2,
   double hipotenusa:

   2 = lado1;
   lado1 = 2
   hipotenusa = sqrt(lado1*lado1 + ladp2*ladp2);
   cout << "La hipotenusa vale  << hipotenusa;
)
```

"Software and cathedrals are much the same. First we build them, then we pray".

▷ ***Errores en tiempo de ejecución (Execution error)***

Se ha generado el programa ejecutable, pero se produce un error durante la ejecución.

```
int dato_entero;
int otra_variable;

dato_entero = 0;
otra_variable = 7 / dato_entero;
```

▷ ***Errores lógicos (Logic errors)***

Se ha generado el programa ejecutable, pero el programa ofrece una solución equivocada.

```
.........
lado1 = 4;
lado2 = 9;
hipotenusa = sqrt(lado1+lado1 + lado2*lado2);
.........
```

I.2.4. Cuidando la presentación

Además de generar un programa sin errores, debemos asegurar que:

- **El código fuente sea fácil de leer por otro programador.**
- **El programa sea fácil de manejar por el usuario.**

I.2.4.1. Escritura de código fuente

A lo largo de la asignatura veremos normas que tendremos que seguir para que el código fuente que escribamos sea fácil de leer por otro programador. Debemos usar espacios y líneas en blanco para separar tokens y grupos de sentencias. El compilador ignora estos separadores pero ayudan en la lectura del código fuente.

> ***Para hacer más legible el código fuente, usaremos separadores como el espacio en blanco, el tabulador y el retorno de carro***

Este código fuente genera el mismo programa que el de la página 10 pero es mucho más difícil de leer.

```
#include<iostream>#include<cmath>using namespace std;
int main(){
   double lado1;double lado2;
   double hipotenusa;
   cout<<"Introduzca la longitud del primer cateto: ";
   cin>>lado1;
   cout<<"Introduzca la longitud del segundo cateto: ";cin>>lado2;
   hipotenusa=sqrt(lado1*lado1+lado2*lado2);
   cout<<"\nLa hipotenusa vale "<<hipotenusa;
}
```

I.2.4.2. Etiquetado de las Entradas/Salidas

Es importante dar un formato adecuado a la salida de datos en pantalla. El usuario del programa debe entender claramente el significado de todas sus salidas.

```
totalVentas = 45;
numeroVentas = 78;
cout << totalVentas << numeroVentas;  // Imprime 4578
```

```
cout << "\nSuma total de ventas = " << totalVentas;
cout << "\nNúmero total de ventas = " << numeroVentas;
```

Las entradas de datos también deben etiquetarse adecuadamente:

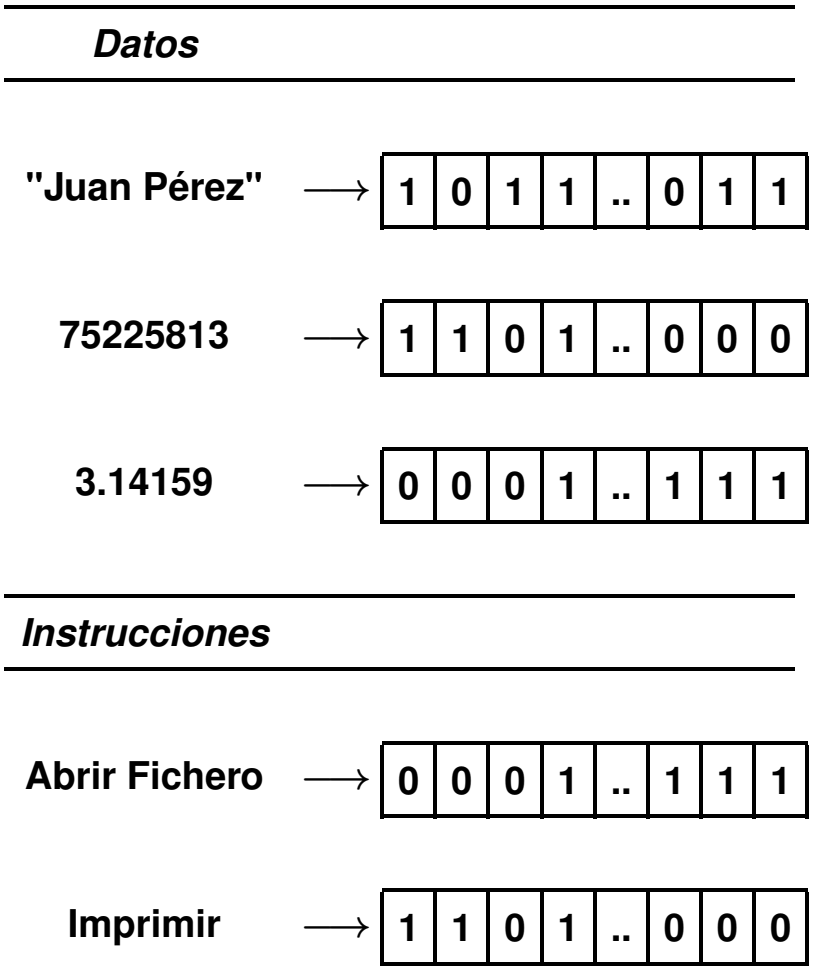
```
cin >> lado1;
cin >> lado2;
```

```
cout << "Introduzca la longitud del primer cateto: ";
cin >> lado1;
cout << "Introduzca la longitud del segundo cateto: ";
cin >> lado2;
```

I.3. Datos y tipos de datos

I.3.1. Representación en memoria de datos e instrucciones

Tanto las instrucciones como los datos son combinaciones adecuadas de 0 y 1.



Nos centramos en los datos.

I.3.2. Datos y tipos de datos

I.3.2.1. Declaración de datos

Al trabajar con un lenguaje de alto nivel, no haremos referencia a la secuencia de 0 y 1 que codifican un valor concreto, sino a lo que representa para nosotros.

Nombre de empleado: Juan Pérez

Número de habitantes : 75225813

π: 3.14159

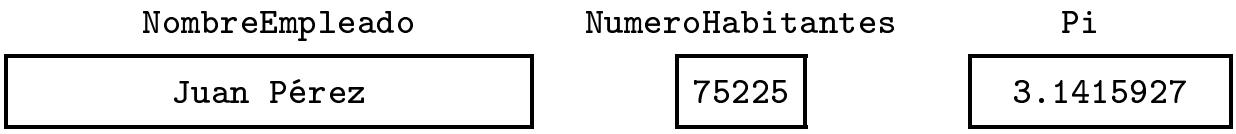
Un programa necesitará representar información de diverso tipo (cadenas de caracteres, enteros, reales, etc) Cada lenguaje de programación ofrece sus propios tipos de datos, denominados *tipos de datos primitivos (primitive data types)* . Por ejemplo, en C++: `string`, `int`, `double`, etc. Además, el programador podrá crear sus propios tipos usando otros recursos, como por ejemplo, las *clases* (ver tema III).

***Declaración de un dato (data declaration)* : Es la sentencia en la que se asigna un nombre a un dato y se le asocia un tipo de dato. El valor que se le puede asignar a un dato depende del tipo de dato con el que es declarado.**

Cuando se declara un dato de un tipo primitivo, el compilador reserva una zona de memoria para trabajar con ella. Ningún otro dato podrá usar dicha zona.

```
string NombreEmpleado;
int    NumeroHabitantes;
double Pi;

NombreEmpleado = "Pedro Ramírez";
NombreEmpleado = "Juan Pérez";    // Se pierde el antiguo
NombreEmpleado = 37;              // Error de compilación
NumeroHabitantes = "75225";       // Error de compilación
NumeroHabitantes = 75225;
Pi = 3.14156;
Pi = 3.1415927;                   // Se pierde el antiguo
```



Podemos declarar varias variables de un mismo tipo en la misma línea. Basta separarlas con coma:

```
double lado1, lado2, hipotenusa;
```

Opcionalmente, se puede dar un valor inicial durante la declaración.

```
<tipo> <identificador> = <valor_inicial>;

double dato = 4.5;
```

equivale a:

```
double dato;
dato = 4.5;
```

Cada dato necesita un identificador único. Un identificador de un dato es un token formado por caracteres alfanuméricos con las siguientes restricciones:

- **Debe empezar por una letra o subrayado (_)**
- **No pueden contener espacios en blanco ni ciertos caracteres especiales como letras acentuadas, la letra eñe, las barras \ o /, etc. Ejemplo:** `lado1` `lado2` `precio_con_IVA`
- **El compilador determina la máxima longitud que pueden tener (por ejemplo, 31 caracteres)**
- **Sensible a mayúsculas y minúsculas.** `lado` **y** `Lado` **son dos identificadores distintos.**
- **No se podrá dar a un dato el nombre de una palabra reservada. No es recomendable usar el nombre de algún identificador usado en las *bibliotecas estándar* (por ejemplo,** `cout`**)**

```
#include <iostream>
using namespace std;
int main(){
   double cout; // Error de sintaxis
   double main; // Error de sintaxis
```

Ejercicio. **Determinar cuáles de los siguientes son identificadores válidos. Si son inválidos explicar por qué.**

a) `registro1` **b)** `1registro` **c)** `archivo_3` **d)** `main`

e) `nombre y direccion` **f)** `dirección` **g)** `diseño`

Cuando se declara un dato y no se inicializa, éste no tiene ningún valor asignado *por defecto*. El valor almacenado es *indeterminado* y puede variar de una ejecución a otra del programa. Lo representaremos gráficamente por ?

***Ejemplo.* Calculad la cuantía de la retención a aplicar sobre el sueldo de un empleado, sabiendo el porcentaje de ésta.**

```
/*
   Programa para calcular la retención a aplicar en
   el sueldo de un empleado
*/
#include <iostream>
using namespace std;

int main(){
   double salario_bruto; // Salario bruto, en euros
   double retencion;     // Retención a aplicar, en euros
```

salario_bruto	retencion
?	?

```
   cout << "Introduzca salario bruto: ";
   cin >> salario_bruto;                  // El usuario introduce 32538

   retencion = salario_bruto * 0.18;
   cout << "\nRetención a aplicar: " << retencion;
}
```

salario_bruto	retencion
32538.0	4229.94

Un error *lógico* muy común es usar un dato no asignado:

```
int main(){
   double salario_bruto; // Salario bruto, en euros
   double retencion;     // Retención a aplicar, en euros


   retencion = salario_bruto * 0.18;   // salario_bruto indeterminado


   cout << "Retención a aplicar: " << retencion;
}
```

Imprimirá un valor indeterminado.

I.3.2.2. Literales

Un *literal (literal)* es la especificación de un valor concreto de un tipo de dato. Dependiendo del tipo, tenemos:

- ▷ ***Literales numéricos (numeric literals)* : son tokens numéricos. Para representar datos reales, se usa el punto `.` para especificar la parte decimal:**

  ```
  2    3    3.1415927
  ```

- ▷ ***Literales de cadenas de caracteres (string literals)* : Son cero o más caracteres encerrados entre comillas dobles:**

  ```
  "Juan Pérez"
  ```

- ▷ **Literales de otros tipos, como *literales de caracteres (character literals)* `'a'`, *Literales lógicos (boolean literals)* `true`, etc.**

I.3.2.3. Datos constantes

Podríamos estar interesados en usar datos a los que sólo permitimos tomar un único valor, fijado de antemano. Es posible con una *constante (constant)* . Se declaran como sigue:

```
const <tipo> <identif> = <expresión>;
```

- **A los datos no constantes se les denomina *variables (variables)* .**
- **A las constantes se les aplica las mismas consideraciones que hemos visto sobre tipo de dato y reserva de memoria.**
- **Los identificadores de las constantes suelen ser sólo en mayúsculas para diferenciarlos de las variables.**

Ejemplo. **Calculad la longitud de una circunferencia y el área de un círculo, sabiendo su radio.**

```
/*
   Programa que pide el radio de una circunferencia
   e imprime su longitud y el área del círculo
*/
#include <iostream>
using namespace std;

int main() {
   const double PI = 3.1416;
   double area, radio, longitud;

   // PI = 3.15;   <- Error de compilación

   cout << "Introduzca el valor del radio ";
   cin >> radio;

   area = PI * radio * radio;
   longitud = 2 * PI * radio;

   cout << "\nEl área del círculo es: " << area;
   cout << "\nLa longitud de la circunferencia es: " << longitud;
}
```

http://decsai.ugr.es/~carlos/FP/I_circunferencia.cpp

Comparar el anterior código con el siguiente:

```
   ......
   area = 3.1416 * radio * radio;
   longitud = 2 * 3.1416 * radio;
```

Ventajas al usar constantes:

- **El nombre dado a la constante (`PI`) proporciona más información al programador y hace que el código sea más legible.**

  Esta ventaja podría haberse conseguido usando un dato variable, pero entonces podría cambiarse su valor por error dentro del código. Al ser constante, su modificación no es posible.

- **Código menos propenso a errores. Para cambiar el valor de `PI` (a `3.1415927`, por ejemplo), sólo hay que modificar la línea de la declaración de la constante.**

  Si hubiésemos usado literales, tendríamos que haber recurrido a un cut-paste, muy propenso a errores.

Fomentaremos el uso de datos constantes en vez de literales para representar toda aquella información que sea constante durante la ejecución del programa.

Ejercicio. **Modificad el ejemplo de la página 28 introduciendo una constante que contenga el valor del porcentaje de la retención (`0.18`).**

```
/*
   Programa para calcular la retención a aplicar en
   el sueldo de un empleado
*/
#include <iostream>
using namespace std;

int main(){
   const double PORCENTAJE_RETENCION = 0.18;  // Porcentaje de retención
   double retencion;          // Retención a aplicar, en euros
   double salario_bruto;      // Salario bruto, en euros
```

salario_bruto	PORCENTAJE_RETENCION	retencion
?	0.18	?

```
   cout << "Introduzca salario bruto: ";
   cin >> salario_bruto;

   retencion = salario_bruto * PORCENTAJE_RETENCION;

   cout << "\nRetención a aplicar: " << retencion;
}
```

salario_bruto	PORCENTAJE_RETENCION	retencion
32538.0	0.18	4229.94

http://decsai.ugr.es/~carlos/FP/I_retencion.cpp

Una sintaxis de programa algo más completa:

[/* Breve descripción en lenguaje natural
de lo que hace el programa */]

[*Inclusión de recursos externos*]

[`using namespace std;` **]**

```
int main(){
```

[*Declaración de constantes*]

[*Declaración de variables*]

[*Sentencias del programa separadas por ;*]

```
}
```

I.3.2.4. Codificando con estilo

- **El identificador de un dato debe reflejar su semántica (contenido). Por eso, salvo excepciones (como las variables contadoras de los bucles -tema II-) no utilizaremos nombres con pocos caracteres**

   `v, l1, l2, hp`

   `voltaje, lado1, lado2, hipotenusa`

- **No utilizaremos nombres genéricos**

   `aux, vector1`

   `copia, calificaciones`

- **Usaremos minúsculas para los datos variables. Las constantes se escribirán en mayúsculas. Los nombres compuestos se separarán con subrayado _:**

   `precioventapublico, tasaanual`

   `precio_venta_publico, tasa_anual`

- **No se nombrarán dos datos con identificadores que difieran únicamente en la capitalización, o un sólo carácter.**

   `cuenta, cuentas, Cuenta`

   `cuenta, coleccion_cuentas, cuenta_ppal`

  Hay una excepción a esta norma. Cuando veamos las clases, podremos definir una clase con el nombre `Cuenta` **y una instancia de dicha clase con el nombre** `cuenta`**.**

- **Cuando veamos clases, funciones y métodos, éstos se escribirán con la primera letra en mayúscula. Los nombres compuestos se separarán con una mayúscula.**

   **Clase** `CuentaBancaria`**, Método** `Ingresa`

En el examen, se baja puntos por no seguir las anteriores normas.

I.4. Operadores y expresiones

I.4.1. Expresiones

Una *expresión (expression)* es una combinación de datos y operadores sintácticamente correcta, que devuelve un valor. El caso más sencillo de expresión es un literal o un dato:

```
3
3 + 5
lado1
```

La aplicación de un operador sobre uno o varios datos es una expresión:

```
lado1 * lado1
lado2 * lado2
```

En general, los operadores y funciones se aplican sobre expresiones y el resultado es una expresión:

```
lado1 * lado1  +  lado2 * lado2
sqrt(lado1 * lado1  +  lado2 * lado2)
```

Una expresión NO es una sentencia de un programa:

- **Expresión:** `sqrt(lado1*lado1 + lado2*lado2)`
- **Sentencia:** `hipotenusa = sqrt(lado1*lado1 + lado2*lado2);`

Las expresiones pueden aparecer a la derecha de una asignación, pero no a la izquierda.

```
3 + 5 = lado1;  // Error de compilación
```

Todo aquello que puede aparecer a la izquierda de una asignación se conoce como *l-value* (left) y a la derecha *r-value* (right)

Cuando el compilador evalúa una expresión, devuelve un valor de un tipo de dato (entero, real, carácter, etc.). Diremos que la expresión es de dicho tipo de dato. Por ejemplo:

`3 + 5` **es una expresión entera**

`3.5 + 6.7` **es una expresión de reales**

Cuando se usa una expresión dentro de `cout`, el compilador detecta el tipo de dato resultante y la imprime de forma adecuada.

```
cout << "\nResultado = " << 3 + 5;
cout << "\nResultado = " << 3.5 + 6.7;
```

Imprime en pantalla:

```
Resultado = 8
Resultado = 10.2
```

A lo largo del curso justificaremos que es mejor no incluir expresiones dentro de las instrucciones `cout` (más detalles en la página 95). Mejor guardamos el resultado de la expresión en una variable y mostramos la variable:

```
suma = 3.5 + 6.7;
cout << "\nResultado = " << suma;
```

> ***Evitad la evaluación de expresiones en una instrucción de salida de datos. Éstas deben limitarse a imprimir mensajes y el contenido de las variables.***

I.4.2. Terminología en Matemáticas

Notaciones usadas con los operadores matemáticos:

- ▷ ***Notación prefija (Prefix notation)*** **. El operador va antes de los argumentos. Estos suelen encerrarse entre paréntesis.**
  `seno(3), tangente(x), media(valor1, valor2)`
- ▷ ***Notación infija (Infix notation)*** **. El operador va entre los argumentos.**
  `3+5 x/y`

Según el número de argumentos, diremos que un operador es:

- ▷ ***Operador unario (Unary operator)*** **. Sólo tiene un argumento:**
  `seno(3), tangente(x)`
- ▷ ***Operador binario (Binary operator)*** **. Tienes dos argumentos:**
  `media(valor1, valor2) 3+5 x/y`
- ▷ ***Operador n-ario (n-ary operator)*** **. Tiene más de dos argumentos.**

I.4.3. Operadores en Programación

Los lenguajes de programación proporcionan operadores que permiten manipular los datos.

- ▷ **Se denotan a través de tokens alfanuméricos o simbólicos.**
- ▷ **Suelen devolver un valor.**

Tipos de operadores:

- ▷ **Los definidos en el núcleo del compilador.**

  No hay que incluir ninguna biblioteca

  Suelen usarse tokens simbólicos para su representación. Ejemplos:

  `+` **(suma),** `-` **(resta),** `*` **(producto), etc.**

  Los operadores binarios suelen ser infijos:

```
3 + 5
lado * lado
```

- ▷ **Los definidos en bibliotecas externas.**

  Por ejemplo, `cmath`

  Suelen usarse tokens alfanuméricos para su representación. Ejemplos:

  `sqrt` **(raíz cuadrada),** `sin` **(seno),** `pow` **(potencia), etc.**

  Suelen ser prefijos. Si hay varios argumentos se separan por una coma.

```
sqrt(4.2)
sin(6.4)
pow(3 , 6)
```

Tradicionalmente se usa el término *operador (operator)* a secas para denotar los primeros, y el término *función (function)* para los segundos.

A los argumentos de las funciones se les denomina *parámetros (parameter)* .

Una misma variable puede aparecer a la derecha y a la izquierda de una asignación:

```
double dato;

dato = 4;           // dato contiene 4
dato = dato + 3;    // dato contiene 7
```

En una sentencia de asignación

```
variable = <expresión>
```

primero se evalúa la expresión que aparece a la derecha y luego se realiza la asignación.

Ejercicio. **Construid un programa en C++ para que lea desde teclado un valor de aceleración y masa de un objeto y calcule la fuerza correspondiente según la segunda ley de Newton:**

$$F = m * a$$

I.5. Tipos de datos simples en C++

El comportamiento de un tipo de dato viene dado por:

- **El *rango (range)* de valores que puede representar, que depende de la cantidad de memoria que dedique el compilador a su representación interna. Intuitivamente, cuanta más memoria se dedique para un tipo de dato, mayor será el número de valores que podremos representar.**
- **El conjunto de operadores que pueden aplicarse a los datos de ese tipo.**

> ***A lo largo de este tema se verán operadores y funciones aplicables a los distintos tipos de datos. No es necesario aprenderse el nombre de todos ellos pero sí saber cómo se usan.***

I.5.1. Los tipos de datos enteros

I.5.1.1. Representación de los enteros

***Propiedad fundamental*: Cualquier entero puede descomponerse como la suma de determinadas potencias de 2.**

$$53 = 0*2^{15}+0*2^{14}+0*2^{13}+0*2^{12}+0*2^{11}+0*2^{10}+0*2^{9}+0*2^{8}+0*2^{7}+$$

$$+0*2^{6}+1*2^{5}+1*2^{4}+0*2^{3}+1*2^{2}+0*2^{1}+1*2^{0}$$

La representación en binario sería la secuencia de los factores (1,0) que acompañan a las potencias:

$$0000000000110101$$

- ▷ **Dos elementos a combinar:** `1, 0`
- ▷ r **posiciones. Por ejemplo,** $r = 16$
- ▷ **Se permiten repeticiones e importa el orden**

  $$0000000000110101 \neq 0000000000110110$$
- ▷ **Número de datos distintos representables =** 2^r

Se conoce como *bit* a la aparición de un valor 0 o 1. Un *byte* es una secuencia de 8 bits.

Esta representación de un entero es válida en cualquier lenguaje de programación.

I.5.1.2. Rango de los enteros

El rango de un *entero (integer)* es un subconjunto del conjunto matemático Z. La cardinalidad dependerá del número de bits (r) que cada compilador utiliza para su almacenamiento.

Los compiladores suelen ofrecer distintos tipos enteros. En C++: `short`, `int`, `long`, **etc. El más usado es** `int`.

El estándar de C++ no obliga a los compiladores a usar un tamaño determinado.

Lo usual es que un `int` **ocupe 32 bits. El rango sería:**

$$\left[-\frac{2^{32}}{2}, \frac{2^{32}}{2} - 1\right] = [-2147483648, 2147483647]$$

```
int entero;
entero = 53;
```

Cuando necesitemos un entero mayor, podemos usar el tipo `long long int`. **También puede usarse la forma abreviada del nombre:** `long long`. **Es un entero de 64 bits y es estándar en C++ 11. El rango sería:**

$$[-9223372036854775808, 9223372036854775807]$$

Algunos compiladores ofrecen tipos propios. Por ejemplo, Visual C++ ofrece `__int64`.

I.5.1.3. Literales enteros

Como ya vimos en la página 29, un literal es la especificación (dentro del código fuente) de un valor concreto de un tipo de dato. Los *literales enteros (integer literals)* se construyen con tokens formados por símbolos numéricos. Pueden empezar con un signo -

```
53      -406778      0
```

***Nota*. En el código se usa el sistema decimal (53) pero internamente, el ordenador usa el código binario (000000000110101)**

Para representar un literal entero, el compilador usará el tipo `int`. Si es un literal que no cabe en un `int`, se usará otro tipo entero mayor.

I.5.1.4. Operadores

Operadores binarios

`+ - * / %`

suma, resta, producto, división entera y módulo.
El operador módulo (%) representa el resto de la división entera
Devuelven un entero.
Binarios. Notación infija: `a*b`

```
int n;
n = 5 * 7;       // Asigna a la variable n el valor 35
n = n + 1;       // Asigna a la variable n el valor 36
n = 25 / 9;      // Asigna a la variable n el valor 2
n = 25 % 9;      // Asigna a la variable n el valor 7
n = 5 / 7;       // Asigna a la variable n el valor 0
n = 7 / 5;       // Asigna a la variable n el valor 1
n = 173 / 10;    // Asigna a la variable n el valor 17
n = 5 % 7;       // Asigna a la variable n el valor 5
n = 7 % 5;       // Asigna a la variable n el valor 2
n = 173 % 10;    // Asigna a la variable n el valor 3
5 / 7 = n;       // Sentencia Incorrecta.
```

Operaciones usuales:

- ▷ **Extraer el dígito menos significativo:** `5734 % 10` → `4`
- ▷ **Truncar desde el dígito menos significativo:** `5734 / 10` → `573`

Operadores unarios de incremento y decremento

++ y -- Unarios de notación postfija.

Incrementan y decrementan, respectivamente, el valor de la variable entera sobre la que se aplican (no pueden aplicarse sobre una expresión).

```
<variable>++;   /* Incrementa la variable en 1
                   Es equivalente a:
                   <variable> = <variable> + 1; */
<variable>--;   /* Decrementa la variable en 1
                   Es equivalente a:
                   <variable> = <variable> - 1; */
```

También existe una versión prefija de estos operadores. Lo veremos en el siguiente tema y analizaremos en qué se diferencian.

```
int dato = 4;
dato = dato+1;        // Asigna 5 a dato
dato++;               // Asigna 6 a dato
```

Operador unario de cambio de signo

- Unario de notación prefija.

Cambia el signo de la variable sobre la que se aplica.

```
int dato = 4, dato_cambiado;
dato_cambiado = -dato;              // Asigna -4 a dato_cambiado
dato_cambiado = -dato_cambiado;     // Asigna 4 a dato_cambiado
```

I.5.1.5. Expresiones enteras

Son aquellas expresiones, que al evaluarlas, devuelven un valor entero.

```
entera       56       (entera/4 + 56)%3
```

El orden de evaluación depende de la *precedencia* de los operadores.

Reglas de precedencia:

```
()
-  (operador unario de cambio de signo)
* / %
+ -
```

Cualquier operador de una fila superior tiene más prioridad que cualquiera de la fila inferior.

```
variable = 3 + 5 * 7;  // equivale a 3 + (5 * 7)
```

Los operadores de una misma fila tienen la misma prioridad. En este caso, para determinar el orden de evaluación se recurre a otro criterio, denominado *asociatividad (associativity)* . Puede ser de izquierda a derecha (LR) o de derecha a izquierda (RL).

```
variable = 3 / 5 * 7;  // / y * tienen la misma precedencia.
                       // Asociatividad LR. Equivale a (3/5)*7
variable = - -5;       // Asociatividad RL. Equivale a - (-5)
```

Ante la duda, forzad la evaluación deseada mediante la utilización de paréntesis:

```
dato = 3 + (5 * 7);     // 38
dato = (3 + 5) * 7;     // 56
```

Ejercicio. **Teniendo en cuenta el orden de precedencia de los operadores, indicad el orden en el que se evaluarían las siguientes expresiones:**

a) `a + b * c - d` **b)** `a * b / c` **c)** `a * c % b - d`

Ejercicio. **Leed un entero desde teclado que represente número de segundos y calcule el número de minutos que hay en dicha cantidad y el número de segundos restantes. Por ejemplo, en 123 segundos hay 2 minutos y 3 segundos.**

Ejemplo. **Incrementar el salario en 100 euros y calcular el número de billetes de 500 euros a usar en el pago de dicho salario.**

```
int salario, num_bill500;
salario = 43000;
num_bill500 = (salario + 100) / 500;  // ok
num_bill500 = salario + 100 / 500;    // Error lógico
```

I.5.2. Los tipos de datos reales

Un dato de tipo *real (float)* tiene como rango un subconjunto *finito* de R

- ▷ **Parte entera de** $4{,}56 \longrightarrow 4$
- ▷ **Parte real de** $4{,}56 \longrightarrow 56$

C++ ofrece distintos tipos para representar valores reales. Principalmente, `float` **(usualmente 32 bits) y** `double` **(usualmente 64 bits).**

```
double valor_real;
valor_real = 541.341;
```

I.5.2.1. Literales reales

Son tokens formados por dígitos numéricos y con un único punto que separa la parte decimal de la real. Pueden llevar el signo `-` **al principio.**

```
800.457      4.0      -3444.5
```

Importante:

- ▷ **El literal** `3` **es un entero.**
- ▷ **El literal** `3.0` **es un real.**

Los compiladores suelen usar el tipo `double` **para representar literales reales.**

También se puede utilizar *notación científica (scientific notation)* :

`5.32e+5` **representa el número** $5{,}32 * 10^5 = 532000$

`42.9e-2` **representa el número** $42{,}9 * 10^{-2} = 0{,}429$

I.5.2.2. Representación de los reales

¿Cómo podría el ordenador representar $541{,}341$?

Lo *fácil* sería:

- ▷ **Representar la parte entera 541 en binario**
- ▷ **Representar la parte real 341 en binario**

De esa forma, con 64 bits (32 bits para cada parte) podríamos representar:

- ▷ **Partes enteras en el rango $[-2147483648, 2147483647]$**
- ▷ **Partes reales en el rango $[-2147483648, 2147483647]$**

Sin embargo, la forma usual de representación no es así. Se utiliza la representación en *coma flotante (floating point)* . La idea es representar un *valor* y la *escala*. En aritmética decimal, la escala se mide con potencias de 10:

$$\begin{array}{lll} 42{,}001 & \rightarrow & \textbf{valor = } 4{,}2001 \quad \textbf{escala = } 10 \\ 42001 & \rightarrow & \textbf{valor = } 4{,}2001 \quad \textbf{escala = } 10^4 \\ 0{,}42001 & \rightarrow & \textbf{valor = } 4{,}2001 \quad \textbf{escala = } 10^{-1} \end{array}$$

El valor se denomina *mantisa (mantissa)* y el coeficiente de la escala *exponente (exponent)* .

En la representación en coma flotante, la escala es 2. A *grosso modo* se utilizan m bits para la mantisa y n bits para el exponente. La forma explícita de representación en binario se verá en otras asignaturas. Basta saber que utiliza potencias inversas de 2. Por ejemplo, 1011 representaría

$$1 * \frac{1}{2^1} + 0 * \frac{1}{2^2} + 1 * \frac{1}{2^3} + 1 * \frac{1}{2^4} =$$

$$= 1 * \frac{1}{2} + 0 * \frac{1}{4} + 1 * \frac{1}{8} + 1 * \frac{1}{16} = 0{,}6875$$

***Problema*: Si bien un entero se puede representar de forma exacta como suma de potencias de dos, un real sólo se puede *aproximar* con suma de potencias inversas de dos.**

Valores tan sencillos como $0{,}1$ o $0{,}01$ no se pueden representar de forma exacta, produciéndose un error de *redondeo (rounding)*

$$0{,}1 \approxeq 1 * \frac{1}{2^4} + 0 * \frac{1}{2^5} + 0 * \frac{1}{2^6} + 0 * \frac{1}{2^7} + 0 * \frac{1}{2^8} + \cdots$$

Por tanto:

¡Todas las operaciones realizadas con los reales pueden devolver valores que sólo sean aproximados!

Especial cuidado tendremos con operaciones del tipo

```
Repite varias veces
   Ir sumándole a una variable_real varios valores reales;
```

ya que los errores de aproximación se irán acumulando.

I.5.2.3. Rango y Precisión

La codificación en coma flotante separa el valor de la escala. Esto permite trabajar (en un mismo tipo de dato) con magnitudes muy grandes y muy pequeñas.

```
double masa_tierra_kg, masa_electron_kg;

masa_tierra_kg = 5.98e24;                // ok
masa_electron_kg = 9.11e-31;             // ok
```

C++ ofrece varios tipos reales: `float` **y** `double` **(y** `long double` **a partir de C++11). Con 64 bits, pueden representarse exponentes hasta** ± 308**.**

Pero el precio a pagar es muy elevado ya que se obtiene muy poca *precisión (precision)* (número de dígitos consecutivos que pueden representarse) tanto en la parte entera como en la parte real.

Tipo	Tamaño	Rango	Precisión
`float`	**4 bytes**	`+/-3.4 e +/-38`	**7 dígitos aproximadamente**
`double`	**8 bytes**	`+/-1.7 e +/-308`	**15 dígitos aproximadamente**

```
double valor_real;

// Datos de tipo double con menos de 15 cifras
// (en su representación decimal)

valor_real = 11.0;                          
// Almacena: 11.0                           Correcto

valor_real = 1.1;                           
// Almacena: 1.1000000000000001             Problema de redondeo

// Datos de tipo double con más de 15 cifras
// (en su representación decimal)

valor_real = 1000000000000000000100.0;      
// Almacena: 1000000000000000000000.0       Problema de precisión

valor_real = 0.10000000000000000009;        
// Almacena: 0.10000000000000001            Problema de precisión

valor_real = 1.00000000000000000009;        
// Almacena: 1.0                            Problema de precisión

valor_real = 10000000001.000000000004;      
// Almacena: 10000000001.0                  Problema de precisión
```

En resumen:

- **Los tipos *enteros* representan datos enteros de forma exacta, siempre que el valor esté en el rango correspondiente.**
- **Los tipos *reales* representan la *parte entera* de forma exacta si el número de dígitos es menor o igual que 7 -16 bits- o 15 -32 bits-.**

  La representación es aproximada si el número de dígitos es mayor.

  La *parte real* será sólo aproximada.

Los reales en coma flotante también permiten representar valores especiales como *infinito (infinity)* y una *indeterminación (undefined)* (*Not a Number*)

Representaremos infinito por `INF` y la indeterminación por `NaN`, pero hay que destacar que no son literales que puedan usarse en el código.

Las operaciones numéricas con infinito son las usuales en Matemáticas (`1.0/INF` es cero, por ejemplo) mientras que cualquier expresión que involucre `NaN`, produce otro `NaN`:

```
double valor_real, divisor = 0.0;

valor_real = 17.5 / divisor;           // Almacena INF
valor_real = 1.5 / valor_real;         // Almacena 0.0
valor_real = divisor / divisor;        // Almacena NaN
valor_real = 1.5 / valor_real;         // Almacena NaN
valor_real = 1e+300;
valor_real = valor_real * valor_real;  // Almacena INF
valor_real = 1.0 / valor_real;         // Almacena 0.0
```

I.5.2.4. Operadores

Los operadores matemáticos usuales también se aplican sobre datos reales:

```
+, -, *, /
```

Binarios, de notación infija. También se puede usar el operador unario de cambio de signo (-). Aplicados sobre reales, devuelven un real.

```
double real;
real = 5.0 * 7.0;    // Asigna a real el valor  35.0
real = 5.0 / 7.0;    // Asigna a real el valor 0.7142857
```

¡Cuidado! El comportamiento del operador / depende del tipo de los operandos: si todos son enteros, es la división entera. Si todos son reales, es la división real.

```
5 / 7     es una expresión entera. Resultado = 0
5.0 / 7.0   es una expresión real. Resultado = 0.7142857
```

Si un argumento es entero y el otro real, la división es real.

```
5 / 7.0   es una expresión real. Resultado = 0.7142857
```

I.5.2.5. Funciones estándar

Hay algunas bibliotecas *estándar* que proporcionan funciones que trabajan sobre datos numéricos (enteros o reales) y que suelen devolver un real. Por ejemplo, `cmath`

```
pow(), cos(), sin(), sqrt(), tan(), log(), log10(), ....
```

Todos los anteriores son unarios excepto `pow`**, que es binario (base, exponente). Devuelven un real.**

Para calcular el valor absoluto se usa la función `abs()`**. Devuelve un tipo real (aún cuando el argumento sea entero).**

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

using namespace std;

int main(){
   double real, otro_real;

   real      = 5.4;
   otro_real = abs(-5.4);
   otro_real = abs(-5);
   otro_real = sqrt(real);
   otro_real = pow(4.3, real);
}
```

Nota:

Observad que una misma función (**abs** por ejemplo) puede trabajar con datos de distinto tipo. Esto es posible porque hay varias sobrecargas de esta función. Posteriormente se verá con más detalle este concepto.

I.5.2.6. Expresiones reales

Son expresiones cuyo resultado es un número real.

`sqrt(real)` **es una expresión real**

`pow(4.3, real)` **es una expresión real**

En general, diremos que las *expresiones aritméticas (arithmetic expression)* o *numéricas* son las expresiones o bien enteras o bien reales.

Precedencia de operadores en las expresiones reales:

```
()
-  (operador unario de cambio de signo)
* /
+ -
```

> ***Consejo*:** *Para facilitar la lectura de las fórmulas matemáticas, evitad el uso de paréntesis cuando esté claro cuál es la precedencia de cada operador.*

Ejemplo. **Construid una expresión para calcular la siguiente fórmula:**

$$\frac{-b+\sqrt{b^2-4ac}}{2a}$$

`-b+sqrt(b*b-4.0*a*c)/2.0*a`	`Error lógico`
`((-b)+(sqrt((b*b) - (4.0*a)*c)))/(2.0*a)`	`Difícil de leer`
`(-b + sqrt(b*b - 4.0*a*c)) / (2.0*a)`	`Correcto`

Ejercicio. **Construid un programa para calcular la posición de un objeto que sigue un movimiento rectilíneo uniforme. La posición se calcula aplicando la siguiente fórmula:**

$$x_o + vt$$

dónde x_0 es la posición inicial, v la velocidad y t el tiempo transcurrido. Suponed que los tres datos son reales.

Ejercicio. **Construid una expresión para calcular la distancia euclídea entre dos puntos del plano $P1 = (x_1, y_1), P2 = (x_2, y_2)$. Usad las funciones `sqrt` y `pow`.**

$$d(P1, P2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

I.5.3. Operando con tipos numéricos distintos

I.5.3.1. Asignaciones a datos de expresiones de distinto tipo

El operador de asignación permite trabajar con tipos distintos en la parte izquierda y derecha.

En primer lugar se evalúa la parte derecha de la asignación.

En segundo lugar, se realiza la asignación. Si el tipo del resultado obtenido en la parte derecha de la asignación es distinto al del dato de la parte izquierda, el compilador realiza una *transformación de tipo (casting)* de la expresión de la derecha al tipo de dato de la parte izquierda de la asignación.

Esta transformación es temporal (mientras se evalúa la expresión)

Si el resultado de la expresión no cabe en la parte izquierda, se produce un un error lógico denominado *desbordamiento aritmético (arithmetic overflow)* . El valor asignado será un valor indeterminado.

Veamos varios casos:

▷ **Como cabría esperar, a un dato numérico de tipo *grande* se le puede asignar cualquier expresión numérica de tipo *pequeño*.**

```
int chico;
long long  grande;

// grande = chico; Sin problemas.

chico  = 5;        // 5 es un literal int
                   // int = int

grande = chico;    // chico int -> chico long long
                   // long long = long long
cout << chico;     // chico sigue siendo int
```

La transformación de `int` a `long long` es inmediata:

```
5 int:
00000000000000000000000000000101
5 long long:
0000000000000000000000000000000000000000000000000000000000000101
```

Otro ejemplo:

```
double real;
int entero;

entero = 5;
real   = entero;   // 5 int -> 5 double
                   // double = double
cout << entero;    // entero sigue siendo int
                   // Imprime 5 y no 5.0
```

▷ **En general, a un dato numérico de tipo *pequeño* se le puede asignar cualquier expresión numérica de tipo *grande*. Si el resultado está en el rango permitido del tipo pequeño, la asignación se realiza correctamente. En otro caso, se produce un desbordamiento aritmético y se almacenará un valor indeterminado.**

```
int chico;
long long  grande;

// chico = grande; Puede desbordarse.

grande = 6000000;  // 6000000 es un literal int
                   // 6000000 int -> 6000000 long long
                   // long long = long long

chico = grande;    // grande long long -> grande int int
                   // El resultado 6000000 cabe en chico
                   // int = int

grande = 36000000000000;
                   // 36000000000000 es un literal long long
                   // long long = long long

chico = grande;    // 36000000000000 no cabe en un int
                   // Desbordamiento.
                   // chico = -415875072
                   // int = int
```

▷ **A un entero se le puede asignar una expresión real. En este caso, se pierde la parte decimal, es decir, se *trunca* la expresión real.**

```
double real;
int entero;

real = 5.3;         // 5.3 es un literal double
                    // double = double

// entero = real; Se trunca el real

entero = real;      // real double (5.3) -> real int (5)
                    // int = int
```

A un dato numérico se le puede asignar una expresión de un tipo distinto. Si el resultado cabe, no hay problema. En otro caso, se produce un desbordamiento aritmético.

Si asignamos una expresión real a un entero, se trunca la parte decimal.

I.5.3.2. Expresiones con datos numéricos de distinto tipo

Muchos operadores numéricos permiten que los argumentos sean expresiones de tipos distintos. Para evaluar el resultado de una expresión que contenga datos con tipos distintos, el compilador realiza un casting para que todos sean del mismo tipo y así poder hacer las operaciones.

Los datos de tipo pequeño se transformarán al mayor tipo de los otros datos de la expresión.

Esta transformación es temporal (mientras se evalúa la expresión)

Ejemplo. **Calcular la cuantía de las ventas totales de un producto a partir de su precio y del número de unidades vendidas.**

```
int unidades_vendidas;
double precio_unidad, venta_total;
......
cin >> precio_unidad;
cin >> unidades_vendidas;

venta_total = precio_unidad * unidades_vendidas;
           // double        * int
           // unidades_vendidas int -> double
           // double        * double
           // El resultado de la expresión es double
           // double = double
```

Si en una expresión todos los datos son del mismo tipo, el compilador no realiza casting.

Ejemplo. **Calcular la media aritmética de la edades de dos personas.**

```
int edad1 = 10, edad2 = 5;
double media;


media = (edad1 + edad2)/2;  // media = 7.0
```

La expresión `edad1 + edad2` **es entera y devuelve** `15`**. Por tanto, los dos operandos de la expresión** `(edad1 + edad2)/2` **son enteros, por lo que el operador de división actúa sobre enteros y es la división entera, devolviendo el entero** `7`**. Al asignarlo a la variable real** `media`**, se transforma en** `7.0`**. Se ha producido un error lógico.**

Posibles soluciones (de peor a mejor)

- **Usar un dato temporal de un tipo mayor.**

```
int edad1 = 10, edad2 = 5;
double media, edad1_tmp;

edad1_tmp = edad1;
media     = (edad1_tmp + edad2)/2;
          // double + int es double
          // double / int es double
          // media = 7.5
```

El inconveniente de esta solución es que estamos representando un mismo dato con dos variables distintas y corremos el peligro de usarlas en sitios distintos.

▷ **Cambiar el tipo de dato original de las variables.**

```
double edad1 = 10, edad2 = 5;
double media;

media = (edad1 + edad2)/2;
      // double + double es double
      // double / int es double
      // media = 7.5
```

Debemos evitar esta solución ya que el tipo de dato asociado a las variables debe depender de la semántica de éstas. En cualquier caso, cabe la posibilidad de reconsiderar la decisión con respecto a los tipos asociados y cambiarlos si la semántica de las variables así lo demanda. No es el caso de nuestras variables de edad, que son enteras.

▷ **Usar un casting manual tal y como se indica en la sección I.5.3.3 (página 69)**

▷ **Forzamos la división real introduciendo un literal real:**

```
int edad1 = 10, edad2 = 5;
double media;

media = (edad1 + edad2)/2.0;
      // int + int es int
      // int / double es double
      // media = 7.5
```

Ejemplo. **¿Qué pasaría en este código?**

```
int chico = 1234567890;
long long grande;

grande = chico * chico;

      // grande = 304084036    Error lógico
```

En la expresión `chico * chico` **todos los datos son del mismo tipo (**`int`**). Por tanto no se produce casting y el resultado se almacena en un** `int`**. La multiplicación correcta es** `1524157875019052100` **pero no cabe en un** `int`**, por lo que se se produce un desbordamiento aritmético y a la variable** `grande` **se le asigna un valor indeterminado (**`304084036`**)**

Observad que el resultado (`1524157875019052100`**) sí cabe en un** `long long` **pero el desbordamiento se produce durante la evaluación de la expresión,** ***antes*** **de realizar la asignación (recordad lo visto en la página 41)**

Posibles soluciones: Las mismas que vimos en el ejemplo anterior (las tres primeras, porque no hay literales involucrados en la expresión)

Nota:

El desbordamiento como tal no ocurre con los reales ya que una operación que de un resultado fuera de rango devuelve infinito (INF)

```
double real, otro_real;

real = 1e+200;
otro_real = real * real;
      // otro_real = INF
```

Durante la evaluación de una expresión numérica en la que intervienen datos de distinto tipo, el compilador realizará un casting para transformar los datos de tipos pequeños al mayor de los tipos involucrados.

Esta transformación es temporal (sólo se aplica mientras se evalúa la expresión).

Pero cuidado: si en una expresión todos los datos son del mismo tipo, el compilador no realiza ninguna transformación, de forma que la expresión resultante es del mismo tipo que la de los datos involucrados. Por tanto, cabe la posibilidad que se produzca un desbordamiento durante la evaluación de la expresión.

I.5.3.3. El operador de casting (Ampliación)

Este apartado es de ampliación. No entra en el examen.

El *operador de casting (casting operator)* permite que el programador pueda cambiar explícitamente el tipo por defecto de una expresión. La transformación es siempre temporal: sólo afecta a la instrucción en la que aparece el casting.

```
static_cast<tipo_de_dato> (expresión)
```

Ejemplo. **Media aritmética:**

```
int edad1 = 10, edad2 = 5;
double media;

media = (static_cast<double>(edad1) + edad2)/2;
```

Ejemplo. **Retomamos el ejemplo de la página 67:**

```
int chico = 1234567890;
long long grande;


grande = static_cast<long long>(chico) * chico;
         // chico int -> chico long long
         // long long * int
         // grande = 1524157875019052100

// chico sigue siendo int después de la instrucción anterior
```

¿Por qué no es correcto lo siguiente?

```
int chico = 1234567890;
long long grande;

grande = static_cast<long long> (chico * chico);

// grande = 304084036
```

En C, hay otro operador de casting que realiza una función análoga a `static_cast`:

```
(<tipo de dato>) expresión
```

```
int edad1 = 10, edad2 = 5;
double media;
media = ((double)edad1 + edad2)/2;
```

I.5.4. El tipo de dato carácter

I.5.4.1. Rango

Frecuentemente, querremos manejar información que podemos representar con un único carácter. Por ejemplo, grupo de teoría de una asignatura, carácter a leer desde el teclado para seleccionar una opción de un menú, calificación obtenida (según la escala ECTS), tipo de moneda, etc.

Los caracteres que pueden usarse son los permitidos en la plataforma en la que se ejecuta el programa. Para representar dichos caracteres se pueden usar distintos tipos de *codificación (coding)* . El estándar aceptado actualmente es *Unicode* (multi-lenguaje, multi-plataforma). Éste establece los caracteres que pueden representarse (incluye caracteres de idiomas como español, chino, árabe, etc), cómo hacerlo (permite tres tamaños distintos: 8, 16 y 32 bits) y asigna un número de orden a cada uno de ellos.

Los primeros 256 caracteres de Unicode coinciden con la codificación antigua denominada *ASCII Extendido (Extended ASCII)* . De éstos, los primeros 32 no son imprimibles. Los caracteres con acentos especiales usados en Europa Occidental (incluida España) están incluidos en el ASCII extendido.

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

C++ permite asignar a cualquier entero uno de estos caracteres, encerrándolo entre comillas simples. El valor que el entero almacena es el número de orden de dicho carácter en la tabla.

```
int letra_piso;
long entero_grande;

letra_piso = 65;      // Almacena 65
letra_piso = 'A';     // Almacena 65
entero_grande = 65;   // Almacena 65
entero_grande = 'A'; // Almacena 65
```

Si nos restringimos a los 256 primeros caracteres, sólo necesitamos 1 byte para representarlo. Por eso, C++ ofrece el tipo de dato `char`:

> ***El tipo de dato*** `char` ***es un entero*** **pequeño** ***sin signo con rango*** $\{0 \cdots 255\}$

```
char letra_piso;

letra_piso = 65;      // Almacena 65
letra_piso = 'A';     // Almacena 65
```

Además, el recurso `cout` se ha programado de la siguiente forma:

- ▷ **Si se pasa a `cout` un dato de tipo `char`, imprime el carácter correspondiente al entero almacenado.**
- ▷ **Si se pasa a `cout` un dato de tipo entero (distinto a `char`), imprime el entero almacenado.**

```
int entero = 'A';
char letra = 'A';

cout << entero;  // Imprime 65
cout << letra;   // Imprime A
```

I.5.4.2. Literales de carácter

Son tokens formados por:

- **Un único carácter encerrado entre comillas simples:**
  `'!' 'A' 'a' '5' 'ñ'`

  Observad que `'5'` es un literal de carácter y `5` es un literal entero

  ¡Cuidado!: `'cinco'` o `'11'` no son literales de carácter.

  Observad la diferencia:

```
double r;
char letra;

letra = 'r';   // literal de carácter 'r'
r = 23.2;      // variable real r
```

- **O bien una *secuencia de escape*, es decir, el símbolo `\` seguido de otro símbolo, como por ejemplo:**

Secuencia	Significado
`\n`	Nueva línea (retorno e inicio)
`\t`	Tabulador
`\b`	Retrocede 1 carácter
`\r`	Retorno de carro
`\f`	Salto de página
`\'`	Comilla simple
`\"`	Comilla doble
`\\`	Barra inclinada

  Las secuencias de escape también deben ir entre comillas simples, por ejemplo, `'\n'`, `'\t'`, etc.

```
#include <iostream>
using namespace std;
int main(){
   const char NUEVA_LINEA = '\n';
   char letra_piso;

   letra_piso = 'B';   // Almacena 66 ('B')
   cout << letra_piso << "ienvenidos";
   cout << '\n' << "Empiezo a escribir en la siguiente línea";
   cout << '\n' << '\t' << "Acabo de tabular esta línea";
   cout << NUEVA_LINEA;
   cout << '\n' << "Esto es una comilla simple: " << '\'';
}
```

Escribiría en pantalla:

```
Bienvenidos
Empiezo a escribir en la siguiente línea
        Acabo de tabular esta línea

Esto es una comilla simple: '
```

Ampliación:

Para escribir un retorno de carro, también puede usarse una constante llamada `endl` en la forma:

```
cout << endl << "Adiós" << endl
```

Esta constante, además, obliga a vaciar el buffer de datos en ese mismo momento, algo que, por eficiencia, no siempre querremos hacer.

I.5.4.3. Funciones estándar y operadores

El fichero de cabecera `cctype` contiene varias funciones para trabajar con caracteres. Los argumentos y el resultado son de tipo `int`. Por ejemplo:

`tolower` `toupper`

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

int main(){
   char letra_piso;

   letra_piso = tolower('A');   // Almacena 97 ('a')
   letra_piso = toupper('A');   // Almacena 65 ('A')
   letra_piso = tolower('B');   // Almacena 98 ('b')
   letra_piso = tolower('!');   // Almacena 33 ('!') No cambia
```

Los operadores aplicables a los enteros también son aplicables a cualquier `char` o a cualquier literal de carácter. El operador actúa siempre sobre el entero de orden correspondiente:

```
    char caracter;      // También valdría cualquier tipo entero;
    int diferencia;

    caracter = 'A' + 1;      // Almacena 66 ('B')
    caracter = 65 + 1;       // Almacena 66 ('B')
    caracter = '7' - 1;      // Almacena 54 ('6')

    diferencia = 'c' - 'a';  // Almacena 2
```

¿Qué imprimiría la sentencia `cout << 'A'+1`? No imprime `'B'` como cabría esperar sino 66 ya que `1` es un literal entero y por tanto de tipo `int`. Un `int` es más grande que un `char`, por lo que `'A'+1` es un `int`.

I.5.5. El tipo de dato cadena de caracteres

Un literal de tipo *cadena de caracteres (string)* es una sucesión de caracteres encerrados entre comillas dobles:
`"Hola"`, `"a"` **son literales de cadena de caracteres**

```
cout << "Esto es un literal de cadena de caracteres";
```

Las secuencias de escape también pueden aparecer en los literales de cadena de caracteres presentes en `cout`

```
int main(){
   cout << "Bienvenidos";
   cout << "\nEmpiezo a escribir en la siguiente línea";
   cout << "\n\tAcabo de tabular esta línea";
   cout << "\n";
   cout << "\nEsto es una comilla simple '";
   cout << " y esto es una comilla doble \"";
}
```

Escribiría en pantalla:

```
Bienvenidos
Empiezo a escribir en la siguiente línea
        Acabo de tabular esta línea

Esto es una comilla simple ' y esto es una comilla doble "
```

Nota:

Formalmente, el tipo **`string`** no es un tipo simple sino compuesto de varios caracteres. Lo incluimos dentro de este tema en aras de simplificar la materia.

Ejercicio. **Determinar cuales de las siguientes son constantes de cadena de caracteres válidas, y determinar la salida que tendría si se pasase como argumento a** `cout`

a) `"8:15 P.M."` **b)** `"'8:15 P.M."` **c)** `'"8:15 P.M."'`

d) `"Dirección\n"` **e)** `"Dirección'n"` **f)** `"Dirección\'n"`

g) `"Dirección\\'n"`

C++ ofrece dos alternativas para trabajar con cadenas de caracteres:

- ▷ ***Cadenas estilo C*: son *vectores de caracteres* con terminador** `'\0'`**. Se verá en la asignatura Metodología de la Programación.**
- ▷ **Usando el tipo** `string` **(la recomendada en esta asignatura)**

```
int main(){
   string mensaje_bienvenida;
   mensaje_bienvenida = "\tFundamentos de Programación\n";
   cout << mensaje_bienvenida;
}
```

Para poder operar con un string debemos incluir la biblioteca `string`**:**

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

int main(){
   string cad;
   cad = "Hola y ";
   cad = cad + "adiós";
   cout << cad;
}
```

Una función muy útil definida en la biblioteca `string` es:

```
to_string( <dato> )
```

donde `dato` puede ser casi cualquier tipo numérico. Para más información:

http://en.cppreference.com/w/cpp/string/basic_string/to_string

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

int main(){
   string cadena;
   int entero;
   double real;

   entero = 27;
   real   = 23.5;
   cadena = to_string(entero);   // Almacena "27"
   cadena = to_string(real);     // Almacena "23.500000"
}
```

Nota:

La función **to_string** es otro ejemplo de función que puede trabajar con argumentos de distinto tipo de dato como enteros, reales, etc (sobrecargas de la función)

También están disponibles funciones para hacer la transformación inversa, como por ejemplo:

```
stoi( <cadena> )      stod( <cadena> )
```

que convierten a `int` y `double` respectivamente:

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

int main(){
   string cadena;
   int entero;
   double real;

   cadena = "27";
   entero = stoi(cadena);          // Almacena 27
   cadena = "23.5";
   real   = stod(cadena);          // Almacena 23.5
   cadena = "  23.5 basura";
   real   = stod(cadena);          // Almacena 23.5
   cadena = "basura 23.5";
   real   = stod(cadena);          // Error de ejecución
}
```

Ampliación:

Realmente, **`string`** es una clase (lo veremos en temas posteriores) por lo que le serán aplicables métodos en la forma:

```
cad = "Hola y ";
cad.append("adiós");     // :-O
```

I.5.6. El tipo de dato lógico o booleano

Es un tipo de dato muy común en los lenguajes de programación. Se utiliza para representar los valores verdadero y falso que suelen estar asociados a una condición.
En C++ se usa el tipo `bool`.

I.5.6.1. Rango

El rango de un dato de tipo *lógico (boolean)* está formado solamente por dos valores: verdadero y falso. Para representarlos, se usan los siguientes literales:

```
true        false
```

I.5.6.2. Funciones standard y operadores lógicos

Una expresión lógica es una expresión cuyo resultado es un tipo de dato lógico.

Algunas funciones que devuelven un valor lógico:

▷ **En la biblioteca** `cctpye`**:**

```
isalpha    isalnum    isdigit   ...
```

Por ejemplo, `isalpha('3')` **es una expresión lógica (devuelve** `false`**)**

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

int main(){
   bool es_alfabetico, es_alfanumerico, es_digito_numerico;

   es_alfabetico      = isalpha('3');   // false
   es_alfanumerico    = isalnum('3');   // true
   es_digito_numerico = isdigit('3');   // true
```

▷ **En la biblioteca** `cmath`**:**

```
isnan    isinf    isfinite   ...
```

Comprueban, respectivamente, si un real contiene `NaN`**,** `INF` **o si es distinto de los dos anteriores.**

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

int main(){
   double real = 0.0;
   bool es_indeterminado;

   real = real/real;
   es_indeterminado = isnan(real);  // true
```

Los operadores son los clásicos de la lógica Y, O, NO que en C++ son los operadores `&&`, `||`, `!` respectivamente.

$$p \equiv \text{Carlos es varón}$$

$$q \equiv \text{Carlos es joven}$$

p	q	p `&&` q	p `||` q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

p	`!` p
true	false
false	true

Por ejemplo, si p es `false`, y q es true, p `&&` q será false y p `||` q será true.

Recordad la siguiente regla nemotécnica:

- ▷ `false &&` ***expresión*** **siempre es `false`.**
- ▷ `true ||` ***expresión*** **siempre es `true`.**

Tabla de Precedencia:

```
!
&&
||
```

Ejercicio. **Declarad dos variables de tipo `bool`, `es_joven` y `es_varon`. Asignadles cualquier valor. Declarad otra variable `es_varon_viejo` y asignadle el valor correcto usando las variables anteriores y los operadores lógicos.**

I.5.6.3. Operadores Relacionales

Son los operadores habituales de comparación de expresiones numéricas.
Pueden aplicarse a operandos tanto enteros, reales, como de caracteres y tienen el mismo sentido que en Matemáticas. El resultado es de tipo `bool`.

`==` (igual), `!=` (distinto), `<`, `>`, `<=` (menor o igual) y `>=` (mayor o igual)

Algunos ejemplos:

- ▷ **La expresión `(4 < 5)` devuelve valor `true`**
- ▷ **La expresión `(4 > 5)` devuelve el valor `false`**
- ▷ **La relación de orden entre caracteres se establece según la tabla ASCII.**
  La expresión `('a' > 'b')` devuelve el valor `false`.

`!=` es el operador relacional distinto.
`!` es la negación lógica.
`==` es el operador relacional de igualdad.
`=` es la operación de asignación.

Tanto en `==` como en `!=` se usan 2 signos para un único operador

```
//   Ejemplo de operadores relacionales

int main(){
   int entero1, entero2;
   double real1, real2;
   bool menor, iguales;

   entero1 = 3;
   entero2 = 5;

   menor = entero1 < entero2;                        // true
   menor = entero2 < entero1;                        // false
   menor = (entero1 < entero2) && !(entero2 < 7);    // false
   menor = (entero1 < entero2) || !(entero2 < 7);    // true
   iguales = entero1 == entero2;                     // false

   real1 = 3.8;
   real2 = 8.1;

   menor = real1 > real2;                            // false
   menor = !menor;                                   // true
}
```

Veremos su uso en la *sentencia condicional*:

```
  if (4 < 5)
     cout << "4 es menor que 5";

  if (!(4 > 5))
     cout << "4 es menor o igual que 5";
```

Tabla de Precedencia:

```
()
!
< <= > >=
==  !=
&&
||
```

A es menor o igual que B y B no es mayor que C

```
int A = 40, B = 34, C = 50;
bool condicion;

condicion = A <= B && !B > C;        // Incorrecto
condicion = (A <= B) && (!(B > C)); // Correcto
condicion = (A <= B) && !(B > C);   // Correcto


condicion = A <= B && B <= C;   // Correcto. Expresión simplificada
```

Consejo: ***Simplificad las expresiones lógicas, para así aumentar su legibilidad.***

Ejercicio. **Escribid una expresión lógica que devuelva `true` si un número entero `edad` está en el intervalo [0,100]**

I.5.7. Lectura de varios datos

Hasta ahora hemos leído/escrito datos uno a uno desde/hacia la consola, separando los datos con `Enter`.

```
int entero, otro_entero;
double real;

cin >> entero;
cin >> real;
cin >> otro_entero;
```

También podríamos haber usado como separador un espacio en blanco o un tabulador. ¿Cómo funciona?

La E/S utiliza un *buffer* intermedio. Es un espacio de memoria que sirve para ir suministrando datos para las operaciones de E/S, desde el dispositivo de E/S al programa. Por ahora, dicho dispositivo será el teclado.

El primer `cin` **pide datos al teclado. El usuario los introduce y cuando pulsa** `Enter`**, éstos pasan al buffer y termina la ejecución de** `cin >> entero;`**. Todo lo que se haya escrito en la consola pasa al buffer, *incluido* el** `Enter`**. Éste se almacena como el carácter** `'\n'`**.**

Sobre el buffer hay definido un *cursor (cursor)* que es un apuntador al siguiente byte sobre el que se va a hacer la lectura. Lo representamos con ↑. Representamos el espacio en blanco con ␣

Supongamos que el usuario introduce `43  52.1<Enter>`

43 ␣ ␣ 52.1 \n	cin >> entero;	␣ ␣ 52.1 \n
↑	entero = 43	↑

El `43` **se asigna a** `entero` **y éste se borra del buffer.**

Las ejecuciones posteriores de `cin` se saltan, previamente, los separadores que hubiese al principio del buffer (espacios en blanco, tabuladores y `\n`). Dichos separadores se eliminan del buffer.

La lectura se realiza sobre los datos que hay en el buffer. Si no hay más datos en él, el programa los pide a la consola.

Ejemplo. **Supongamos que el usuario introduce** `43  52.1<Enter>`

```
cin >> entero;
   // Usuario:  43    52.1<Enter>
   // Buffer:  [43    52.1\n]
   // entero =  43
   // Buffer:  [    52.1\n]
cin >> real;
   // real = 52.1
   // Buffer:  [\n]
cin >> otro_entero;
   // Buffer:  []
```

Ahora el buffer está vacío, por lo que el programa pide datos a la consola:

```
   // Usuario: 37<Enter>
   // otro_entero = 37;
   // Buffer:  [\n]
```

Esta comunicación funciona igual entre un fichero y el buffer. Para que la entrada de datos sea con un fichero en vez de la consola basta ejecutar el programa desde el sistema operativo, redirigiendo la entrada:

```
C:\mi_programa.exe < fichero.txt
```

Contenido de `fichero.txt`:

```
43     52.1\n37
```

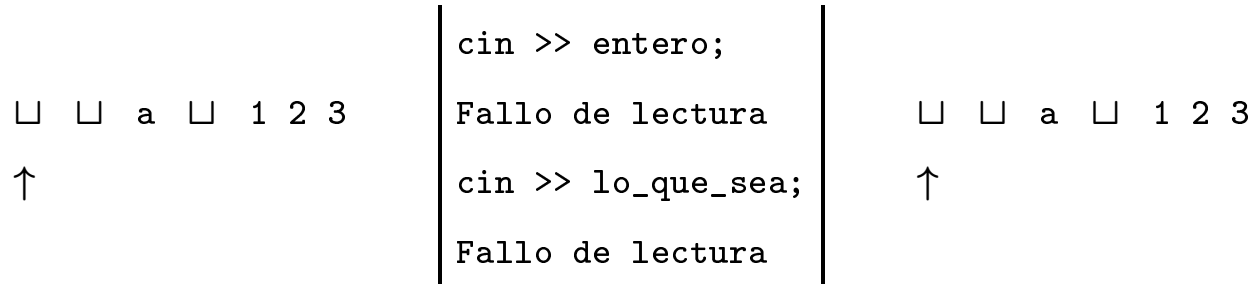
Desde un editor de texto se vería lo siguiente:

```
43     52.1
37
```

La lectura sería así:

```
cin >> entero;
   // Buffer:  [43     52.1\n37]
   // entero =  43
   // Buffer:  [     52.1\n37]
cin >> real;
   // real = 52.1
   // Buffer:  [\n37]
cin >> otro_entero;
   // otro_entero = 37;
   // Buffer:  []
```

¿Qué pasa si queremos leer un entero pero introducimos, por ejemplo, una letra? Se produce un error en la lectura y a partir de ese momento, todas las operaciones siguientes de lectura también dan fallo y el cursor no avanzaría.



Se puede *resetear* el estado de la lectura con `cin.clear()` y consultarse el estado actual (error o correcto) con `cin.fail()`. En cualquier caso, para simplificar, a lo largo de este curso asumiremos que los datos vienen en el orden correcto especificado en el programa, por lo que no será necesario recurrir a `cin.clear()` ni a `cin.fail()`.

Si vamos a leer sobre un tipo `char` debemos tener en cuenta que `cin` siempre se salta los separadores que previamente hubiese:

```
char caracter;
```

␣ a 1 2 3	cin >> caracter;	1 2 3
↑	caracter = 'a'	↑

Si queremos leer los separadores en una variable de tipo `char` debemos usar `cin.get()`:

␣ a 1 2 3	caracter = cin.get();	a 1 2 3
↑	caracter = ' '	↑

a 1 2 3	caracter = cin.get();	1 2 3
↑	caracter = 'a'	↑

Lo mismo ocurre si hubiese un carácter de nueva línea:

\n \n a \n	caracter = cin.get();	\n a \n
↑	caracter = '\n'	↑

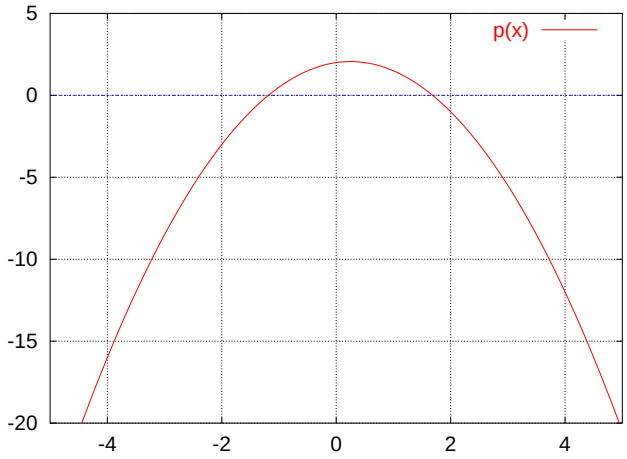
Ampliación:

Para leer una cadena de caracteres (**string**) podemos usar la función **getline** de la biblioteca **string**. Permite leer caracteres hasta llegar a un terminador que, por defecto, es el carácter de nueva línea **'\n'**. La cadena a leer se pasa como un parámetro por referencia a la función. Este tipo de parámetros se estudian en el segundo cuatrimestre.

I.6. El principio de una única vez

Ejemplo. ***Calcular las raíces de una ecuación de 2º grado.***

$$p(x) = ax^2 + bx + c = 0$$



Algoritmo: **Raíces de una parábola**

- ***Entradas*****: Los parámetros de la ecuación a, b, c.**
  Salidas**: Las raíces de la parábola $r1, r2$**
- ***Descripción*****:**
  Calcular $r1, r2$ en la forma siguiente:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

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

int main(){
   double a, b, c;          // Parámetros de la ecuación
   double raiz1, raiz2;     // Raíces obtenidas

   cout << "\nIntroduce coeficiente de 2º grado: ";
   cin >> a;
   cout << "\nIntroduce coeficiente de 1er grado: ";
   cin >> b;
   cout << "\nIntroduce coeficiente independiente: ";
   cin >> c;

   // Se evalúa dos veces la misma expresión:
   raiz1 = (-b + sqrt(b*b + 4*a*c) ) / (2*a);
   raiz2 = (-b - sqrt(b*b + 4*a*c) ) / (2*a);

   cout << "\nLas raíces son: " << raiz1 << " y " << raiz2;
}
```

En el código anterior se evalúa dos veces la expresión `sqrt( b*b + 4*a*c )`. Esto es nefasto ya que:

- **El compilador pierde tiempo al evaluar dos veces una misma expresión. El resultado es el mismo ya que los datos involucrados no han cambiado.**
- **Mucho más importante: Cualquier cambio que hagamos en el futuro nos obligará a modificar el código en dos sitios distintos. De hecho, había un error en la expresión y deberíamos haber puesto: `b*b - 4*a*c`, por lo que tendremos que cambiar dos líneas.**

Para no repetir código usamos una variable para almacenar el valor de la expresión que se repite:

```
int main(){
   double a, b, c;          // Parámetros de la ecuación
   double raiz1, raiz2;     // Raíces obtenidas
   double radical, denominador;

   cout << "\nIntroduce coeficiente de 2º grado: ";
   cin >> a;
   cout << "\nIntroduce coeficiente de 1er grado: ";
   cin >> b;
   cout << "\nIntroduce coeficiente independiente: ";
   cin >> c;

   // Cada expresión sólo se evalúa una vez:

   denominador = 2*a;
   radical = sqrt(b*b - 4*a*c);

   raiz1 = (-b + radical) / denominador;
   raiz2 = (-b - radical) / denominador;
   cout << "\nLas raíces son: " << raiz1 << " y " << raiz2;
}
```

http://decsai.ugr.es/~carlos/FP/I_ecuacion_segundo_grado.cpp

Nota:

Observad que, realmente, también se repite la expresión `-b`. Debido a la sencillez de la expresión, se ha optado por mantenerla duplicada.

Principio de Programación:

Una única vez (Once and only once)

Cada descripción de comportamiento debe aparecer una única vez en nuestro programa.

O dicho de una manera informal:

Jamás ha de repetirse código

La violación de este principio hace que los programas sean difíciles de actualizar ya que cualquier cambio ha de realizarse en todos los sitios en los que está repetido el código. Ésto aumenta las posibilidades de cometer un error ya que podría omitirse alguno de estos cambios.

En el tema III (Funciones y Clases) veremos herramientas que los lenguajes de programación proporcionan para poder cumplir este principio. Por ahora, nos limitaremos a seguir el siguiente consejo:

Si el resultado de una expresión no cambia en dos sitios distintos del programa, usaremos una variable para almacenar el resultado de la expresión y utilizaremos su valor tantas veces como queramos.

Bibliografía recomendada para este tema:

- **A un nivel menor del presentado en las transparencias:**
  - **Primer capítulo de Deitel & Deitel**
  - **Primer capítulo de Garrido.**
- **A un nivel similar al presentado en las transparencias:**
  - **Capítulo 1 y apartados 2.1, 2.2 y 2.3 de Savitch**
- **A un nivel con más detalles:**
  - **Los seis primeros capítulos de Breedlove.**
  - **Los tres primeros capítulos de Gaddis.**
  - **Los tres primeros capítulos de Stephen Prata.**
  - **Los dos primeros capítulos de Lafore.**

Índice alfabético