Tema 2. Introducción a un Lenguaje de Programación

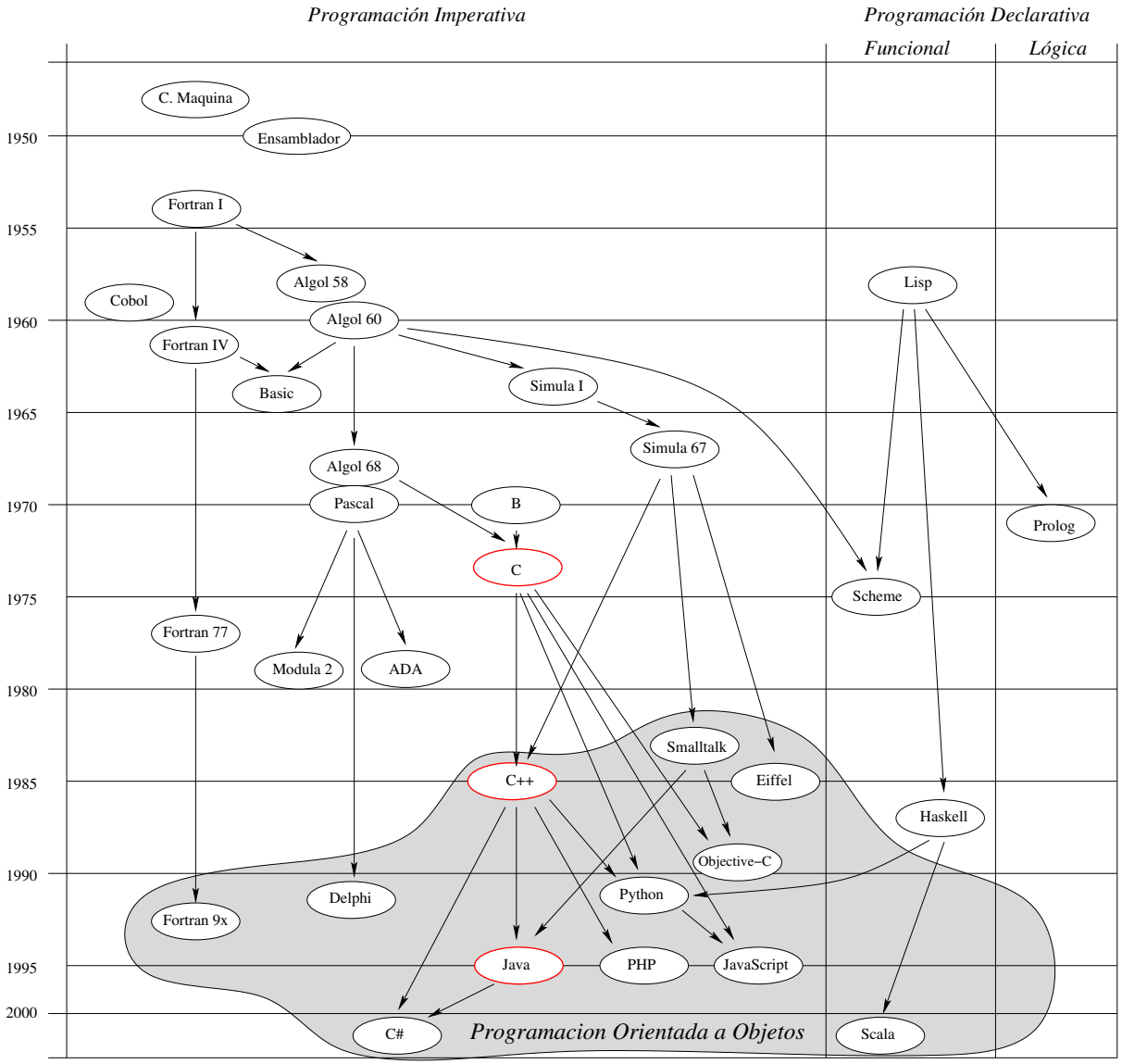
Vicente Benjumea García

Introducción a la Programación
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

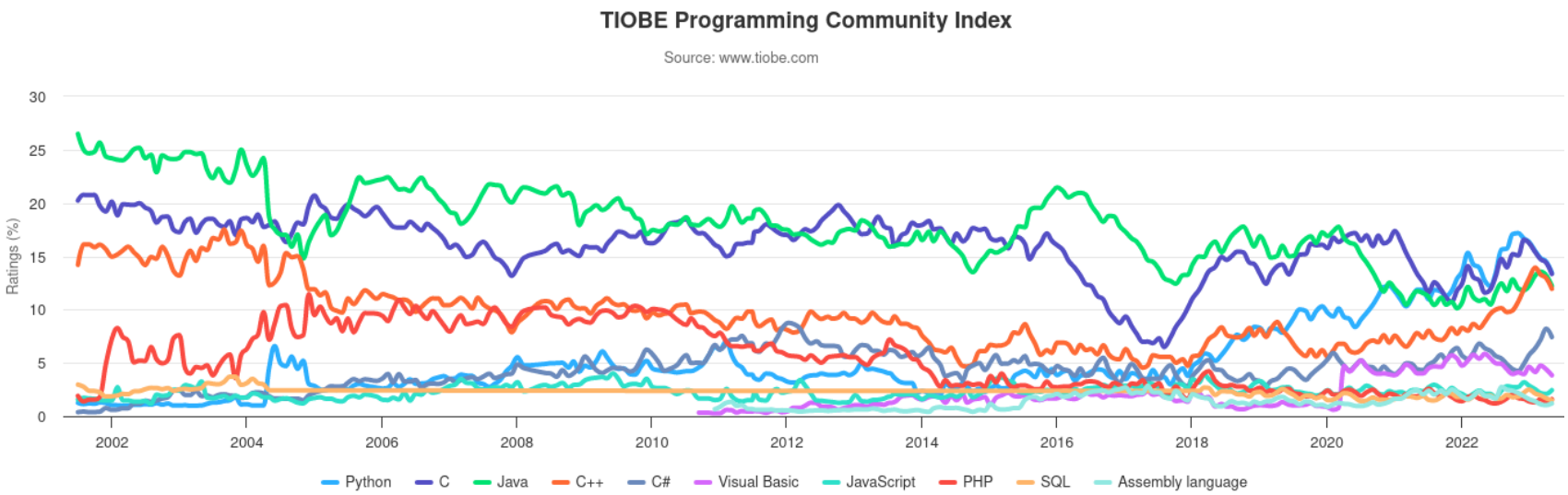
Tema 2. Introducción a un lenguaje de programación

- Introducción a C++.
- Tipos de datos simples predefinidos.
- Constantes, variables y asignaciones.
- Entrada y salida de datos básica.
- Expresiones Lógicas o Booleanas.
- Control del flujo de ejecución.
- Estructuras de selección.
- Estructuras de iteración.
- Control de errores y excepciones.
- Entrada y salida de datos avanzada.
- Tipos de datos.
- Problemas derivados de la implementación de los números.
- Errores frecuentes.

- Influencias de los lenguajes de programación:



- Clasificación de los lenguajes más *populares* (2023) según *Tiobe* (`https://www.tiobe.com/tiobe-index/`):



Programa

Un **programa** consiste en la implementación de un **algoritmo**, codificado un lenguaje de programación concreto, adecuado para ser ejecutado en una computadora, con el objetivo de resolver un problema determinado.

En este curso utilizaremos el *lenguaje de programación* **C++**.

Elementos de un Programa

Un programa se compone de los siguientes elementos:

- Las **DECLARACIONES** y **DEFINICIONES** especifican conceptos y elementos.
- Las **ACCIONES** permiten manipular la información.
- Los **OBJETOS** almacenan información, y son manipulados por la acciones.
- Estructuras de **CONTROL** del flujo de ejecución secuencial.
- Mecanismos para la **MODULARIZACIÓN** adecuada del programa.

Esquema General de un Programa en C++

- En la asignatura utilizaremos un **subconjunto simple** del lenguaje de programación **C++** para describir los algoritmos.

```
// Inclusion de módulos de biblioteca necesarios
#include <iostream>
#include <string>

using namespace std; // Uso del espacio de nombres de las bibliotecas estándares

// Declaraciones y definiciones de Constantes y Tipos

// Declaraciones y definiciones de Subprogramas

int main()        // Definición del Programa Principal
{
    // Declaración de variables locales

    // Secuencia de sentencias (acciones)
}
```

Ejemplo de un Programa en C++

- Diseñe un programa que lee del teclado una cantidad de *pesetas* y muestra en pantalla su equivalente en *euros*, sabiendo que 1 *euro* equivale a 166.386 *pts*.

Algoritmo:

1. Mostrar mensaje en pantalla solicitando la cantidad de pesetas a convertir.
2. Leer la cantidad de pesetas de teclado y almacenarla en una variable
3. Calcular la cantidad de euros equivalente
   1. Dividir la cantidad de pesetas entre 166.386.
   2. Almacenar el resultado en una variable.
4. Mostrar en pantalla la cantidad de euros resultado de la conversión.

Ejecución del Programa

```
Introduzca cantidad de pts: 500 ↵
500 Pts equivalen a 3.00506 Euros
```

El valor 500 lo introduce el usuario desde el teclado
El símbolo ↵ representa la tecla Entrar / Enter

Ejemplo de un Programa en C++

- Diseñe un programa que lee del teclado una cantidad de *pesetas* y muestra en pantalla su equivalente en *euros*, sabiendo que 1 *euro* equivale a 166.386 *pts*.

```
#include <iostream>
using namespace std;
// -- Constantes -------
const double PTS_1_EUR = 166.386;
/* -- Programa Principal -------- */
int main()
{
    int pts;
    cout << "Introduzca cantidad de pts: ";
    cin >> pts;
    double euros = pts / PTS_1_EUR;
    cout << pts << " Pts equivalen a " << euros << " Euros" << endl;
}
```

El programa C++ se debe COMPILAR para generar un programa ejecutable

Ejecución del Programa

```
Introduzca cantidad de pts: 500 ↵
500 Pts equivalen a 3.00506 Euros
```

El valor 500 lo introduce el usuario desde el teclado
El símbolo ↵ representa la tecla Entrar / Enter

Compilación de un programa C++ en el entorno de desarrollo

- Un programa codificado en **C++** está codificado en un lenguaje de programación de **alto nivel**, que el procesador no puede ejecutar directamente.
- El proceso de **compilación** traduce el programa en **C++** a un programa equivalente en **código máquina**, que sí es ejecutable por el procesador.
- El entorno de desarrollo **Code::Blocks** tiene un **editor** y un **compilador de C++** integrado, que facilitará realizar esta tarea. Sin embargo, es necesario **configurar** adecuadamente el proceso de compilación con algunas **opciones** importantes que permitirán realizar una **compilación más segura**.
  - `-O2`: aumenta el nivel de análisis y optimización del código.
  - `-Wall`: permite detectar errores frecuentes durante la compilación.
  - `-Wextra`: permite detectar errores adicionales durante la compilación.
  - `-Werror`: compilación más estricta, trata los *avisos* como **errores**.
  - `-D_GLIBCXX_DEBUG`: permite comprobar los accesos indexados en `string` y `array` durante la ejecución del programa (tema 4).

  Véase el documento correspondiente a la Guía del Entorno de Programación CodeBlocks para saber como se activan.

Elementos Básicos de un Programa en C++

- Palabras reservadas. Ej: `main`, `const`, `int`, `double`
- Tipos. Ej: `int`, `double`
- Identificadores. Ej: `pts`, `euros`, `PTS_1_EUR`
- Constantes literales. Ej: `166.386`, `" Pts equivalen a "`
- Constantes simbólicas. Ej: `PTS_1_EUR`
- Variables. Ej: `pts`, `euros`
- Operadores. Ej: `= + - * / % >> <<`
- Delimitadores. Ej: `( ) { } ; ,`
- Dos clases de comentarios:
  - `//` Comentario de línea
  - `/*` Comentario de bloque `*/`

Las Acciones Manipulan la Información

- Se ejecutan en **SECUENCIA**: cuando se termina de ejecutar la sentencia actual, entonces se ejecuta la siguiente sentencia.
- Se representan mediante **SENTENCIAS** ejecutables:
  - Sentencias de Inicialización y Asignación.
  - Sentencias de Entrada y Salida de Datos.
  - Sentencias de Control del Flujo de Ejecución secuencial.
  - Llamadas a Subprogramas.

Información: Variables y Constantes

- Las **variables** y **constantes** almacenan y representan la **información**, en forma de **datos**, que manipula el programa.
  - Las **Constantes** representan **información** cuyo **valor** se establece antes de la ejecución del programa, en **Tiempo de Compilación**, y **permanece constante** (no cambia) durante la ejecución del mismo, por ejemplo 1 euro equivale a 166.386 pesetas.
  - Las **Variables** almacenan **información** cuyo **valor** se establece durante la ejecución del programa, en **Tiempo de Ejecución**, y **sí puede cambiar** durante la ejecución del mismo, por ejemplo, el valor de `pts` y `euros`.

Ejemplo de un Programa en C++

- Desarrolle un programa que lee del teclado una cantidad de *gramos* y muestra en pantalla su desglose en *toneladas*, *kilogramos* y *gramos*.

Análisis de la Solución

- Gramos en 1 Kilogramo: *1000*
- Kilogramos en 1 Tonelada: *1000*
- Gramos en 1 Tonelada: $1000 \times 1000 = 1000000$
- Por ejemplo para un valor de 3254875 gramos:

```
3254875 | 1000000        254875 | 1000
 254875   3 (Tn)     (gr) 875    254 (Kg)
```

Ejecución del Programa

```
Introduzca la cantidad de gramos: 3254875 ↵
3254875 gr equivalen a 3 Tn, 254 Kg, y 875 gr
```

El valor 3254875 lo introduce el usuario desde el teclado

Ejemplo de un Programa en C++

- Desarrolle un programa que lee del teclado una cantidad de *gramos* y muestra en pantalla su desglose en *toneladas*, *kilogramos* y *gramos*.

```
#include <iostream>
using namespace std;
// -- Constantes -------
const int GRAMOS_1_KILOGRAMO = 1000;
const int KILOGRAMOS_1_TONELADA = 1000;
const int GRAMOS_1_TONELADA = GRAMOS_1_KILOGRAMO * KILOGRAMOS_1_TONELADA;
/* -- Programa Principal -------- */
int main()
{
    int gramos_totales;
    cout << "Introduzca la cantidad de gramos: ";
    cin >> gramos_totales;
    int toneladas = gramos_totales / GRAMOS_1_TONELADA;
    int resto_gramos = gramos_totales % GRAMOS_1_TONELADA;
    int kilos = resto_gramos / GRAMOS_1_KILOGRAMO;
    int gramos = resto_gramos % GRAMOS_1_KILOGRAMO;
    cout << gramos_totales << " gr equivalen a "
         << toneladas << " Tn, " << kilos << " Kg, y " << gramos << " gr"
         << endl;
}
```

Tipos de datos simples predefinidos

Tipo de Datos

- Define el **conjunto de valores** que puede tomar el objeto.
- Determina las **operaciones** que se pueden aplicar al objeto.
- Determina el **espacio** que es necesario para almacenar su valor en memoria.
- Determina la **interpretación** del valor binario almacenado en memoria.

Tipos de Datos Simples o Escalares

- Formados por elementos **indivisibles** y **ordenados**, es decir, les son aplicables los operadores relacionales (== != < > <= >=).
- Predefinidos: `bool`, `char`, `short`, `unsigned short`, `int`, `unsigned`, `long`, `unsigned long`, `float`, `double`, `long double`.
- Definidos por el programador: **`enum`** (tipo enumerado).

Tipos de Datos Compuestos o Estructurados (Tema 4)

- Predefinidos: Cadena de Caracteres (`string`).
- Definidos por el programador:
  - Estructuras (registros).
  - Arrays.

Tipos de datos simples predefinidos

Tamaño de los tipos de datos simples predefinidos

Tipo	Valores	Tamaño	Intervalo Valores
bool	Lógicos	1 byte	`false true`
char	Caracteres	1 byte	Tabla ASCII (256 valores)
short	Enteros	2 bytes	±32767
unsigned short	Naturales	2 bytes	+65535
enum	Enumerados	4 bytes	Valores de la enumeración
int	Enteros	4 bytes	±2147483647
unsigned	Naturales	4 bytes	+4294967295
long	Enteros	4/8 bytes	±9223372036854775807
unsigned long	Naturales	4/8 bytes	+18446744073709551615
long long	Enteros	8 bytes	±9223372036854775807
unsigned long long	Naturales	8 bytes	+18446744073709551615
float	Reales	4 bytes	±1.17549e±38 (7 dec.)
double	Reales	8 bytes	±1.79769e±308 (15 dec.)
long double	Reales	10/16 bytes	±1.18973e±4932 (18 dec.)

- El operador `sizeof(tipo)` devuelve el número de bytes necesario para representar un valor del tipo especificado.

```
cout << "Tamaño del tipo int: " << sizeof(int) << endl;
```

Tipos, Constantes, Variables y Asignaciones

Constantes y Variables

- **Almacenan y representan la información** que manipula el programa.
- **Constantes**: representan información cuyo **valor** se establece **antes de la ejecución** del programa, y no cambia durante la ejecución del mismo.
- **Variables**: almacenan información cuyo **valor** se establece **durante la ejecución** del programa, y sí puede cambiar durante la ejecución del mismo.

- **Nombre** (identificador): identifica cada entidad de nuestro programa y permite referenciarlo. Comienza por **letra**, seguida por **letras**, **dígitos** y **_**
- **Tipo**: que define las características de los datos que son manipulados.

`bool`	`char`	`int`	`double`
Lógicos	Caracteres	N^{os}Enteros	N^{os}Reales
false y true	Tabla ASCII	$\pm$2147483647	$\pm$1.79769e$\pm$308
1 byte	1 byte	4 bytes	8 bytes

Constantes

Constantes Literales

Las **constantes literales** aparecen representadas con su valor de forma dispersa en el programa. Normalmente, salvo excepciones, su uso **no** es recomendable.

```
media = (num1 + num2) / 2.0; // 2.0 es constante literal
```

Constantes Simbólicas

Las **constantes simbólicas** asocian un *nombre* y un *significado* a un determinado valor constante del tipo especificado. Se definen en la zona de constantes al principio del programa, y su ámbito y tiempo de vida abarca todo el programa.

```
const tipo NOMBRE = valor ;
```

Ejemplo

```
const bool DEPURACION = true;
const char LETRA = 'a';
const int MAXIMO = 1000;
const double ERROR_PRECISION = 1.0e-6; // 1.0E-6 equivale a 1.0×10^-6
```

Variables

- Una **variable almacena un valor** según su **tipo**. El valor almacenado **puede cambiar** durante la ejecución del programa. De todos los valores que especifica su tipo, en un momento dado, una variable **sólo almacena uno** de ellos.
- Las **variables locales** se definen dentro del **cuerpo** de los **subprogramas** y de **main**. Su ámbito y tiempo de vida abarca el bloque en el que están definidas.
- Las **variables globales** se definen **fuera del cuerpo** de los **subprogramas** y de **main**. Su ámbito y tiempo de vida abarca todo el programa.
- **La definición de variables globales está prohibida**.
- Una variable tiene asignada una zona de memoria donde se almacena el valor que toma en un momento dado. Se puede especificar el valor que almacenará **inicialmente**, en otro caso, el valor que almacena queda **inespecificado**.

```
tipo nombre ;
```

```
tipo nombre = expresión ;
```

Ejemplo

```
int main() {
    bool encontrado = false;   // valor inicial false
    char letra = 'z';          // valor inicial 'z'
    int cnt_1, cnt_2;          // valor inicial inespecificado
    int maximo = 56;           // valor inicial 56
    double media;              // valor inicial inespecificado
```

Inicialización, Asignación y Expresiones Aritméticas

- La **inicialización** asigna a una **variable** un valor **inicial** resultado de evaluar una expresión.
- La **asignación** asigna a una **variable** un **nuevo valor** resultado de evaluar una expresión. El valor que estaba anteriormente almacenado se **pierde**.

```
nombre_de_variable = expresión ;
```

- La evaluación de una expresión se realiza según unas reglas de precedencia y de asociatividad.

de mayor a menor	Precedencia	Asociat.
paréntesis	( )	No
unarios	! -	Dch
multiplicativos	* / %	Izq
sumatorios	+ -	Izq
relacionales	< > <= >=	No
igualdad	== !=	No
lógico y	&&	Izq
lógico o	||	Izq

```
int main()
{
    int x1 = 4 + 2 * 6;

    x1 =   2 +   x1 * 3  / 2   - 3;

    x1 =   2 +  (x1 * 3) / 2   - 3;
    x1 =   2 + ((x1 * 3) / 2)  - 3;
    x1 =  (2 + ((x1 * 3) / 2)) - 3;
    x1 = ((2 + ((x1 * 3) / 2)) - 3);
}
```

- Nótese la diferencia entre los símbolos de **asignación** (=) y de **igualdad** (==).
- A todos los tipos simples (incluido el `char`) se le pueden aplicar los operadores relacionales (== != < > <= >=).

Operadores

Operador	Significado
`( expr )`	Paréntesis, modifica el orden de evaluación.
`! X`	Negación lógica unaria (véase tabla de verdad)
`- X`	Menos unario, negativo, cambio de signo.
`X * Z`	Multiplicación aritmética.
`X / Z`	División aritmética (entera o real).
`X % Z`	Resto de la división aritmética entera (módulo).
`X + Z`	Suma aritmética.
`X - Z`	Resta aritmética.
`X > Z`	Mayor, `true` si `X` mayor que `Z`, `false` en otro caso.
`X >= Z`	Mayor o igual, `true` si `X` mayor o igual que `Z`, `false` en otro caso.
`X < Z`	Menor, `true` si `X` menor que `Z`, `false` en otro caso.
`X <= Z`	Menor o igual, `true` si `X` menor o igual que `Z`, `false` en otro caso.
`X == Z`	Igualdad, `true` si `X` igual a `Z`, `false` en otro caso.
`X != Z`	Distinto, `true` si `X` distinto a `Z`, `false` en otro caso.
`X && Z`	Conjunción (Y)/(AND) lógica (véase tabla de verdad).
`X || Z`	Disyunción (O)/(OR) lógica (véase tabla de verdad).

Asignacion y Expresiones Aritméticas

Sentencias de Asignación Especiales

```
int main()
{
    int x1 = 4 + 2 * 6; // Inicialización (16)

    ++x1;               // x1 = x1 + 1;

    --x1;               // x1 = x1 - 1;

    x1++;               // x1 = x1 + 1;

    x1--;               // x1 = x1 - 1;

    x1 += expr;         // x1 = x1 + ( expr );

    x1 -= expr;         // x1 = x1 - ( expr );

    x1 *= expr;         // x1 = x1 * ( expr );

    x1 /= expr;         // x1 = x1 / ( expr );

    x1 %= expr;         // x1 = x1 % ( expr );
}
```

Conversiones de Tipo

- En las expresiones, cuando se evalúa una operación aplicada sobre **operandos de tipos distintos**, se realiza una conversión de tipo **automática** al tipo simple más **amplio** de los operandos.
  - bool ▸ char ▸ short ▸ enum ▸ int ▸ unsigned ▸ long ▸ unsigned long ▸ float ▸ double ▸ long double

```
double a = 5 / 2;    // valor de a ? -> 2.0 = (5 / 2)     división entera
double b = 5 / 2.0;  // valor de b ? -> 2.5 = (5.0 / 2.0) división real
```

- En la evaluación de expresiones, a veces nos puede interesar que un determinado operando o expresión sea **evaluado** a un valor con un **tipo distinto al suyo propio**. En estos casos, se realiza una **conversión explícita** de tipos (*casting*). Para ello, se precede el valor o expresión entre paréntesis por el símbolo del tipo al que queremos convertir:

```
int      a = int(true);      // produce el entero 1
bool     b = bool(0);        // produce el valor lógico false

int      c = int('a');       // produce el entero 97    (según tabla ASCII)
char     d = char(65);       // produce el carácter 'A' (según tabla ASCII)

double   i = double(2);      // produce el real 2.0
int      j = int(3.6);       // produce el entero 3 (pierde decimales)
```

Conversiones de Tipo. Ejemplo

- ¿ Cuales serán los valores que tomen las variables z1, z2 y z3 ?

```
#include <iostream>
using namespace std;
int main()
{
    int    x  = 1;
    int    y  = 2;
    double z1 = x / y * 1e6;
    double z2 = double(x / y) * 1e6;
    double z3 = double(x) / double(y) * 1e6;
}
```

La división es una operación propensa a errores

Atención a la **división por cero**, y a la **división entera** en vez de la **división real**.

Entrada y Salida de Datos Básica

- Para realizar operaciones de entrada y salida básicas es necesario **incluir** la biblioteca estándar **<iostream>** al principio del programa, y usar el **espacio de nombres std**.

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

- Se puede utilizar para todos los **tipos simples predefinidos**.
- **No** está definida para el **tipo enumerado** (tipo definido por el programador).
- También se puede utilizar para las **cadenas de caracteres** (véase tema 4).

Entrada de Datos >>

El **operador de entrada** (>>) permite asignar a una **variable** un determinado valor *entrado* desde el flujo de entrada de teclado (**cin**). Ej: `cin >> x1 >> x2 ;`

Salida de Datos <<

El **operador de salida** (<<) permite **mostrar** en el flujo de salida a pantalla (**cout**) el valor de una constante, variable o expresión (**endl** ≡ *Salto-de-Línea*).
Ej: `cout << "Valor: " << x1 << " " << (x2 * 3) << endl ;`

Entrada y Salida de Datos Básica

Ejemplo

```
#include <iostream>
using namespace std ;
int main()
{
  int x1, x2 ;
  cout << "Introduce dos números: " ;
  cin >> x1 >> x2 ;

  cout << "X " << x1 << " " << (3*x2) << endl;
  cout << "FIN" << endl;// endl Salto-de-Línea
}
```

```
#include <iostream>
using namespace std ;
int main()
{
  int x1, x2 ;
  cout << "Introduce dos números: " ;
  cin >> x1 ;
  cin >> x2 ;
  cout << "X " << x1 << " " << (3*x2) << endl;
  cout << "FIN \n" ;// "\n" Salto-de-Línea
}
```

Ejecución del Programa

```
Introduce dos números: 12␣15 ↵
X 12 45
FIN
```

Los valores 12 y 15 los introduce el usuario desde el teclado
Los datos de entrada se separan por espacios o ENTER
Al final de la entrada se pulsa la tecla ENTER

```
Introduce dos números: 12 ↵
15 ↵
X 12 45
FIN
```

Los valores 12 y 15 los introduce el usuario desde el teclado
Los datos de entrada se separan por espacios o ENTER
Al final de la entrada se pulsa la tecla ENTER
El símbolo ↵ representa la tecla Entrar / Enter

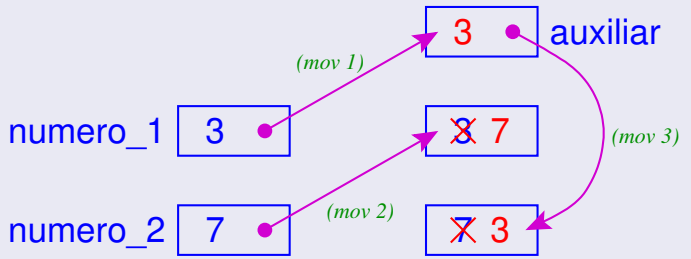
Ejemplos Simples

1. Programa que lea dos números enteros y los intercambie.
2. Programa que lea un número entero y muestre **`true`** si el número leído es par, y muestre **`false`** si es impar.
3. Programa que lea un carácter, suponemos que es una letra mayúscula, la convierta a minúscula y muestra el resultado.
   - Los dígitos del `'0'` al `'9'` están consecutivos en la tabla ASCII.
   - Las letras de la `'A'` a la `'Z'` están consecutivas en la tabla ASCII.
   - Las letras de la `'a'` a la `'z'` están consecutivas en la tabla ASCII.

Ejemplo 1

Programa que lea dos números enteros y los intercambie.

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

int main()
{
    int numero_1, numero_2;
    cout << "Introduzca dos números: ";
    cin >> numero_1 >> numero_2;

    int auxiliar = numero_1;          // (mov 1)
    numero_1 = numero_2;              // (mov 2)
    numero_2 = auxiliar;              // (mov 3)

    cout << "Resultado: " << numero_1 << " " << numero_2 << endl;
}
```



Ejemplo 2

Programa que lea un número entero y muestre **`true`** si el número leído es par, y muestre **`false`** si es impar.

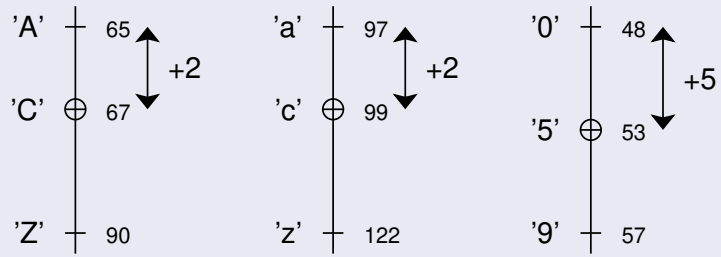
```
#include <iostream>
using namespace std;

int main()
{
    int numero;
    cout << "Introduzca un número: ";
    cin >> numero;
    int resto = numero % 2;
    bool par = (resto == 0);  // par = ((numero % 2) == 0);
    cout << "El número " << numero << " es par: " << boolalpha << par << endl;
}
```

- `cout << boolalpha` permite mostrar los valores lógicos (booleanos) como `false` o `true` (en vez de 0 y 1).

Ejemplo 3

Programa que lea un carácter, suponemos que es una letra mayúscula, la convierta a minúscula y muestra el resultado.

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

int main()
{
    char letra_mayuscula;
    cout << "Introduzca una letra mayúscula: ";
    cin >> letra_mayuscula;
    // Suponemos que es una letra mayúscula
    int distancia = int(letra_mayuscula) - int('A');
    char letra_minuscula = char(int('a') + distancia);
    // letra_minuscula = char(int('a') + (int(letra_mayuscula) - int('A')));
    // letra_minuscula = char('a' + (letra_mayuscula - 'A'));
    cout << letra_mayuscula << " -> " << letra_minuscula << endl;
}
```



En nuestros programas, nunca debemos trabajar directamente con los valores de la tabla ASCII, ya que son valores que podemos calcular directamente dentro de nuestro código. Respecto a la tabla ASCII, sólo debemos tener en consideración lo siguiente:

- Los dígitos del `'0'` al `'9'` están consecutivos en la tabla ASCII.
- Las letras de la `'A'` a la `'Z'` están consecutivas en la tabla ASCII.
- Las letras de la `'a'` a la `'z'` están consecutivas en la tabla ASCII.

Expresiones Lógicas o Booleanas

- Una **expresión lógica** es una expresión que será evaluada a un valor lógico *verdadero* o *falso* (`true` o `false`). Por Ejemplo: `(((x + 6) >= 10) && (z < 20))`
- Permiten comprobar el **estado** de la computación en un momento determinado. Son la base para la **toma de decisiones** en nuestros programas.
- La evaluación de una expresión lógica se realiza según unas reglas de precedencia y de asociatividad, y la tabla de verdad de los operadores lógicos.
- Los operadores relacionales (`< <= > >= == !=`) producen un valor lógico como resultado de la comparación.
- Los operadores lógicos (`! && ||`) producen un valor lógico como resultado.
- Los operadores lógicos (`&& ||`) se evalúan en **CORTOCIRCUITO**.

Tabla de verdad de los operadores lógicos

x	y	x && y	x || y	! x
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Evaluación en Cortocircuito de Expresiones Lógicas

- Los operadores **AND** y **OR** (`&&` , `||`) se evalúan en **CORTOCIRCUITO**.

- Cuando ya se conoce el resultado de la operación lógica trás la evaluación del primer operando, entonces el **segundo operando NO se evalúa**.
  - Para el operador **AND**, cuando el primer operando se evalúa a **false**, entonces el resultado de la operación AND es **false** sin necesidad de evaluar el segundo operando.
  - Para el operador **OR**, cuando el primer operando se evalúa a **true**, entonces el resultado de la operación OR es **true** sin necesidad de evaluar el segundo operando.

- Sin embargo, si trás la evaluación del primer operando no se puede calcular el resultado de la operación lógica, entonces el **segundo operando SÍ se evalua**.
  - Para el operador **AND**, cuando el primer operando se evalúa a **true**, entonces el resultado de la operación AND es el resultado de evaluar el segundo operando.
  - Para el operador **OR**, cuando el primer operando se evalúa a **false**, entonces el resultado de la operación OR es el resultado de evaluar el segundo operando.

Expresiones Lógicas: Ejemplos

- Codifique las siguientes expresiones lógicas.

```
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    bool valor1 = ...;            // Toma el valor falso
    bool valor2 = ...;            // Toma el valor verdadero
    bool par = ... ;              // True sí NUM es par
    bool tres_digitos = ...;      // True sí NUM tiene tres dígitos
    bool tres_digitos_par = ...; // True sí NUM tiene tres dígitos y es par

    bool primo_10 = ...;          // True sí NUM un número primo menor que 10
    bool divisor_100 = ...;       // True sí NUM es un divisor de 100
}
```

- Desarrolle un programa que lea un número entero y muestre `true` si tiene tres dígitos, y además es capicúa. En otro caso muestra `false`.

Expresiones Lógicas: Ejemplo 1

- Codifique las siguientes expresiones lógicas.

```
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    bool valor1 = false;
    bool valor2 = true;
    bool par = (num % 2) == 0;
    bool tres_digitos = (num >= 100) && (num <= 999);
    bool tres_digitos_par = tres_digitos && par;
    bool tres_digitos_par = ((num >= 100)&&(num <= 999)) && ((num % 2) == 0);
    bool primo_10 = ((num >= 2) && (num <= 3)) || (num == 5) || (num == 7);
    bool divisor_100 = (num != 0) && ((100 % num) == 0); // CORTOCIRCUITO
}
```

Expresiones Lógicas: Ejemplo 2

- Desarrolle un programa que lea un número entero y muestre `true` si tiene tres dígitos, y además es capicúa. En otro caso muestra `false`.

```
int main()
{
    int numero;
    cout << "Introduce un número: ";
    cin >> numero;

    int digito_1 = numero % 10;
    int digito_3 = (numero / 100) % 10;
    bool tres_cap = (numero >= 100) && (numero <= 999)
                  && (digito_1 == digito_3);

    cout << numero << " tiene 3 dígitos y es capicúa: "
         << boolalpha << tres_cap << endl;
}
```

Expresiones Lógicas. Equivalencias

`! ( ! b)`	$\Leftrightarrow$	`(b)`
`! (a && b)`	$\Leftrightarrow$	`( ! a || ! b)`
`! (a || b)`	$\Leftrightarrow$	`( ! a && ! b)`
`! (x == y)`	$\Leftrightarrow$	`(x != y)`
`! (x > y)`	$\Leftrightarrow$	`(x <= y)`
`! (x < y)`	$\Leftrightarrow$	`(x >= y)`
~~`(b == true)`~~	$\Leftrightarrow$	`(b)`
~~`(b == false)`~~	$\Leftrightarrow$	`( ! b)`
~~`(b != true)`~~	$\Leftrightarrow$	`( ! b)`
~~`(b != false)`~~	$\Leftrightarrow$	`(b)`
`((a && b) || (a && c))`	$\Leftrightarrow$	`a && (b || c)`
`((a || b) && (a || c))`	$\Leftrightarrow$	`a || (b && c)`

Expresiones Lógicas. Ejercicios

- Desarrolle un programa que lea de teclado datos dos números enteros (`x` e `y`) y un carácter (`c`), y muestre `true` por cada una de las siguientes propiedades que cumplan, y `false` en caso contrario.
  1. **x** $\in \{ 3, 4, 5, 6, 7 \}$
  2. **x** $\in \{ 1, 2, 3, 7, 8, 9 \}$
  3. **x** $\in \{ 1, 3, 5, 7, 9 \}$
  4. **x** $\in \{ 2, 5, 6, 7, 8, 9 \}$
  5. **x** $\in \{ 3, 4, 6, 8, 9 \}$, **y** $\in \{ 6, 7, 8, 3 \}$
  6. Ni **x** ni **y** sean mayores que **10**
  7. **x** no sea multiplo de **y**
  8. **c** es una letra mayúscula
  9. **c** es una letra (mayúscula o minúscula)
  10. **c** es un alfanumérico (mayúscula, minúscula o dígito)

- Simplifique las siguientes expresiones lógicas:
  1. `! (x > y) && (x < z)`
  2. `! ((x > y) && (x < z))`
  3. `! ((x == y) || (x != z))`
  4. `! ((x == y) || ((x < y) && (x > z)))`
  5. `(x == y) || (((x < y) && (x > z)) == false)`
  6. `((x != y) && (y < z)) || (((x == y) || (x > z)) != true)`

Control del Flujo de Ejecución de las Sentencias

- El flujo de ejecución de nuestros programas es **secuencial**.
  - Cuando se termina de ejecutar una sentencia, entonces se ejecuta la siguiente.
- Hasta ahora, el flujo de ejecución de las sentencias coincide con el **orden** en el que han sido codificadas.
  - Esto no es suficiente para resolver problemas.
  - Necesitamos nuevas sentencias que permitan **controlar el flujo de ejecución** bajo determinadas **condiciones** (expresiones lógicas).

Control del Flujo de Ejecución de las Sentencias

Secuencia

- Cuando se termina de ejecutar una sentencia, entonces se ejecuta la siguiente.

Sentencias de Selección

- Permiten **seleccionar** la ejecución **alternativa** y **excluyente** de unas sentencias u otras dependiendo del valor de una **expresión lógica**.

Sentencias de Iteración

- **Repiten** la ejecución de una secuencia de sentencias.
- El **número de iteraciones** depende de una **expresión lógica** cuyo valor evoluciona durante la ejecución del cuerpo del bucle.

Anidamientos

- Estas estructuras de control se pueden **anidar**, es decir, se pueden poner unas dentro de otras para que el flujo de ejecución sea el adecuado para resolver el problema.
- El ámbito de vida de una variable abarca el bloque { } en el que está definida.
  - Cuando se declara una variable **dentro de un bucle**, entonces la variable se **crea** y se **destruye** en **cada iteración** (el valor **no perdura** entre iteraciones).

Ejercicios

1. Programa que lea un número, calcule su valor absoluto y lo muestre en pantalla.
2. Programa que lea dos números enteros y muestre un mensaje indicando si *num1* es divisible por *num2*.
3. Programa que lee un carácter, si el caracter leído es una letra minúscula, la convierte a mayúscula, si el caracter leído es una letra mayúscula, la convierte a minúscula, en otro caso no la modifica. Finalmente, muestra el resultado de la transformación.
4. Desarrolle un programa que lea un número real comprendido entre 0 y 10 y muestre la nota asociada.
   - Matricula de Honor ($n = 10$), Sobresaliente ($9 \leq n < 10$), Notable ($7 \leq n < 9$), Aprobado ($5 \leq n < 7$), Suspenso ($0 \leq n < 5$), Error.
5. Desarrolle un programa que muestre el mayor de tres números leídos de teclado.
6. Desarrolle un programa que muestre el mayor de cuatro números leídos de teclado.
7. Calcular el resultado de una operación aritmética (`+ - * /`) y dos números leídos de teclado.
8. Desarrolle un programa que lea de teclado tres números (día, mes y año), si la fecha es correcta mostrará el mensaje **`Fecha Correcta`**, en otro caso mostrará **`Error`**.

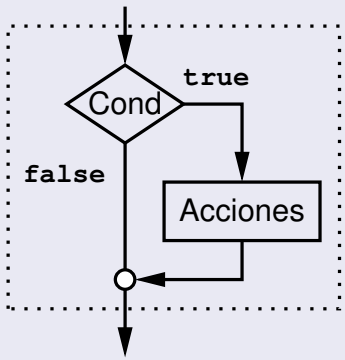
Sentencias de Selección

- Permiten **seleccionar** la ejecución **alternativa** y **excluyente** de unas sentencias u otras dependiendo del valor de una **expresión lógica**.

Sentencia de Selección Simple (`if`)

- Se evalúa la condición (expresión lógica).
  - Si el resultado es `true`, entonces se ejecutan las sentencias del **cuerpo del if**.
  - En otro caso, **no se ejecuta nada**.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

```
int main()
{
    // Sentencia de Selección Simple
    if ( EXPR_LOGICA_1 ) {
        // Acciones_1
    }
    // Siguiente sentencia
}
```



Sentencia de Selección Simple (`if`)

- Calcular el valor absoluto de un número entero.

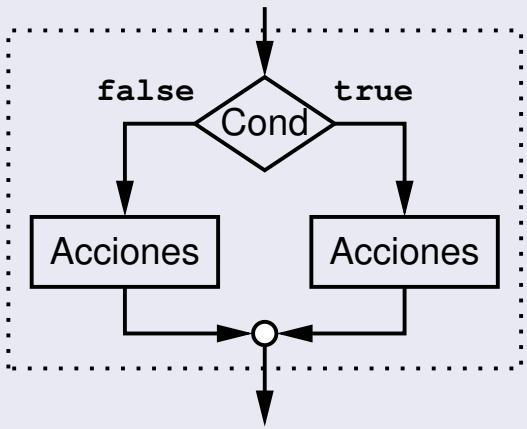
```
#include <iostream>
using namespace std;

int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    if (num < 0) {
        num = -num;
    }
    cout << num << endl;
}
```

Sentencias de Selección

Sentencia de Selección Compuesta (`if-else`)

- Se evalúa la condición (expresión lógica).
  - Si el resultado es `true`, entonces se ejecutan las sentencias del **cuerpo del if**.
  - En otro caso, se ejecutan las sentencias del **cuerpo del else**.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

```
int main()
{
    // Sentencia de Selección Compuesta
    if ( EXPR_LOGICA_1 ) {
        // Acciones_1
    } else {
        // Acciones_otro_caso
    }
    // Siguiente sentencia
}
```



Sentencia de Selección Compuesta (if-else)

- Calcular si un número es divisible por otro.

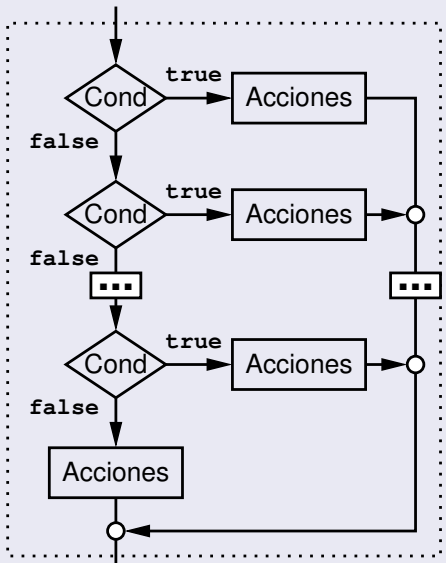
```
#include <iostream>
using namespace std;

int main()
{
    int num1, num2;
    cout << "Introduce dos números: ";
    cin >> num1 >> num2;
    if ((num2 != 0) && ((num1 % num2) == 0)) {
        cout << num1 << " es divisible por " << num2 << endl;
    } else {
        cout << num1 << " no es divisible por " << num2 << endl;
    }
}
```

Sentencias de Selección

Sentencia de Selección Compuesta Encadenada (`if-else-if`)

- Se van evaluando las expresiones lógicas, en el orden especificado, si la evaluación de una expresión logica es **false**, se continúa evaluando la siguiente expresión lógica, hasta que la evaluación de una expresión logica sea **true**, entonces se ejecutan las acciones correspondientes a dicha expresión lógica, y el resto de expresiones lógicas y acciones correspondientes son desechadas y no se ejecutan.
- Si ninguna expresión lógica se ha evaluado a **true**, entonces se ejecutan las acciones correspondientes a la cláusula **else**, en caso de que no exista clausula **else**, entonces no se hace nada.
- Finalmente, se pasa a ejecutar la siguiente sentencia.

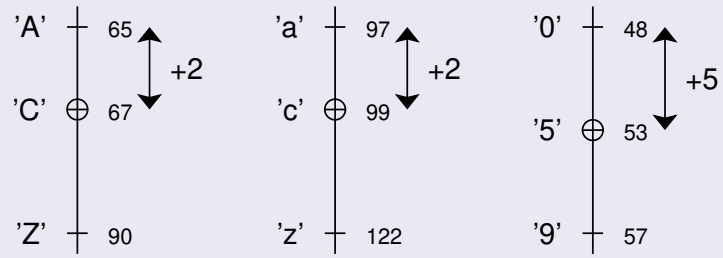
```
int main()
{
    // Sentencia de Selección Encadenada
    if ( EXPR_LOGICA_1 ) {
        // Acciones_1
    } else if ( EXPR_LOGICA_2 ) {
        // Acciones_2
    // <...> todas las condiciones que se deseen
    } else if ( EXPR_LOGICA_3 ) {
        // Acciones_3
    } else {
        // Acciones_otro_caso
    }
    // Siguiente sentencia
```



Sentencias de Selección. Ejemplo 1

- Programa que lee un carácter, si el caracter leído es una letra minúscula, la convierte a mayúscula, si el caracter leído es una letra mayúscula, la convierte a minúscula, en otro caso no la modifica. Finalmente, muestra el resultado de la transformación.

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

int main()
{
    char caracter, nuevo;
    cout << "Introduzca un caracter: ";
    cin >> caracter;
    if ('A' <= caracter && caracter <= 'Z') {
        nuevo = char('a' + (caracter - 'A'));
    } else if ('a' <= caracter && caracter <= 'z') {
        nuevo = char('A' + (caracter - 'a'));
    } else {
        nuevo = caracter;
    }
    cout << "Resultado: " << nuevo << endl;
}
```



Sentencias de Selección. Ejemplo 2

- Desarrolle un programa que lea un número real comprendido entre 0 y 10 y muestre la nota asociada.
  - Matricula de Honor ($n = 10$), Sobresaliente ($9 \leq n < 10$), Notable ($7 \leq n < 9$), Aprobado ($5 \leq n < 7$), Suspenso ($0 \leq n < 5$), Error.

```
#include <iostream>
using namespace std;
int main()
{
    double nota;
    cout << "Introduce la nota: ";
    cin >> nota;
    if ( ! ((nota >= 0.0) && (nota <= 10.0))) {
        cout << "Error: 0 <= n <= 10" << endl;
    } else if (nota == 10.0) {
        cout << "Matricula de Honor" << endl;
    } else if (nota >= 9.0) {
        cout << "Sobresaliente" << endl;
    } else if (nota >= 7.0) {
        cout << "Notable" << endl;
    } else if (nota >= 5.0) {
        cout << "Aprobado" << endl;
    } else {
        cout << "Suspenso" << endl;
    }
}
```

Sentencias de Selección. Ejemplo 3 (v1)

- Desarrolle un programa que muestre el mayor de tres números leídos de teclado.

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

int main()
{
    int a, b, c, mayor;
    cout << "Introduce tres números: ";
    cin >> a >> b >> c;
    if (a >= b && a >= c) {
        mayor = a;
    } else if (b >= a && b >= c) {
        mayor = b;
    } else {
        mayor = c;
    }
    cout << mayor << endl;
}
```

Sentencias de Selección. Ejemplo 3 (v2)

- Desarrolle un programa que muestre el mayor de tres números leídos de teclado. *En este caso, esta solución es más flexible y adaptable*.

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

int main()
{
    int a, b, c, mayor;
    cout << "Introduce tres números: ";
    cin >> a >> b >> c;
    mayor = a;
    if (b > mayor) {
        mayor = b;
    }
    if (c > mayor) {
        mayor = c;
    }
    cout << mayor << endl;
}
```

Sentencias de Selección. Ejemplo 4

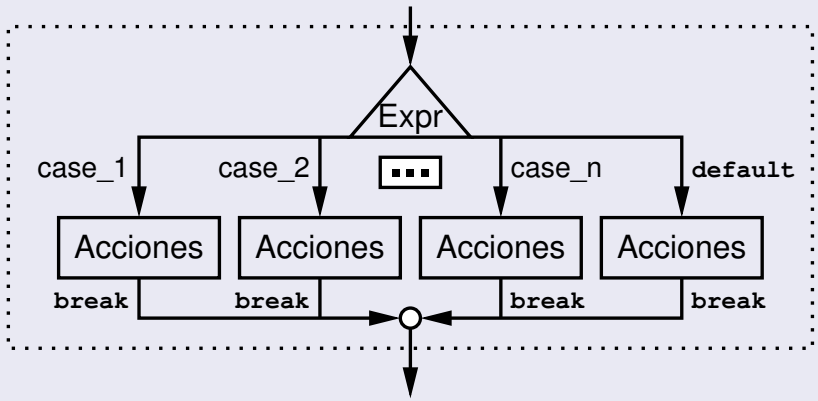
- Desarrolle un programa que muestre el mayor de cuatro números leídos de teclado.

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

int main()
{
    int a, b, c, d, mayor;
    cout << "Introduce cuatro números: ";
    cin >> a >> b >> c >> d;
    mayor = a;
    if (b > mayor) {
        mayor = b;
    }
    if (c > mayor) {
        mayor = c;
    }
    if (d > mayor) {
        mayor = d;
    }
    cout << mayor << endl;
}
```

Sentencia de Selección Múltiple (`switch`)

- Se evalúa la expresión y se compara su resultado con los valores de los diferentes **casos**.
  - En caso de que el valor de la expresión sea igual al valor de algún caso, entonces se ejecutan las sentencias del **cuerpo del caso** (hasta el **break**, que no se debe **olvidar**).
  - Si el valor de la expresión no es igual al ningún valor de los casos, entonces se ejecutan las sentencias del **cuerpo del default** (hasta el **break**, que no se debe **olvidar**).
- Finalmente, se pasa a ejecutar la siguiente sentencia.
- La expresión y los valores constantes de los *casos* deben ser del mismo **tipo simple**, los números reales (`float`, `double`, `long double`) no están permitidos.
- **NO** está permitido declarar variables dentro de los casos de la sentencia `switch`.

```
int main() {
    switch ( EXPRESION ) {
    case CTE_1:
        // Acciones_CTE_1
        break;
    case CTE_2:
    case CTE_3:
        // Acciones_CTE_2_CTE_3
        break;
    default:
        // Acciones_otro_caso
        break;
    }
```



Sentencias de Selección. Ejemplos

- Desarrolle un programa que lea un número que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

1	–>	Lunes
2	–>	Martes
3	–>	Miércoles
4	–>	Jueves
5	–>	Viernes
6	–>	Sábado
7	–>	Domingo
otro caso	–>	Error

- Desarrolle un programa que lea una letra que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

l L	–>	Lunes
m M	–>	Martes
x X	–>	Miércoles
j J	–>	Jueves
v V	–>	Viernes
s S	–>	Sábado
d D	–>	Domingo
otro caso	–>	Error

Sentencias de Selección. Ejemplo 1

- Desarrolle un programa que lea un número que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

```
#include <iostream>
using namespace std;
int main()
{
    int numero_semana;
    cout << "Introduce el número: ";
    cin >> numero_semana;
    switch (numero_semana) {
    case 1:
        cout << "Lunes" << endl;
        break;
    case 2:
        cout << "Martes" << endl;
        break;
    case 3:
        cout << "Miércoles" << endl;
        break;
    case 4:
        cout << "Jueves" << endl;
        break;
    case 5:
        cout << "Viernes" << endl;
        break;
    case 6:
        cout << "Sábado" << endl;
        break;
    case 7:
        cout << "Domingo" << endl;
        break;
    default:
        cout << "Error" << endl;
        break;
    }
}
```

Sentencias de Selección. Ejemplo 2

- Desarrolle un programa que lea una letra que designe el día de la semana y muestre por pantalla el nombre del día de la semana correspondiente.

```
#include <iostream>
using namespace std;
int main()
{
    char letra_semana;
    cout << "Introduce la letra: ";
    cin >> letra_semana;
    switch (letra_semana) {
    case 'l':
    case 'L':
        cout << "Lunes" << endl;
        break;
    case 'm':
    case 'M':
        cout << "Martes" << endl;
        break;
    case 'x':
    case 'X':
        cout << "Miércoles" << endl;
        break;
    case 'j':
    case 'J':
        cout << "Jueves" << endl;
        break;
    case 'v':
    case 'V':
        cout << "Viernes" << endl;
        break;
    case 's':
    case 'S':
        cout << "Sábado" << endl;
        break;
    case 'd':
    case 'D':
        cout << "Domingo" << endl;
        break;
    default:
        cout << "Error" << endl;
        break;
    }
}
```

Sentencias de Selección. Ejemplo 3

- Calcular el resultado de una operación aritmética (+ - * /) y dos números leídos de teclado.

```
#include <iostream>
using namespace std;
int main() {
    char operador; double num1, num2, resultado; bool ok = true; // compacto por espacio
    cout << "Introduce el operador y dos operandos: ";
    cin >> operador >> num1 >> num2;
    switch (operador) {
    case '+': resultado = num1 + num2;
              break;
    case '-': resultado = num1 - num2;
              break;
    case '*': resultado = num1 * num2;
              break;
    case '/': if (num2 != 0) {
                resultado = num1 / num2;
              } else {
                ok = false;
              }
              break;
    default:  ok = false;
              break;
    }
    if (ok) {
        cout << "Resultado: " << resultado << endl;
   } else {
        cout << "Error" << endl;
   }
}
```

Sentencias de Selección. Anidamientos. Ejemplo 1

Programa que lea de teclado tres números enteros (día, mes y año), si la fecha es correcta mostrará el mensaje `Fecha Correcta`, en otro caso mostrará `Error`, considerando que:

1. Cualquier valor del año es correcto.
2. Sólo son correctos los meses desde el número **1** hasta el número **12**, ambos inclusive, de tal forma que el número **1** corresponde al mes de **Enero**, el número **2** al mes de **Febrero**, y así sucesivamente hasta el número **12** que corresponde al mes de **Diciembre**.
3. El número correspondiente al día debe ser **mayor que cero**, y menor o igual al número de días del mes en que se encuentra, teniendo en cuenta si el año en que se encuentra es **bisiesto** o no.
   a. Los meses *Enero*, *Marzo*, *Mayo*, *Julio*, *Agosto*, *Octubre*, y *Diciembre* tienen **31** días.
   b. Los meses *Abril*, *Junio*, *Septiembre* y *Noviembre* tienen **30** días.
   c. El mes *Febrero* tiene **29** días si el año en que se encuentra es **bisiesto**, y en otro caso tiene **28** días.
   d. Un año es bisiesto si es divisible entre **400**, o si es divisible entre **4** y no es divisible entre **100**.

Sentencias de Selección. Anidamientos. Ejemplo 1

```
#include <iostream>
using namespace std;
int main()
{
    int dia, mes, anyo;
    cout << "Introduzca día, mes y año: ";
    cin >> dia >> mes >> anyo;
    bool ok = false;
    if (1 <= mes && mes <= 12) {
        int ndias;   // su ámbito de vida abarca el bloque { } en el que está definida
        if (mes == 2) {
            if ((anyo % 400 == 0)||((anyo % 4 == 0)&&(anyo % 100 != 0))) {
                ndias = 29;
            } else {
                ndias = 28;
            }
        } else if ((mes == 4)||(mes == 6)||(mes == 9)||(mes == 11)) {
            ndias = 30;
        } else {
            ndias = 31;
        }
        ok = (1 <= dia && dia <= ndias);
    }
    if ( ok ) {
        cout << "Fecha Correcta" << endl;
    } else {
        cout << "Error" << endl;
    }
}
```

```
if (1 <= dia && dia <= ndias) {
  ok = true;
} else {
  ok = false;
}
```

Sentencias de Selección. Anidamientos. Ejemplo 1

```
int main() {
    int dia, mes, anyo;
    cout << "Introduzca día, mes y año: ";
    cin >> dia >> mes >> anyo;
    bool ok = false;
    if (1 <= mes && mes <= 12) {
        int ndias;   // su ámbito de vida abarca el bloque { } en el que está definida
        switch (mes) {
        case 2:
            if ((anyo % 400 == 0)||((anyo % 4 == 0)&&(anyo % 100 != 0))) {
                ndias = 29;
            } else {
                ndias = 28;
            }
            break;
        case 4: case 6: case 9: case 11:    // compacto por falta de espacio
            ndias = 30;
            break;
        default:
            ndias = 31;
            break;
        }
        ok = (1 <= dia && dia <= ndias);
    }
    if ( ok ) {
        cout << "Fecha Correcta" << endl;
    } else {
        cout << "Error" << endl;
    }
}
```

El Operador Condicional Ternario (?:)

El operador condicional ternario (?:)

`(cond ? op2 : op3)`

- El operador condicional ternario `(cond ? op2 : op3)` evalúa la condición (cond), si es verdadera, entonces el resultado de la expresión es el valor del segundo operando (op2). En otro caso, el resultado es el valor del tercer operando (op3).
  - Se debe utilizar de forma simple, ya que puede dar lugar a expresiones complejas.

```
int mayor = (x > y) ? x : y;     // Asigna el valor mayor de X e Y
int menor = (x < y) ? x : y;     // Asigna el valor menor de X e Y
```

Sentencias de Iteración. Bucles

- **Repiten** la ejecución de una secuencia de sentencias, el **cuerpo del bucle**.
- El **número de iteraciones** depende de una *expresión lógica*, denominada **condición de control del bucle**, cuyo valor evoluciona durante la ejecución del cuerpo del bucle.
- El número de iteraciones realizadas por un bucle puede ser:
  - **Predeterminado**: el número de iteraciones a realizar se conoce con anterioridad al inicio de las iteraciones. Bucle **for**
  - **No predeterminado**: el número de iteraciones a realizar depende de la propia ejecución de las sentencias iterativas. Bucles **while** y **do-while**

Diseño de Bucles

- Para el diseño correcto de un bucle se deben considerar los siguientes aspectos:
  1. **Control de la iteración**.
     1. **¿Cuando debe terminar la ejecución del bucle?**
        Estudiar la condición de control del bucle y cuando debe terminar la iteración.
     2. **Inicialización de la condición de control del bucle.**
        Las variables implicadas en la condición deben tener valores iniciales adecuados.
     3. **Evolución de la condición de control del bucle.**
        Las variables implicadas en la condición deben evolucionar durante el proceso iterativo, desde sus valores iniciales, hasta que finalmente tomen valores que permitan terminar la iteración al evaluar la condición de control del bucle.
  2. **Proceso iterativo**.
     1. **Acción iterativa**
        ¿Cual es el proceso cuya ejecución queremos repetir ?
     2. **Inicialización del proceso iterativo.**
        Las variables implicadas en el proceso iterativo deben tener valores iniciales adecuados.
     3. **Evolución del proceso iterativo.**
        Las variables implicadas en el proceso iterativo pueden evolucionar durante el mismo.
- A veces se entremezclan ambos aspectos, el control de la iteración y el proceso iterativo.

Sentencias de Iteración. Bucles. Ejercicios

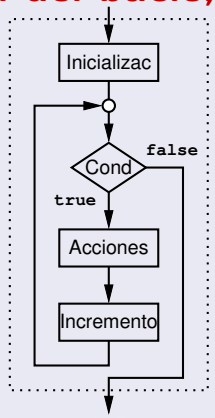
1. Programa que muestra los N primeros números enteros mayores o iguales a cero, siendo N un número leído desde teclado, ej. para *N=6*: `0 1 2 3 4 5`
2. Programa que muestra los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para *N=6*: `5 4 3 2 1 0`
3. Programa que muestra los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para *N=6*: `1 3 5 7 9 11`

Sentencias de Iteración. Bucles

Sentencia de Iteración For

- Se utiliza cuando el **número de iteraciones** que se van a realizar está **predeterminado** antes de su ejecución, y no depende de la ejecución del cuerpo del bucle.
- El proceso iterativo está **controlado por los valores** que toma una **variable de control**.
- INICIALIZACIÓN: **inicializa la variable de control**. Se ejecuta **una única vez** antes del proceso iterativo. El ámbito de vida de la variable de control abarca todo el proceso iterativo.
- EXPR_LÓGICA: una **expresión lógica** que utiliza la **variable de control** para determinar el **número de iteraciones**. En cada iteración, se **evalúa y comprueba antes** de ejecutar el cuerpo del bucle, si es **Verdadera**, se ejecuta el cuerpo del bucle. Si es **Falsa**, termina el bucle.
- INCREMENTO: **evoluciona la variable de control**. En cada iteración, se ejecuta **justo después** de ejecutar el cuerpo del bucle.
- El **cuerpo del bucle** se ejecuta **para cada valor** que toma la **variable de control**.
- **En el cuerpo del bucle, no deben ser modificadas ni la variable de control del bucle, ni el valor de la expresión que determina el número de iteraciones.**

```
int main()
{
    for ( INICIALIZACION; EXPR_LOGICA; INCREMENTO ) {
        // Acciones (cuerpo del bucle)
    }
}
```



Sentencias de Iteración. Bucle For. Ejemplo 1

- Programa que muestra los N primeros números enteros mayores o iguales a cero, siendo N un número leído desde teclado, ej. para *N=6*: 0 1 2 3 4 5

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cout << i << " ";
    }
    cout << endl;
}
```

Sentencias de Iteración. Bucle For. Ejemplo 2 (v1)

- Programa que muestra los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para *N=6*: 5 4 3 2 1 0

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cout << (n - i - 1) << " ";
    }
    cout << endl;
}
```

n	*i*	*v*
6	*0*	*5*
6	*1*	*4*
6	*2*	*3*
6	*3*	*2*
6	*4*	*1*
6	*5*	*0*

v == n-i-1

Sentencias de Iteración. Bucle For. Ejemplo 2 (v2)

- Programa que muestra los N primeros números enteros mayores o iguales a cero (en orden **inverso**), siendo N un número leído desde teclado, ej. para *N=6*: 5 4 3 2 1 0

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    for (int i = n-1; i >= 0; --i) {
        cout << i << " ";
    }
    cout << endl;
}
```

Sentencias de Iteración. Bucle For. Ejemplo 3 (v1)

- Programa que muestra los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para *N=6*: 1 3 5 7 9 11

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

int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    for (int i = 0; i < n; ++i) {
        cout << ((i * 2) + 1) << " ";
    }
    cout << endl;
}
```

n	*i*	*v*
6	*0*	*1*
6	*1*	*3*
6	*2*	*5*
6	*3*	*7*
6	*4*	*9*
6	*5*	*11*

*v == i*2+1*

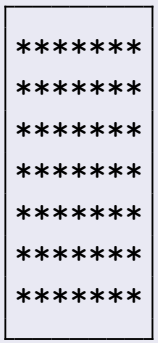
Sentencias de Iteración. Bucle For. Ejemplo 3 (v2)

- Programa que muestra los N primeros números enteros impares mayores que cero, siendo N un número leído desde teclado, ej. para *N=6*: 1 3 5 7 9 11

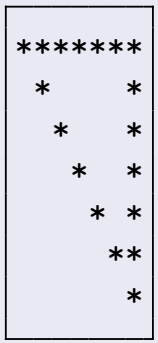
```
#include <iostream>
using namespace std;

int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    for (int i = 1; i < 2 * n; i += 2) {
        cout << i << " ";
    }
    cout << endl;
}
```

Sentencias de Iteración. Anidamientos. Ejemplos

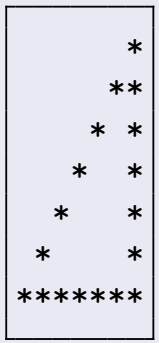
- Desarrolle varios programas donde cada uno lee de teclado un número y muestra una figura como en los siguientes ejemplos para el número **7**.



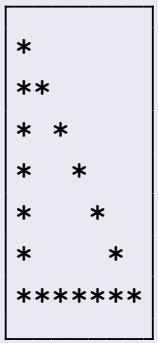
prg.1



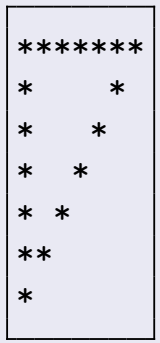
prg.2



prg.3



prg.4



prg.5

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Anidamientos. Ejemplo 1

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número **5**.

```
#include <iostream>
using namespace std;
const char SIMBOLO = '*';
int main()
{
    int n;
    cout << "Introduzca un número: ";
    cin >> n;
    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            cout << SIMBOLO;
        }
        cout << endl;
    }
}
```

```
*****↲
*****↲
*****↲
*****↲
*****↲
```

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Anidamientos. Ejemplo 2

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número **5**.

```
#include <iostream>
using namespace std;
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    cout << "Introduzca un número: ";
    cin >> n;
    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            if ((f == 0)||(c == n-1)||(c == f)) {
                cout << SIMBOLO;
            } else {
                cout << ESPACIO;
            }
        }
        cout << endl;
    }
}
```

```
*****↲
*   *↲
* * *↲
**  *↲
*   *↲
```

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Anidamientos. Ejemplo 3

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número **5**.

```
#include <iostream>
using namespace std;
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    cout << "Introduzca un número: ";
    cin >> n;
    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n; ++c) {
            if ((f == n-1)||(c == n-1)||(c == n-f-1)) {
                cout << SIMBOLO;
            } else {
                cout << ESPACIO;
            }
        }
        cout << endl;
    }
}
```

```
    *↲
   **↲
  * *↲
 *  *↲
*****↲
```

n	f	c
5	0	4
5	1	3
5	2	2
5	3	1
5	4	0

$c == n-f-1$

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Anidamientos. Ejemplo 4

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número **5**.

```
#include <iostream>
using namespace std;
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    cout << "Introduzca un número: ";
    cin >> n;
    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < f+1; ++c) {
            if ((f == n-1)||(c == 0)||(c == f)) {
                cout << SIMBOLO;
            } else {
                cout << ESPACIO;
            }
        }
        cout << endl;
    }
}
```

```
*↲
**↲
* *↲
*  *↲
*****↲
```

n	*f*	*columnas*
5	0	1
5	1	2
5	2	3
5	3	4
5	4	5

columnas == f+1

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Anidamientos. Ejemplo 5

- Desarrolle un programa que lea de teclado un número y muestre una figura como en el siguiente ejemplo para el número **5**.

```
#include <iostream>
using namespace std;
const char SIMBOLO = '*';
const char ESPACIO = ' ';
int main()
{
    int n;
    cout << "Introduzca un número: ";
    cin >> n;
    for (int f = 0; f < n; ++f) {
        for (int c = 0; c < n-f; ++c) {
            if ((f == 0)||(c == 0)||(c == n-f-1)) {
                cout << SIMBOLO;
            } else {
                cout << ESPACIO;
            }
        }
        cout << endl;
    }
}
```

```
*****↲
*   *↲
* *↲
**↲
*↲
```

n	f	$columnas$
5	0	5
5	1	4
5	2	3
5	3	2
5	4	1

$columnas == n-f$

- Nótese que después del último asterisco de cada fila debe haber un salto de línea (`endl`).

Sentencias de Iteración. Bucles. Ejercicios

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

Sentencia de Iteración For con Final Anticipado

- Se utiliza cuando el **número máximo de iteraciones** que se pueden realizar está **predeterminado** antes de su ejecución, pero es posible **finalizar anticipadamente** las iteraciones, dependiendo de alguna **condición extra**.
- La expresión lógica que controla la iteración tiene dos partes, **conectadas mediante el operador lógico** `&&` (**NO** se deben conectar con `||`):
  - La **primera parte** está destinada a controlar el **número máximo de iteraciones** que se pueden realizar, utilizando para ello la **variable de control** del bucle.
  - La **segunda parte** añade la **condición** necesaria para **finalizar anticipadamente** el proceso iterativo.
- El **cuerpo del bucle** sí podrá modificar los valores que afectan a la condición de finalización anticipada.
- **En el cuerpo del bucle, no deben ser modificadas ni la variable de control del bucle, ni el valor de la expresión que determina el número máximo de iteraciones.**

```
int main()
{
    for (int i = 0; (i < MAX_ITER) && ( cond_final_anticipado ); ++i) {
        // Acciones (cuerpo del bucle)
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v1)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    int cnt = 0; // cuenta de divisores, inicialmente a cero
    for (int div = 1; div <= num; ++div) {
        if ((num % div) == 0) {
            ++cnt; // si tiene algún divisor, incrementar la cuenta de divisores
        }
    }
    if (cnt == 2) {
        // si sólo tiene dos divisores entre 1 y NUM, entonces es primo
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v2)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    int cnt = 0;
    for (int div = 2; div <= num/2; ++div) {
        if ((num % div) == 0) {
            ++cnt;
        }
    }
    if ((num > 1)&&(cnt == 0)) {
        // si (NUM>1) y no tiene ningún divisor entre 2 y NUM/2, entonces es primo
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v3)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    int cnt = 0;
    for (int div = 2; (div <= num/2) && (cnt == 0); ++div) { // final anticipado
        if ((num % div) == 0) {
            ++cnt;
        }
    }
    if ((num > 1)&&(cnt == 0)) {
        // si (NUM>1) y no tiene ningún divisor entre 2 y NUM/2, entonces es primo
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v4)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    bool es_primo = (num > 1); // para ser primo, debe ser mayor que 1
    for (int div = 2; (div <= num/2) && es_primo ; ++div) { // final anticipado
        if ((num % div) == 0) {
            es_primo = false; // si tiene algún divisor, entonces no es primo
        }
    }
    if (es_primo) {
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

Sentencias de Iteración. Bucles. Ejercicios

- El *máximo común divisor* de dos números enteros es el mayor número entero que es divisor de ambos números.
- Se puede calcular el *máximo común divisor* de dos números enteros positivos mediante el *algoritmo de Euclides*:
  - Dados dos números enteros positivos, si son distintos, entonces se realiza un proceso iterativo hasta que los dos números sean iguales, en el cual, en cada iteración, al número mayor se le restará el número menor.
  - Al finalizar el proceso iterativo, el *máximo común divisor* será igual al valor de ambos números *(que ahora son iguales)*.
- Calcular el **máximo común divisor**, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.

	M	N	
	24	18	
24-18 ⇒	6	18	
	6	12	⇐ 18-6
	6	6	⇐ 12-6

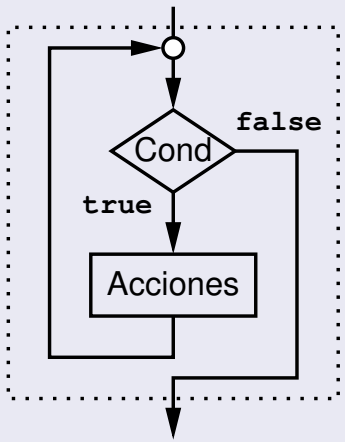
Sentencias de Iteración While y Do-While

- Las sentencias **WHILE** y **DO-WHILE** se utilizan cuando el número de iteraciones que se van a realizar **no está predeterminado** antes de su ejecución, sino que depende de la propia ejecución del proceso iterativo.
  - Una condición (expresión lógica) controla la iteración, si es **verdadera**, entonces continúa el proceso iterativo.
  - En el cuerpo del bucle habrá sentencias que dirijan la **evolución** de la iteración afectando al valor de la condición de control.

Sentencia de Iteración While

- La condición de control se evalúa **antes** de ejecutar el cuerpo del bucle.
  - Si la condición se evalúa a **Verdadera**, entonces **se ejecuta** el cuerpo y se **repite** el proceso.
  - Si la condición se evalúa a **Falsa**, entonces **NO se ejecuta** el cuerpo y **finaliza** la ejecución del bucle.

```
int main()
{
    while ( EXPR_LOGICA ) {
        // Acciones (cuerpo del bucle)
    }
}
```



Sentencias de Iteración. Ejemplo 1 (v1)

- Calcular el *máximo común divisor*, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.
  Para ello, se le debe restar **sucesivamente** al número **mayor** el número **menor**, hasta que los dos números sean iguales. El valor final es el **MCD**.

```
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    cout << "Introduzca dos números positivos: ";
    cin >> num1 >> num2;
    if ((num1 <= 0) || (num2 <= 0)) {
        cout << "Error. Números erróneos" << endl;
    } else {
        while (num1 != num2) {
            if (num1 > num2) {
                num1 = num1 - num2;
            } else {
                num2 = num2 - num1;
            }
        }
        cout << "El MCD es: " << num1 << endl;
    }
}
```

Sentencias de Iteración. Ejemplo 1 (v2)

- Calcular el *máximo común divisor*, según el método de Euclides, de dos números enteros (> 0) leídos de teclado.
  Para ello, se le debe restar **sucesivamente** al número **mayor** el número **menor**, hasta que los dos números sean iguales. El valor final es el **MCD**.

```
#include <iostream>
using namespace std;
int main()
{
    int num1, num2;
    cout << "Introduzca dos números positivos: ";
    cin >> num1 >> num2;
    while ((num1 <= 0) || (num2 <= 0)) {
        cout << "Error. Introduzca dos números positivos: ";
        cin >> num1 >> num2;
    }
    while (num1 != num2) {
        if (num1 > num2) {
            num1 = num1 - num2;
        } else {
            num2 = num2 - num1;
        }
    }
    cout << "El MCD es: " << num1 << endl;
}
```

Sentencias de Iteración. Ejemplo 2 (v1)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    bool es_primo = (num > 1);
    int div = 2;
    while ((div <= num/2) && es_primo) {
        if ((num % div) == 0) {
            es_primo = false;
        }
        ++div;
    }
    if (es_primo) {
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

Sentencias de Iteración. Ejemplo 2 (v2)

- Un número entero es un **número primo** si es **mayor que 1** y sólo tiene **dos divisores**, el propio número y el número 1. Desarrolle un programa que muestre si un número entero leído de teclado es primo.

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce un número: ";
    cin >> num;
    int div = 2;
    while ((div <= num/2)&&((num % div) != 0)) { // si encuentro un divisor
        ++div;                                   // (NUM%DIV==0), entonces
    }                                            // terminar el bucle
    if (div == num/2+1) {
        // si (DIV == NUM/2+1), entonces no se ha encontrado ningún
        // divisor entre 2 y NUM/2, por lo tanto es primo
        cout << "El número " << num << " es primo" << endl;
    } else {
        cout << "El número " << num << " no es primo" << endl;
    }
}
```

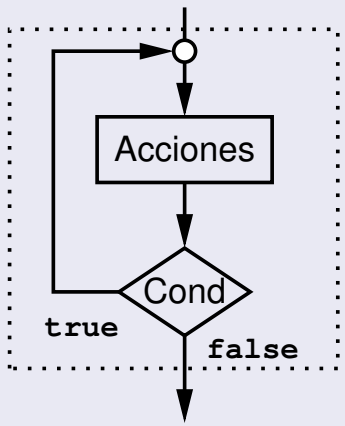
Sentencias de Iteración. Bucles. Ejercicios

- Programa que lee dos números enteros (numerador y denominador). En la lectura del denominador, si el número leído es igual a cero, entonces se repetirá el proceso de lectura del denominador hasta que el número introducido sea distinto de cero. Finalmente se mostrará el resultado de dividir el numerador entre el denominador.

Sentencia de Iteración Do-While

- La condición de control se evalúa **después** de ejecutar el cuerpo del bucle.
  - Primero **se ejecuta** el cuerpo, y posteriormente, si la condición se evalúa a **Verdadera**, entonces se **repite** el proceso.
  - Si la condición se evalúa a **Falsa**, entonces **finaliza** la ejecución del bucle.

```
int main()
{
    do {
        // Acciones (cuerpo del bucle)
    } while ( EXPR_LOGICA ) ;  // Nótese el punto-y-coma
}
```



Sentencias de Iteración. Ejemplo 1

- Programa que lee dos números enteros (numerador y denominador). En la lectura del denominador, si el número leído es igual a cero, entonces se repetirá el proceso de lectura del denominador hasta que el número introducido sea distinto de cero. Finalmente se mostrará el resultado de dividir el numerador entre el denominador.

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

int main()
{
    int numerador, denominador;
    cout << "Introduce el numerador: ";
    cin >> numerador;
    do {
        cout << "Introduce el denominador: ";
        cin >> denominador;
    } while (denominador == 0);
    int resultado = numerador / denominador;
    cout << "Resultado: " << resultado << endl;
}
```

Sentencias de Iteración. Ejercicios

1. Multiplicar, mediante sumas, dos números enteros (≥ 0) leídos de teclado.

$$2 \times 5 \equiv 2 + 2 + 2 + 2 + 2$$

2. Potencia de la base elevada al exponente, dos números enteros (≥ 0) leídos de teclado (error si 0^0). En C++ **x^y no** es equivalente a *x elevado a y*.

$$2^5 \equiv 2 \times 2 \times 2 \times 2 \times 2$$

3. Calcular el factorial de un número (≥ 0) leído de teclado.

$$6! \equiv 1 \times 2 \times 3 \times 4 \times 5 \times 6$$

4. Dividir, mediante restas, dos números enteros (≥ 0) leídos de teclado.

$$\begin{array}{lll} 11 \div 4 \Longrightarrow & 11 - 4 \equiv 7 & \\ & 7 - 4 \equiv 3 & \Longrightarrow \textit{resto} \equiv 3; \textit{cociente} \equiv 2 \end{array}$$

Sentencias de Iteración. Ejercicio 1

- Multiplicar, mediante sumas, dos números enteros (≥ 0) leídos de teclado.

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

int main()
{
    int m, n;
    cout << "Introduce dos números: ";
    cin >> m >> n;
    if ((m < 0)||(n < 0)) {
        cout << "Error, números negativos" << endl;
    } else {
        // Sumar: m + m + m + ··· + m   (n veces)
        int total = 0; // 1 es el elemento neutro de la suma
        for (int i = 0; i < n; ++i) {
            // Proceso iterativo: Añadir el valor de 'm' al total
            total = total + m;      // total += m;
        }
        cout << "Resultado: " << total << endl;
    }
}
```

Sentencias de Iteración. Ejercicio 2

- Potencia de la base elevada al exponente, dos números enteros (≥ 0) leídos de teclado (error si 0^0). En C++ **x^y no** es equivalente a *x elevado a y*.

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

int main()
{
    int base, exp;
    cout << "Introduce base y exponente: ";
    cin >> base >> exp;
    if ((base < 0)||(exp < 0)||(base == 0 && exp == 0)) {
        cout << "Error, números no válidos" << endl;
    } else {
        // Multiplicar: base * base * base * ··· * base   (exp veces)
        int total = 1; // 1 es el elemento neutro de la multiplicación
        for (int i = 0; i < exp; ++i) {
            // Proceso iterativo: Multiplicar el valor de 'base' al total
            total = total * base;       // total *= base;
        }
        cout << "Resultado: " << total << endl;
    }
}
```

Sentencias de Iteración. Ejercicio 3

- Calcular el factorial de un número (≥ 0) leído de teclado.

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

int main()
{
    int n;
    cout << "Introduce un número: ";
    cin >> n;
    if (n < 0) {
        cout << "Error, número negativo" << endl;
    } else {
        // Multiplicar: 1 2 3 4 5 6 7 ··· n
        long total = 1;  // el tipo LONG permite almacenar números mayores
        for (int i = 2; i <= n; ++i) {
            // Proceso iterativo: multiplicar numero y acumular
            total = total * i;      // total *= i;
        }
        cout << "Factorial: " << total << endl;
    }
}
```

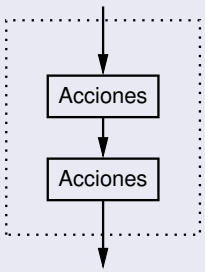
Sentencias de Iteración. Ejercicio 4

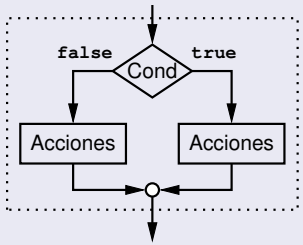
- Dividir, mediante restas, dos números enteros (≥ 0) leídos de teclado.

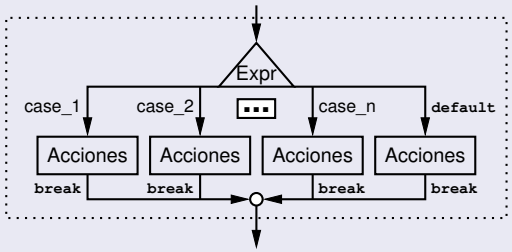
```
#include <iostream>
using namespace std;

int main()
{
    int dividendo, divisor;
    cout << "Introduce el dividendo y el divisor: ";
    cin >> dividendo >> divisor;
    if ((dividendo < 0)||(divisor <= 0)) {
        cout << "Error, números erróneos" << endl;
    } else {
        int resto = dividendo;
        int cociente = 0;
        while (resto >= divisor) {
            resto = resto - divisor;    // resto -= divisor;
            ++cociente;
        }
        cout << "Cociente: " << cociente << endl;
        cout << "Resto: " << resto << endl;
    }
}
```

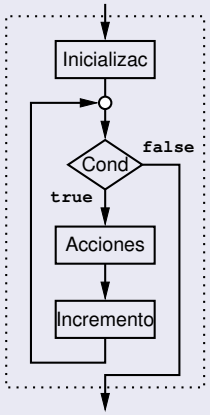
Programación Estructurada

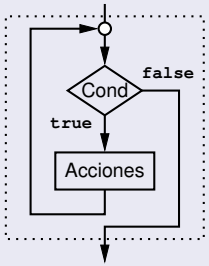
- Un programa sigue una metodología de **programación estructurada** si **todas** las estructuras de control que se utilizan (secuencia, selección e iteración) tienen un **único punto de entrada** y un **único punto de salida**.

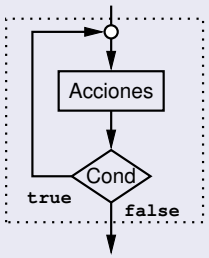












- Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras. Es la base de la **programación estructurada**. Un algoritmo que use tan sólo las estructuras de control tratadas en este tema, se denomina estructurado.
- **Sólo están permitidas las sentencias y estructuras de selección e iteración que aparecen en los apuntes y que se han visto en clase (`if`, `switch`, `while`, `do-while`, `for`, etc.). No se podrán usar otras como `goto`, `continue`, etc. La sentencia `break` sólo se podrá utilizar en la estructura `switch`.**

Control de Errores. Excepciones

- Durante la ejecución de un programa, se pueden dar **situaciones inesperadas o errores inesperados** para los cuales el programa no tiene un comportamiento adecuado.
  - **Abortarán** la ejecución del programa. Ej. división por cero.
  - Provocarán un **malfuncionamiento** del programa Ej. lectura de un número negativo cuando se desea leer número positivo.
- El programador puede utilizar las estructuras de selección y de iteración para controlar algunos de estos errores.

```
#include <iostream>
using namespace std;
int main()
{
    int numerador, denominador, resultado;
    cout << "Introduce el numerador y denominador: ";
    cin >> numerador >> denominador;
    if (denominador == 0) {
        cout << "Error: división por cero" << endl;
    } else {
        resultado = numerador / denominador;
        cout << "Resultado: " << resultado << endl;
    }
}
```

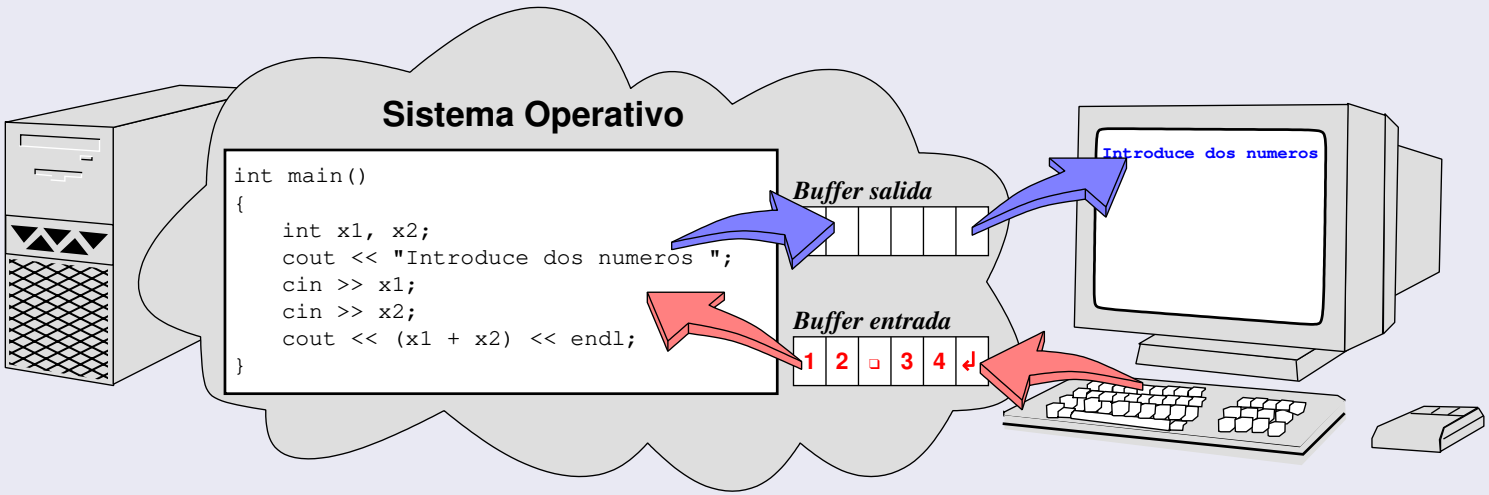
Lanzamiento de Excepciones

- Si no es posible recuperar la situación inesperada o el error inesperado, entonces podemos **lanzar (`throw`) una excepción que informe de la situación de error excepcional**.
  - Normalmente, si se lanza una excepción, entonces la ejecución del programa **aborta** en ese punto.
  - Hay situaciones avanzadas (que no veremos en este curso) en las cuales se puede capturar la excepción y recuperar la situación de error.

```
#include <iostream>
using namespace std;
int main()
{
    int numerador, denominador, resultado;
    cout << "Introduce el numerador y denominador: ";
    cin >> numerador >> denominador;
    if (denominador == 0) {
        throw  "Error: división por cero";
    }
    // Nótese que al lanzar la excepción, el programa aborta la ejecución (no continúa)
    resultado = numerador / denominador;
    cout << "Resultado: " << resultado << endl;
}
```

El Buffer de Entrada

- La entrada y salida de datos se realiza indirectamente a través de **buffers** controlados por el *Sistema Operativo* y son independientes de nuestro programa.
- Cuando se pulsa alguna tecla, los caracteres correspondientes a las teclas pulsadas se almacenan en una zona de memoria intermedia: el **buffer de entrada**. Cuando un programa realiza una operación de entrada de datos, accede al buffer de entrada y obtiene los caracteres allí almacenados si los hubiera, o esperará hasta que los haya (cuando se pulsen una serie de teclas seguida por la tecla **ENTER**).
- Una vez obtenidos los caracteres asociados a las teclas pulsadas, se **convertirán** a un valor del tipo de la variable especificada por la operación de entrada, **asignándole** dicho valor a la misma.



Entrada de Datos Avanzada

- La entrada de datos (>>) se realiza **eliminado** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos (datos).
- Después se **leen** caracteres hasta encontrar algún carácter no adecuado para el tipo de datos de la lectura (para el tipo `char` sólo se lee un único carácter).
- Si el dato leído se puede convertir al tipo de la variable que lo almacenará, entonces la operación será correcta, los datos leídos se **convierten** al tipo de la variable y se **asignan** a la variable. En otro caso la operación será incorrecta, el flujo de entrada pasa a modo **erróneo**, y cualquier entrada siguiente **fallará**.

```
int main() {
    int x;  char letra1, letra2;
    cout << "Introduce número seguido por letra espacio letra: ";
    cin >> x >> letra1 >> letra2;
    cout << "Resultado:" << x << " [" << letra1 << "] [" << letra2 << "]" << endl;
}
```

Ejecución del Programa

```
Introduce número seguido ...: ▫▫▫123z▫x ↲
Resultado: 123 [z] [x]
```

3 espacios número 123 letra z espacio letra x ENTER

Entrada de Caracteres Avanzada

- La sentencia `cin.get(c)` **lee** el siguiente carácter (de tipo `char`) **sin eliminar** los espacios en blanco ni saltos de línea **iniciales**.

- La sentencia `cin >> ws` **elimina** los espacios en **blanco** y saltos de línea **iniciales** hasta encontrar algún carácter distinto de ellos.

```
int main() {
    char c1, c2, c3 ;
    cout << "Introduzca 3 caracteres: ";
    cin >> ws ;
    cin.get(c1) ;
    cin.get(c2) ;
    cin.get(c3) ;
    cout << "Resultado: " << c1 << c2 << c3 << endl;
}
```

Ejecución del Programa

```
Introduzca 3 caracteres: ▫▫▫a▫b ↵
Resultado: a▫b
```

Se introducen 3 espacios letra a espacio letra b ENTER
Se han leído los tres caracteres consecutivos

Ejemplos. Procesamiento Secuencial

1. Programa que lee de teclado una secuencia de números enteros terminada en el número cero, y muestra el resultado de la suma de todos los números de la secuencia (sin incluir al cero terminador), por ejemplo, para la secuencia de números *20 34 12 54 0* muestra el resultado **120**.

```
Introduce una secuencia de números terminada en cero:
20▫34▫12▫54▫0 ↵                    | números separados por espacios y ENTER
Resultado: 120
```

2. Programa que lee de teclado una secuencia de números enteros terminada en el número cero, y muestra aquellos números de la secuencia de entrada que son pares (sin incluir al cero terminador), por ejemplo, para la secuencia de números *11 12 13 14 15 16 17 18 19 0* muestra el resultado **12 14 16 18**.

```
Introduce una secuencia de números terminada en cero:
11▫12▫13▫14▫15▫16▫17▫18▫19▫0 ↵     | números separados por espacios y ENTER
Resultado: 12 14 16 18
```

3. Programa que lee de teclado una secuencia de números enteros terminada en el número cero, y muestra el mayor, menor y la media de todos los números de la secuencia (sin incluir al cero terminador), por ejemplo, para la secuencia de números *20 34 12 54 17 0* muestra el resultado **54 12 27.4**.

```
Introduce una secuencia de números terminada en cero:
20▫34▫12▫54▫17▫0 ↵                 | números separados por espacios y ENTER
Resultado: 54 12 27.4
```

Entrada de Datos Avanzada

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 1

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    int suma = 0;
    cout << "Introduce una secuencia de números terminada en cero: " << endl;
    cin >> num ;
    while (num != 0) {
        suma = suma + num;   // suma += num;
        cin >> num ;
    }
    cout << "Resultado: " << suma << endl;
}
```

Ejecución del Programa

```
Introduce una secuencia de números terminada en cero:
20▫34▫12▫54▫0 ↵
Resultado: 120
```

cinco números separados por espacios y ENTER
los caracteres tecleados se almacenan en el Buffer
los números serán leídos desde el Buffer

Entrada de Datos Avanzada

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 2

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Introduce una secuencia de números terminada en cero: " << endl;
    cin >> num ;
    cout << "Resultado: " ;
    while (num != 0) {
        if (num % 2 == 0) {
            cout << num << " ";
        }
        cin >> num ;
    }
    cout << endl;
}
```

Ejecución del Programa

```
Introduce una secuencia de números terminada en cero:
11 12 13 14 15 16 17 18 19 0 ↵
Resultado: 12 14 16 18
```

diez números separados por espacios y ENTER
los caracteres tecleados se almacenan en el Buffer
los números serán leídos desde el Buffer

Entrada de Datos Avanzada

El Buffer de Entrada. Procesamiento Secuencial. Ejemplo 3

```
#include <iostream>
using namespace std;
int main()
{
    int num;
    int cnt = 0;
    int suma = 0;
    cout << "Introduce una secuencia de números terminada en cero: " << endl;
    cin >> num ;
    int menor = num;
    int mayor = num;
    while (num != 0) {
        ++cnt;
        suma += num;
        if (num > mayor) {
            mayor = num;
        } else if (num < menor) {
            menor = num;
        }
        cin >> num ;
    }
    if (cnt == 0) {
        cout << "Secuencia vacía" << endl;
    } else {
        double media = double(suma) / double(cnt);
        cout << "Resultado: " << mayor << " " << menor << " " << media << endl;
    }
}
```

Entrada y Salida Formateada de Datos (I)

- Entrada y salida de *valores lógicos* (*true*, *false*).

```
#include <iostream>
using namespace std;
int main()
{
    bool x;
    cin >> boolalpha >> x;      // lee los valores lógicos como 'false' o 'true'
    cout << boolalpha << x;     // muestra los valores lógicos como 'false' o 'true'

}
```

- Representación de los *números enteros* (*decimal*, *hexadecimal*, *octal*).

```
#include <iostream>
using namespace std;
int main()
{
    cout << dec << 27;          // muestra 27 (decimal)
    cout << hex << 27;          // muestra 1b (hexadecimal)
    cout << oct << 27;          // muestra 33 (octal)
}
```

- Estos manipuladores *mantienen su estado* hasta que vuelva a ser cambiado.

Salida Formateada de Datos (II)

- Es posible especificar la *anchura de campo* y la *precisión*.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    // Anchura de campo
    cout << setw(5) << 234;                    // muestra □□234
    cout << left  << setw(5) << 234;           // muestra 234□□
    cout << right << setw(5) << 234;           // muestra □□234
    cout << setfill('#') << setw(5) << 234;    // muestra ##234

    // Precisión
    cout << 0.00006789 << 0.0006789 << 678900.0 << 6789000.0 ;
    //      6.789e-05     0.0006789    678900      6.789e+06    PRECISION(6)
    cout << setprecision(3);
    cout << 0.00006789 << 0.0006789 << 678900.0 << 6789000.0 ;
    //      6.79e-05      0.000679     6.79e+05    6.79e+06     PRECISION(3)
}
```

- `setw()` sólo afecta al siguiente campo de salida.
- Los demás manipuladores *mantienen su estado* hasta que vuelva a ser cambiado.
- `setprecision()`, `setw()` y `setfill()` requieren incluir la biblioteca estándar `<iomanip>`.
  - La precisión especifica el número de cifras significativas. La precisión por defecto es 6.
  - Si ($10^{-5} < |valor| < 10^{precision}$) entonces usa *punto-fijo*, en otro caso usa *punto-flotante*.

Salida Formateada de Datos (III)

- Representación de los *números reales* (*precisión*, *punto fijo*, *punto flotante*).

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setprecision(6) << defaultfloat ;
    cout << 0.00006789   << 0.0006789    << 678900.0      << 6789000.0 ;
    //      6.789e-05       0.0006789       678900           6.789e+06
    cout << setprecision(6) << fixed ;
    cout << 0.00006789   << 0.0006789    << 678900.0      << 6789000.0 ;
    //      0.000068        0.000679        678900.000000    6789000.000000
    cout << setprecision(6) << scientific ;
    cout << 0.00006789   << 0.0006789    << 678900.0      << 6789000.0 ;
    //      6.789000e-05    6.789000e-04    6.789000e+05     6.789000e+06
}
```

- Estos manipuladores *mantienen su estado* hasta que vuelva a ser cambiado.
- `setprecision()` requiere incluir la biblioteca estándar `<iomanip>`.
- El modo por defecto es `defaultfloat`. La precisión por defecto es 6.
- `defaultfloat`:
  - Si $(10^{-5} < |valor| < 10^{precision})$ entonces usa *punto-fijo*, en otro caso usa *punto-flotante*.
  - La precisión especifica el número de cifras significativas.
- `fixed`:
  - Representa los números en *punto-fijo*.
  - La precisión especifica el número de cifras decimales exactas.
- `scientific`:
  - Representa los números en *punto-flotante*.
  - La precisión especifica el número de cifras decimales exactas.

Tildes en el Sistema Operativo Windows

- Tanto el IDE *Code::Blocks*, como *VsCode*, en el sistema operativo *Windows*, suelen dar problemas en el tratamiento y manipulación de caracteres con tildes. Para resolver este problema, se puede incluir la biblioteca `<windows.h>` e invocar a los procedimientos que se indican en el siguiente ejemplo:

```
#include <iostream>
#include <windows.h>
/* resto del programa */
int main()
{
    SetConsoleOutputCP(1252);
    SetConsoleCP(1252);
    /* resto del programa */
}
```

- Además de lo anterior, en el IDE *VsCode* será necesario activar la siguiente opción: **Archivo->Preferencias->Configuración**. Buscar la opción **files.encoding**, y seleccionar la codificación **Windows 1252**.

Tipos de Datos Simples: Tipos Enumerados

- Los **tipos simples predefinidos son suficientes** para resolver problemas que sólo requieran tipos de datos simples.
- Sin embargo, también es posible definir otros tipos simples para especificar la interpretación de un valor, o para especificar los valores posibles.

- El programador puede definir un **tipo enumerado** como una secuencia de símbolos separados por comas. Estos símbolos son los *"valores"* asociados al tipo definido.

```
enum Colores { rojo, verde, azul } ;
enum Palos { oros, copas, espadas, bastos } ;
```

- Se pueden **declarar** variables y constantes de un tipo enumerado definido. Una variable del tipo enumerado puede **tomar el valor** de cualquier símbolo de la enumeración.

```
Colores mi_color = verde ;
```

- El tipo enumerado es un **tipo ordinal**, se le pueden aplicar los operadores relacionales.
- Es posible la **conversión** al tipo natural o entero. A cada elemento de la enumeración le corresponde un número según la posición que ocupa en la secuencia:

`0 ≡ int(rojo)` `2 ≡ int(espadas)` `verde ≡ Colores(1)`

- **No** es posible realizar ni **entrada** ni **salida** directa de valores enumerados.

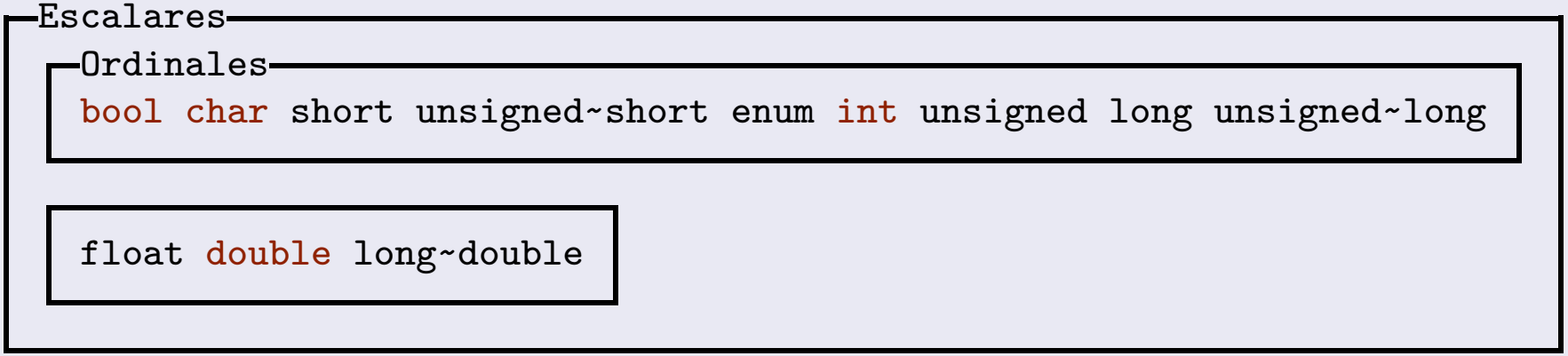
Tipos de Datos Simples

Propiedades de los Tipos Escalares

- Formados por elementos **indivisibles** (simples).
- Formados por elementos **ordenados**, es decir, les son aplicables los operadores relacionales (== != < > <= >=).

Propiedades de los Tipos Ordinales

- Los tipos ordinales son tambien tipos escalares, por lo que, además de las propiedades anteriores, también tienen las siguientes propiedades:
  - Cada valor tiene un **predecesor** y un **sucesor** únicos (excepto el primero y el último, respectivamente).



Tipos de Datos Simples

Tipo Lógico: `bool`

- Representa los valores *falso* y *verdadero*, resultados de expresiones relacionales y lógicas.
- Los valores se representan en el lenguaje de programación con los símbolos **`false`** y **`true`** para representar los valores *falso* y *verdadero* respectivamente.
- La codificación interna es un número binario (de 8 bits) donde el número cero representa **`false`** y el número uno representa **`true`**.
- Operadores aplicables:
  - Operadores relacionales: `==` `!=` `<` `>` `<=` `>=`
  - Operadores lógicos: `!` `&&` `||`
  - Operadores E/S: `>>` `<<`
  - Conversión: es posible convertirlos al tipo entero (`int`), `false` corresponde con el valor `0` y `true` corresponde con el valor `1`, y a la inversa.
- Tabla de verdad de los operadores lógicos:

x	y	x && y	x || y	! x
F	F	F	F	V
F	V	F	V	V
V	F	F	V	F
V	V	V	V	F

Tipos de Datos Simples

Tipo Carácter: `char`

- Representa los símbolos (denominados **caracteres**) utilizados para la entrada y salida de datos. Por ejemplo de teclado y a pantalla.
- Los valores se representan en el lenguaje de programación como el símbolo entre **comillas simples**: `'a'`, `'F'`, `'5'`, `'.'`, `'+'`, etc.
- Cada símbolo se codifica internamente como un número binario (de 8 bits) que se corresponde con la posición del símbolo en la **tabla ASCII**.
- Operadores aplicables:
  - Operadores relacionales: `==` `!=` `<` `>` `<=` `>=`
  - Incremento y decremento: `++` `--`
  - Operadores E/S: `>>` `<<`
  - Conversión: es posible convertirlos al tipo entero (`int`), obteniendo su posicion en la tabla de conversión (ASCII), y a la inversa.
- Véase la tabla ASCII.
  - Los dígitos del `'0'` al `'9'` están consecutivos en la tabla ASCII.
  - Las letras de la `'A'` a la `'Z'` están consecutivas en la tabla ASCII.
  - Las letras de la `'a'` a la `'z'` están consecutivas en la tabla ASCII.
  - La tabla ASCII no representa la letra Ñ/ñ ni las vocales con tildes.

Tipos de Datos Simples

- Tabla ASCII para la codificación de caracteres básicos (tipo `char`)

Cod	Car	Cod	Car	Cod	Car	Cod	Car	Cod	Car	Cod	Car
32	SP	48	0	64	@	80	P	96	‘	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

Tipos de Datos Simples

Tipo Entero: `short`, `int`, `long`, `long long`

- Representa el concepto matemático de **número entero**, limitado a un intervalo de valores máximo: $\{ -2^{n-1} \cdots 2^{n-1}-1 \}$.
- Los valores se representan en el lenguaje de programación como una secuencia de dígitos, opcionalmente precedidos por el símbolo negativo: −1245
- La codificación interna es un número binario en **complemento-a-dos**, con un número de bits limitado (`short`: 16 bits, `int`: 32 bits, `long`: 32/64 bits, `long long`: 64 bits).
- Operadores aplicables:
  - Operadores relacionales: == != < > <= >=
  - Operadores aritméticos: + − ∗ % / (**división entera**)
  - Incremento y decremento: ++ −−
  - Operadores E/S: >> <<
  - Conversión: es posible convertirlos a otros tipos escalares, si y solo si están dentro del intervalo correcto, y a la inversa.

Tipos de Datos Simples

Tipo Natural: `unsigned short`, `unsigned`, `unsigned long`

- Representa el concepto matemático de **número natural**, incluido el cero, limitado a un intervalo de valores máximo: $\{ 0 \cdots 2^n\text{-}1 \}$.
- Los valores se representan en el lenguaje de programación como una secuencia de dígitos: 1245
- La codificación interna es un número **binario puro**, con un número de bits limitado (`unsigned short`: 16 bits, `unsigned`: 32 bits, `unsigned long`: 32/64 bits).
- Operadores aplicables:
  - Operadores relacionales: == != < > <= >=
  - Operadores aritméticos: + - * % / (**división entera**) (**aritmética modular**)
  - Incremento y decremento: ++ --
  - Operadores E/S: >> <<
  - Conversión: es posible convertirlos a otros tipos escalares, si y solo si están dentro del intervalo correcto, y a la inversa.
- Si se van a realizar **operaciones aritméticas** con variables de tipo `unsigned`, y el resultado pudiera ser **negativo**, entonces se deben **convertir** los valores.

```
int    resultado1 = int(menor) - int(mayor);
double resultado2 = double(menor) - double(mayor);
```

Tipos de Datos Simples

Tipo Real: `float`, `double`, `long double`

- Representan el concepto matemático de **números reales**, con representación **inexacta**, precisión limitada e intervalo de valores máximo.
- Los valores se representan en el lenguaje de programación como una secuencia de dígitos, opcionalmente precedidos por el símbolo negativo, seguidos por una parte decimal y un factor de escala opcionales:
  - -1245.678e2 ($\equiv$ -1245.678$\times 10^{2}$), -1245.678e-3 ($\equiv$ -1245.678$\times 10^{-3}$)
- La codificación interna es un número binario en **coma-flotante** (mantisa y exponente), con un número de bits limitado (`float`: 32 bits, `double`: 64 bits, `long double`: 80/128 bits).
- Operadores aplicables:
  - Operadores relacionales: == != < > <= >= (**comparaciones inexactas**)
  - Operadores aritméticos: + - * / (**división real**) (**cálculos inexactos**)
  - Incremento y decremento: ++ --
  - Operadores E/S: >> <<
  - Conversión: es posible convertirlos a otros tipos escalares, si y solo si están dentro del intervalo correcto, y a la inversa. La parte decimal se pierde.

Problemas Derivados de la Implementación de los Números

Desbordamiento

- El número de bits utilizados para representar los números *naturales*, *enteros* y *reales* es limitado.
- El resultado de aplicar algún operador a dos valores de un determinado tipo puede dar lugar a un valor no representable (fuera de límites) dando lugar a lo que se conoce como **desbordamiento** (**overflow**).
- Por ejemplo, la siguiente operación matemática produce un resultado erróneo.

```
#include <iostream>
using namespace std;
int main()
{
    unsigned resultado = 7 + 4294967290;
    cout << resultado << endl; // Muestra 1 debería mostrar 4294967297
}
```

- Ejemplo, si la representación de números naturales fuese de 4 bits:

```
    7        0111
+ 10     + 1010
------   -------
   17    1 0001
```

  - El resultado final en realidad es 1 (y no 17)

Problemas Derivados de la Implementación de los Números

Representación aproximada de números reales y pérdida de precisión

- El número de bits utilizado para representar la parte fraccionaria de los números reales es limitado.
  - Implica una **representación aproximada** y una **pérdida de precisión**.
- Esta pérdida de precisión afecta a los cálculos realizados con números reales.
- Debido a la pérdida de precisión en las operaciones con números reales, nunca debemos realizar comparaciones de **igualdad** (==/!=) con el resultado de éstas.
- Por ejemplo, la siguiente operación matemática produce un resultado erróneo.

```
#include <iostream>
using namespace std;
int main()
{
    bool ok = ((3.0 * 0.1) / 3.0) == (3.0 * (0.1 / 3.0));
    cout << boolalpha << ok << endl; // Muestra false
}                                    // Matemáticamente debería ser true
```

- Ejemplo, si la precisión fuese de 4 dígitos:
  - 3.0000 × 0.1000 = 0.3000 -> 0.3000 / 3.0000 = 0.1000
  - 0.1000 / 3.0000 = 0.0333 -> 3.0000 × 0.0333 = 0.0999

El tipo unsigned

- La representación de los números de tipo `unsigned` se realiza en **binario puro**, y la representación de los números de tipo `int` se realiza en **complemento-a-dos**:
- Cuando se realizan operaciones aritméticas o comparaciones entre una variable de tipo **int** y otra de tipo **unsigned**, el valor de la variable de tipo **int** se convierte a **unsigned**, que en caso de ser un valor negativo, no puede ser representado adecuadamente, por lo que puede producir resultados erróneos.
- Por ello, el tipo **unsigned** sólo se debe utilizar en aquellas situaciones donde se desee realizar aritmética modular o manipulaciones de bits.

- Ejemplo de representación de números de 3 bits:

Binario	unsigned	int
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

- Las siguientes operaciones producen resultados erróneos.

```
#include <iostream>
using namespace std;
int main()
{
  int num21 = -4;
  unsigned num22 = 2;
  double num23 = num21 + num22;        // ERROR: 4.29497e+09 vs. -2
  cout << num23 << endl;               // ERROR: 4.29497e+09 vs. -2
  int num24 = 1.0 * (num21 + num22); // ERROR: 2147483647 vs. -2
  cout << num24 << endl;               // ERROR: 2147483647 vs. -2
  if (num21 > num22) {                 // ERROR: ¿ (-4 > 2) ? vs. false
      cout << "ERROR" << endl;         // ERROR: (-4 < 2) vs. false
  }
}
```

Problemas Derivados de la Implementación de los Números

El tipo `float`

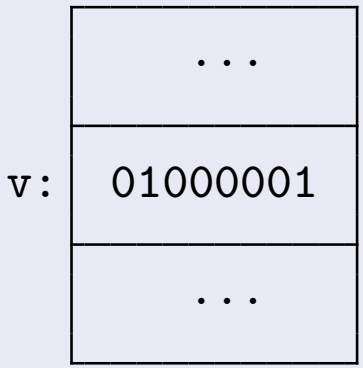
- El tipo **float** no puede representar a todos los valores del tipo **int**.
  - El tipo **float** sólo utiliza 32 bits para representar los números en formato de punto flotante, de los cuales, se utilizan 8 bits para el exponente, y 24 bits para la mantisa.
  - El tipo **int** utiliza también 32 bits en formato **complemento-a-dos** para representar los números enteros.
  - Por lo tanto, los 24 bits de la mantisa del tipo **float** no son suficientes para representar a todos los numeros enteros de tipo **int**, ya que son necesarios 32 bits, por lo que se produce una pérdida de precisión en la conversión entre numeros de tipo **int** y números de tipo **float**. El número 16777215 es el mayor número **int** que puede representar el tipo **float** sin pérdida de precisión.
- En el siguiente ejemplo, el resultado de la multiplicación de los dos números **int** (9527×15937) es 151831799, que cuando se convierte a **float**, se produce una pérdida de precisión (pasa a tomar el valor 151831792).

```
#include <iostream>
using namespace std;
int main()
{
    int num21 = 9527;              // 9527 * 15937 = 151831799
    int num22 = 15937;
    float res2a = num21 * num22;          // ERROR: 151831792
    cout << fixed << res2a << endl;       // ERROR: 151831792 vs. 151831799
    float num23 = 1.0;
    int res2i = num21 * num22 * num23;    // ERROR: 151831792
    cout << res2i << endl;                // ERROR: 151831792 vs. 151831799
    double res2d = num21 * num22 * num23; // ERROR: 151831792
    cout << fixed << res2d << endl;       // ERROR: 151831792 vs. 151831799
}
```

Tipos de Datos

Interpretación del Valor Binario Almacenado en Memoria

- Internamente toda la información se representa en **binario** como una secuencia de bits (de un tamaño especificado).
- El **tipo**, asociado a esa representación, permite interpretarlo y asociarlo con la información abstracta que representa.
- ¿ Que valor contiene la variable v ?



Si v es de tipo entero, entonces v almacena el número 65

Si v es de tipo carácter, entonces v almacena la letra 'A' ya que el código binario 01000001 (65) es el valor de la letra 'A' en la tabla ASCII para el tipo carácter

Errores de Codificación Frecuentes

1. Usar el operador de **asignación** = en lugar del operador de **igualdad** ==, cuando se desea realizar una comparación de igualdad.
2. Usar el operador de **asignación** con **más** =+ en lugar del operador de **incremento** +=, cuando se desea incrementar el valor de la variable.
3. Usar el operador de **asignación** con **menos** =- en lugar del operador de **decremento** -=, cuando se desea decrementar el valor de la variable.
4. Usar el símbolo `&` en lugar de `&&`. Usar el símbolo `|` en lugar de `||`.
5. Usar el símbolo ^ para calcular la potencia.
6. Definir una operación relacional erróneamente, por ejemplo `(0 <= x <= 9)` en vez de `((0 <= x) && (x <= 9))`.
7. Errores al crear un expresión lógica sin tener en cuenta que se **evalúa en cortocircuito**. Por ejemplo, codificar `(((num1 % num2) == 0) && (num2 != 0))` en lugar de `((num2 != 0) && ((num1 % num2) == 0))`
8. Es fácil equivocarse cuando se mezclan los operadores lógicos (`! && ||`) en la misma expresión lógica compleja.
9. Olvidar la sentencia `break` al final de las sentencias `case` del `switch`.
10. Es un error poner `;` en la sentencia `if`, `else`, `for` o `while`:
    `if (...) ; {...} else ; {...}` en lugar de `if (...) {...} else {...}`
    `for (...) ; {...}` en lugar de `for (...) {...}`
    `while (...) ; {...}` en lugar de `while (...) {...}`