UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y CC. Computación
E.T.S.I. Telecomunicación

CONCEPTOS BÁSICOS DE PROGRAMACIÓN

Tema 2

Programación I

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

- RESOLUCIÓN DE PROBLEMAS

  - Programación:
    Establecer una **secuencia de acciones** que trás ser ejecutadas por un **procesador** resuelva un determinado **problema**.

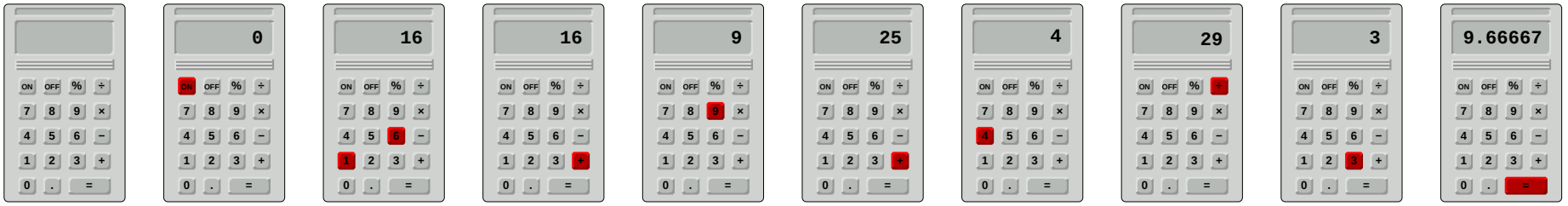
  - Fases:

    1. Análisis del problema
    2. Estudio de su solución
    3. Diseño del **Algoritmo**
    4. Codificación del programa
    5. Depuración y prueba

- EJEMPLOS INTUITIVOS DE ALGORITMOS

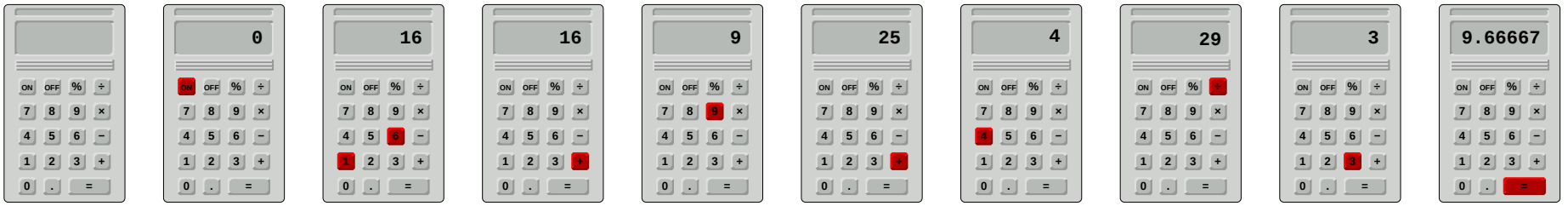
  - PROBLEMA: calcular la media aritmética de tres números cualesquiera utilizando una calculadora básica.

    - Fases 1 y 2: Análisis y Estudio de la solución (ejemplo 16, 9, 4)

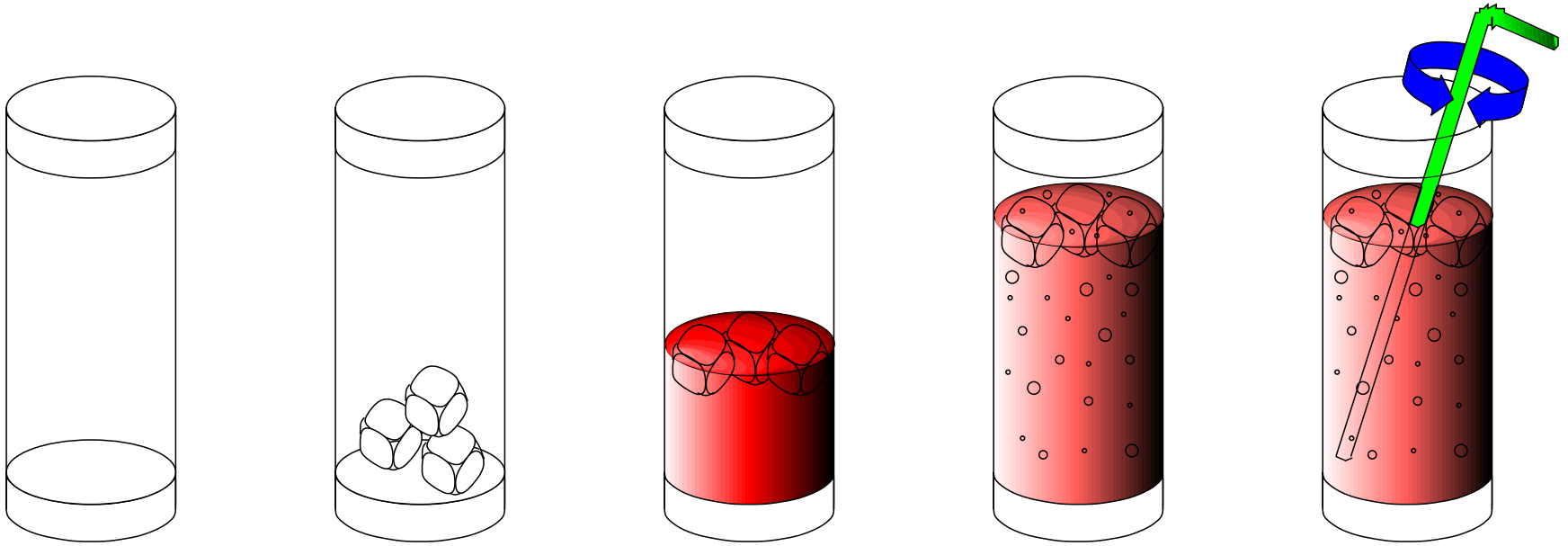


- Fase 3: Diseño del **Algoritmo**

1. Pulsar la tecla .ºN"
2. Teclear el primer número
3. Pulsar la tecla -"
4. Teclear el segundo número
5. Pulsar la tecla -"
6. Teclear el tercer número
7. Pulsar la tecla "÷"
8. Pulsar la tecla "3"
9. Pulsar la tecla -"
10. La media de los tres números aparece en la pantalla
11. Pulsar la tecla .ºFF"

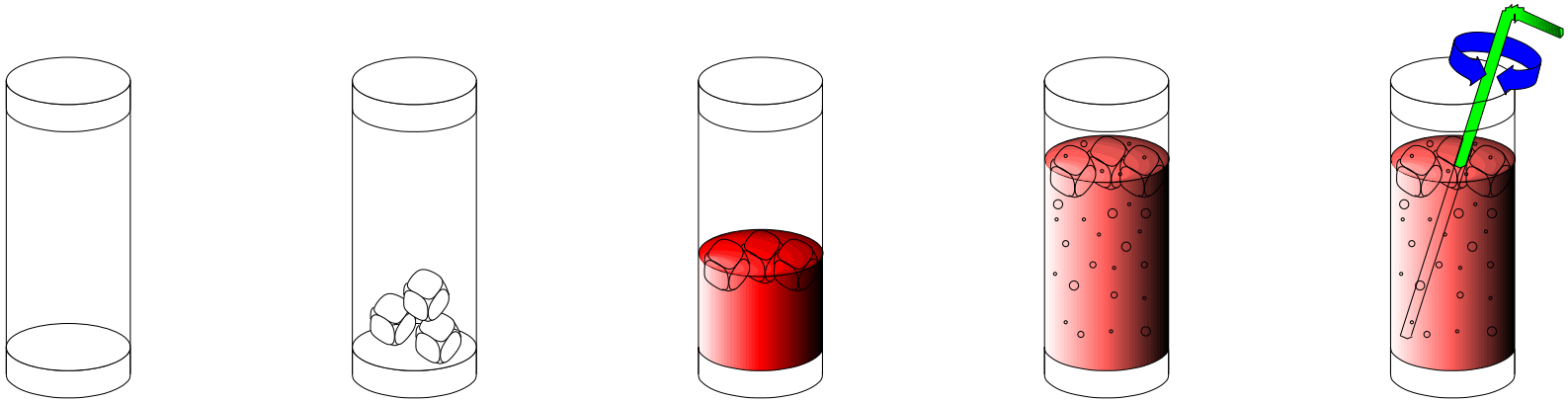


- EJEMPLOS INTUITIVOS DE ALGORITMOS
  - PROBLEMA: Preparación de un "Tinto de Verano"
    - Fases 1 y 2: Análisis y Estudio de la solución



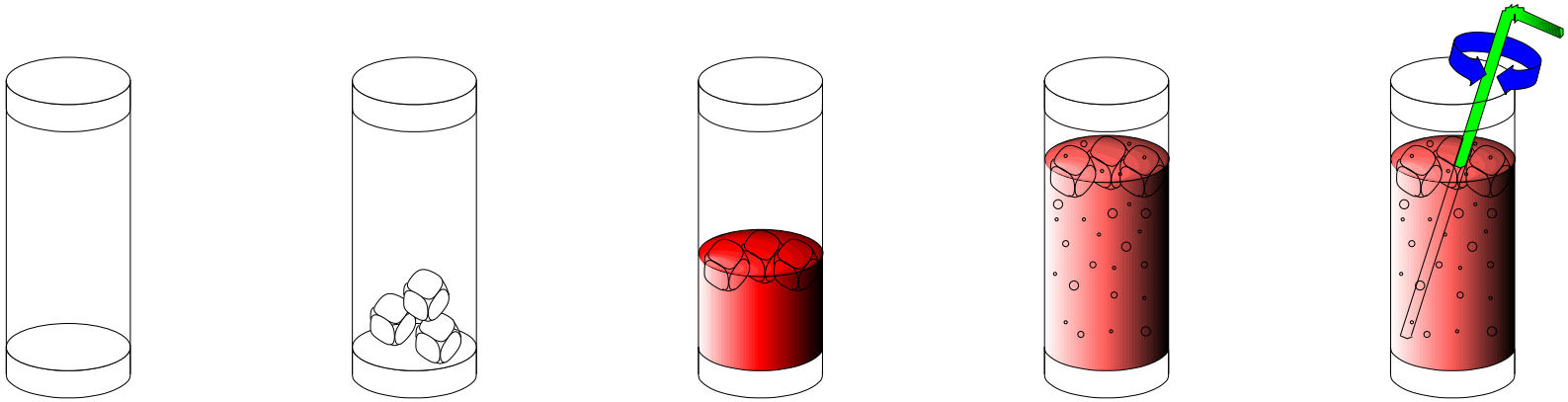
- Fase 3: Diseño del **Algoritmo**

  1. Tomar un vaso.
  2. Colocar algunos cubitos de hielo en el vaso.
  3. Echar vino tinto en el vaso.
  4. Añadir gaseosa al contenido del vaso.
  5. Agitar el contenido.



- Fase 3: Diseño del **Algoritmo**

  1. Tomar un vaso **vacío**.
  2. Colocar **tres** cubitos de hielo en el vaso.
  3. Echar vino tinto **hasta la mitad** del vaso.
  4. Añadir gaseosa **hasta llenar** el vaso.
  5. Agitar **tres segundos** el contenido.



- DEFINICIONES
  - Procesador
    - Entidad capaz de **entender** una secuencia finita de acciones y **ejecutarlas** en la forma en que se especifican
  - Entorno
    - Conjunto de **condiciones** necesarias para la ejecución de un algoritmo
  - Acciones primitivas
    - Son acciones que el procesador es capaz de **entender y ejecutar directamente**

- ALGORITMO
  - Dado un procesador y un entorno bien definido, es el enunciado de una **secuencia finita** de **acciones primitivas** que **resuelven** un determinado problema
    - Lenguaje simbólico a utilizar
    - Acciones Primitivas
    - Representación de los datos.

- Fase 3: Diseño del **Algoritmo** (acciones primitivas)

  1. Tomar un vaso vacío.
  2. Colocar tres cubitos de hielo en el vaso.
  3. Echar vino tinto hasta la mitad del vaso.
  4. Añadir gaseosa hasta llenar el vaso.
  5. Agitar tres segundos el contenido.

- Fase 3: Diseño del **Algoritmo** (acciones primitivas)

  1. Tomar un vaso vacío.
  2. Colocar tres cubitos de hielo en el vaso.
     a) Sacar la cubitera del congelador.
     b) Rociar la parte inferior con agua.
     c) REPETIR
     d) Extraer un cubito.
     e) Echarlo al vaso.
     f) HASTA QUE el nº de cubitos sea 3.
     g) Rellenar los huecos de la cubitera con agua.
     h) Meter de nuevo la cubitera en el congelador.
  3. Echar vino tinto hasta la mitad del vaso.
  4. Añadir gaseosa hasta llenar el vaso.
  5. Agitar tres segundos el contenido.

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

- LENGUAJE DE PROGRAMACIÓN

  - El **significado** de un determinado programa viene definido por la **semántica formal** del lenguaje.

  - Un lenguaje de programación se define mediante una **Gramática**, que establece **formalmente** los símbolos y reglas para describir de forma **no ambigua** los PROGRAMAS.

    - Elementos Léxicos: unidad mínima con significado

    - Reglas Sintácticas: establecen como se combinan los elementos léxicos

    - Reglas Semánticas: establecen el significado consistente de las estructuras sintácticas

- PROGRAMA

  - Descripción de un algoritmo mediante un lenguaje de programación real.

  - Elementos de un programa:
    - DECLARACIONES y DEFINICIONES especifican conceptos y elementos
    - ACCIONES permiten manipular la información
    - OBJETOS sobre los que recaen la acciones
    - CONTROL de la secuencia (flujo) de ejecución
    - MODULARIZACIÓN

- PROGRAMA

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

  - **Compilacion**: `g++ -ansi -Wall -Wextra -Werror -D_GLIBCXX_DEBUG -o programa programa.cpp`

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
// - Tipos ------
// - Subalgoritmos --
// --- [Variables y Acciones]
// - Principal ----
int main ()
{
   // Variables
   // Acciones
}
```

- PROGRAMA

  - En la asignatura utilizaremos un **subconjunto** del lenguaje de programación **C++** para describir los algoritmos. En el sistema operativo Windows (CodeBlocks, Dev-C++, etc):

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
// - Constantes ----
// - Tipos ------
// - Subalgoritmos --
// --- [Variables y Acciones]
// - Principal ----
int main ()
{
   // Variables
   // Acciones
   system("pause");
}
```

PROGRAMA: EJEMPLO

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

- **Compilacion**: `g++ -ansi -Wall -Wextra -Werror -D_GLIBCXX_DEBUG -o programa programa.cpp`

```
#include <iostream>
#include <string>
using namespace std;
// - Constantes ----
const double PTS_1_EUR = 166.386;
// - 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;
}
```

- ACCIONES (permiten manipular la información)
  - Se ejecutan en SECUENCIA
  - Se representan mediante SENTENCIAS ejecutables
    - Sentencias de Incialización y Asignación
    - Sentencias de Entrada/Salida
    - Sentencias de Control de Flujo
    - Llamadas a Subprogramas
    - Nota: en C++ las definiciones de objetos son también sentencias ejecutables (construyen objetos en memoria)

- OBJETOS (sobre los que recaen la acciones)
  - Representan la información que manipula el programa
  - Nombre *Identifica cada objeto y los referencia (Identificadores)*
    - Comienza por **letra**, seguida por una secuencia de **letras**, **dígitos** y **'_'**.
    - Ejemplos: `PTS_1_EUR`, `pts`, `euros`
  - Tipo *define el conjunto de valores que puede tomar y las operaciones aplicables*

`bool`	`char`	`unsigned`	`int`	`double`
Lógicos	*Caracteres*	*Naturales* ($\mathbb{N}$)	*Enteros* ($\mathbb{Z}$)	*Reales* ($\mathbb{R}$)
`false` y `true`	*Tabla ASCII*	$+4294967295$	± 2147483647	$\pm 1.79769313E\pm 308$
`!` `&&` `||`	`==` `!=` `<` `>` `<=` `>=` `+` `-` `*` `/` `%` `++` `--` `>>` `<<`			

  - Valor
    - Constante *no varía durante la ejecución del programa*
    - Variable *puede cambiar durante la ejecución del programa*

OBJETOS: CONSTANTES LITERALES Y SIMBÓLICAS

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

- Las constantes simbólicas asocian un significado a un determinado valor constante. Se definen en la zona de constantes según el siguiente ejemplo y su ámbito de visibilidad abarca todo el programa:

```
// - Constantes ----
const bool DEPURACION = true;
const char LETRA = 'a';
const unsigned KBYTE = 1024;
const int ESCALA = -1;
const double ERROR_PRECISION = 1.56E-7; // 1.56e-7 equivale a 1.56 × 10^-7
const double MAX_VAL = 74E3; // 74e3 equivale a 74.0 × 10^3
// - Principal ----
int main ()
{
   // Acciones
}
```

OBJETOS: VARIABLES

- Su valor **puede variar** durante la ejecución del programa. De entre todos los valores que especifica su tipo, en un momento dado **sólo tomará uno** de ellos.
- Tiene asignado una zona de memoria donde se almacena la representación del valor que toma en un momento dado. Se puede especificar su valor inicial, en otro caso, su valor queda **INESPECIFICADO**
- Las variables se definen dentro del cuerpo del programa principal y del cuerpo de los subalgoritmos. Su ámbito de visibilidad abarca el cuerpo (bloque) en el que están definidas.

```
// - Principal ----
int main ()
{
   bool logico = false;
   char caracter = 'z';
   unsigned natural_1, natural_2;
   int entero = 56;
   double real;
   // Acciones
}
```

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

INICIALIZACIÓN, ASIGNACION Y EXPR. ARITMÉTICAS

- La **inicialización** asigna a un **objeto** un valor inicial resultado de evaluar una expresión aritmética.
- La **asignación** asigna a una **variable** un **nuevo** valor resultado de evaluar una expresión aritmética. El valor anterior **se pierde**.
- La evaluación de una expresión aritmética (expresada en notación **infija**) se realiza según unas reglas de precedencia y de asociatividad

```
// - Principal ----
int main ()
{
   int x1 = 4 + 2 * 6; // Inicializac
   x1 = x1 * 3 % 2 - 3; // Asignación
   ++x1;          // x1 = x1 + 1;
   --x1;          // x1 = x1 - 1;
   x1 += 5;       // x1 = x1 + 5;
   x1 -= 3;       // x1 = x1 - 3;
   x1 *= 2 + 5; // x1 = x1 * (2 + 5);
   x1 /= 2;       // x1 = x1 / 2;
   x1 %= 2;       // x1 = x1 % 2;
}
```

	Precedencia	Aso
paréntesis	( )	NO
unarios	! -	Dch
multiplicativos	* / %	Izq
sumatorios	+ -	Izq
relacionales	< > <= >=	NO
igualdad	== !=	NO
lógico_y	&&	Izq
lógico_o	||	Izq

CONVERSIÓN DE TIPO IMPLÍCITA Y EXPLÍCITA

- Se puede realizar conversión implícita entre tipos simples cuando el tipo destino utiliza una representación *"mayor"* que el tipo origen.

- En otro caso, se debe realizar una conversión explícita de tipos.

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    char letra;
    cin >> letra;
    int valor = int(letra) + 1;
    letra = char(valor);
    letra = char(letra + 1);
    cout << letra << endl;
}
```

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

SENTENCIAS DE ENTRADA/SALIDA

- Sentencia de **entrada** (`cin >>`): permite asignar a una **variable** un determinado valor leído de teclado

- Sentencia de **salida** (`cout <<`): permite mostrar en pantalla el valor de un objeto o expresión (`endl` ≡ *Salto-Línea*)

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int x1, x2;
   cout << "Introduzca dos numeros: ";
   cin >> x1 >> x2;
   cout << "Valor: " << x1 << ' ' << (x2 * 3) << endl;
   cout << "FIN" << endl;
   char c;
   cin.get(c);
}
```

```
Introduzca dos numeros:  12 □ 15 ↩
Valor: 12 45
FIN
```

ENTRADA DE DATOS. 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 (se pulsen una serie de teclas seguida por la tecla `ENTER` $\boxed{\hookleftarrow}$).

- 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

  - La entrada de datos (`>>`) se realiza saltando los caracteres en blanco y saltos de línea hasta encontrar datos. Si el dato leído se puede convertir al tipo de la variable que lo almacenará, entonces la operación será correcta, en otro caso la operación será incorrecta.

  - La entrada de datos (`get`) permite leer el siguiente carácter (`char`) sin saltar los separadores de la entrada de datos.

  - La sentencia `cin >> ws`; permite saltar los separadores de la entrada de datos hasta algún carácter que no sea separador.

SALIDA FORMATEADA DE DATOS

- Es posible especificar el formato aplicado a la salida de datos. Para ello, se debe incluir la biblioteca estándar **iomanip**.

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   cout << boolalpha; // escribe los valores booleanos como 'false' o 'true'
   cout << dec << 27; // escribe 27 (decimal)
   cout << hex << 27; // escribe 1b (hexadecimal)
   cout << oct << 27; // escribe 33 (octal)
   cout << setprecision(2) << 4.567; // escribe 4.6
   cout << setw(5) << 234; // escribe ␣␣234
   cout << setfill('#') << setw(5) << 234; // escribe ##234
}
```

- EJEMPLO
  - Programa que lea dos números enteros y los divida

- EJEMPLO
  - Programa que lea dos números enteros y los divida

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int dividendo, divisor;
    int cociente;
    cin >> dividendo >> divisor;
    cociente =  /  dividendo divisor;
    cout << cociente << endl;
}
```

ERROR SINTÁCTICO

- EJEMPLO
  - Programa que lea dos números enteros y los divida

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int dividendo, divisor;
   const int cociente = 3;
   cin >> dividendo >> divisor;
   cociente = dividendo  /  divisor;
   cout << cociente << endl;
}
```

ERROR SEMÁNTICO

- EJEMPLO
  - Programa que lea dos números enteros y los divida

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int dividendo, divisor;
    int cociente;
    cin >> dividendo >> divisor;
    cociente = dividendo  /  divisor;
    cout << cociente << endl;
}
```

```
RT_ERROR: DIVISIÓN POR CERO
```

- EJEMPLO
  - Programa que lea dos números enteros y los divida

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int dividendo, divisor;
    int cociente;
    cin >> dividendo >> divisor;
    cociente = dividendo  %  divisor;
    cout << cociente << endl;
}
```

ERROR DE PROGRAMACIÓN

- EJEMPLO
  - Programa que lea dos números enteros y los intercambie

- EJEMPLO
  - Programa que lea dos números enteros y los intercambie

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int numero_1, numero_2;
    cin >> numero_1 >> numero_2;
    int auxiliar = numero_2;
    numero_2 = numero_1;
    numero_1 = auxiliar;
    cout << numero_1 << ' ' << numero_2 << endl;
}
```

- EJEMPLO
  - Programa que lea un número entero y muestre si es par o no

- EJEMPLO
  - Programa que lea un número entero y muestre si es par o no

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int numero;
    cin >> numero;
    int resto = numero  %  2;
    bool par = (resto  ==  0);
    // par = ((numero  %  2)  ==  0);
    cout << numero << " es par: " << boolalpha << par << endl;
}
```

- EJEMPLO

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

Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb
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

- EJEMPLO

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

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    char letra_mayuscula;
    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;
}
```

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

EXPRESIÓN LÓGICA

- Es una afirmación que una vez evaluada puede ser verdadera o falsa (`true` o `false`).

- Permite evaluar el estado de la computación en un momento determinado. Es la base para la toma de decisiones

- La evaluación de una expresión lógica (expresada en notación **infija**) se realiza según unas reglas de precedencia y de asociatividad, y la tabla de verdad de los operadores lógicos

	Precedencia	Aso
paréntesis	( )	NO
unarios	! -	Dch
multiplicativos	* / %	Izq
sumatorios	+ -	Izq
relacionales	< > <= >=	NO
igualdad	== !=	NO
lógico_y	&&	Izq
lógico_o	||	Izq

x	y	! x	x && y	x || y
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

Los operadores lógicos (&& ||) se evalúan en **CORTOCIRCUITO**

EVALUACIÓN EN CORTOCIRCUITO DE EXPR. LÓGICAS.

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

- La evaluación en *cortocircuito* significa que 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 **SÍ** es necesario evaluar el segundo operando para calcular el resultado de la expresión lógica.
  - 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

- Las siguientes son expresiones booleanas que pueden ser utilizadas donde su valor sea necesario

  (Asignación, Expr_Bool, SI, MIENTRAS y REPETIR)

```
// - Principal ----
int main ()
{
   int num;
   cin >> num;
   bool error = false;
   bool par = ...;
   bool tres_digitos = ...;
   bool tres_digitos_par = ...;
   bool primo_10 = ...;
   bool divisor_10 = ...;
}
```

EXPRESIONES LÓGICAS. EJEMPLOS

- Las siguientes son expresiones booleanas que pueden ser utilizadas donde su valor sea necesario

  (Asignación, Expr_Bool, SI, MIENTRAS y REPETIR)

```
// - Principal ----
int main ()
{
   int num;
   cin >> num;
   bool error = false;
   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_10 = (num > 0) && ((10 % num) == 0); // Válido gracias al CORTOCIRCUITO
}
```

- EJEMPLO

  - Programa que lea un número entero y compruebe si tiene tres dígitos, y en ese caso, si es capicúa

- EJEMPLO

  - Programa que lea un número entero y compruebe si tiene tres dígitos, y en ese caso, si es capicúa

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int numero;
    cin >> numero;
    int digito_1 = numero  %  10;
    int digito_3 = numero  /  100;
    bool tres_cap = (numero  >=  100)  &&  (numero  <=  999)  &&  (digito_1  ==  digito_3);
    cout << numero << " tiene 3 dígitos y es capicua: " << boolalpha << tres_cap << endl;
}
```

EXPRESIONES LÓGICAS. EQUIVALENCIAS

$$\texttt{!}\ (\ \texttt{!}\ b) \Leftrightarrow b$$
$$\texttt{!}\ (a\ \texttt{\&\&}\ b) \Leftrightarrow (\ \texttt{!}\ a\ \texttt{||}\ \ \texttt{!}\ b)$$
$$\texttt{!}\ (a\ \texttt{||}\ b) \Leftrightarrow (\ \texttt{!}\ a\ \texttt{\&\&}\ \ \texttt{!}\ b)$$

$$\texttt{!}\ (x\ \texttt{==}\ y) \Leftrightarrow (x\ \texttt{!=}\ y)$$
$$\texttt{!}\ (x\ \texttt{>}\ y) \Leftrightarrow (x\ \texttt{<=}\ y)$$
$$\texttt{!}\ (x\ \texttt{<}\ y) \Leftrightarrow (x\ \texttt{>=}\ y)$$

$$(b\ \texttt{==}\ \texttt{true}) \Leftrightarrow (b)$$
$$(b\ \texttt{==}\ \texttt{false}) \Leftrightarrow (\ \texttt{!}\ b)$$

$$(b\ \texttt{!=}\ \texttt{true}) \Leftrightarrow (\ \texttt{!}\ b)$$
$$(b\ \texttt{!=}\ \texttt{false}) \Leftrightarrow (b)$$

$$((a\ \texttt{\&\&}\ b)\ \texttt{||}\ (a\ \texttt{\&\&}\ c)) \Leftrightarrow a\ \texttt{\&\&}\ (b\ \texttt{||}\ c)$$
$$((a\ \texttt{||}\ b)\ \texttt{\&\&}\ (a\ \texttt{||}\ c)) \Leftrightarrow a\ \texttt{||}\ (b\ \texttt{\&\&}\ c)$$

EXPRESIONES LÓGICAS. EJERCICIOS

- Programa que lea datos y muestre si cumplen las siguientes propiedades: (x, y enteros y c carácter)

  1. $\texttt{x} \in \{3,4,5,6,7\}$
  2. $\texttt{x} \in \{1,2,3,7,8,9\}$
  3. $\texttt{x} \in \{1,3,5,7,9\}$
  4. $\texttt{x} \in \{2,5,6,7,8,9\}$
  5. $\texttt{x} \in \{3,4,6,8,9\}$, $\texttt{y} \in \{6,7,8,3\}$
  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
  10. `c` es un alfanumérico

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

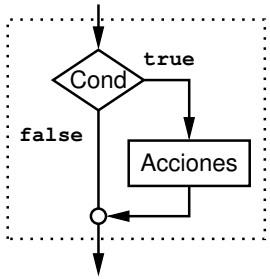
1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

- SENTENCIAS DE SELECCIÓN.
  - Permiten seleccionar la ejecución **alternativa** y **excluyente** de unas sentencias u otras dependiendo del valor de una expresión
    - SI
    - CASO

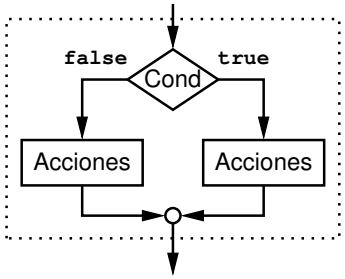
SENTENCIA DE SELECCIÓN **SI**

Sintaxis

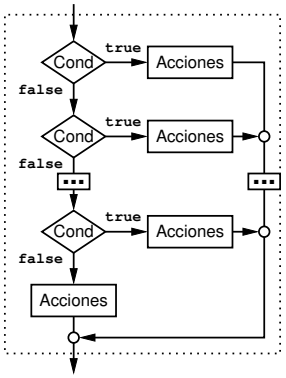
```
// - Principal ----
int main ()
{
   // SIMPLE
   if ( EXPR_LOGICA_1 ) {
      // Acciones_1
   }
}
```




```
// - Principal ----
int main ()
{
   // COMPUESTA
   if ( EXPR_LOGICA_1 ) {
      // Acciones_1
   } else {
      // Acciones_otro_caso
   }
}
```




```
// - Principal ----
int main ()
{
   // MULTIPLE
   if ( EXPR_LOGICA_1 ) {
      // Acciones_1
   } else if ( EXPR_LOGICA_2 ) {
      // Acciones_2
   } else if ( EXPR_LOGICA_3 ) {
      // Acciones_3
   } else {
      // Acciones_otro_caso
   }
}
```



- SENTENCIA DE SELECCIÓN **SI**. EJEMPLO

  - Programa que lea un número Real comprendido entre 0 y 10 e imprima la nota asociada

$$
\begin{aligned}
n = 10 &\ -> \ \textit{Matricula de Honor} \\
9 \leq n < 10 &\ -> \ \textit{Sobresaliente} \\
7 \leq n < 9 &\ -> \ \textit{Notable} \\
5 \leq n < 7 &\ -> \ \textit{Aprobado} \\
0 \leq n < 5 &\ -> \ \textit{Suspenso}
\end{aligned}
$$

■ SENTENCIA DE SELECCIÓN **SI**. EJEMPLO

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   double 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;
   }
}
```

SENTENCIA DE SELECCIÓN **SI**. EJEMPLO

- Programa que imprime el mayor de tres números

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int a, b, c;
   cin >> a >> b >> c;
   int mayor = a;
   if (b > mayor) {
      mayor = b;
   }
   if (c > mayor) {
      mayor = c;
   }
   cout << mayor << endl;
}
```

SENTENCIA DE SELECCIÓN **SI**. EJEMPLO

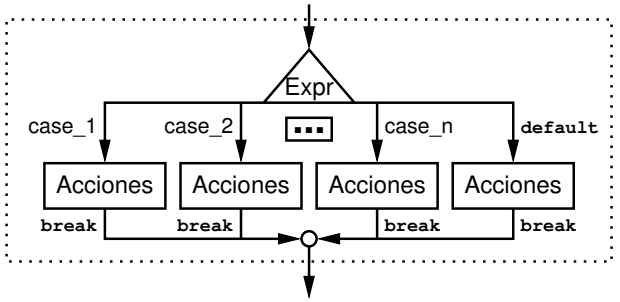
- Programa que imprime el mayor de cuatro números

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int a, b, c, d;
   cin >> a >> b >> c >> d;
   int mayor = a;
   if (b > mayor) {
      mayor = b;
   }
   if (c > mayor) {
      mayor = c;
   }
   if (d > mayor) {
      mayor = d;
   }
   cout << mayor << endl;
}
```

SENTENCIA DE SELECCIÓN **CASO**

Sintaxis

```
// - Principal ----
int main ()
{
   switch ( EXPRESION ) {
   case CTE_1:
      // Acciones (EXPR == CTE_1)
      break;
   case CTE_2:
      // Acciones (EXPR == CTE_2)
      break;
   case CTE_3:
   case CTE_4:
      // Acciones ((EXPR == CTE_3) || (EXPR == CTE_4))
      break;
   default:
      // Acciones_otro_caso
      break;
   }
}
```



- **Nota: no** está permitido *declarar variables* dentro de los casos de la sentencia `switch`

- SENTENCIA DE SELECCIÓN **CASO**. EJEMPLO

  - Programa que lea una letra que designe el dia de la semana y muestre por pantalla el nombre del dia.

l	$->$	Lunes
m	$->$	Martes
x	$->$	Miercoles
j	$->$	Jueves
v	$->$	Viernes
s	$->$	Sabado
d	$->$	Domingo

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    char letra;
    cin >> letra;
    switch (letra) {
    case 'l':
        cout << "Lunes" << endl;
        break;
    case 'm':
        cout << "Martes" << endl;
        break;
    case 'x':
        cout << "Miercoles" << endl;
        break;
    case 'j':
        cout << "Jueves" << endl;
        break;
    case 'v':
        cout << "Viernes" << endl;
        break;
    case 's':
        cout << "Sabado" << endl;
        break;
    case 'd':
        cout << "Domingo" << endl;
        break;
    default:
        cout << "Error" << endl;
        break;
    }
}
```

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

- SENTENCIAS DE ITERACIÓN. BUCLES

  - Repiten la ejecución de una secuencia de sentencias (el cuerpo del bucle)

  - El número de iteraciones depende de una condición (expresión lógica) cuyo valor evoluciona durante la ejecución del cuerpo del bucle

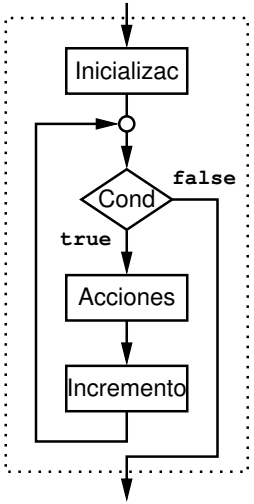
  - Tres sentencias de iteración diferentes

    `for`, `while`, `do`–`while`

SENTENCIA DE ITERACIÓN. **FOR**

- Se utiliza cuando el número de iteraciones que se van a realizar está **predeterminado** antes de su ejecución. El proceso iterativo está controlado por los valores que toma una **variable** de control del bucle.
  - INICIALIZACIÓN: inicializa la variable de control. Se ejecuta **una única vez** antes del proceso iterativo. Si se define la variable, su ámbito de visibilidad es el cuerpo del bucle.
  - EXPR_LÓGICA: controla la iteración, si es **Verdadera**, entonces se ejecuta el cuerpo del bucle. Se evalúa **iterativamente antes** de ejecutar el cuerpo del bucle. Contiene una expresión aritmética que determina un límite máximo de iteraciones.
  - INCREMENTO: evoluciona la variable de control. Se ejecuta **iterativamente 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 límite maximo de iteraciones.
- Sintaxis

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



SENTENCIA DE ITERACIÓN. **FOR**. EJEMPLO

- Programa que imprime los **N** primeros números enteros mayores o iguales a cero:

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int n;
   cin >> n;
   for (int i = 0; i < n; ++i) {
      cout << i << " ";
   }
   cout << endl;
}
```

- SENTENCIA DE ITERACIÓN. **FOR**. EJEMPLO

  - Programa que escribe los **n** primeros números enteros impares mayores que cero [1 3 5 7 $\cdots$]

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int n;
   cin >> n;
   for (int i = 0; i < n; ++i) {
      cout << ((i * 2) + 1) << " ";
   }
   cout << endl;
}
```

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int n;
   cin >> n;
   for (int i = 1; i < 2 * n; i += 2) {
      cout << i << " ";
   }
   cout << endl;
}
```

- 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**

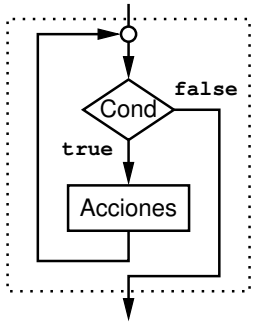
  - La condición de control se evalúa **antes** de ejecutar el cuerpo del bucle.

  - 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.

  - Sintaxis

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



SENTENCIA DE ITERACIÓN **WHILE**. EJEMPLO

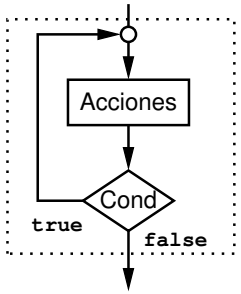
- Escribir el primer número mayor que 1 que sea divisor de un número entero leído de teclado (si es 0 ó 1, entonces escribirá 1)

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int num, divisor;
   cin >> num;
   if (num <= 1) {
      divisor = 1;
   } else {
      divisor = 2;
      while ((num % divisor) != 0) {
         ++divisor;
      }
   }
   cout << "El primer divisor de " << num << " es " << divisor << endl;
}
```

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.
- Sintaxis

```
// - Principal ----
int main ()
{
   do {
      // Acciones (cuerpo del bucle)
   } while ( EXPR_LOGICA );
}
```



- SENTENCIA DE ITERACIÓN **DO-WHILE**. EJEMPLO

  - Leer un número entero que sea par y mostrarlo en pantalla

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int num;
    do {
        cin >> num;
    } while ((num % 2) != 0);
    cout << "El numero par es " << num << endl;
}
```

- DISEÑO DE BUCLES.

  - Para el diseño correcto de un bucle se deben considerar los siguientes aspectos:

    1. Control de la **iteración**

       a) Condición de finalización

       b) Inicialización y evolución de dicha condición.

    2. Proceso repetitivo

       a) Acción iterativa

       b) Inicialización y actualización.

  - A veces ambos aspectos se entremezclan

- EJERCICIOS

  1. Programa que multiplique 2 números enteros mediante sumas
  2. Programa que calcule el factorial de un número
  3. Programa que divida 2 números enteros mediante restas

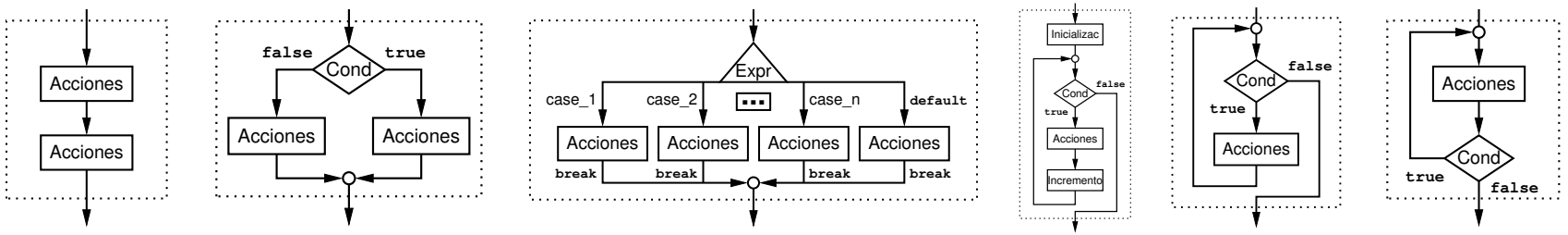
```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int m, n;
    cin >> m >> n;
                          n veces
    // Sumar: m + m + m + ··· + m
    int total = 0;
    for (int i = 0; i < n; ++i) {
        // Proceso iterativo: Añadir el valor de 'm' al total
        total = total + m;
    }
    cout << total << endl;
}
```

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   int n;
   cin >> n;
   // Multiplicar: 1 2 3 4 5 6 7 ··· n
   int fact = 1;
   for (int i = 1; i <= n; ++i) {
      // Proceso iterativo: multiplicar numero
      fact = fact * i;
   }
   cout << fact << endl;
}
```

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
    int dividendo, divisor;
    cin >> dividendo >> divisor;
    if (divisor == 0) {
        cout << "El divisor no puede ser cero" << endl;
    } else {
        int resto = dividendo;
        int cociente = 0;
        while (resto >= divisor) {
            resto -= divisor;
            ++cociente;
        }
        cout << cociente << ' ' << resto << endl;
    }
}
```

PROGRAMACIÓN ESTRUCTURADA

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



- Esta característica hace posible que se pueda aplicar la **abstracción** para su diseño y desarrollo
- La abstracción se basa en la identificación de los elementos a un determinado nivel, ignorando los detalles especificados en niveles inferiores.
- Un algoritmo que use tan sólo las estructuras de control tratadas en este tema, se denomina **estructurado**.
- Bohm y Jacopini demostraron que todo problema computable puede resolverse usando únicamente estas estructuras. Esta es la base de la **programación estructurada**.
- La **programación modular** va un paso más allá y considera que cada nivel de refinamiento corresponde con un módulo abstracto independiente

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]

- TIPO (I)
  - Define el conjunto de valores que puede tomar un determinado objeto.
  - Determina las operaciones que se pueden aplicar a un objeto.
  - Define la interpretación del valor almacenado en memoria.
  - Define el espacio que será necesario reservar en memoria para albergar a un objeto.
  - Tipos de datos básicos predefinidos en **C++**:

`bool`	`char`	`unsigned`	`int`	`double`
8 bits	*8 bits*	*32 bits*	*32 bits*	*64 bits*
Lógicos	*Caracteres*	*Naturales* ($\mathbb{N}$)	*Enteros* ($\mathbb{Z}$)	*Reales* ($\mathbb{R}$)
`false` y `true`	*Tabla ASCII*	*Binario*	*Bin Cmpl 2*	*Pto Flotante*
		$+4294967295$	± 2147483647	$\pm 1.79769313E\pm 308$
0 / 1		$0 \cdots 2^n - 1$	$-2^{n-1} \cdots 2^{n-1} - 1$	*Mant:Exp*
`!` `&&` `||`	`==` `!=` `<` `>` `<=` `>=` `+` `-` `*` `/` `%` `++` `--` `>>` `<<`			

TIPO (II)

Tipos simples predefinidos en C++:

Lógicos	*Caracteres*	*Naturales* ($\mathbb{N}$)	*Enteros* ($\mathbb{Z}$)	*Reales* ($\mathbb{R}$)

`bool`
8 bits
`false` y `true`
	`char`
	8 bits
	Tabla ASCII
		`unsigned` [`int`]
		32 bits
		+4294967295
			`int`
			32 bits
			±2147483647
				`double`
				64 bits
				±1.79769E±308

	`signed char` *8 bits*	`unsigned short` [`int`] *16 bits* +65535	`short` [`int`] *16 bits* ±32767	`float` *32 bits* ±3.40282E±38
	`unsigned char` *8 bits*	`unsigned long` [`int`] *32/64 bits* +18446744073709551615	`long` [`int`] *32/64 bits* ±9223372036854775807	`long double` *96/128 bits* ±1.18973E±4932
Binario *0 / 1*	*Tabla ASCII*	*Binario* $0 \cdots 2^n - 1$	*Bin Cmpl 2* $-2^{n-1} \cdots 2^{n-1} - 1$	*Pto Flotante* *Mant:Exp*
`!` `&&` `||`	`==` `!=` `<` `>` `<=` `>=` `+` `-` `*` `/` `%` `++` `--` `>>` `<<`			

Es posible definir nombres alternativos para los tipos:

- `typedef unsigned uint ;`
- `typedef unsigned long ulong ;`

INTERPRETACIÓN DE LA REPRESENTACIÓN

- Internamente toda la información se representa 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.

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   bool cond = true;
   unsigned natural = 1;
   char letra = 'A';
   int entero = 65;
}
```

true	0000:0001
1	0000:0001
'A'	0100:0001
65	0100:0001

CLASIFICACIÓN DE LOS TIPOS

<table>
<tr><th></th><th>Simples</th><th>Compuestos</th></tr>
<tr><td>Predefinidos</td><td>

- Lógico: `bool`
- Carácter: `char`
- Natural: `unsigned`
- Entero: `int`
- Real: `double`

</td><td>

- Cadena: `string`

</td></tr>
<tr><td>Definidos</td><td>

- Puntero:

```
typedef Persona* PPersona;
```

</td><td>

- Registro o Estructura:

```
struct Persona {
    string nombre;
    int edad;
};
```

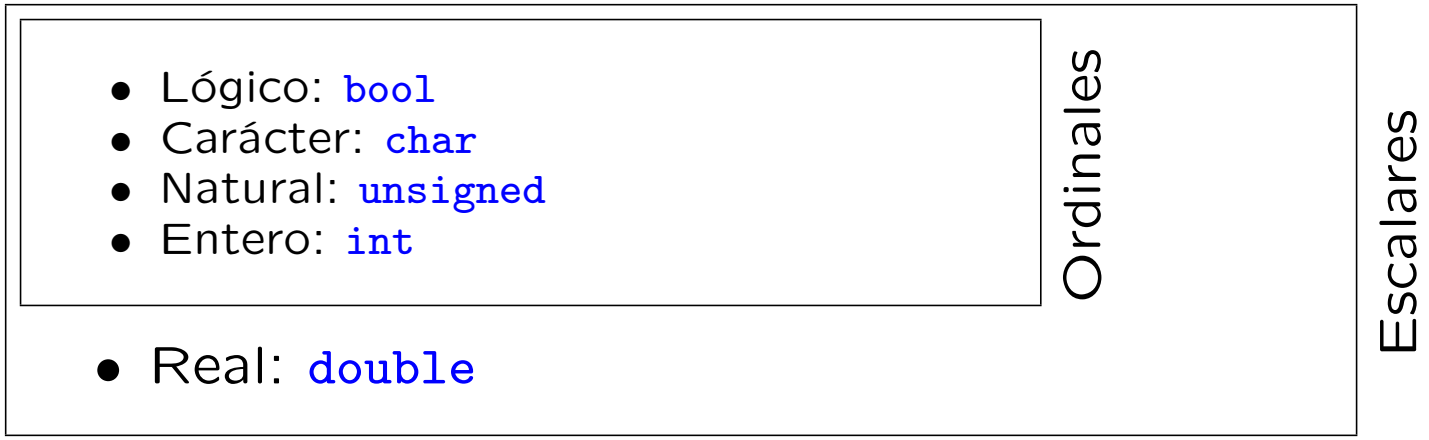
- Array:

```
typedef array<Persona, NALUM> Alumnos;
```

</td></tr>
</table>

TIPOS ESCALARES Y ORDINALES

- Escalares:
  - Fomados por elementos indivisibles (simples).
  - Formados por elementos ordenados, es decir, le son aplicables los operadores relacionales `==` , `!=` , `<` , `>` , `<=` , `>=`
- Ordinales (Integrales): Además de las propiedades anteriores:
  - Cada valor tiene un predecesor y un sucesor únicos (excepto el primero y el último, respectivamente).



- TIPO LÓGICO (BOOLEANO)
  - Representa los valores de verdad: `false` y `true`
  - Resultado de expresiones relacionales y lógicas
  - Operadores aplicables: operadores lógicos `!` , `&&` , `||`
  - Tabla de Verdad:

x	y	`!` x	x `&&` y	x `||` y
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

  - Entrada/Salida: `>>` , `<<`

- TIPO CARÁCTER

  - Representa los símbolos 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 apostrofes: `'a'`, `'F'`, `'5'`, `'.'`, `'+'`, etc.

  - Necesitan de una tabla de conversión para la representación interna. La más utilizada la tabla ASCII asocia a cada carácter una posición ordenada (número de orden).

- TIPO CARÁCTER
  - Operadores aplicables:
    - Relacionales: `==` , `!=` , `<` , `>` , `<=` , `>=`
    - Conversion: Es posible convertirlos al tipo *entero*, obteniendo su posicion en la tabla de conversión (ASCII), y a la inversa.
    - Incremento/Decremento: `++`, `--`
    - Entrada/Salida: `>>` , `<<`

- TABLA ASCII (representación de caracteres) [0..31] Car. Control

Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb	Rep	Simb
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

- TIPO NATURAL

  - Representa el concepto matemático de *Número Natural* limitado a un rango de valores

  - Los valores se representan en el lenguaje de programación como una secuencia de dígitos: `1245`

  - Representación interna en binario puro para un número de bits concreto que depende de la palabra de CPU

  - Rango de valores representados: $[0..2^n - 1]$

- TIPO NATURAL
  - Operadores aplicables:
    - Aritméticos: `+` , `-` , `*` , `/` , `%`
    - Relacionales: `==` , `!=` , `<` , `>` , `<=` , `>=`
    - Conversion: Es posible convertirlos a otros tipos escalares si y solo si están dentro del rango correcto y a la inversa.
    - Incremento/Decremento: `++`, `--`
    - Entrada/Salida: `>>` , `<<`

- TIPO ENTERO

  - Representa el concepto matemático de *Número Entero* limitado a un rango de valores

  - Los valores se representan en el lenguaje de programación como una secuencia de dígitos, opcionalmente precedidos por el símbolo negativo: `-1245`

  - Representación interna en complemento a dos para un número de bits concreto que depende de la palabra de CPU

  - Rango de valores representados: $[-2^{n-1}..2^{n-1}-1]$

- TIPO ENTERO
  - Operadores aplicables:
    - Aritméticos: `+` , `-` , `*` , `/` , `%`
    - Relacionales: `==` , `!=` , `<` , `>` , `<=` , `>=`
    - Conversion: Es posible convertirlos a otros tipos escalares si y solo si están dentro del rango correcto y a la inversa.
    - Incremento/Decremento: `++`, `--`
    - Entrada/Salida: `>>` , `<<`

- TIPO REAL

  - Representa el concepto matemático de *Número Real* limitado a un rango de valores y con precisión limitada

  - Los valores se representan en el lenguaje de programación como una secuencia de dígitos, opcionalmente precedidos por el símbolo negativo, una parte decimal y un factor de escala:
    - `-1245.678E2` (equivalente a -1245.678×10^2),
    - `-1245.678E-3` (equivalente a -1245.678×10^{-3})

  - Representación interna en coma flotante (mantisa, exponente) para un número de bits concreto que depende de la palabra de CPU.

  - En la representación interna, ambas partes son limitadas por lo que la representación es INEXACTA

- TIPO REAL
  - Operadores aplicables:
    - Aritméticos: `+` , `-` , `*` , `/`
    - Relacionales: `==` , `!=` , `<` , `>` , `<=` , `>=`
    - Conversion: Es posible convertirlos a los tipos *Natural* y *Entero* si y solo si están dentro del rango correcto y a la inversa. La parte decimal se pierde.
    - Incremento/Decremento: `++`, `--`
    - Entrada/Salida: `>>` , `<<`

- COMPATIBILIDAD DE TIPOS

  - Los operadores tienen que aplicarse a objetos de tipos de datos compatibles.

  - La información del tipo de dato ayuda a los compiladores a detectar operaciones inapropiadas con tipos de datos no compatibles.

  - Hay lenguajes de programación más estrictos que otros en cuanto al chequeo de tipos.

- CONVERSIONES ENTRE TIPOS

  - Es posible convertir implícitamente valores de un tipo escalar a otro tipo escalar **si y solo si están dentro del rango correcto**, según el siguiente esquema:

Tipo Origen \ *Tipo Destino*	`bool`	`char`	`unsigned`	`int`	`double`
`bool`	*	SI	SI	SI	SI
`char`	NO	*	SI	SI	SI
`unsigned`	NO	NO	*	NO	SI
`int`	NO	NO	NO	*	SI
`double`	NO	NO	NO	NO	*

  - Para convertir explícitamente, se precede el valor entre paréntesis por el símbolo del tipo al que queremos convertir: `char`(48) `unsigned`('0') `double`(50) `int`(3.1416) `int`(valor * 45.56)

- PROBLEMAS DERIVADOS DE LA IMPLEMENTACIÓN DE LOS TIPOS NUMÉRICOS

  - El número de bits utilizados para representar los números *Naturales*, *Enteros* y *Reales* es limitado (palabra de CPU).

  - El resultado de aplicar algún operador a dos valores de un determinado tipo puede dar lugar a un valor no representable (fuera de rango) dando lugar a lo que se conoce como **OVERFLOW**

  - Ejemplo para una representación de Naturales de 4 bits:

```
   7       0111
+ 10     + 1010
-----   ---------
  17      10001
```

- # PROBLEMAS DERIVADOS DE LA IMPLEMENTACIÓN DE LOS TIPOS NUMÉRICOS

  - El número de bits utilizado para representar la parte fraccionaria de los números *Reales* es limitado. Implica **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 deberemos realizar comparaciones de igualdad (==) con el resultado de éstas.
  - Por ejemplo, las siguientes operaciones matemáticas equivalentes producen un resultado inesperado.

```
#include <iostream>
#include <string>
using namespace std;
// - Principal ----
int main ()
{
   bool ok = (3.0 * (0.1 / 3.0)) == ((3.0 * 0.1) / 3.0);
   cout << boolalpha << ok << endl; // Escribe 'false'. Matematicamente deberia ser 'true'
}
```

Tema 2: CONCEPTOS BÁSICOS DE PROGRAMACIÓN

1. Resolución de problemas. Concepto de Algoritmo
2. Programa. Acciones y Objetos
3. Inicialización, Asignación y Expresiones Aritméticas
4. Entrada/Salida básica
5. Expresiones Lógicas
6. Estructuras de control: Selección
7. Estructuras de control: Iteración
8. Tipos de datos simples
9. Bibliografía: [DALE89a], [BROO95], [GOLD88], [JOYA03]