UD1: El lenguaje de programación C++

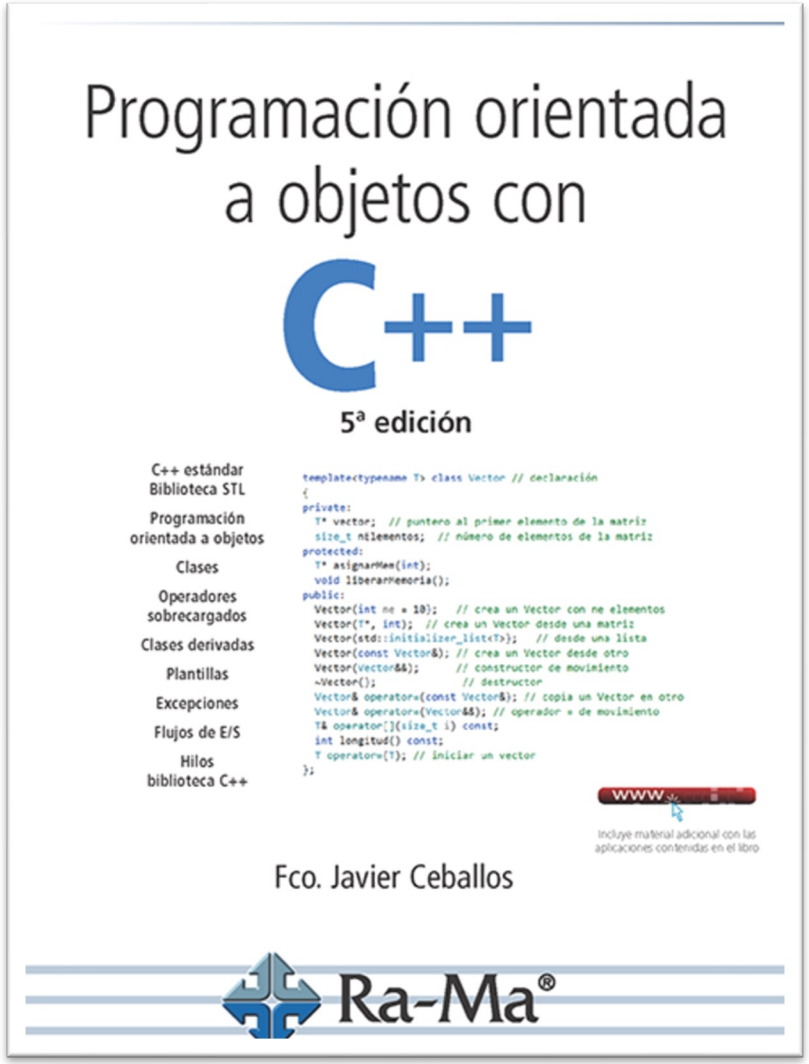
Tema 1 - Introducción a C++

« Me lo explicaron y lo olvidé, lo vi y lo aprendí, lo hice y lo entendí.»

- Confucio -

Bibliografía - Ceballos

- Capítulos 1, 3 y apéndice A

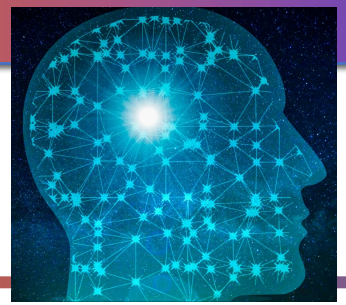


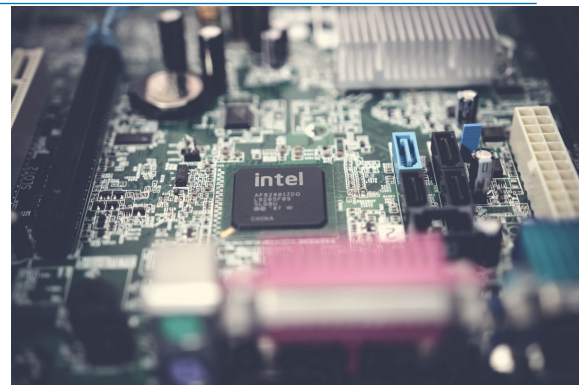
Universidad Politécnica de Cartagena

Dominio del problema

Dominio de la solución (ordenador)

Traducir





Interpretar

Requiere estas habilidades:

- Entender el problema
- Ser capaz de resolver el problema
- Diseñar un proceso sistemático (algoritmo) a partir de la solución. Requiere **pensamiento lógico y matemático**

- Conocer las características del lenguaje de programación
- Ser capaz de utilizar correctamente el lenguaje de programación
- Ser capaz de traducir el algoritmo al lenguaje de programación. Requiere **pensamiento lógico y matemático**

El código sigue al problema, no al revés. El código imita (o embebe/replica) la estructura, entidades y las reglas del dominio del problema para implementar un procedimiento que solucione el problema. El programador tiene que saber qué quiere hacer

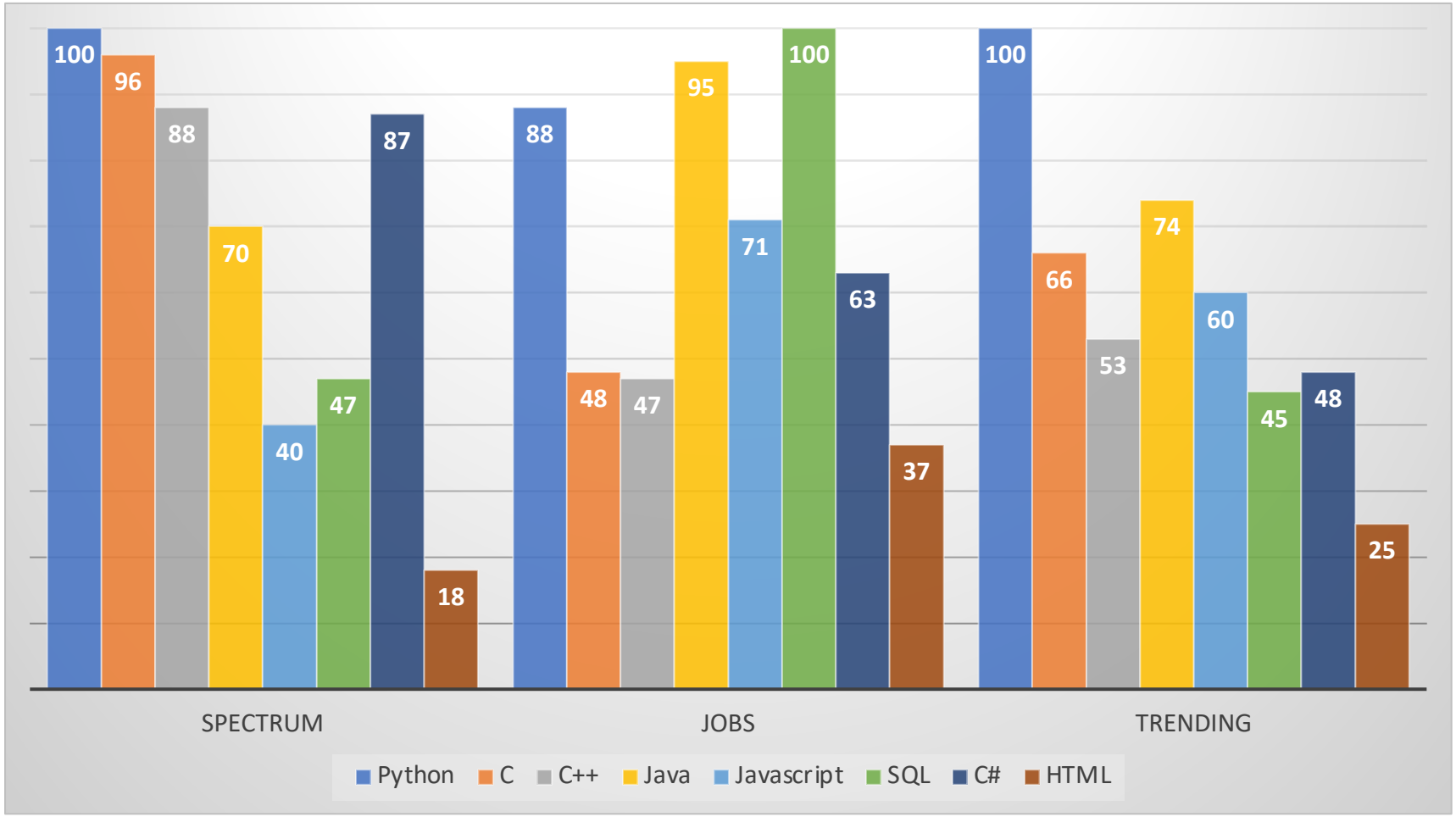
¿Qué es programar? Mecanismos del lenguaje

1. Encontrar una representación "informática" de los datos del problema
   - Tipos primitivos: enteros, reales, caracteres
   - Tipos compuestos: enumerados, vectores, estructuras y clases, plantillas, punteros y referencias
2. Manipular dichas representaciones para generar resultados
   - Expresiones (aritmético-lógicas) para realizar cálculos
   - Sentencias de selección y repetición para controlar la ejecución de sentencias
   - Funciones como unidad de descomposición, reutilización y organización

Abstracción de datos: la implementación interna de los datos (secuencias de bits) y su manejo se oculta al programador

Abstracción de control (acciones): alterar, de forma controlada, el orden secuencial del programa sin necesidad de conocer la estructura interna de los computadores

IEEE Top 10 programming languages 2022



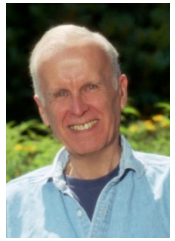
https://spectrum.ieee.org/top-programming-languages-2022

¿Qué es C++?

- No es solo una versión mejorada de C, sino que tiene su propia forma de realizar las cosas. Hay que aprender "*the C++ way*"
- A veces, la compatibilidad y el parecido con C es un lastre y una fuente de errores para el programador novel, que hace las cosas "*the C way*"
- Su diseño comenzó en 1979 y es estándar ISO/IEC 14882:1998
- Compatible hacia atrás con C++ y con gran parte de C
- Lenguaje dominante en muchos sectores productivos (videojuegos, sistemas operativos, infraestructura, bases datos, redes, etc.)
- C++ es * lenguajes de programación en uno (multi-paradigma):
  - Imperativo y estructurado: C
  - Orientado a objetos
  - Funcional
  - Programación de plantillas
  - \+ Una gran colección de librerías

La historia de C++

Universidad Politécnica de Cartagena



Abstracciones específicas del dominio

Fortran

1957

1960

Cobol

Abstracciones de propósito general

Simula

1962

Traducción directa al HW

Assembler → BCPL → C

1966

1970

C++

1979

C++11

Java

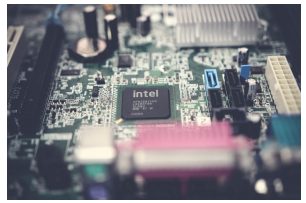
1995

C#

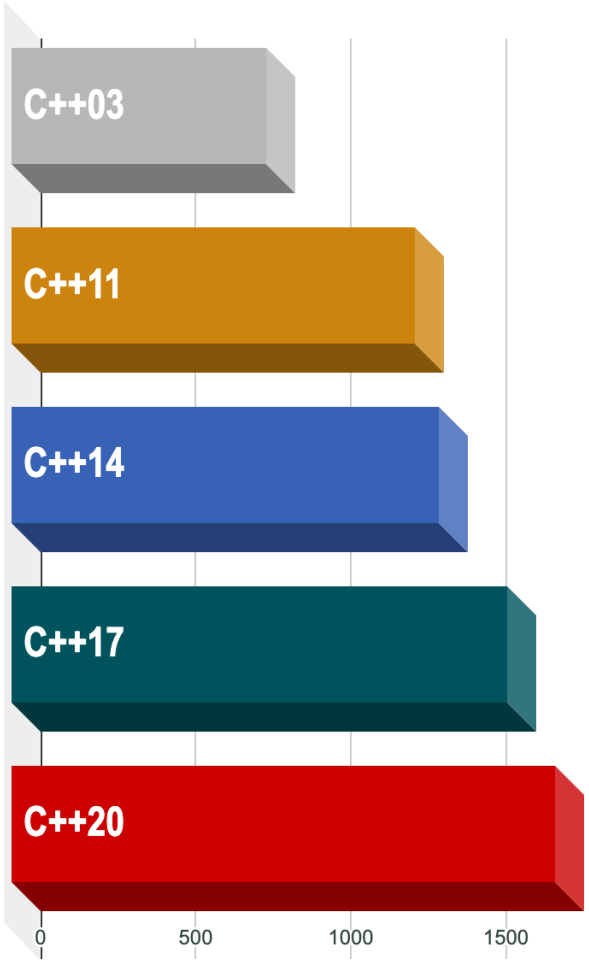
https://www.youtube.com/watch?v=15Wiurmu1oM

Dominio del problema

Dominio de la solución (ordenador)



Número de páginas de los estándares de C++



Sintaxis del lenguaje C/C++

- Define cómo se construyen frases válidas combinando los elementos léxicos (*tokens*) del lenguaje
- **Sentencia simple**: agrupación válida de *tokens* acabada en ‘**;**’
  - Declaración de variables, expresiones de asignación, invocación de funciones, sentencias **return**
- **Sentencia compuesta**: agrupación válida de sentencias, simples o compuestas, delimitada por un bloque ‘**{**‘ ‘**}**’
  - Definición de funciones, sentencias de control de ejecución (selección y repetición), definición de algunos tipos de dato, inicialización de variables
- *Expresión*: conjunto de identificadores o literales unidos por operadores que, tras ser evaluado, produce un valor

La función es la unidad fundamental de organización de código en C/C++ ($programa = \sum funciones$). Las funciones NO contienen funciones

Los *tokens* en los lenguajes de programación

- El *token* es la unidad de información más pequeña de un lenguaje
- Hay 5 tipos de *token* en C++:
  - **Identificadores:** nombran variables, funciones, espacios de nombres, tipos y plantillas. Es un secuencia de letras (a-z, A-Z, _) y números (0-9) que no comienzan por un número.
  - **Palabra clave**: tienen significado especial para el lenguaje → semántica
  - Un **literal** describe un valor "tal cual": 1 (entero), 3.14 (real), 'a' (caracter), "texto" (*string*), true (*bool*), nullptr, {*valores*} (struct/class), y literales del usuario (http://en.cppreference.com/w/cpp/language/user_literal)
  - Los **operadores** se utilizan en expresiones: conjunto de identificadores of literales unidos por operadores que generan un valor al ser evaluadas
  - Los **separadores** separan *tokens*: espacio, tabulador, nueva línea, { } [ ] , ; ( )

¿Qué código puedo escribir en la definición de una función?

① Declaración de variables locales con/sin inicialización (valor generado mediante una expresión). Las variables tienen que declararse antes de ser utilizadas en expresiones:

```
double a, b, c=9;
int vector[3] = {1,2,3};
```

EXPRESIÓN: conjunto de identificadores (variables locales, variables globales, argumentos o funciones) y literales, unidos mediante operadores, que genera un valor

② Expresión de asignación:

```
a = (b+c)/9 * cos(b/3);
```

③ Invocación de otras funciones (de biblioteca o creadas por nosotros), suministrando valores (mediante expresiones) a sus parámetros y gestionando en valor retornado:

```
printf("a es %d, 2*a es %d\n", a, 2*a);
```

④ Sentencias de selección (`if/else`, `switch`) o repetición (`while`, `do/while`, `for`), en secuencia o anidadas unas dentro de otras:

- Las condiciones de estas sentencias son expresiones lógicas
- Son sentencias compuestas, de bloque. Dentro del bloque se pueden utilizar todos los elementos enumerados en esta transparencia
- En sentencias de repetición, la condición debe hacerse falsa en algún momento para evitar ∞

⑤ Sentencia `return`, con o sin valor de retorno (expresión), que termina la ejecución de la función. El tipo del valor de retorno debe coincidir con el tipo de retorno de la función

Ejemplo: algoritmo babilónico $\sqrt{x}$ (X a.C.)

$$aprox_raiz_{n+1} = \frac{aprox_raiz_n + \frac{numero}{aprox_raiz_n}}{2}$$

⏎ es ENTER

```
Numero para calcular raiz cuadrada? -3⏎
Numero para calcular raiz cuadrada? 81⏎
Aproximacion 0: 40.500000000000000
Aproximacion 1: 21.250000000000000
Aproximacion 2: 12.530882352941177
Aproximacion 3: 9.497456198181656
Aproximacion 4: 9.013027839452249
Aproximacion 5: 9.000009415515176
Aproximacion 6: 9.000000000004924
Aproximacion 7: 9.000000000000000
```

Solo 26 iteraciones son necesarias para calcular $\sqrt{9987665524241}$

```
#include <iostream> // librería de E/S
#include <cmath>    // librería matemática

using namespace std; // facilita la vida (ya se contará)

int main() {
    double v_actual, v_sig, numero; // variables
    int i;
    bool continuar=true; // nuevo tipo dato
    do {
      cout<<"Numero para calcular raiz cuadrada? ";
      cin >> numero;
    } while (numero<0);
    v_actual = numero/2; // suposicion 1
    cout << "Aproximacion 0: "<< v_actual<<'\n';
    for (i=0; i<100 && continuar; ++i) {
        v_sig = (v_actual+(numero/v_actual))/2;
        if (fabs(v_sig-v_actual) < 1e-9)
          { continuar=false; }
        v_actual=v_sig;
        cout << "Aproximacion "<< i+1<<": "<< v_actual<<'\n';
    }
    return 0;
}
```

asegurar que el número es positivo

repito hasta encontrar el valor "correcto"

¿estoy lo suficientemente cerca del valor "correcto"?

Palabras reservadas (95+)

https://en.cppreference.com/w/cpp/keyword

alignas	const	for	private	thread_local
alignof	constexpr	friend	protected	throw
and	const_cast	goto	public	true
and_eq	continue	if	~~register~~	try
asm	decltype	inline	reinterpret_cast	typedef
auto	default	int	requires	typeid
bitand	delete	long	return	typename
bitor	do	mutable	short	union
bool	double	namespace	signed	unsigned
break	dynamic_cast	new	sizeof	using
case	else	noexcept	static	virtual
catch	enum	not	static_assert	void
char	explicit	not_eq	static_cast	volatile
char16_t	export	nullptr	struct	wchar_t
char32_t	extern	operator	switch	while
class	false	or	template	xor
compl	float	or_eq	this	xor_eq
concept				

Programa = datos + algoritmos (Niklaus Wirth)

Datos: auto, bool, char, char16_t, char32_t, const, constexpr, const_cast, decltype, delete, double, dynamic_cast, enum, extern, false, float, int, long, new, nullptr, register, reinterpret_cast, short, signed, static, static_cast, thread_local, true, typedef, union, unsigned, void, volatile, wchar_t

- **Clases (OO)**: class, explicit, friend, mutable, private, protected, public, struct, this, virtual

Algoritmo: break, case, catch, continue, default, do, else, for, goto, if, return, switch, throw, try, while

- **Operadores**: alignas, alignof, and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, sizeof, typeid, typename, xor, xor_eq

Otros: asm, concept, export, inline, namespace, noexcept, operator, requires, static_assert, template, using

Operadores

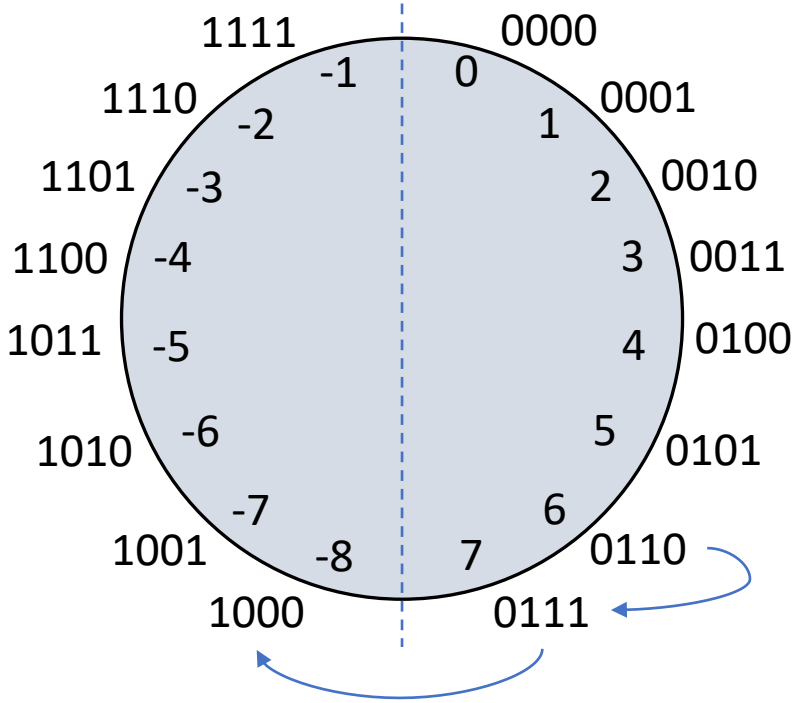
Asignación	Incremento decremento	Aritméticos	Lógicos	Comparación	Acceso a miembros	Otros

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`? :`

Palabras clave: **`and`, `bitor`, `or`, `xor`, `compl`, `bitand`, `and_eq`, `or_eq`, `xor_eq`, `not`, `not_eq`**

Números enteros

- El bit más significativo define el signo (0 positivo, 1 negativo) y el resto la magnitud (que está en Complemento-a-2 si el número es negativo)
- **`short`** ±32000, **`int`** ±2000000000, **`long`** $\pm 9.\cdot 10^{18}$
- Todos los tipos enteros primitivos tienen riesgo de desbordamiento: operar (+, -, *) con números positivos y obtener uno negativo o viceversa. Con la multiplicación es muy complicado detectarlo
- <u>Consejo</u>: prescindir de los números sin signo (unsigned) y nunca mezclarlos en expresiones con números con signo. Si lo hace, reflexione y asegúrese de que es correcto y que no van a generar problemas

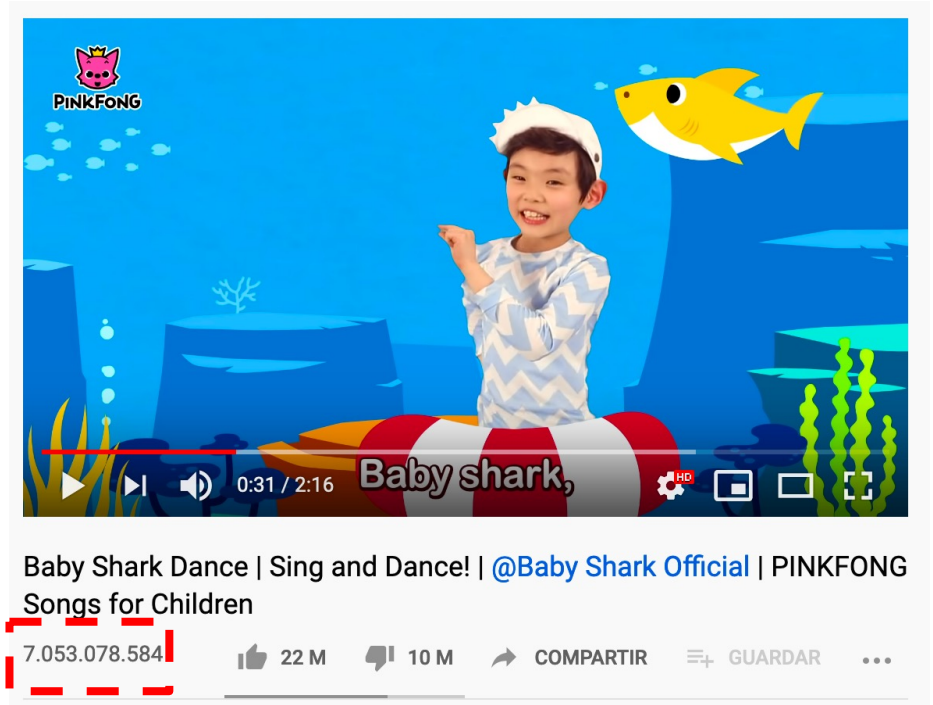


6 + 2 = -8??
0110 + 0010 = 1000

Desbordamiento contador Youtube

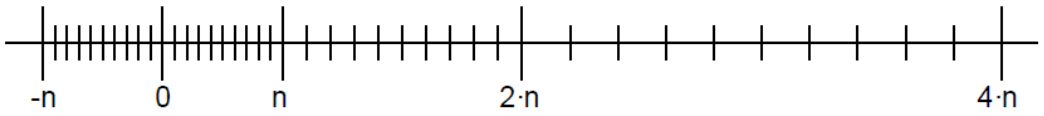
2014

2021



Números reales: IEEE 754

- Hay un problema: matemáticamente, entre cada dos números reales hay infinitos números intermedios, pero el ordenador no tiene memoria infinita para representarlos ni gestionarlos
- Solución: números con precisión finita y fija, de forma que el espacio entre dos números representables consecutivos no es uniforme y depende del exponente (longitud del número)



- La cantidad de opciones para representar números es 2^{32} (**`float`**), 2^{64} (**`double`**), 2^{80} (**`long double`**)

	lowest()	min()	max()	epsilon()
`float`	-3.40282e+38	1.17549e-38	3.40282e+38	1.19209e-07 (≈6 dígitos de precisión)
`double`	-1.79769e+308	2.22507e-308	1.79769e+308	2.22045e-16 (≈15 dígitos de precisión)

```
Inicializados a 1234567890.123456789
float: 1234567936, float+1: 1234567936
double: 1234567890.12345671653747558593 75, double+1: 1234567891.12345671653747558593 75
```

Caracteres: tabla ASCII

https://unicode-table.com/es/alphabets/
https://unicode-table.com/es/sets/

Caracteres de control ASCII

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

Caracteres ASCII imprimibles

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

elCodigoASCII.com.ar

ASCII extendido

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

(1) Variables

Obligatorio
Opcional

Variable: *nombre simbólico* que denomina a una información cuyo valor puede cambiar

Toda variable debe ser *declarada* antes de ser utilizada

```
modificador tipo identificador = expresión;
```

modificadores: **auto, extern, static, const, signed, unsigned, volatile, register**

```
int x= 4;
char c_99 = 'r';        //literal char
unsigned long tmp = 8883337; //literal entero
float a = 4.44433e-9;    //literal real
char valorHex = 0xa5;    //literal hexadecimal
```

V = <N, T, R, K>

- **N** ≡ identificador de la variable (usuario)
- **T** ≡ tipo de la variable (usuario)
- **R** ≡ dirección de memoria (compilador)
- **K** ≡ valor actual de la variable (usuario)

	Celdas de memoria	Direcciones de memoria
	0001 0100	…
x	**0000 0000**	**0xA3B2**
	0101 0110	0xA3B3
	1111 1111	0xA3B4
	…	0xA3B5
		…

Tipos en C++: primitivos (números enteros o reales), puntero/referencia, vector, enumerado, estructura/clase

Modificadores en declaración de variables

- Modificadores de tipo: **`signed, unsigned, long, short, const, volatile`**
- Modificadores de almacenamiento: ~~**`register`**~~**`, static, extern, mutable`**
- **`static`**: indica que el tiempo de vida de una variable es el programa (si se utiliza en una variable local), o que la variable (global) solo es visible para otras funciones dentro del mismo fichero. Su valor se fija en tiempo de compilación
- **`const`**: indica que el valor almacenado en la variable es constante, y por tanto no se puede modificar una vez asignado por primera vez (a la vez que la variable es declarada). Muy utilizada en los parámetros de funciones
- **`volatile`**: indica al compilador que la variable no debe ser optimizada y que todos los valores deben verse reflejados en memoria. Utilizada en acceso directo al HW
- Algunos modificadores se utilizan también en los parámetros de las funciones y en los atributos de las clases (**`mutable`**)

Literales de tipos primitivos

```
#include <iostream>

using namespace std; // evito tener que poner el std::
int main() {
  cout << true << " | " << std::boolalpha << true << endl;
  int d = 42;
  int o = 052; // octal
  int x = 0x2a, X = 0X2A; // hexadecimal
  cout << x << " | " << std::hex << X << '\n';
  int b = 0b101010; // binario
  char c = 'a'; // caracter
  long entero = 1'090'345; // con separadores
  cout << std::dec << entero << '\n';
  double real = 1'234.56e-3; // notación científica y separadores
  cout << real << '\n';
}
```

```
1 | true
42 | 2a
1090345
1.23456
```

Punteros y referencias

- Una variable **tipo puntero o tipo referencia** almacena la dirección de memoria de otra variable. Es posible declararlas para cualquier tipo de dato, incluso los definidos por el usuario

```
long a, b, *c; // solo la vble. c es un puntero
```

- El tamaño de un puntero depende de la arquitectura de la máquina, normalmente 32 o 64 bits (siempre entero sin signo)
- El literal `nullptr` indica puntero nulo (*dangling pointer*). Soluciona errores que había en C. En expresiones lógicas se evalúa a `false`
- Los tipos **referencia** son realmente punteros enmascarados (*puntero constante a variable*), como si fuera un *alias*. Se utilizan principalmente para pasar valores por referencia a funciones y despreocuparse de la farragosa notación de punteros

```
long &d=a; // d es una referencia, inicialización obligatoria
```

Punteros y memoria

	Dirección memoria	Valor memoria
`long x = 175;`	0x77FFAB	175
`long *px;`	0x7800AC	-
`px = &x;`	0x77FFAB	175
& es operador *dirección-de*	0x7800AC	0x77FFAB
`*px = 77;`	0x77FFAB	77
* es operador *indirección*	0x7800AC	0x77FFAB

```
int main(int argc, const char *argv[])
```

```
C:\> miPrograma.exe hola tiempo real
```

*argv[] →

argc	argv[argc]
0	“miPrograma”
1	“hola”
2	“tiempo”
3	“real”

Cuidado con el error clásico: la declaración de un puntero solo reserva memoria para el puntero. Se tiene que tener memoria también para almacenar el valor al que se quiere apuntar (la variable apuntada)

Conversiones de tipos

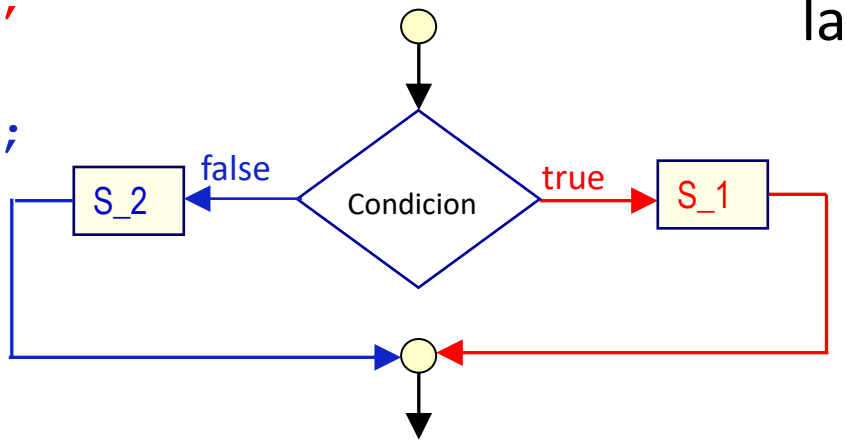
- El tipado fuerte es una característica importante, pero a veces es deseable relajarla cuando se opera con los tipos numéricos primitivos
- C++ define dos tipos de conversiones entre tipos de dato:
  - **Implícitas**: realizadas automáticamente en determinadas situaciones:

    `short → int → long → float → double → long double`
  - **Explícitas**: realizadas por el programador. Se supone que en programa bien diseñado apenas hay que utilizarlos. Siempre con cuidado:
    - `dynamic_cast <new_type> (expression)`, en OO permite *upcasting* y *downcasting*
    - `static_cast <new_type> (expression)`, la conversión clásica de C
    - `const_cast <new_type> (expression)`, elimina cv-qualification (**const** o **volatile**). Evitar salvo que se esté manejando un API que gestiona cv-qualification incorrectamente (muy raro)
    - `reinterpret_cast <new_type> (expression)`, reinterpreta la secuencia de bits subyacente. Evitar salvo en casos muy, muy particulares (que no se van a mostrar)

(4) Sentencias de selección: **if/else**

```
if (condicion){
   sentencia/s_1;
} else {
   sentencia/s_2;
}
```

Obligatorio
Opcional



Operadores de comparación
`a == b`
`a != b`
`a < b`
`a > b`
`a <= b`
`a >= b`

Operadores lógicos
`!a`
`not a`
`a && b`
`a and b`
`a || b`
`a or b`

En C++, el literal **0**, **false** y **nullptr** se evalúan a **false**. Cualquier otro valor es **true**

Los **if/else** anidados facilitan la escritura y lectura del código:

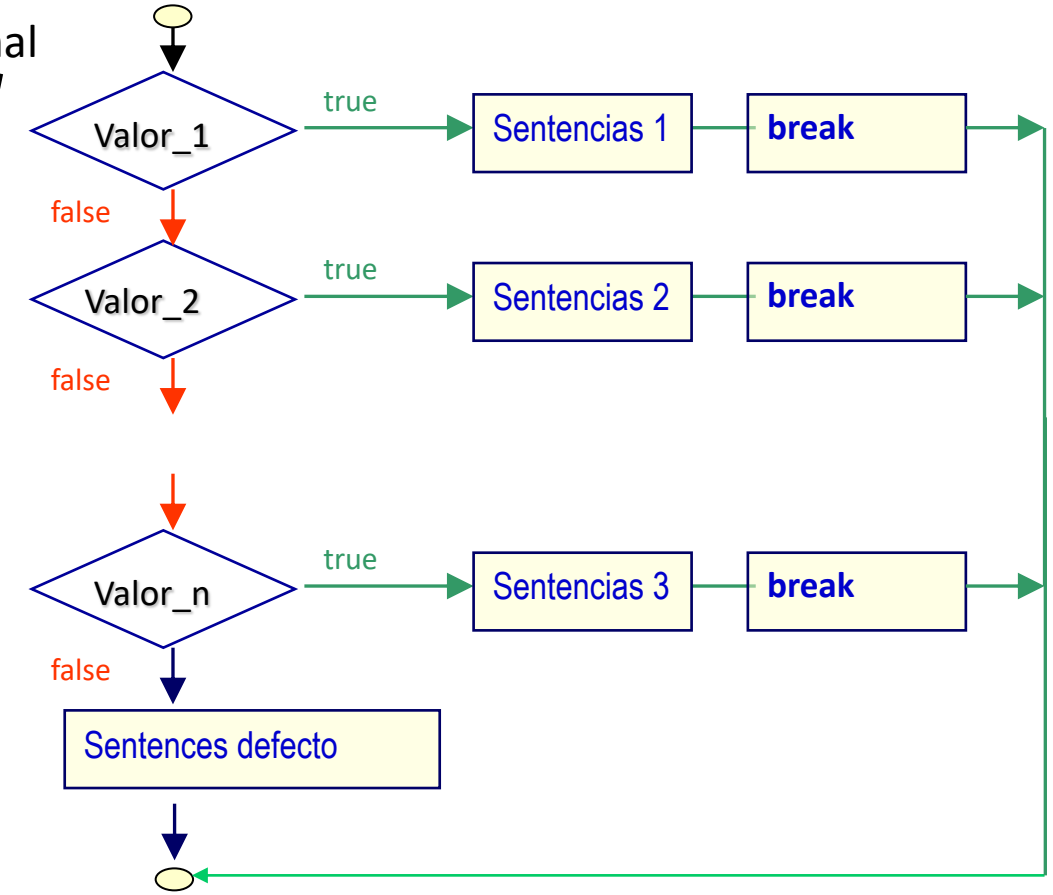
```
if (nota >= 9){
   cout << "Sobresaliente";
} else if (nota >= 7){
   cout << "Notable" ;
} else if (nota >= 5){
   cout << "Aprobado";
} else {
   cout << "Suspenso";
}
```

Solo cuando las condiciones son mutuamente excluyentes. Es decir, cuando hay una única opción posible

(4) Sentencias de selección: **switch**

```
switch (variable) {
  case valor_1: [sentencias_1;]
          [break;]
  case valor_2: [sentencias_2;]
          [break;]
   ...
  case valor_n: [sentencias_n;]
          [break;]
  [default: sentences_defecto;]
}
```

Opcional



Si no escribe un '**break**' al final de cada '**case**', la ejecución "atraviesa" los '**case**' que quedan en el '**switch**' hasta alcanzar un '**break**' o el final del '**switch**'

¡Mal hábito y mala práctica de programación!

(4) Sentencias de selección con inicializador

- C++ incluye la opción de declarar e inicializar una variable en la estructura de selección, limitando su existencia a dicha sentencia:

```
if (init; expresion) {
   ...
}
```

```
switch (init; variable) {
   ...
}
```

- Facilita el control del ámbito y tiempo de vida de variables que se van a utilizar solo en el contexto de la sentencia de selección

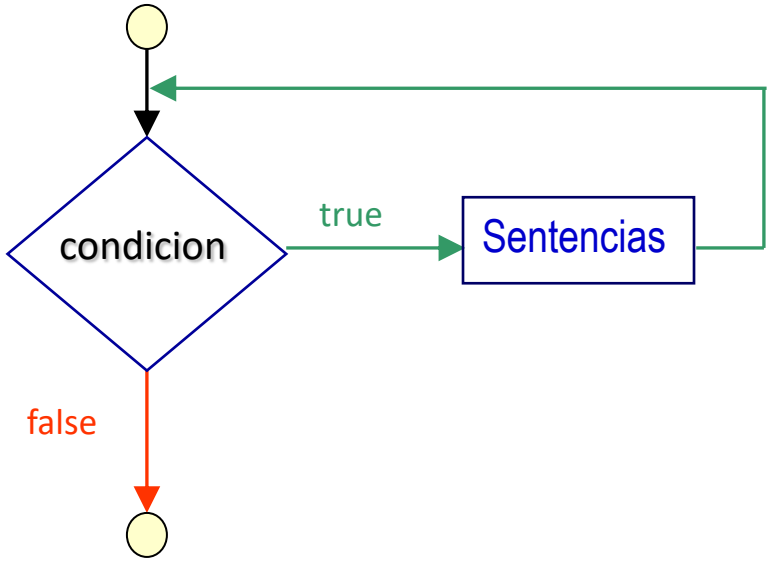
```
double num;
if (int tam = XXX(); tam < 100) {
   ...
}
// 'tam' ya no existe aquí, pero sí 'num'
```

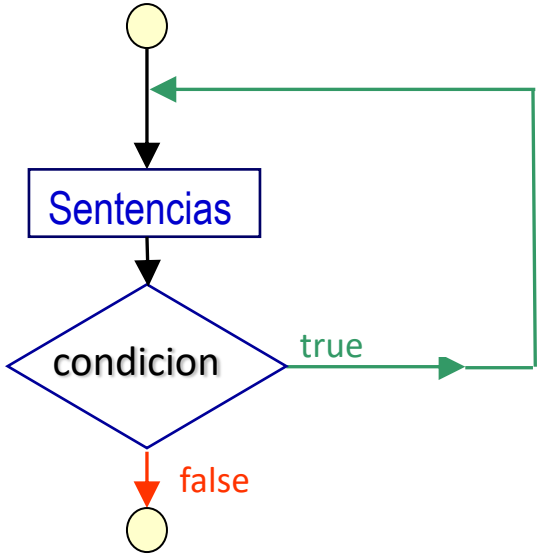
(4) Bucles: **while, do/while,** bucle general

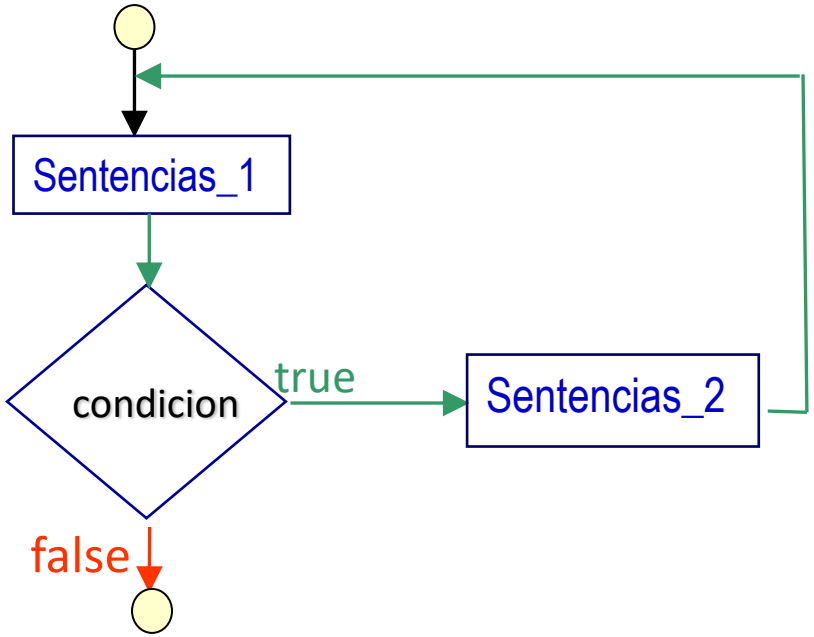
```
while (condicion) {
   sentencias;
}
```

```
do{
   sentencias;
} while (condicion);
```

```
while (1) {
   sentencias_1;
   if (!condicion)
    { break; }
   sentencias_2;
}
```



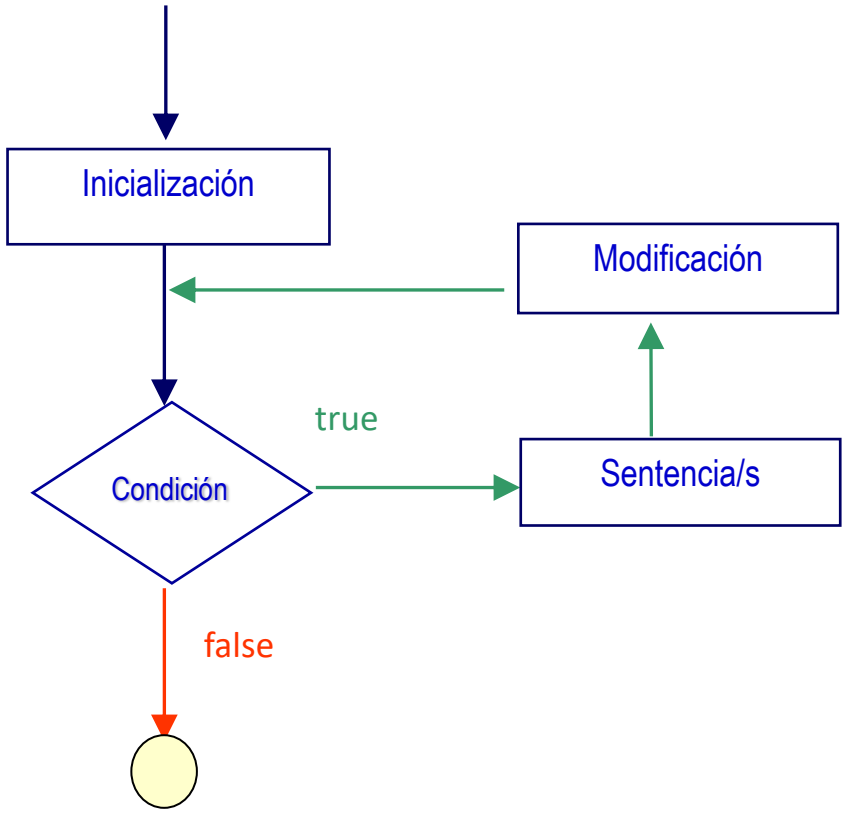




(4) Bucle controlado por contador: **for**

Obligatorio
Opcional

```
for ( inicialización_contadores ;
      condicion ;
      modificación_contadores ) {
          sentencia/s ;
}
```



- Se pueden declarar la/s variable/s contador/es directamente en el bucle '**for**':

```
for (int i=0; i<10; ++i)
```

- Los tres campos son opcionales, pero para mejorar la inteligibilidad del código es mejor utilizarlos todos
- La ejecución de *modificación_contadores* es automática al final de cada ciclo → el programados no debe modificarlo en el código del bucle

Funciones o sub-rutinas (I)

- Facilitan la reutilización de código y ayudan a organizar los programas. Proporcionan una vista de "caja negra": *sé lo que hace el código pero no cómo lo hace*

```
tipo_dato_retorno id_funcion (lista_parametros) modificadores { ... }
```

obligatorio

opcional

- Existe el tipo de retorno especial **`void`**, que significa que la función no retorna valor
- Puede tener parámetros con valores por defecto, pero deben aparecer al final:

```
void incrementar (int &var, const int valor=1) { }
```

- Cuando se implementa una función, se asume que los parámetros tienen valor (aunque no sepamos cuál): los proporcionados por la función invocante
- Las variables declaradas en una función son locales (salvo que se utilice **`static`**) y se eliminan cuando la función termina

Funciones o sub-rutinas (II)

- Una función termina cuando retorna valor (**`return`**) o lanza excepción (**`throw`**)
- El tipo del valor retornado con **`return`** debe coincidir con `tipo_dato_retorno`
- Si una función llega al '**}**' que la define, ejecuta un **`return`** sin valor. Solo para funciones **`void`**

- Una función se invoca escribiendo su nombre (identificador) seguido de un conjunto de valores separados por comas, uno por cada parámetro de la función, entre paréntesis. Si la función no tiene parámetros, los paréntesis se dejan vacíos
- La función invocante puede ignorar el valor de retorno, almacenarlo en una variable, o utilizarlo como parte de una expresión

- *Sobrecarga* (*overloading*): varias funciones pueden tener el mismo nombre siempre que la lista de parámetros sea diferente (no cuenta el valor de retorno)

Paso por valor/copia y por referencia

- C/C++ define dos mecanismos para pasar valores a/desde una función:
  - **Paso por valor**: se realiza una copia de la memoria donde está almacenado el valor. Las modificaciones se hacen sobre la copia, no sobre el original
  - **Paso por referencia**: se envía la dirección de memoria donde se encuentra la variable, es más eficiente. Las modificaciones afectan a la variable original, salvo que se utilice una referencia constante, en cuyo caso el compilador genera un error al intentar modificarla
- Por defecto, referencias y punteros (y "arrays de C") se pasan y retornan *por referencia*; el resto de tipos de dato se pasan *por valor*
- **NUNCA** se debe retornar de una función una referencia a una variable que no se haya creado dinámicamente, ya que el original se eliminará al terminar la función

Código: variables, referencias y punteros

```
void intercambio1 (int a, int b) {
  int tmp = a;  a = b;    b = tmp;
}
void intercambio2 (int &a, int &b) {
  int tmp = a;  a = b;    b = tmp;
}
void intercambio3 (int *a, int *b) {
  int tmp = *a;  *a = *b;  *b = tmp;
}
int main() {
  int a=1, b=2;  intercambio1(a, b);
  cout << "I1: " << a << "|" << b << endl;
  a=1;   b=2;  intercambio2(a, b);
  cout << "I2: " << a << "|" << b << endl;
  a=1;   b=2;  intercambio3(&a, &b);
  cout << "I3: " << a << "|" << b << endl;
}
```

```
I1: 1|2
I2: 2|1
I3: 2|1
```

```
int incrementar1 (int a) { return ++a; }
void incrementar2 (int &a) { ++a; }
int& max (int &a, int &b) {
  cout << "Funcion MAX\n";
  if(a>b) return a;
  else return b;
}
int main() {
  int a=1, b=2;
  a=1;  incrementar1(a); cout << a << endl;
  a=1; a=incrementar1(a); cout << a << endl;
  a=1;  incrementar2(a); cout << a << endl;
  a=1;
  int c = max (a,b);
  cout << "MAX: " << c << endl;
  max(a,b) = 1000; // ¿?
  cout << a << "|" << b << endl;
}
```

La función retorna una referencia, y las referencias pueden utilizarse a la izda. de la asignación

```
1
2
2
Funcion MAX
MAX: 2
Funcion MAX
1|1000
```

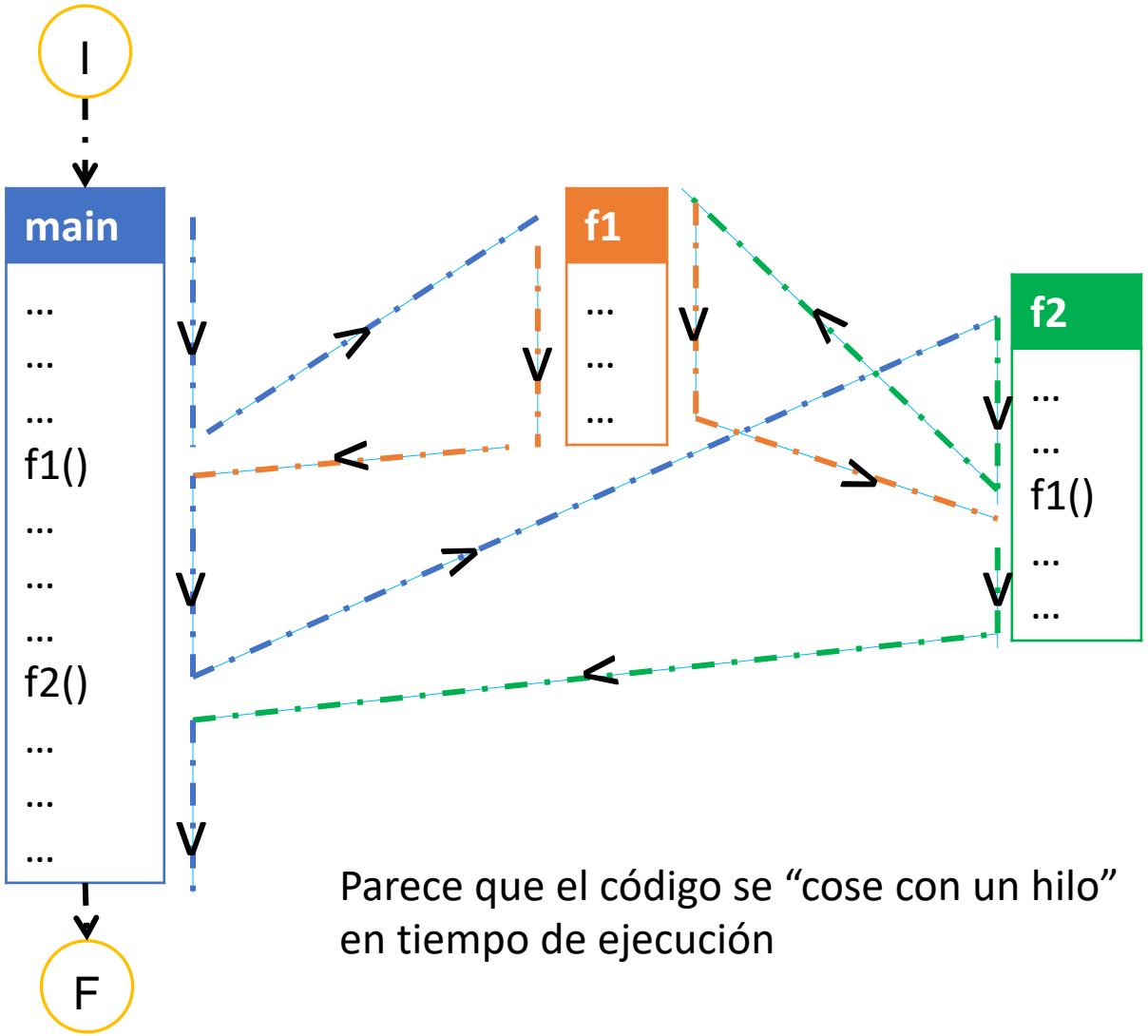
Ámbito (*scope*) de un identificador

- **Ámbito** es la porción de código fuente, posiblemente discontinua, donde un identificador (variable, función, espacio de nombres, tipo de dato o plantilla) es utilizable. Siempre delimitado por **{ }**. Es una propiedad estática del identificador (no cambia al ejecutarse el código)
- Los identificadores definidos dentro de un mismo ámbito pueden ser utilizados directamente, siempre que hayan sido declarados previamente al punto en que se quieren utilizar (salvo ámbito **mixto**)
- Los identificadores definidos en distintos ámbitos tienen la visibilidad que describe la siguiente tabla:

Público	**Privado**	**Mixto**
• Global • Namespace • Enumeration	• Function parameter • Template • Block (*"clásico" y básico; si no es ningún otro de la tabla*)	• Class

Flujo de ejecución de un programa

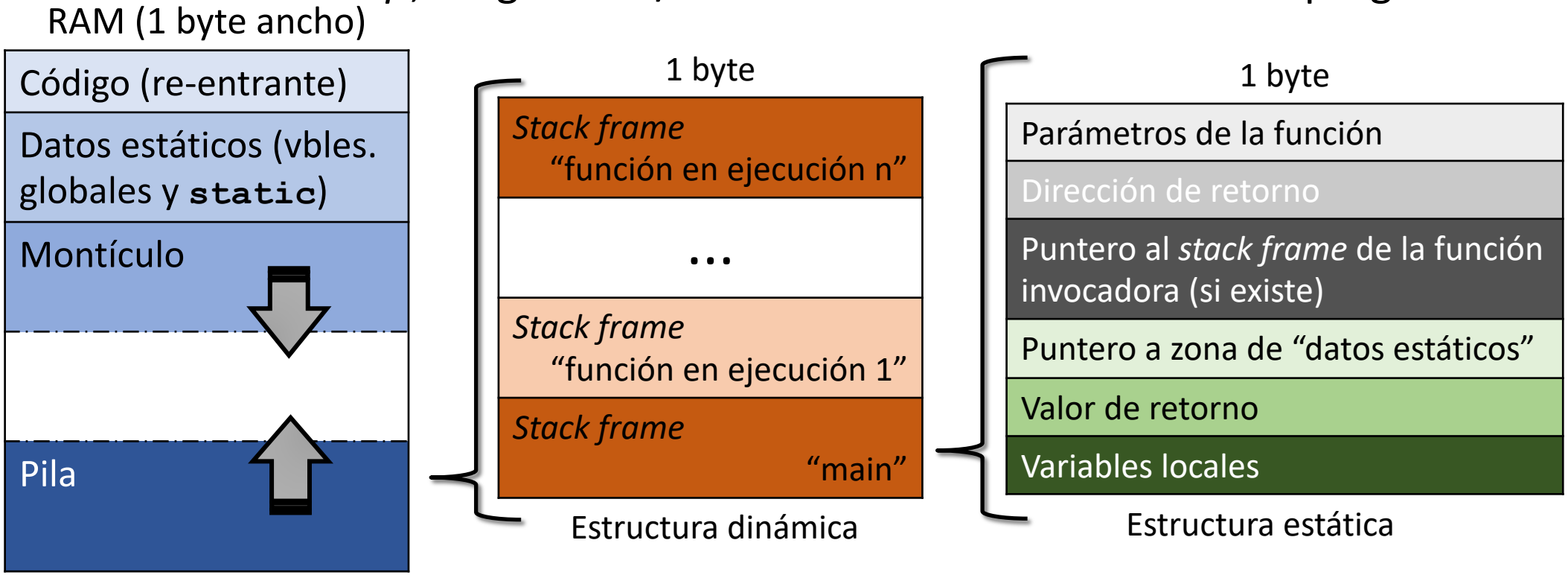
- El flujo de ejecución representa la ejecución en el tiempo de las sentencias de un programa
- Comienza en la función ‘main’
- Cada vez que se invoca una función, el flujo de ejecución salta a la posición de memoria donde está almacenada, interrumpiendo momentáneamente la ejecución de la función actual
- Cuando la función invocada termina, el flujo de ejecución retorna al punto original donde realizó el salto y continua la ejecución por donde se quedó



Parece que el código se “cose con un hilo” en tiempo de ejecución

Memoria de un programa en ejecución

- Cada programa en ejecución (proceso) tiene asignado un *run-time storage* (RAM)
  - **Pila** (*stack*): almacena en un *stack frame* la información de las funciones en ejecución
  - **Montículo** (*heap*): bloque de memoria que almacena variables creadas de forma dinámica. Gestionado por el programador. C (*malloc()* y *free()* <stdlib.h>), C++ (**new** y **delete**)
- Las variables locales se guardan en pila; el programador gestiona el tiempo de vida de las variables del *heap*; las globales/**static** sobreviven hasta el fin del programa



Pila y montículo (I)

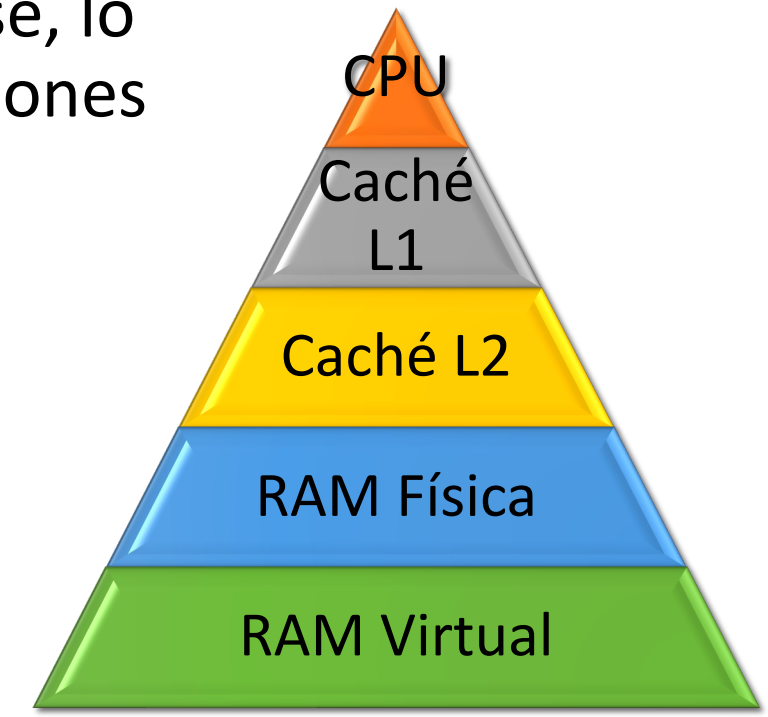
- El funcionamiento (la electrónica) de la CPU está muy ligado al uso de estructuras tipo pila (LIFO, acceso secuencial) → *Stack Pointer*
- El concepto "función", que dirige la ejecución de un programa, se implementa <u>eficientemente</u> con una pila (el rendimiento importa)
- Pero la pila tiene un problema: liga el tiempo de vida (*tv*) de las variables declaradas en una función al *tv* de la propia función → obliga a copiar los valores que necesitan un *tv* mayor → muy ineficiente
- Para mitigarlo, se diseñó el montículo: un espacio de memoria en el que el programador puede gestionar el *tv* de las variables

Pila y montículo (II)

- Hay muchos paradigmas de programación, pero el HW de ejecución funciona como funciona → C/C++ se ajusta muy bien
- El espacio de memoria asignado a la pila se reparte entre todas las funciones en ejecución, pero el montículo es común para todas ellas
- El montículo almacena solo variables, mientras que la pila dedica espacio a almacenar información que necesita la CPU para operar
- El acceso a las variables almacenadas en la pila es directo, pero al montículo se accede solo a través de punteros; además, la pila suele estar en caché → el acceso a la pila suele ser “doblemente” rápido

Pila y montículo (III)

- Puesto que la pila está asociada a la infraestructura de ejecución de código en la CPU, el desbordamiento de pila (*stack overflow*) es un error potencialmente peligroso (el SO suele abortar el programa, pero a veces el error puede acarrear más consecuencias)
- No es tan problemático si se llena el montículo
- La memoria del montículo puede fragmentarse, lo que puede reducir rendimiento de las aplicaciones
- Y, para terminar de complicarlo todo, la arquitectura HW (fundamentalmente, la pirámide de memoria) influye muchísimo en la velocidad de ejecución de un programa



Consejos para diseño de funciones

- Consejos generales:
  - Una función debe realizar una y sólo una tarea, mostrando una alto grado de cohesión. Aumenta su reutilización
  - Salvo casos concretos, las funciones deben tener menos de 20 líneas de código
  - Las funciones deben exhibir un bajo grado de acoplamiento entre ellas (ser independientes). Aumenta su reutilización

Acoplamiento y cohesión en funciones (Aula Virtual)

- Separar la entrada de datos de su procesamiento y de la presentación de resultados (patrón *Modelo-Vista-Controlador*)
- Definir *contratos* (pre-condiciones, invariantes y pos-condiciones), así como políticas de gestión de errores y excepciones para todo el programa

Análisis de uso de recursos (I)

https://valgrind.org

- **Valgrind** es una librería de instrumentación para construir herramientas de análisis en tiempo de ejecución. Detecta fallos en la gestión de memoria y de concurrencia, y ayuda a perfilar el rendimiento (velocidad de ejecución) del SW

```
int* f() {
  int *a = new int; // heap
  *a=9;
  return a;
}
int main() {
  for (int i=0; i<100; ++i) {
    int *ptr = f();
    // hacer algo con ptr
  }
  return 0;
}
```

Ámbito de **ptr**. Se borra la variable, pero no la memoria a la que apunta

```
$ valgrind --tool=memcheck ./a.out
==525== Memcheck, a memory error detector
==525== HEAP SUMMARY:
==525==   in use at exit: 400 bytes in 100 blocks
==525==  total heap usage: 101 allocs, 1 frees,
               73,104 bytes allocated
==525== 400 bytes in 100 blocks are definitely lost
==525==  at 0x4835DEF: operator new(unsigned long)
==525==  by 0x401163: f() (in /a.out)
==525==  by 0x401191: main (in /a.out)
==525== LEAK SUMMARY:
==525==  definitely lost: 400 bytes in 100 blocks
==525==  indirectly lost: 0 bytes in 0 blocks
==525==   possibly lost: 0 bytes in 0 blocks
==525==  still reachable: 0 bytes in 0 blocks
==525==     suppressed: 0 bytes in 0 block
```

Análisis de uso de recursos (II)

https://valgrind.org

```
int* f() {
  int *a = new int; // equivalente a malloc() en C
  *a=9;
  return a;
}
int main() {
  for (int i=0; i<100; ++i) {
    int *ptr = f();
    // hacer algo con ptr
    delete ptr; // "alguien" tiene que liberar la memoria cuando ptr ya
          //  no hace falta. "alguien" tiene que ser responsable
  }
  return 0;
}
```

Para vectores: **new[]** y **delete[]**

resource owner

```
root@edb6da6b31de:/# valgrind --tool=memcheck ./a.out
==533== Memcheck, a memory error detector
==533== HEAP SUMMARY:
==533==   in use at exit: 0 bytes in 0 blocks
==533==  total heap usage: 101 allocs, 101 frees, 73,104 bytes allocated
==533==
==533== All heap blocks were freed -- no leaks are possible
```

Espacio de nombres (`namespace`)

- C++ incluye espacio de nombres (**`namespace`**), que es una región que proporciona un ámbito a los identificadores (tipos de datos, funciones, variables, etc.) contenidos en él
- Se utilizan para organizar el código en grupos lógicos y evitar colisiones de nombres, que pueden ocurrir especialmente cuando se incluyen varias librerías (por ejemplo, si una librería incluye una función `print` y tu código también …)
- Para acceder a un identificador desde fuera del espacio de nombres se tiene que utilizar el nombre completo (`std::cout`, `std::string`), o bien utilizar **`using`** para un único identificador (**`using`** `std::string`) o para todo el espacio de nombres (**`using namespace`** `std`)
- Es posible anidar espacios de nombres en otros espacios de nombres
- `std` es el espacio reservado para la librería estándar

```
namepace upct {
  namespace tr {
   XXXX
  }
}
```

El modificador auto

- Desde C++11, **auto** activa la inferencia automática de tipos en el compilador, liberando al usuario de esta necesidad

```
auto var = 3.14; // el compilador infiere que var debe ser double
```

- El tipo de la variable **auto** se deduce en tiempo de compilación, no de ejecución, y dicho tipo se utiliza en el resto de código a compilar
- Solo se puede utilizar **auto** si la variable se inicializa en su declaración
- Se puede utilizar también en la declaración de funciones y en el *enlace estructurado* con estructuras y pares/tuplas
- Se puede utilizar como tipo de retorno de una función. **En esta asignatura, este uso está prohibido**
- Las nuevas versiones de C++ van aumentando los usos de **auto**

Tipos enumerados

- A veces, los valores de un tipo no son numéricos: colores, palos de una baraja, etc. → declarar un tipo enumerado, que mejora la legibilidad de un programa:

```
enum semaforo_t {verde, amarillo, rojo};
```

- Disponibles en C/C++. C++ proporciona además **enum class**, que fuerza el tipado fuerte en los valores
- En ambos casos, internamente, los valores enumerados se representan mediante números enteros, comenzando por cero, aunque es configurable:

```
enum class palos_t {oros=10, copas, bastos=100, espadas};
```

Tipo_dato identificador { valor_inicial }

```
semaforo_t sem1 {amarillo};
sem1 {semaforo_t::verde};
switch (sem1) {
  case amarillo:
    ...
}
sem = static_cast<semaforo_t> (0);
```

```
palos_t pl {palos_t::copas}; // obligatorio palos_t::
pl = oros; // error, requiere nombre completo
pl = static_cast<palos_t> (11); // con valor numérico
switch (pl) {
  case palos_t::copas:
    ...
}
```

Código: tipo enumerado

```
#include <iostream>
#include <string>

using namespace std;

enum class T_RGB { red, green, blue };

T_RGB leer() {
  string str;  // cadena de texto
  cin >> str;
  if (str == "RED")
    { return T_RGB::red; }
  else if (str == "GREEN")
    { return T_RGB::green; }
  else { return T_RGB::blue; }
  // falta algo ¿?
}
```

```
string to_string (const T_RGB val) {
  switch (val) {
    case T_RGB::red   : return "RED";
    case T_RGB::green : return "GREEN";
    case T_RGB::blue : return "BLUE";
    default: return "ERROR";
  }
}

int main() {
  T_RGB val = T_RGB::red;
  cout << to_string(val) << endl;
  cout << "Valor: ";
  val=leer();
  cout << to_string(val) << endl;
  return 0;
}
```

<cmath>: constantes y funciones

Symbol	Expression	Value
M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.30258509299404568402
M_PI	pi	3.14159265358979323846
M_PI_2	pi/2	1.57079632679489661923
M_PI_4	pi/4	0.785398163397448309616
M_1_PI	1/pi	0.318309886183790671538
M_2_PI	2/pi	0.636619772367581343076
M_2_SQRTPI	2/sqrt(pi)	1.12837916709551257390
M_SQRT2	sqrt(2)	1.41421356237309504880
M_SQRT1_2	1/sqrt(2)	0.707106781186547524401

- **Basic functions**: abs, fabs, fmod, remainder, remquo, fma, fmax, fmin, fdim, div
- **Exponential functions**: exp, exp2, expm1, log, log10, log2, log1p
- **Power functions**: pow, sqrt, cbrt, hypot
- **Trigonometric functions**: sin, cos, tan, asin, acos, ~~atan~~, atan2
- **Hyperbolic functions**: sinh, cosh, tanh, asinh, acosh, atanh
- **Error and gamma functions**: erf, erfc, tgamma, lgamma
- **Rounding functions**: ceil, floor, trunc, round, rint, nearbyint
- **Floating point functions**: frexp, ldexp, modf, scalbn, logb, ilogb, copysign
- **Classificacion and comparison functions**: isfinite, isinf, isnan, isnormal, isgreater, islower, signbit, isunordered
- La librería **<complex>** define funciones para números complejos