[Organización de Computadoras]

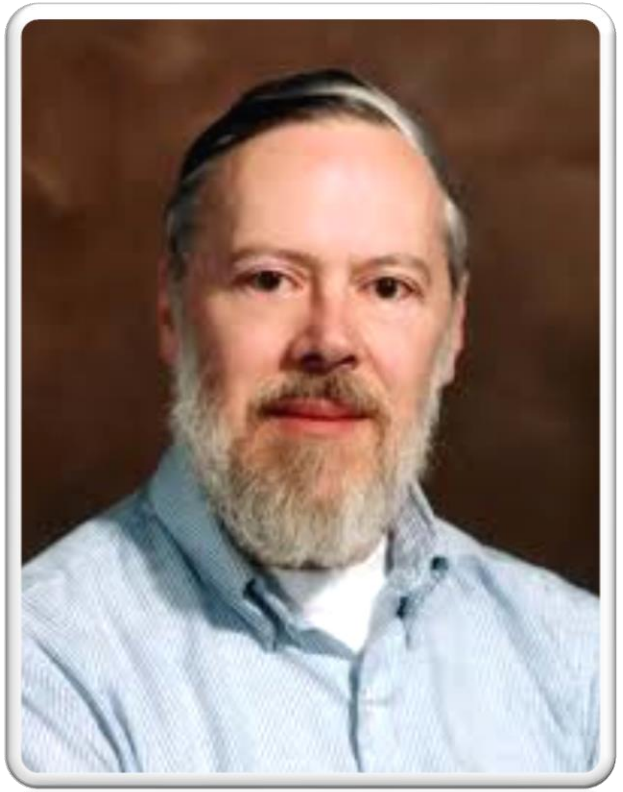
EL LENGUAJE DE PROGRAMACIÓN C (PARTE 1/2).

Copyright

Introducción

Lenguaje C :: origen.

- El **lenguaje C** fue diseñado por **Dennis Ritchie** en el año **1972**.
- Deriva de un lenguaje anterior llamado **B**.
- Se usó para **implementar** gran parte del sistema operativo **UNIX**.
- Hoy en día **se lo sigue utilizando** para implementar todo tipo de sistemas:
  - Por caso, el sistema operativo **GNU/Linux.**

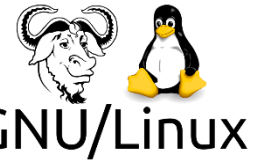


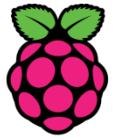
Lenguaje C :: filosofía de diseño

- C es un lenguaje de programación **minimalista.**
- Intenta satisfacer múltiples **objetivos**:
  - Poder ser **compilado** usando un compilador **sencillo** y **simple**.
  - Brindar acceso de **bajo nivel al hardware** de la computadora.
  - Poder **traducir** cada instrucción de C en pocas instrucciones de **lenguaje máquina**.
  - **No requerir** mucha ayuda adicional en tiempo de ejecución.

Lenguaje C :: algunos mitos

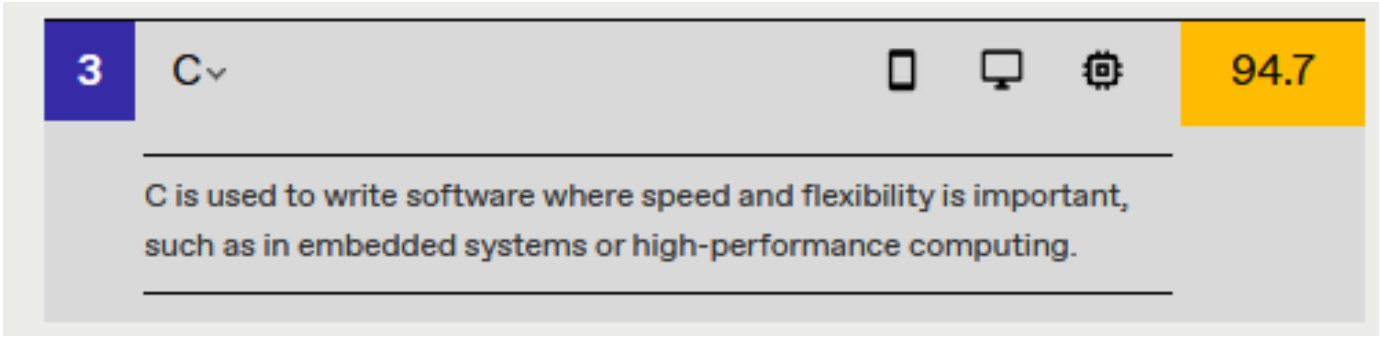
- *C no se usa. ¿Por qué programar en C?*
- El lenguaje ha sido usado para desarrollar:
  - Sistemas Operativos (casi **todos**, **GNU/Linux**, OS X, Windows, Android, **QNX**, Solaris)
  - Otros **lenguajes** como **Perl**, **PHP**, **Python** y **Ruby.**
  - **Java** Virtual Machine (si, está escrita en C).
  - Sistemas **Embebidos** (Arduino, Galileo, Raspberry).
  - **Controladores** de dispositivos (drivers).
  - Motores de **bases de datos** (MySQL, PostgreSQL, etc.).
  - Cualquier sistema de **alto** desempeño.





Lenguaje C :: algunos mitos

- *No hay trabajo en **C**. ¿Por qué programar en **C**?*
- Según Spectrum IEEE:



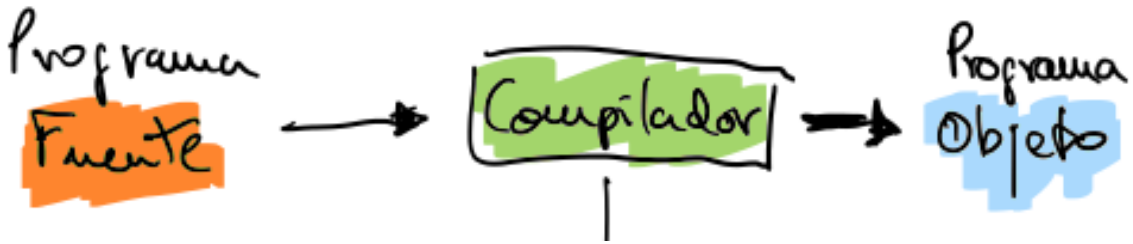
Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.4
3	C		94.7
4	C++		92.4
5	JavaScript		88.1
6	C#		82.4
7	R		81.7
8	Go		77.7
9	HTML		75.4
10	Swift		70.4

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

Lenguaje C :: características

- C es un lenguaje **compilado**.
- Cuenta con **excelentes compiladores optimizantes.**
  - Ejemplo de esto es el compilador **gcc.**
- El código escrito en este lenguaje es altamente **portable.**
- **Ninguna** característica del sistema operativo **se ve reflejada** en el lenguaje.



Lenguaje C :: características

- Elementos de un programa:
  - Palabras **reservadas** (muy pocas).
  - **Variables** y **funciones** definidas por el programador.
  - Funciones de librería **estándar.**
- Los bloques de código se **delimitan** entre **llaves.**
- Cada sentencia finaliza en **punto y coma**.
- Al igual que **UNIX**, el lenguaje C es **sensible** a **mayúsculas** y **minúsculas.**

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

```
24  int main(){
25      int i, id;
26      char name [51];
27
28      //CREA COUNT REGISTROS CON DATOS DEL USUARIO
29      for(i=0; i<count; i++){
30          printf("Introduzca un ID: ");
31          scanf("%d",&id); fflush(stdin);
32
33          printf("Introduzca un Nombre (max. 50 caracteres): ");
34          scanf("%50[^\n]", name); fflush(stdin);
35          data_set[i] = new_data(id, name);
36      }
```

Lenguaje C :: características

- El **código fuente** en **C** tiene tres tipos de **contenido**:
  - **Comentarios**.
  - **Código** de programa (escrito en C).
  - **Directivas** al preprocesador.

```
/* Clásico hola mundo escrito en C
para comprensión de los alumnos de OC
*/
#include <stdio.h>
int main() {
  // Una llamada a función
  printf("¡Hola mundo!\n");
  return 0;
}
```

Lenguaje C :: características

- El **código fuente** en **C** tiene tres tipos de **contenido**:
  - **Comentarios**.
  - **Código** de programa (escrito en C).
  - **Directivas** al preprocesador.

Comentarios de múltiple línea.

```
/* Clásico hola mundo escrito en C
para comprensión de los alumnos de OC
*/
#include <stdio.h>
int main() {
  // Una llamada a función
  printf("¡Hola mundo!\n");
  return 0;
}
```

Lenguaje C :: características

- El **código fuente** en **C** tiene tres tipos de **contenido**:
  - **Comentarios**.
  - **Código** de programa (escrito en C).
  - **Directivas** al preprocesador.

Comentarios de línea simple.

```
/* Clásico hola mundo escrito en C
para comprensión de los alumnos de OC
*/
#include <stdio.h>
int main() {
  // Una llamada a función
  printf("¡Hola mundo!\n");
  return 0;
}
```

Lenguaje C :: características

- El **código fuente** en **C** tiene tres tipos de **contenido**:
  - **Comentarios**.
  - **Código** de programa (escrito en C).
  - **Directivas** al preprocesador.

Código del programa.

```
/* Clásico hola mundo escrito en C
para comprensión de los alumnos de OC
*/
#include <stdio.h>
int main() {
  // Una llamada a función
  printf("¡Hola mundo!\n");
  return 0;
}
```

Lenguaje C :: características

- El **código fuente** en **C** tiene tres tipos de **contenido**:
  - **Comentarios**.
  - **Código** de programa (escrito en C).
  - **Directivas** al preprocesador.

Directiva al preprocesador: se solicita la incorporación de funcionalidad de la librería de entrada/salida estándar.

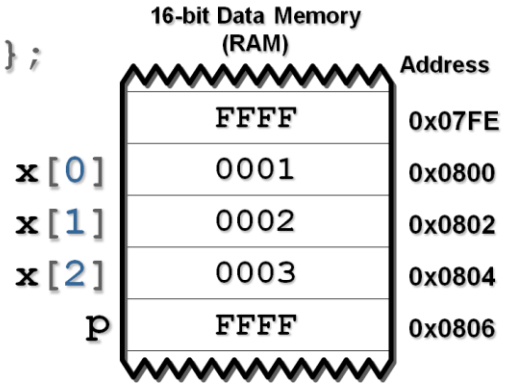
```
/* Clásico hola mundo escrito en C
para comprensión de los alumnos de OC
*/
#include <stdio.h>
int main() {
  // Una llamada a función
  printf("¡Hola mundo!\n");
  return 0;
}
```

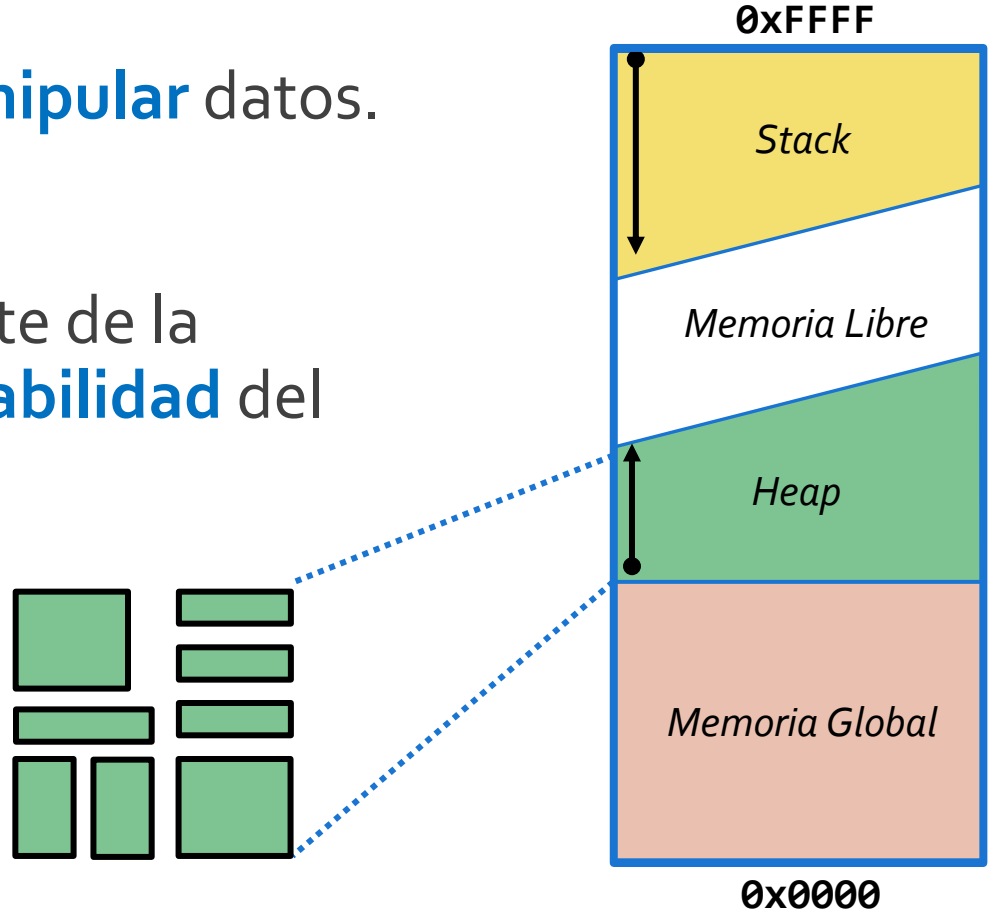
Lenguaje C :: características relevantes para OC

- En el **contexto de la materia**, haremos énfasis en dos cuestiones que resultan de interés.
- La **aritmética** de **punteros** y su uso para **manipular** datos.
- La **administración de memoria**.
  - A **diferencia** de otros lenguajes, en C parte de la **administración de memoria** es **responsabilidad** del programador.

```
int x[3] = {1,2,3};
int *p;

p = &x;
p++;
```





Variables & Expresiones.

DECLARACIÓN.

TIPOS DE DATOS Y MODIFICADORES.

USO Y OPERADORES.

Declaración de variables

- Las variables se declaran indicando su **tipo** seguido de su **nombre**:
  - `char letra;`
  - `int contador;`
- La declaración **múltiple** es posible separando las distintas variables con una coma:
  - `char letra, inicial;`
  - `int i, j, k;`

Inicialización de variables

- La **asignación** es una expresión y **puede** ser parte de una expresión mas compleja.
- El **operador** de **asignación** es el **=**
  - El **primer** elemento a **izquierda** **debe** ser el nombre de una **variable**.
  - Tiene asociatividad a **derecha**.
  - El **resultado** de la **evaluación de una asignación**, además del cambio en la variable, es el valor asignado a la variable.

Inicialización de variables

- **Declaración** e **inicialización** se pueden **combinar** usando el operador de asignación:
  - `char letra = 'A';`
  - `int i = 1, j = 10;`
- La asignación como operador da una **gran flexibilidad**, si bien es **a su vez fuente de problemas** (si no se utiliza adecuadamente):
  - `a = b = 2;`
  - `a = (b = 2) + 2;`

Tipos de datos elementales

- Los **tipos elementales** en **C** son los siguientes:
  - **Enteros** (**`int`**)
  - **Reales** en precisión **simple** (**`float`**)
  - **Reales** en precisión **doble** (**`double`**)
  - **Caracteres** y **enteros** de 1 byte (**`char`**)
  - **Punteros** (*)
- Nótese que **no existen** los **booleanos** ni las **cadenas de caracteres** como tipos elementales.

Modificadores de tipos de datos

- Los **tipos elementales** admiten distintos **modificadores**:
  - **`unsigned`**: para representar **sólo** valores **positivos** (no aplica a **`float`** y **`double`**)
  - **`signed`**: para representar valores **positivos** y **negativos** (activo por **defecto**).
  - **`long`**: para representar enteros **largos** (sólo puede aplicarse al tipo **`int`**).
- Ejemplos de aplicación de modificadores:
  - `unsigned int contador;`
  - `signed char letra;`
  - `long int balance;`

Alcance de una declaración

- En el lenguaje C **no es posible anidar** funciones.
- Así, existen **sólo dos alcances** posibles para una declaración:
  - **Alcance global:** la declaración es **visible** desde **todas** las funciones del programa.
  - **Alcance local:** la declaración **sólo es visible** dentro de la función en la que aparece y tiene **precedencia por sobre** las declaraciones globales.

```
int mi_funcion(){
    float otra_funcion(){
      ...
    }
}
```

```
unsigned int a, b;
char c;

int main(){
  float a, c, z;
  ...
}
```

Las variables a, b y c son tienen alcance global.

Las variables a, c y z tienen alcance local al bloque main.
Dentro de main las variables a y c refieren a float.
Fuera de main la variable z no existe.

Expresiones constantes

- Formulación de expresiones constantes:
  - Los reales (**`float`**) se pueden expresar tanto en **notación decimal** como en **notación científica.**
  - Los datos de doble precisión (**`long`**) se denotan agregando una **`L`** al final.
  - Los caracteres (**`char`**) se definen directamente usando comillas simples.
  - Las **cadenas de caracteres** se definen usando comillas dobles.

```
float a = 2,56
float b = 2,45E-4;
long int c = 200L;
char a = ‘a’;
char cadena [] = “hola”;
```

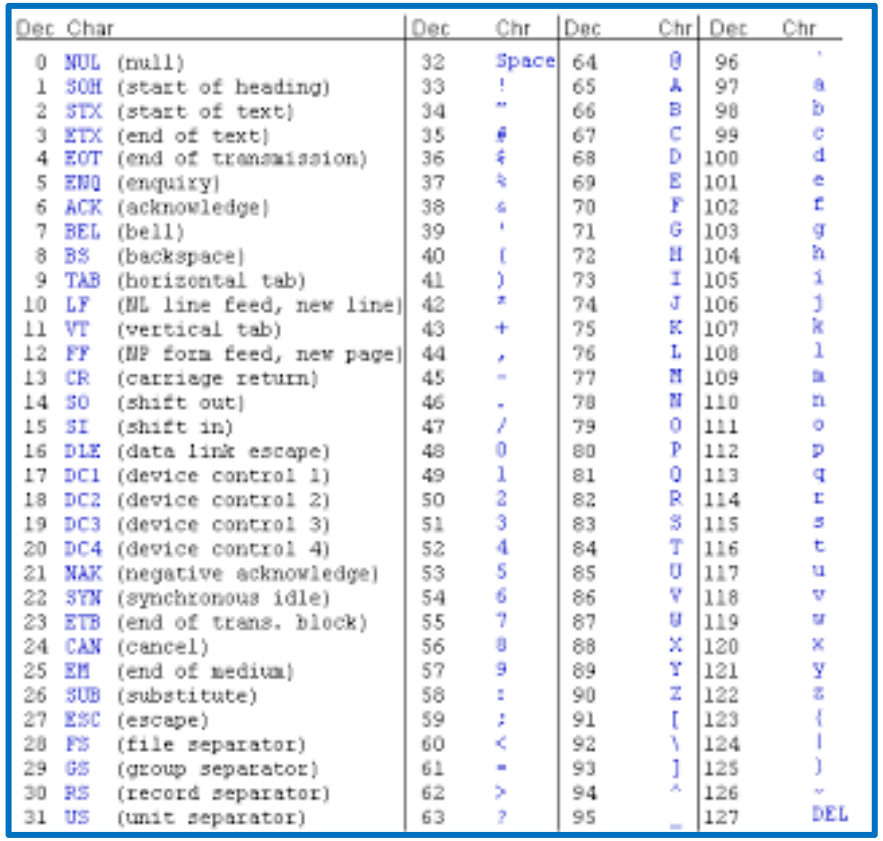
Expresiones constantes

- Una constante **entera** puede expresarse en diferentes bases:
  - **Decimal** (por defecto).
  - **Hexadecimal**, anteponiendo **0x** a la declaración.
  - **Octal**, anteponiendo **0** a la declaración.
  - **Binario**, anteponiendo **0b** a la declaración.

```
int a = 15
int a = 0xF;
int a = 017;
Int a = 0b1111;
```

Expresiones constantes

- Las constantes de tipo **carácter especial** se definen usando secuencias de **escape**:
  - **‘\n’**: línea nueva.
  - **‘\r’**: retorno de carro.
  - **‘\t’**: tabulador.
  - **‘\\’**: barra invertida.
  - **‘\0’**: carácter nulo.
  - **‘\nnn’**: carácter **ASCII** nnn (octal).
  - **‘\xnn’**: carácter **ASCII** nn (hexadecimal).



Dec	Char		Dec	Chr	Dec	Chr	Dec	Chr
0	NUL	(null)	32	Space	64	@	96	`
1	SOH	(start of heading)	33	!	65	A	97	a
2	STX	(start of text)	34	"	66	B	98	b
3	ETX	(end of text)	35	#	67	C	99	c
4	EOT	(end of transmission)	36	$	68	D	100	d
5	ENQ	(enquiry)	37	%	69	E	101	e
6	ACK	(acknowledge)	38	&	70	F	102	f
7	BEL	(bell)	39	'	71	G	103	g
8	BS	(backspace)	40	(	72	H	104	h
9	TAB	(horizontal tab)	41	)	73	I	105	i
10	LF	(NL line feed, new line)	42	*	74	J	106	j
11	VT	(vertical tab)	43	+	75	K	107	k
12	FF	(NP form feed, new page)	44	,	76	L	108	l
13	CR	(carriage return)	45	-	77	M	109	m
14	SO	(shift out)	46	.	78	N	110	n
15	SI	(shift in)	47	/	79	O	111	o
16	DLE	(data link escape)	48	0	80	P	112	p
17	DC1	(device control 1)	49	1	81	Q	113	q
18	DC2	(device control 2)	50	2	82	R	114	r
19	DC3	(device control 3)	51	3	83	S	115	s
20	DC4	(device control 4)	52	4	84	T	116	t
21	NAK	(negative acknowledge)	53	5	85	U	117	u
22	SYN	(synchronous idle)	54	6	86	V	118	v
23	ETB	(end of trans. block)	55	7	87	W	119	w
24	CAN	(cancel)	56	8	88	X	120	x
25	EM	(end of medium)	57	9	89	Y	121	y
26	SUB	(substitute)	58	:	90	Z	122	z
27	ESC	(escape)	59	;	91	[	123	{
28	FS	(file separator)	60	<	92	\	124	|
29	GS	(group separator)	61	=	93	]	125	}
30	RS	(record separator)	62	>	94	^	126	~
31	US	(unit separator)	63	?	95	_	127	DEL

Conversión explícita de tipos

- La **conversión explícita** de tipos (type **casting**) de expresiones y variables se denota de la siguiente forma:
  - `int a; float b; char c;`
  - `b = 65.0;`
  - `a = (int) b; // a ahora vale 65.`
  - `c = (char) a; // c ahora vale ‘A’ (esto es, el ASCII 65).`

Operadores aritméticos

- Los operadores **aritméticos** básicos disponibles son los usuales:
  - Suma (**+**) y Resta (**–**).
  - Multiplicación (*****) y División (**/**).
  - Módulo o resto (**%**).
- El **tipo del resultado** en una **división** depende del tipo de los operandos:
  - `4   / 3    da  1      (entero)`
  - `4.0 / 3    da  1.333  (real)`
  - `4   / 3.0  da  1.333  (real)`
  - `4.0 / 3.0  da  1.333  (real)`

Operadores de incremento y decremento

- Los operadores unarios (++) y (--) representan las operaciones de **incremento** y **decremento** respectivamente
  - **`x++;`** y **`++x;`** equivalen a la asignación **`x = x + 1;`**
  - **`y--;`** y **`--y;`** equivalen a la asignación **`y = y – 1;`**
- A partir de estos, las notaciones **sufijas** y **prefijas**:
  - `a=3;  b=a++;` **`// a vale 4, b vale 3`**
  - `a=3;  b=++a;` **`// a vale 4, b vale 4`**
- No cualquier cosa es válida:
  - `--b=a++;`

Operaciones a nivel bit

- El lenguaje C brinda un conjunto de operadores numéricos a **nivel de bit**:
  - Negación: **`~x;`**
  - Conjunción: **`x & y;`**
  - Disyunción: **`x | y;`**
  - Disyunción exclusiva: **`x ^ y;`**
  - Desplazamiento a izquierda: **`x << n;`**
  - Desplazamiento a derecha: **`x >> n;`**

```
int x = 26;
int y = 61;

// x          = 00011010
// y          = 00111101

// x & y      = 00011000
// x | y      = 00111111
// x ^ y      = 00100111
// ~x         = 11100101
// x>>2       = 00000110
```

Operadores relacionales

- Los operadores **relacionales** disponibles son:
  - Igual (**==**) y distinto (**!=**).
  - Mayor (**>**) y menor (**<**).
  - Mayor o igual (**>=**) y menor o igual (**<=**).
- El **resultado** de una comparación es un **entero**, considerando la semántica:
  - El **cero** (**0**) denotará el valor **falso.**
  - **Cualquier** otro valor denotará el valor **verdadero**.

*De aquí que el valor booleano **false** se note con un **entero igual a 0**, mientras que con cualquier otro valor entero se represente el valor booleano **true**.*

Operadores lógicos

- Sobre los datos de "tipo" **booleano** (enteros), se definen los siguientes operadores:
  - La **conjunción** lógica (**&&**)
  - La **disyunción** lógica (**||**)
  - La **negación** lógica (**!**)
- Para pensar: ¿a cuánto evalúa la siguiente expresión?
  - `res = (3 > 2 || 5 == 4) && !1;`

Sentencias de control

CONDICIONAL.

REPETICIÓN.

ALTERNATIVAS MÚLTIPLES.

Sentencia condicional

- La **sentencia condicional** tiene la siguiente sintaxis:

```
if (condicion) {

  // bloque then.

} else {

  // bloque else.

}
```

```
#include <stdio.h>
main() {
  int a = 2;
  if (a % 2){
    printf("dos es impar\n");
  }else{
    printf("dos es par\n");
  }
}
```

Sentencia condicional

- Las construcciones del siguiente tipo se pueden **abreviar** convenientemente con expresiones **condicionales**:

```
if (condicion) {

  res = exp-1;

} else {

  res = exp-2;
}
```

```
res =  condicion ? exp-1 : exp-2;
```

Sentencias de repetición

- La **sentencia de repetición desde** tiene la siguiente sintaxis:

```
for (cont = 0; cont < tope; cont++) {

  // sentencias a ejecutar en cada iteración

}
```

Sentencias de repetición

- La **sentencia de repetición `mientras`** tiene la siguiente sintaxis:

```
while(condicion) {

  // sentencias a ejecutar en cada iteración

}
```

Sentencias de repetición

- La **sentencia de repetición `repetir`** tiene la siguiente sintaxis:

```
do{

  // sentencias a ejecutar en cada iteración

} while(condicion);
```

Sentencias de control

- Al estar **dentro** del ámbito de una **sentencia** de **repetición**, se puede hacer uso de dos sentencias de **control** especiales:
  - **`break`**: esta sentencia permite **romper** una **repetición**, retomando la ejecución en la instrucción inmediata siguiente.
  - **`continue`**: esta sentencia permite indicar que la iteración actual **ya fue completada** y que se desea considerar la siguiente iteración.
- Independientemente de que el lenguaje ofrece este tipo de sentencias, desde la cátedra **desaconsejamos** su uso ya que modifican el comportamiento esperado de otras sentencias, **complejizando** la **lectura** y **mantenimiento** del código fuente.

Alternativas múltiples

- La **sentencia switch** tiene la siguiente sintaxis:

```
switch (alternativa) {
  case caso-1: { // bloque-1 }
  case caso-2: { // bloque-2 }
  ...
  case caso-n: { // bloque-n }
  default: { // bloque por defecto }
}
```

El tipo de dato evaluado como alternativa, debe ser un tipo de dato simple.
Por ejemplo, no se puede utilizar como alternativa un dato del tipo cadena de caracteres.

E/S Estándar

Entrada y salida estándar

- Las funciones de **entrada/salida estándar** en C **no pertenecen** al lenguaje en sí, sino que son provistas por diversas **funciones** de **librería**.
  - Recordemos que uno de los **objetivos** de diseño es lograr una **alta portabilidad**, por lo que el lenguaje debe ser **absolutamente independiente** del sistema operativo.
- Funciones de **entrada**:
  - `getchar(), scanf(),` etc.
- Funciones de **salida**:
  - `putchar(), printf(),` etc.

La función `printf()`

- La invocación a la función **`printf()`** guarda la siguiente sintaxis:

```
printf( formato-a-utilizar, exp-1, exp-2, ..., exp-n);
```

- El **formato a utilizar** es una **cadena** que describe **cómo** mostrar la información.
- Las **expresiones** son los **datos** que se desean mostrar siguiendo el formato.

```
int x = 3; float y = 23.0; char z = 'A';

printf("Hola mundo!!\n");

printf("x vale %d\n", x);

printf("y vale %f,\n...y z vale %c.\n", y, z);
```

Expresiones de formatos

- En la **E/S estándar**, se puede hacer uso de las siguientes expresiones de formato:
  - **%c**: carácter.
  - **%i**: entero decimal.
  - **%d**: entero decimal.
  - **%x**: entero hexadecimal.
  - **%f**: real.
  - **%p**: puntero.
  - **%s**: cadena de caracteres.
  - Entre **otras**.

```
int x = 3; float y = 23.0; char z = ‘A’;
printf(“Hola mundo!!\n”);
printf(“x vale %d\n”, x);
printf(“y vale %f,\n...y z vale %c.\n”, y, z);
```

Expresiones de formatos

- Existen **muchas** otras expresiones que controlan otros aspectos del formato:
  - La **precisión** (número de decimales)
  - La **justificación** (a izquierda o a derecha)
  - Poder escribir ciertos caracteres **especiales** (por ejemplo, el carácter %)
- Consultar la referencia de esta función en la bibliografía recomendada:
  - ***The C Programming Language***, escrito por B. Kernighan y D. Ritchie.

La función scanf()

- La invocación a la función **scanf()** guarda la siguiente sintaxis:

```
scanf( formato-a-utilizar, dest-1, dest-2, ..., dest-n);
```

- El **formato a utilizar** es una **cadena** que describe **cómo** obtener la información.
- Los **destinos** son las **variables** donde se almacenan los datos obtenidos.

```
int x; float y;
printf("Ingrese un número entero: ");
scanf("%d", &x); // ¡x se modifica!
printf("\nOtro entero y un real: ");
scanf("%u %f", &x, &y);
```

*En la sección donde se presentan la **definición de funciones** y el **pasaje de parámetros**, analizaremos qué es lo que sucede con el operador **&** que antecede las variables donde se almacenan los datos obtenidos por la función **scanf()**.*

Funciones

DECLARACIÓN VS. DEFINICIÓN.

PASAJE DE PARÁMETROS.

Definición de una función

- La definición de una **función** guarda la siguiente **estructura**:

```
tipo nombre (parametros) {

  // cuerpo de la función

}
```

```
int suma (int op1, int op2) {
  return op1 + op2;
}

void imprimir_error(int err){
  ...
}
```

- Las **funciones** admiten **llamadas recursivas**.
- Los **procedimientos**, si bien **no contemplados** en el lenguaje, se pueden definir indicando que una cierta función **no retorna resultado.**
  - Para esto se debe hacer uso del tipo especial **`void`.**

Llamada a funciones

- Una **función** es **llamada** dentro de una expresión, con sus **parámetros** definidos de forma **posicional**, separados por comas (**,**):
  - `c = suma(a, b);`
  - `imprimir_error(404);`
- Como las sentencias son expresiones **sin limitaciones**, las funciones pueden ser llamadas **descartando** el resultado.
  - `suma(a, b);`
  - En esencia, esto equivale a contar con un **procedimiento.**

Pasaje de parámetros

- Pasaje por **valor** o por **copia**:
  - Al pasar un parámetro **por valor** estamos pasando **una copia** del contenido, por lo que **las modificaciones** que haga la rutina **en esa copia no** se verán reflejadas en el **original.**
- Pasaje por **referencia**:
  - Al pasar un parámetro **por referencia** estamos mostrando **cómo acceder a ese argumento** pues estamos pasando la **dirección en memoria** del mismo. Naturalmente, **las modificaciones** que haga la rutina **se ven** inmediatamente reflejadas en el **original.**
- En C el pasaje de parámetros es **por valor**.

Pasaje de parámetros

- Si bien C **no cuenta** con pasaje de parámetros **por referencia**, cuenta con un **operador** que permite **simular** ese pasaje de parámetros
  - El operador **&** determina la **dirección en memoria** de la expresión a su derecha.

```
int a = 5;
printf("a vale %i\n", a);
printf("&a vale %p\n", &a); // // ¿por qué %i se cambia por %p?
```

Definición y uso de funciones: `factorial()`

```
int factorial(int n) {
  int res = 1;
  while (n > 1)
    res *= n--;
  return(res);
}

int main() {
  printf(“%d! = %d\n”, 5, fact(5));
}
```

Definición y uso de funciones: **swap()**

```
void swap(int * a, int * b) {
  int aux = *a;
  *a = *b;
  *b = aux;
}

int main() {
  int x = 5, y = 1;
  printf("(x,y) = (%d,%d)\n", x,y);
  swap(&x, &y);
  printf("(x,y) = (%d,%d)\n", x,y);
  return 0;
}
```

*¿Qué **operación** aplica el * sobre las variables **a** y **b**? Esto será discutido con más detalle en las clases de teoría, cuando se aborde el tópico **punteros**.*

*En esta sentencia, tanto **&x** como **&y** son **copiados** en los parámetros **formales a** y **b** de la función **swap**.*
*Como **&x** y **&y** generan las direcciones de memoria de **x** e **y**, entonces cuando en **swap** se modifique **a** o **b**, en realidad, se estarán modificando los valores **referenciados** por **x** e **y** en **main**.*
*De esta forma, se logra simular el pasaje de parámetros por **referencia**.*

Definición vs. declaración

- C **sólo** permite **invocar** a funciones **previamente** definidas:
  - Por ejemplo, no se puede invocar **`suma(a,b)`** si esta función no está definida con **anterioridad** a la sentencia de invocación.
  - ¿Cómo hacer para definir programas que hagan uso de una **recursión cruzada**?
- Es posible **separar** espacialmente la **declaración** de una función de su **definición**:
  - La **definición** explicita el código asociado a la función.
  - La **declaración** sólo explicita la **cabecera o prototipo** de la misma.

Definición vs. declaración

```
// prototipo de impar()
int impar(int n);

// definición de par()
int par(int n) {
  return !n ? 1 : impar(--n);
}

// definición de impar()
int impar(int n) {
  return !n ? 0 : par(--n);
}
```

Archivos

Gestión de archivos

- **C** comparte la filosofía **UNIX** en el sentido de considerar prácticamente todo como si fuera un **archivo**:
  - La **lectura** de información desde un **archivo convencional** **no difiere** del ingreso de datos a través del teclado.
  - La **escritura** de información hacia un **archivo convencional** **no difiere** de la salida de datos por pantalla.

Funciones de librería

- **C** cuenta con diversas **funciones** de librería sobre archivos:
  - `FILE* fopen(char*, char*)`: **apertura** de un archivo para lectura o escritura.
  - `int fclose(FILE*)`: **cierre** de un archivo.
  - `int fprintf(FILE*, ...)`: **escritura** con formato.
  - `int fscanf(FILE*, ...)`: **lectura** con formato.
  - `int feof(FILE*)`: determina si se ha alcanzado el **final** de un archivo.

Archivos especiales

- Todo programa asocia tres constantes de tipo **`FILE*`** por defecto a los siguientes archivos:
  - **`stdin`**: entrada estándar (descriptor 0)
  - **`stdout`**: salida estándar (descriptor 1).
  - **`stderr`**: error estándar (descriptor 2).
- Por caso:
  - `fprintf(1, “Usuario: ”); // Similar a printf(“Usuario”)`
  - `fscanf(0, “%s”, user); // Similar a scandf(“%s”, &user)`
  - `fprintf(stderr, “Error: No válido”);`

Buenas prácticas

Prácticas esperadas en la materia

- Definir las **constantes** como macros.
- **Macros** en **mayúsculas** separados por **guiones**.
  - `#define PI 3.1416`
- Identificadores de **variables** y **funciones significativos,** en **minúsculas** separadas por **guiones.**
  - `int contador, float promedio, char * cadena`
  - `int es_valido`
  - `void funcion_de_nombre_largo()`
- Recordar que **C** es **case sensitive** para sus identificadores. Usar **mayúsculas** es una práctica que **desaconsejamos** ya que presta rápidamente a errores.

Prácticas esperadas en la materia

- Largo de las líneas de código: un máximo de entre **79** y **100** caracteres.
- El código debe **compilar** sin **warnings**.
  - Habilitar el **`flag -wall`** en la IDE o en consola al momento de compilar.
- **No** realizar asignaciones dentro de expresiones:
  - `a = (b = 2) + 2;`
- **Documentar** internamente el código de forma **simple, clara** y **compacta**
  - Dirigida a **programadores**.
  - El comentario debe **sumar información**, no **generar nueva** que es **redundante** (por ejemplo, el siguiente comentario carece de todo sentido: `int i = 0 // Inicializa la variable i en cero`)

Prácticas esperadas en la materia

- Elegir una **convención** y **respetarla** a lo largo de todo el código:
  - Uso de **llaves** en los bloques **Ifs**.
  - **Ubicación** de las **llaves** para todo bloque de código (siempre a continuación de una sentencia, o siempre debajo de la misma).
  - Cantidad de **tabuladores** homogéneo para identar el código.

Fin de la presentación.