Apuntes de Programación de Software de Sistemas `CC3301`

Eduardo Reyes R

5 de mayo de 2022

INTRODUCCIÓN

El presente documento contiene apuntes del curso 3301-1 (Otoño 2022) que pretenden servir como documentación o guía a lectores que cursan o preparan el ramo.

En el capítulo primero (Clase 1) encontraremos la bibliografía sugerida (este semestre) para el curso en la sección 10 del mismo. Sin embargo, el curso posee material adicional antiguo. Se recomienda consultar material actualizado y que esté cubierto bajo el programa del curso pues se han removido y modificado contenidos con el cada semestre que se imparte CC3301.

Para consultar los apuntes del profesor Luis Mateu, el link es el siguiente: `https://wiki.dcc.uchile.cl/cc3301/temario` [1] (Mateu, 2021). En la misma página encontrará apuntes de José Piquer y del docente Patricio Poblete. [2]

[1] Un atajo a cada capítulo del apunte es visible en el Apéndice A
[2] Un atajo a sus apuntes está disponible en el Apéndice J y corresponden al año 2001 y 2002

ÍNDICE

I

CLASE 1: INTRODUCCIÓN/MOTIVACIÓN

1 Presentación

Estos son los apuntes no oficiales del curso CC3301-1 cursado en Otoño 2022 bajo las cátedras del docente Luis Mateu B. El material usado es directamente obtenido de cátedras, en adición a los materiales del curso. (Mateu, 2022). Apunte oficial en la sección 10 del capítulo.

2 Propósito del curso

El propósito del curso CC3301 Programación de software de sistemas (PSS) es que los/las estudiantes escriban y mejoren programas en lenguaje C, que requieren hacer un uso eficiente de la plataforma, y por lo tanto necesitan conocer su arquitectura de hardware y la interfaz de programación de aplicaciones (API) del sistema operativo.

3 Resultados de aprendizaje

En este curso aprenderán a:

- Escribir programas eficientes en el lenguaje C
- Usar el *debugger* **ddd** para diagnosticar errores
- Usar *sanitize* para detectar errores de programación como fugas de memoria o punteros locos
- Identificar a qué instrucciones assembler **Risc-V** se compilan los programas en C
- Escribir trozos de programas en assembler **Risc-V**
- Escribir programas que usan directamente los servicios provistos por *Linux* (el núcleo de sistemas operativos como *Debian*, *Ubuntu*, *Android*, etc.)
- Crear procesos paralelos para aprovechar todos los cores del computador y así disminuir el tiempo de ejecución

4 Acerca del profesor

Resumen bajo palabras de Mateu.

- Profesor de "jornada parcial" del DCC
- Solo me dedico a los 3 cursos que dicto para la facultad
- Los otros 2 cursos son arquitectura de computadores y sistemas operativos
- Trabajé 5 años en Synopsys: empresa con sede en California y centros de investigación y desarrollo en varios países, incluyendo uno en Chile
- Synopsys es líder en software para diseñar circuitos digitales, usado por Intel, AMD, nVidia, Qualcomm, etc.
- Trabajé en el mantenimiento de Design Compiler, un compilador de Verilog/Vhdl de unas 10 millones de líneas de código en C que corre principalmente bajo Linux

5 ¿Por qué aprender a programar en C?

- Es el lenguaje preferido cuando se necesita eficiencia, por ejemplo para programar un decodificador de video
- Existe una amplia base de software escrito en C y que necesita mejorarse:
  - ✓ Resolver bugs
  - ✓ Agregar funcionalidades
  - ✓ Portar a nuevas plataformas
  - ✓ Mejorar la eficiencia
- Ejemplos: los núcleos de Linux (C) y Windows (C y C++), el intérprete de Python, el compilador just-intime de Java, administradores de bases de datos, etc.
- Desventajas:
  - × No es robusto
  - × Es inseguro
- Futuro ficción: será reemplazado por Rust

6 ¿Por qué aprender assembler Risc-V?

- Porque es el assembler más fácil de aprender
- Hace más fácil aprender assemblers más complejos como x86 y Arm
- Se requiere programar en assembler para usar de la manera más eficiente las instrucciones especializadas de los procesadores como manejo de vectores
- Algunos bugs son tan complejos que requieren revisar el assembler generado por el compilador
- Risc-V se posiciona en términos de uso como la tercera plataforma después de Arm y x86
- Y no parará de crecer porque un fabricante de chips no necesita pagar una licencia por usar el set de instrucciones de Risc-V
- Hay diseños avanzados de Risc-V open source y gratuitos
- Diseñado para que los fabricantes puedan agregar instrucciones aceleradoras del cálculo de operaciones gráficas, de redes neuronales, de encriptación, etc.

7 ¿Por qué aprender un sistema operativo basado en Linux?

- Linux es una reimplementación del núcleo del sistema operativo del legado Unix
- Android, el sistema operativo de los celulares, usa un núcleo derivado de Linux
- iOS y OS X (de Apple) son derivados de Unix
- La mayoría de los servidores en Internet usan un sistema operativo basado en Linux: CentOS, RedHat o Debian
- Los notebooks y computadores de escritorio típicamente usan el sistema operativo Windows, pero WSL 2 de Microsoft también permite correr en Windows los sistemas operativos Debian, Ubuntu y otros basados en Linux, con todas sus aplicaciones

8 ¿Qué es la programación de software de sistemas?

De acuerdo a Wikipedia se trata de programar software:

- Que provee servicios a otro software
- Que requiere alto desempeño
- O ambos

Ejemplos: núcleos de sistemas operativos, compiladores, aplicaciones de ciencia computacional, motores de juegos, automatización industrial, etc.

Típicamente se programan en el lenguaje C

Lo opuesto es la programación de software de aplicaciones como un sistema de administración para una empresa, scripts para páginas web, un juego de salón, etc.

Finalmente: ¡es subjetivo!

9 Programa del curso

- Programación en el lenguaje C: tipos, sintaxis de las instrucciones, direcciones de memoria, punteros, errores típicos
- Arquitectura de computadores: Risc vs. Cisc, assembler Risc-V, compilación de programas en C a instrucciones Risc-V
- Implementación de la CPU: circuitos digitales, la memoria, diseño de la cpu, ejecución secuencial, en pipeline, superescalar y fuera de orden.
- Linux: estudio de la API de Linux/Unix (applicaction programming interface), el shell de comandos, cómo se crean los procesos y se cargan los programas, paralelización con múltiples procesos

10 Bibliografía

- Se publicarán en la sección novedades de U-cursos:
  - ✓ Videos de las clases de este semestre y/o de semestres anteriores
  - ✓ Pdf de las presentaciones
  - ✓ Material para probar los ejemplos de las clases
- En página Web: `https://users.dcc.uchile.cl/~lmateu/CC3301/`
  - ✓ Instrucciones para instalar la distribución oficial de Linux en este curso: Debian 11
  - ✓ Controles de semestres pasados
- Material complementario:
  - ✓ Apuntes del curso (actualización necesaria)
  - ✓ Kernighan, B y Ritchie, D (1988) "The C Programming Language"
  - ✓ Richard Stones, Neil Matthew (2003) "Beginning Linux Programming (Programmer to Programmer"

11 Evaluación

En página Web: `https://www.u-cursos.cl/ingenieria/2022/1/CC3301/1/datos_curso/`

12 Consejos y reglas importantes

- No intente resolver las tareas sin haber estudiado la materia correspondiente primero. Cometerá errores que no será capaz de entender y terminará perdiendo más tiempo que el que se ahorró al no estudiar.
- Si sobrepasa el tiempo nominal publicado para resolver la tarea, pida ayuda.
- Pedir ayuda no es copiar cuando el código lo escribió Ud. mismo.
- Copiar un fragmento de código de la tarea de un compañero sí es copia.
- Puede pedir ayuda a un compañero.
- O al profesor de cátedra o los auxiliares en alguno de los horarios de consultas o por correo
- Una tarea en C puede correr exitosamente en su computador y fallar completamente en plataformas ligeramente distintas. Su tarea será corregida bajo Debian 11. Asegúrese de que funciona en esa plataforma

13 Trabajo para la casa

- Instale el Linux oficial del curso (Debian 11): siga las instrucciones que aparecen en esta página en la sección cómo correr Debian en su computador.
- Estudie este tutorial que le enseñará los comandos básicos de Linux. Abra un terminal y experimente con estos comandos.
- Antes de la clase del jueves: estudie en los apuntes del curso la sección principios básicos del lenguaje C
  - Compile y ejecute todos los ejemplos dados
  - Complete el ejercicio final: factorial

II

CLASE 2

1 Esquema de la clase

- Enteros sin signo
- Enteros con signo
- Representación de enteros en base 2
- Representación de enteros negativos en complemento de 2
- Números reales y su representación
- Representación de caracteres en ASCII

2 Tipos de datos en C

- El sistema de tipos de un lenguaje incluye:
  - Tipos de datos primitivos
  - Expresiones y operadores
  - Reglas de inferencia para el tipo de una expresión
  - Mecanismos para definir nuevos tipos
- Tipos primitivos en C:
  - Enteros con signo: char, short int, int, long int, long long int
  - Abreviados: char, short, int, long, long long
  - Enteros sin signo: anteponer atributo unsigned, por ejemplo unsigned int
  - Reales: float, double
  - Punteros
- C no define un tipo especial para los valores de verdad o para strings (Java sí define los tipos boolean y String)

3 Enteros sin signo

- Los enteros sin signo se representan internamente en binario (base 2)
- Usaremos la notación (x)b para indicar que la constante x esta representada en base b. Si no se indica la base, se supone base 10
- Ejemplo: $(13)_{10} = (1101)_2 = 1 \cdot 2^3 + 1 \cdot 2^3 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 0 + 1 = 13$
- Aunque la representación interna es base 2, en el lenguaje las constantes se anotan en base 10
- ¡También se puede usar base 8 anteponiendo el prefijo 0! El 013 no es el mismo número que 13 porque está en octal: $013 = (13)_8 (011011)_2 = 8 + 2 + 1 = 11$
- Un cero a la izquierda sí vale en C. Es decir

  `int i= 010;` es equivalente a `int i= 8;`
- Un número en hexadecimal siempre comienza con el prefijo 0x y cada cifra representa 4 bits:

  $(2001)_{10} = (11111010001)_2 = (7d1)_{16} = 0x7d1$
- Se puede usar base 16 anteponiendo el prefijo 0x: $0x13 = (13)_{16} = (00010011)2 = 16 + 2 + 1 = 19$
- No existe 0b0110 para binarios, ¡pero lo usaremos en las explicaciones!

4 Conversiones

- Para convertir un número binario $x_{n+1}...x_0$ a base 10 hay que calcular $\sum_{i=0}^{n-1} x_i 2^i$
- Ejemplo: $(1100101)_2 = 1 \cdot 2^0 + 1 \cdot 2^2 + 1 \cdot 25 + 1^26 = 1 + 4 + 32 + 64 = 101$
- Para convertir un número en base 10 a base 2, dividir repetidamente por 2, anotando el resto de la división, hasta llegar a 0. El número en base 2 se obtiene leyendo los restos en orden inverso.
- Ejemplo: $25 = (?)_2$

división	resultado	resto
25 / 2	12	1
12 / 2	6	0
6 / 2	3	0
3 / 2	1	1
1 / 2	0	1

Figura 2.1: Ejemplo: $25 = (?)_2$

- Respuesta: $25 = (11001)_2$

5 Rango de representación de enteros sin signo

C ofrece enteros sin signo anteponiendo el atributo unsigned al tipo. La siguiente tabla describe los enteros sin signo. (Mateu, 2021)

tipo	Espacio en bytes Rango	Espacio en bytes Rango	Espacio en bytes Rango
	Máquinas de 64 bits	Máquinas de 32 bits	Máquinas de 16 bits
unsigned char	1	1	1
	[0, 2^8[≡ [0, 255]	[0, 2^8[	[0, 2^8[
unsigned short	2	2	2
	[0, 2^{16}[≡ [0, 65535]	[0, 2^{16}[	[0, 2^{16}[
unsigned int	4	4	**2**
	[0, 2^{32}[≡ [0, 4.294.967.295]	[0, 2^{32}[	**[0, 2^{16}[**
unsigned long	8	**4**	**4**
	[0, 2^{64}[	**[0, 2^{32}[**	**[0, 2^{32}[**
unsigned long long	8	8	8
	[0, 2^{64}[	[0, 2^{64}[	[0, 2^{64}[

Figura 2.2: Rango de representación de enteros sin signo

- Que una máquina sea de n bits significa que los punteros son de n bits y por lo tanto puede direccionar hasta 2^n bytes de memoria
- En Windows de 64 bits el tipo long es de 32 bits
- Las máquinas de 64 bits pueden correr los programas de las máquinas de 32 bits
- Muchos sistemas embebidos y dispositivos usan procesadores de 16 bits por razones de costo (mouse y teclado por ejemplo)
- Observe que si x e y son sin signo, x-y todavía se calcula como x+∼y+1.

6 Rango de representación de enteros con signo

tipo	Espacio en bytes Rango	Espacio en bytes Rango	Espacio en bytes Rango
	Máquinas de 64 bits	Máquinas de 32 bits	Máquinas de 16 bits
char	1	1	1
	$[-2^7, 2^7[\equiv [-128, 127]$	$[-2^7, 2^7[\equiv [-128, 127]$	$[-2^7, 2^7[\equiv [-128, 127]$
short	2	2	2
	$[-2^{15}, 2^{15}[\equiv$ [-32768, 32767]	$[-2^{15}, 2^{15}[\equiv$ [-32768, 32767]	$[-2^{15}, 2^{15}[\equiv$ [-32768, 32767]
int	4	4	2
	$[-2^{31}, 2^{31}[\equiv$ [-2.147.483.648, 2.147.483.647]	$[-2^{31}, 2^{31}[\equiv$ [-2.147.483.648, 2.147.483.647]	$[-2^{15}, 2^{15}[$
long	8	4	4
	$[-2^{63}, 2^{63}[$	$[-2^{31}, 2^{31}[$	$[-2^{31}, 2^{31}[$
long long	8	8	8
	$[-2^{63}, 2^{63}[$	$[-2^{63}, 2^{63}[$	$[-2^{63}, 2^{63}[$

Figura 2.3: Rango de representación de enteros con signo

- Observe que el rango de representación no es simétrico: se puede representar el -128 en un char pero no el +128
- Los enteros positivos se representan en base 2 como si fuesen enteros sin signo
- Los enteros negativos se representan en complemento de 2
- Al operar con enteros, si se produce un desborde en la representación no se genera ningún tipo de error, ¡pero el resultado es incorrecto!
- El tipo char es unsigned en algunas plataformas (use signed char)
- En la plataforma Windows de 64 bits, el tipo long es de 32 bits (no es de 64 bits como en Unix).
- A partir del estandar C99 existe el tipo long long. Se especifica que debe ser de al menos 8 bytes.
- Observe que aún cuando los procesadores de PCs y smartphones son de 64 bits, la mayoría de los smartphones funcionan en modo 32 bits, a no ser que tengan al menos 4 GB de memoria RAM.
- Tampoco se fabrican PCs de 16 bits, pero se venden muchos procesadores para sistemas embebidos que son de 16 bits, con precios insignificantes al lado de sus hermanos de 32 o 64 bits. Por razones de costos nadie colocaría un procesador de 32 o 64 bits para controlar una lavadora.

Advertencia: Cuando se opera con números enteros y se produce un desborde, el runtime de C no genera ningún tipo de error. El resultado simplemente se trunca al tamaño que debe poseer el resultado. El tamaño está especificado por las reglas de inferencia de tipos que veremos en el curso. [1]

[1] El contenido en azul es extracción directa del apunte de Luis Mateu

7 Representación de números negativos (Apunte)

Los números negativos se representan en complemento de 2. Esto significa que si se representa un número positivo x en 8 bits, entonces -x se representa negando todos los sus bits (lo que se denomina complemento de 1) y sumando 1. Ejemplos: (Mateu, 2021)

valor positivo	representación en binario	valor negativo	comp. de 1	comp. de 2
0	00000000	-0	11111111	000000000
1	00000001	-1	11111111	111111111
2	00000010	-2	11111111	111111110
3	00000011	-3	11111111	111111101
...	...	...	...	...
127	01111111	-127	10000000	100000001
-	-	-128	-	100000000

Cuadro 2.1: Tabla de Representación de números negativos (comp. = complemento)

- Observe que todos los números positivos tienen el primer bit en 0
- Todos los negativos tienen el primer bit en 1
- Se puede representar el -128, pero no el 128
- En C la expresión x entrega el complemento de 1 de x (la negación bit a bit)
- la expresión -x entrega el complemento de 2 de x: x + 1 (es decir le cambia el signo)

¿Por qué se escogió esta representación? El número de transistores que tenían los primeros computadores era muy limitado. La operación más compleja era lejos la suma y no quedaban transistores para implementar una resta, multiplicación o división. La gracia de la representación en complemento de 2 es que:

```
x-y == x+(-y) == x+(~y+1)
```

Es decir que podemos hacer la resta sumando el complemento de 2. No se necesita un nuevo circuito para hacer la resta. Por otra parte la multiplicación se puede realizar haciendo sumas, y la división haciendo sumas y restas. ¡La economía en transistores es mayúscula!

8 Representación de negativos en complemento de 2

¿Cómo se representa el -28 en binario en un char?

- Método:
  - Tomar valor absoluto: 28
  - Representar en binario (dividir por 2 repetidamente llegar a 0): 11100 (16+8+4)
  - Extender a 8 bits: 00011100
  - Calcular el complemento de 1 (convertir 0s a 1 y 1s a 0): 11100011
  - Sumar 1 en binario: 11100100
- En resumen para representar un entero negativo, calcular complemento de 1 + 1 del valor positivo
- ¿Por qué se llama complemento de 2?
- Porque complemento de 1 + 1 = complemento de (1+1) = complemento de 2
- Es humor tecnológico
- En C: $-x \equiv \sim x + 1$

9 Tabla de valores y sus equivalencias

Número binario	Valor sin signo	Valor con signo
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
00000100	4	4
...		
01111111	127	127
10000000	128	-128
10000001	129	-127
...	...	...
11111100	252	-4
11111101	253	-3
11111110	254	-2
11111111	255	-1

Figura 2.4: Tabla de valores y sus equivalencias

- El -1 es 111....1111 en todos los tamaños (int, long, short, etc.)
- El primer bit de los negativos es siempre 1
- Por eso se llama el bit de signo

10 Conversión a base 10 de enteros con signo

- ¿Qué valor con signo representa el 11101101?
- Para convertir un número binario de n bits $x_{n-1}...x_0$ a base 10:
  - Si bit de signo $x_{n-1} \equiv 0 : calcular \sum_{i=0}^{n-1} x_i 2^i$
  - Si bit de signo $x_{n-1} \equiv 1 : calcular \sum_{i=0}^{n-1} x_i 2^i - 2^n$
- Para 11101101, $n \equiv 8 : \sum \equiv 128 + 64 + 32 + 8 + 4 + 1 = 237$
- Como el bit de signo es $1 : 237–256 = -19$
- ¿Por qué se eligió el complemento de 2?
- (La alternativa es signo y magnitud.)
- ¡Porque la suma de los enteros con signo es la misma que la suma de los enteros sin signo!
- Porque $x–y \equiv x+\sim y + 1$
- ¡El mismo sumador sirve para sumar o restar enteros con o sin signo!

11 Representación de números reales en punto fijo ¡No se usa!

- Se destina una cantidad fija de bits para la parte fraccional
- Por ejemplo 6.25 en punto fijo de 32 bits con una fracción de 16 bits sería:

  0000000000000110 . 0100000000000000 porque su valor es $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 0 \cdot 2^{-1} + 1.2^{-2}$
- El punto va en un lugar fijo
- En general si:

  $x = 15...x_0 x_{-1}...x_{-16}$ el valor sería $\sum_{i=-16}^{15} x_i 2^i$
- Habría que destinar otro bit para indicar el signo

12 Representación de números reales en punto flotante de 32 bits

- Sea $x \neq 0$ un número real expresado en base 2
- Ejemplo: $6.25 = (110{,}01)_2$
- Hay que normalizar el número: se reescribe de manera que esté en el formato $1.bbbb... \cdot 2^e$
- Ejemplo: $6.25 = (1.1001)_2 \cdot 2^2$
- En general x estará en el formato:

  $$\text{signo} \cdot 1\ .\ m_{-1}m_{-2}...m_{-23} \cdot 2^e$$
- En donde signo puede ser 1 o -1
- Los bits $m_{-1}m_{-2}...m_{-23}$ se llaman la mantisa
- El número x se representa como: s $e_7...e_0$ $m_{-1}m_{-2}...m_{-23}$
- Con s=0 si signo es 0 o s=1 si signo es -1
- La parte 1. no se incluye porque es siempre lo mismo
- El valor sin signo de $e_7...e_0$ es e + 127 (129 para 6.25)
- Por lo tanto 6.25 = 0100 0000 1100 1000 00000000 00000000

13 Representación de números reales en punto flotante

- Los casos $e_7...e_0 = 0$ o 255 son especiales
- El 0 se representa como 000...0 (solo ceros)
- Hay una representación para el NaN: *not a number*
- El tipo float: entrega unos 7 dígitos de precisión
  - Ocupa 32 bits, la mantisa es de 23 bits y el exponente de 8
  - La máxima magnitud representable es $\sim 3.4 \cdot 10^{38}$
  - La mínima magnitud es $\sim 1.18 \cdot 10^{-38}$
- El tipo double: entrega unos 15 dígitos de precisión
  - Ocupa 64 bits, la mantisa es de 52 bits y el exponente de 11
  - La máxima magnitud representable es $\sim 1.79 \cdot 10^{308}$
  - La mínima magnitud es $\sim 2.23 \cdot 10^{-308}$
- Ud. encontrará más detalles en la Wikipedia
- Cuidado: 0.1 no es representable de manera exacta
- Cuidado: ¡Nunca escriba $x == y$! Use $|x - y| < \epsilon$
- Debido a las imprecisiones del cálculo x será aproximadamente y, no igual
- Conjetura: el error de $x + y + z + w$ es mayor al de $(x + y) + (z + w)$

14 Representación de caracteres

- Se usa la codificación ASCII
- Las constantes 'a' 'b' 'c' ... 'z' son 97 98 99 ... 122
- 'A' 'B' 'C' ... 'Z' son 65 66 67 ... 90
- '0' '1' '2' ... '9' son 48 49 50 ... 57
- '!' es 33 "es 34 ... etc.
- '\n' es 10
- Note que 'A'+1 es 'B' y que '0'+4 es '4'

En la página siguiente una tabla de ejemplo.

b7 b6 b5 / BITS / b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0 0	16 DLE 10 20	32 SP 20 40	48 0 30 60	64 @ 40 100	80 P 50 120	96 ‘ 60 140	112 p 70 160
0 0 0 1	1 SOH 1 1	17 DC1 11 21	33 ! 21 41	49 1 31 61	65 A 41 101	81 Q 51 121	97 a 61 141	113 q 71 161
0 0 1 0	2 STX 2 2	18 DC2 12 22	34 " 22 42	50 2 32 62	66 B 42 102	82 R 52 122	98 b 62 142	114 r 72 162
0 0 1 1	3 ETX 3 3	19 DC3 13 23	35 # 23 43	51 3 33 63	67 C 43 103	83 S 53 123	99 c 63 143	115 s 73 163
0 1 0 0	4 EOT 4 4	20 DC4 14 24	36 $ 24 44	52 4 34 64	68 D 44 104	84 T 54 124	100 d 64 144	116 t 74 164
0 1 0 1	5 ENQ 5 5	21 NAK 15 25	37 % 25 45	53 5 35 65	69 E 45 105	85 U 55 125	101 e 65 145	117 u 75 165
0 1 1 0	6 ACK 6 6	22 SYN 16 26	38 & 26 46	54 6 36 66	70 F 46 106	86 V 56 126	102 f 66 146	118 v 76 166
0 1 1 1	7 BEL 7 7	23 ETB 17 27	39 ' 27 47	55 7 37 67	71 G 47 107	87 W 57 127	103 g 67 147	119 w 77 167
1 0 0 0	8 BS 8 10	24 CAN 18 30	40 (28 50	56 8 38 70	72 H 48 110	88 X 58 130	104 h 68 150	120 x 78 170
1 0 0 1	9 HT 9 11	25 EM 19 31	41) 29 51	57 9 39 71	73 I 49 111	89 Y 59 131	105 i 69 151	121 y 79 171
1 0 1 0	10 LF A 12	26 SUB 1A 32	42 * 2A 52	58 : 3A 72	74 J 4A 112	90 Z 5A 132	106 j 6A 152	122 z 7A 172
1 0 1 1	11 VT B 13	27 ESC 1B 33	43 + 2B 53	59 ; 3B 73	75 K 4B 113	91 [5B 133	107 k 6B 153	123 { 7B 173
1 1 0 0	12 FF C 14	28 FS 1C 34	44 , 2C 54	60 < 3C 74	76 L 4C 114	92 \ 5C 134	108 l 6C 154	124 | 7C 174
1 1 0 1	13 CR D 15	29 GS 1D 35	45 − 2D 55	61 = 3D 75	77 M 4D 115	93] 5D 135	109 m 6D 155	125 } 7D 175
1 1 1 0	14 SO E 16	30 RS 1E 36	46 . 2E 56	62 > 3E 76	78 N 4E 116	94 ^ 5E 136	110 n 6E 156	126 ~ 7E 176
1 1 1 1	15 SI F 17	31 US 1F 37	47 / 2F 57	63 ? 3F 77	79 O 4F 117	95 _ 5F 137	111 o 6F 157	127 DEL 7F 177

LEGEND: dec / CHAR / hex oct

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

Figura 2.5: ASCII CONTROL CODE CHART

III

CLASE 3

Este capítulo está construido bajo las explicaciones de la clase grabada de 2020-1. El texto oficial de referencia se encuentra aquí.

1 Esquema de la clase

1) Asociatividad / Precedencia

2) Sistema de tipos

3) Operadores

2 Precedencia

Para entender bien este concepto pensemos en el siguiente caso: ¿Qué pasa si escribimos la expresión $a + b * c$?

Conocemos dos interpretaciones (con paréntesis como la orden de operación en aritmética), la primera $(a + b) * c$, la segunda es $a + (b * c)$. En este caso el orden que preferimos es la segunda opción.

Notemos que la multiplicación $(*)$ en este caso tiene más precedencia que la suma, pero en el caso de la división, por ejemplo, esta tiene la misma precedencia que la multiplicación de este caso.

En los apuntes podemos encontrar la precedencia de todos los operadores (un montón, pues C posee una basta cantidad de operadores). Pero la precedencia no resuelve la ambigüedad que existe en algunas expresiones, atendamos el caso de la sección siguiente.

3 Asociatividad

Basándonos en el ejemplo de $a+b*c$ donde elegimos la precedencia de la multiplicación, nos damos ahora la expresión $a-b+c$. De esta expresión resultan los casos: $(a-b)+c$ y $a-(b+c)$ donde no podemos usar el concepto de parecencia para escoger entre estos dos casos. En estos casos el concepto que nos quita este problema es el concepto de **Asociatividad**.

Hay dos posibilidades cuando hay dos operadores con la misma preferencia: Se asocia a la izquierda o se asocia a la derecha. En el caso de la suma y la resta la asociatividad ocurre hacia la izquierda: $(a - b) + c$. Gran mayoría de operadores asocia hacia la izquierda pero hay excepciones, e.g: el operador de asignación.

En el caso del operador de asignación tenemos, por ejemplo: $a = b = c$, que nuevamente genera los casos $(a = b) = c$ y $a = (b = c)$, donde la segunda es la interpretación correcta, pues el lenguaje C exige que lo que esté al lado izquierdo sea una variable (véase el concepto de *left value*).

4 Sistema de tipos (Borrador)

Nota: Esto es solo un borrador de la clase dictada en video, por ello se recomienda leer estas secciones directamente del apunte disponible en la web.

Supongamos que tenemos declarada una variable:

```
double x = ...;
int k = (int)(x+5)/3
```

Código 1: Ejemplo de código donde int es una *cast* de tipo primitivo y x+5 es de tipo double

Nos preguntamos ¿Cómo se determina el tipo de dicha expresión?. Recordemos que en el lenguaje C los tipos son estáticos, esto quiere decir que los tipos de cada expresión son determinados en tiempos de compilación (en *Python* son dinámicos y se determinan en tiempo de ejecución)

Para ellos tenemos que saber que aquí se usa la operación *value* en x (para el caso de 5 es una constante y ya es un valor, no hace falta pedir *value*).

Aquí x es un real y 5 un entero, entonces hace falta una conversión. Para ello se aplica un *cast* de tipo primitivo implícito (double) a 5. Una vez tenemos el mismo tipo a ambos lados, podemos realizar la operación double+ (suma real).

Después de realizar la suma existe un *cast* de tipo explícito en el código (int) para convertir este número real en un entero, y dado que 3 es entero y la suma real se ha convertido en entero, se realiza una división entera (entre int(x+5) y 3)

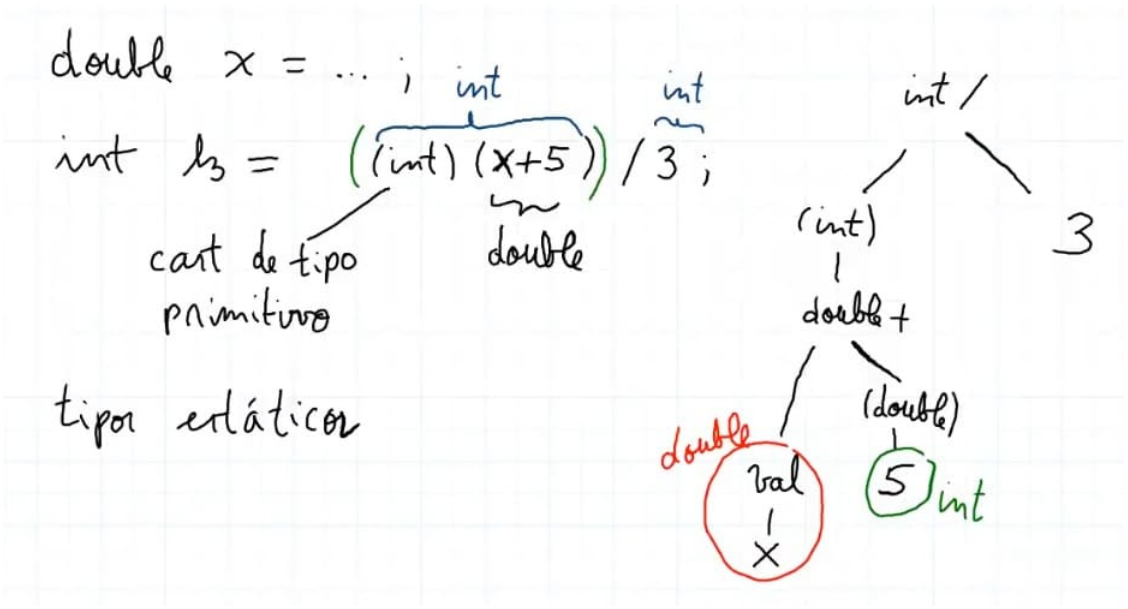


Figura 3.1: Diagrama de árbol y tipos. (Mateu, 2021)

4.1 Inferencia de tipos

Consideremos otro ejemplo. Este ejemplo se encuentra detallado en el apunte disponible en `https://wiki.dcc.uchile.cl/cc3301/temario`. Para el propósito de la clase el capítulo de tipos está disponible aquí bajo el mismo nombre de esta subsección.

```
int a=1;
int b=2;
double x = a/b;
```

Código 2: Segundo ejemplo

¿Cuanto vale x? Parece obvio: 0.5. ¡Pero vale 0! La razón es que el tipo de a/b se determina a partir del tipo de sus operandos, no a partir del uso que se de al resultado. Ya que el destino es una variable real, se tendería a pensar que la división debería ser con números reales, pero así no razona el compilador. La regla es que si ambos operandos son enteros, entonces la división es entera y el resultado tendrá 32 bits. Por eso 1/2 es 0. (Mateu, 2021)

Otra regla para los operadores binarios es que si un operando es double, el otro operando se convierte implícitamente a double también. Por lo tanto para lograr la divisón real hay que usar un cast para convertir uno de los operandos a double: (Mateu, 2021)

```
double x= (double)a / b;
```

Aquí (double) es un cast y tiene mayor precedencia que /. El código es equivalente a:

```
double x= ((double)a) / ((double)b);
```

Cuidado: ¡el siguiente código también entrega 0!

```
double x= (double) (a / b);
```

Consideremos otro ejemplo:

```
char a= 127;
char b= 1;
int c= a+b;
```

¿Cuanto vale c? ¿128? ¿o podría ser -128? De acuerdo al texto de más arriba, se podría pensar que la suma debería realizarse en 8 bits con signo. Pero 128 no es representable en 8 bits con signo. De hecho el resultado de la suma sería el valor binario 10000000, ¡que resulta ser -128 en 8 bits con signo! Pero el valor de c sí resulta ser 128. ¿Por qué?

Hay una regla en C que dice que todas las operaciones aritméticas deben considerar al menos el número de bits del tipo int, es decir 32 bits (casi siempre). Por lo tanto la asignación de c es equivalente a:

```
int c= (int)a + (int)b;
```

Pero cuidado, el número positivo más grande representable en 32 bits es 2147483647. Considere este código:

```
int a= 2147483647;
double x= a + 1;
```

En este caso, la asignación de x es equivalente a:

```
double x= (double)(a+1);
```

Por lo tanto a+1 se realiza en 32 bits con signo. El resultado en binario es un 1 seguido de 31 ceros, que corresponde al valor entero -2147483648. Ese es el valor incorrecto que queda almacenado finalmente en x. Y no 2147483648 como debería ser.

Ejercicio: Reescriba la instrucción de asignación cambiando todas las conversiones implícitas a conversiones explícitas.

```
double x;
char c;
long long ll;
...
int i= ll + c/2 + x*2;
```

```
double x= (double)(a+1);
```


```
double x= (double)(a+1);
```


4.1.1 *Resumen*

- El rango de un tipo numérico es el intervalo de números que puede representar.
- Los tipos numéricos están estrictamente ordenados por su rango:

  char <short <= int <= long <= long long <float <double
- Cuando se realiza una operación numéricas entre tipos distintos, el operando de un tipo con rango menor se convierte implícitamente al tipo del otro operando.
- No se realizan operaciones aritméticas con un tipo de rango inferior a int. Es decir cuando un operando es de tipo char o short se convierte implícitamente al menos a un int.

char <short <= int <= long <= long long <float <double Cuando se realiza una operación numéricas entre tipos distintos, el operando de un tipo con rango menor se convierte implícitamente al tipo del otro operando. No se realizan operaciones aritméticas con un tipo de rango inferior a int. Es decir cuando un operando es de tipo char o short se convierte implícitamente al menos a un int.

4.1.2 *Mezcla de operandos con y sin signo*

A partir de Ansi C se especifica que si un operando es con signo y el otro sin signo, entonces el operando sin signo se convierte implícitamente a un tipo con signo y la operación se realiza con signo. Esto significa que si un operando es unsigned int y se suma con un int, entonces el primero se convierte a int. ¡Cuidado! En esta conversión se podría producir un desborde. (Mateu, 2021)

4.2 El costo de las operaciones

Dado que C es un lenguaje en donde interesa la eficiencia, resulta importante conocer cual es el costo de las operaciones aritméticas en términos de su latencia. La latencia de una operación es la cantidad de ciclos del reloj del procesador que deben pasar para poder usar el resultado de esa operación. (Mateu, 2021)

Símbolo	Tipo de datos	Latencia (ciclos)	Observaciones
+, -	int	1	text
*	int	3	en un procesador reciente
/	int	8, 16 o 32	1 ciclo por cada 1, 2 o 4 bits del divisor
+, -, *	float/double	3	en un procesador reciente
/	float	8 a 32 ciclos	1 ciclo por cada 1, 2 o 4 bits del divisor
/	double	16 a 64 ciclos	1 ciclo por cada 1, 2 o 4 bits del divisor

Observe que la suma y la resta de enteros son las operaciones más eficientes, seguidas de la suma, resta y multiplicación de números en punto flotante. Por otro lado, la división es lejos la operación más lenta. Esto se debe a que su implementación en circuitos requiere un ciclo por cada 4 bits del divisor, en los procesadores más recientes (Haswell). En los procesadores menos recientes (Sandy Bridge) pueden ser 1 ciclo por cada 2 bits del divisor e incluso 1 ciclo por cada bit del divisor (más lento).

4.3 Definición de nuevos tipos

En C se definen nuevos tipos con struct, union, enum y typedef. Estos temas serán abordados más adelante. (Mateu, 2021)

IV
CLASE 4

Para esta clase hay un material audio/visual preparado en un formato especial. Es muy similar a una clase con pizarra. Las instrucciones para bajar la clase y el programa que se necesita para ver la clase están acá, en la sección clases disponibles con el relator, operaciones con bits. En los apuntes corresponde al capítulo de operaciones con bits. (Mateu, 2022)

Es importante que revisen este material para la clase auxiliar del viernes 18, porque será sobre operaciones con bits.

1 Operaciones con bits

1.1 Tabla de operadores lógicos

Símbolo	Operación	Significado	Ejemplo (1100 op 1010)
&	and	**y** bit a bit	1000
|	or	**o** bit a bit	1110
∧	xor	**o exclusivo** bit a bit	0110

1.2 Tabla de la negación bit a bit

Símbolo	Operación	Significado	Ejemplo (10110111)
∼	not	negación bit a bit	01001000

1.3 Tabla de desplazamientos

Símbolo	Operación	Significado	Ejemplo (10111011 op 2)	Equivalencia
<<	shift left	desp. de bits a la izquierda	11101100	$x<<y \;\; x \cdot 2^y$
>>	unsigned shift right	desp. de bits a la derecha	00101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$
>>	signed shift right	desp. de bits a la derecha	11101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$

1.4 Desplazamiento a la derecha de números negativos

Note que si desplazamos -5 en un bit, el resultado es -3 y no -2, pues $\lfloor \frac{-5}{2^1} \rfloor = \lfloor -2{,}5 \rfloor = -3$.

En matemáticas $\lfloor -2{,}5 \rfloor = \text{máx}\,\{x \mid x \in \mathbb{Z} \wedge x \leq -2{,}5\}$

1.4.1 *Ejercicio*

Tome un tiempo para verificar que $\underbrace{11111011}_{-5} >> 1 \equiv \underbrace{11111101}_{-3}$

2 Casos de borde de los desplazamientos

2.1 Desplazamientos mayores al tamaño de un entero

Se debe tener cuidado con los desplazamientos, por ejemplo:

Consideremos la siguiente asignación de variables.

```
int a = 1; // a de 32 bits
int b = a << 32;
```

Frente a este desplazamiento de **b** nos preguntamos. ¿Cuánto vale b?.

Se espera que se naturalmente nuestra respuesta sea 0, pero sucede que este valor de b calculado de esta manera no está especificado en C. En procesadores intel o amd b es 1.

La regla general es que los resultados de **x** << **n** y x >> n no están especificados si **n** es mayor o igual al tamaño de **x** en bits o **n** < 0. Tampoco están definidos para valores negativos de **n**.

2.2 Por qué desplazamientos no especificados

2.2.1 *La culpa del legado*

¿Qué acaba de ocurrir? Descompongamos **n** en sus 32 bits:

$$n = n_{31}n_{30}...n_5n_4...n_0$$

Resulta que en procesadores *intel* cuando se desplaza un entero **x** de 32 bits en **n** solo se consideran los 5 bits menos significativos de **n**. Como 32 se representa en binario como 32 = 00...10...0, sus 5 bits menos significativos son ceros y, en consecuencia, se termina desplazando en cero bits, por eso el resultado es el mismo x.

Esto no es un error, sino que es culpa del legado. En los años 70, el presupuesto en transistores para los primeros procesadores de *intel* eran apenas 8000 transistores, de usar más transistores el chip resultaría muy caro, debido a ello se debía optar por una política de economizar de transistores como fuera posible. Por ejemplo: Al desplazar, ignorando los bits más significativos de n, puesto que desplazar en el tamaño de x o más no tenía utilidad práctica.

El problema es que una vez tomada esa decisión, intel tendría que calcular para siempre los desplazamientos de esa manera. Después de todo, intel debe su éxito debido a la compatibilidad que han tenido sus procesadores nuevos con sus procesadores antiguos. Cambiar el cómo se hacen los desplazamientos hubiese roto unos cuantos programas.

En otros procesadores el resultado sí es el esperado (ejemplo a << 32 sí es 0). Si C especificara que el resultado es 0 los desplazamientos en C serían caros en procesadores intel, pues habría que consultar si n ≥ |x| (montón de ciclos y el desplazamiento sería mayor a un ciclo del reloj). Por eso se prefirió no especificarlo en C, de esta forma los desplazamientos son eficientes en todos los procesadores a pesar de que puedan dar resultados distintos en casos de borde.

3 Ejemplo: conjuntos de frutas con mapas de bits

A continuación veremos un primer ejemplo de aplicación de las operaciones con bits. Consiste en programar las operaciones sobre conjuntos representados por medio de mapas de bits. Las típicas operaciones sobre conjuntos son: Agregar un elemento, determinar pertenencia, unir conjuntos, etc.

3.1 El tipo Fruta

Considere la siguiente definición de tipo en C.

```
typedef enum fruta {
    manzana,pera,guinda,durazno,damasco}
    Fruta;
Fruta f = durazno; // f ≡ 3
f = f + 1; // f ≡ 4 ≡ damasco
```

Código 3: Ejemplo de definición de tipo en C

Sin entrar en detalles de sintaxis de esta declaración, aseguremos que la enumeración de frutas es manzana (constante 0), pera (constante 1), guinda (constante 2), durazno y damasco. El nuevo tipo se llama Fruta (tipo entero)

Si se declara una fruta `f` con valor inicial durazno, esto es equivalente a que el valor inicial del entero `f` es 3, porque durazno es la constante 3. Es perfectamente legítimo escribir `f=f+1`, con lo que `f` será 4 (guarda un damasco). Por supuesto, este ejemplo no tiene mucho sentido, pero se hace énfasis en que el lenguaje lo permite y el resultado es siempre el mismo.

Visitemos un ejemplo con sentido. Se define el nuevo tipo conjunto, será un conjunto de frutas y se representa mediante un entero de tipo int. Se declara el conjunto `c`, la variable `c` es simplemente un int destinado a guardar conjuntos de frutas. Se descompone el entero c en todos sus bits. La idea es que el bit menos significativos representa la presencia (1) o ausencia (0) de una manzana en el conjunto.

3.2 El tipo conjuntos de frutas

```
typedef int Conjunto // de frutas
Conjunto c;
    c
```

3.3 Mapas de bits

Se hace la descomposición de `c` en bits ($c = ...c_5c_4c_3c_2c_1c_0$). Basándonos en lo anterior, c_1 representa presencia de la pera, por ejemplo.

Ejercicio ¿Qué frutas están presentes en el conjunto ...010010?. [1]

Esta representación para un conjunto se llama **mapa de bits** y es muy eficiente para representar conjuntos de toda clase

3.4 Implementación eficiente de operaciones sobre conjuntos

Calculemos 1 desplazado en f bits. ¿Qué representa?. Por ejemplo 1 << 3 (3 representa al durazno).

Entonces 1 desplazado en 3 es ...001000. Lo que es equivalente al **singleton durazno** ≡ {durazno}.

Entonces 1 desplazado en f bits corresponde a contruir un conjunto con solo `f` como fruta. Ponga atención a la siguiente función. La gracia de representar esta manera los conjuntos de fruta es que el costo de singleton es solo un ciclo del reloj (lo que toma el desplazamiento)

```
Conjunto singleton(Fruta f) {
    return 1 << f;
}
```

[1] La respuesta es la pera y el damasco

3.4.1 *Más operaciones (Unión, Agregar, Pertenece)*

```
Conjunto Union(Conjunto r, Conjunto s) {
    return r | s;
}
```

```
Conjunto agregar(Fruta f, Conjunto s) {
    return Union(s, singleton(f));
}
```

```
int pertenece(Frtuta f, Conjunto s) {
    return s & singleton(f);
}
```

3.4.2 *Complemento de un conjunto*

```
Conjunto complemento(Conjunto s) {
    return  s
}
```

4 Mostrar un conjunto y ejemplo de uso

4.1 Mostrar un conjunto

```
void monstar(Conjunto s) {
    for(Fruta f = manzana; f <= damasco; f++) {
        if(pertenece(f,s)) {
            switch(f)   {
            case manzana: printf("manzana");break;
            case pera: printf("pera");break;
            case guinda: printf("guinda");break;
            case durazno: printf("durazno");break;
            case damasco: printf("damasco");break;
            }
        }
    }
    printf("\n")
}
```

4.2 Ejemplo de uso de conjuntos

```
int main() {
    Conjunto s1 = agregar(manzana,
                  agregar(guinda,
                    singleton(damasco)));
    Conjunto s2 = complemento(s1);
    mostrar(s1);
    mostrar(s2);
    return 0;
}
```

4.3 Ejercicios

- Programe una función que entregue la intersección de 2 conjuntos (en no más de 1 ciclo del reloj)
- Programe una función que entregue la diferencia (de 2 conjuntos) (en no más de 2 ciclos del reloj)

5 Ejemplo: extracción de bits usando máscaras

A menudo, en aplicaciones que manipulan grandes volúmenes de información, se requiere ser muy cuidadosos con el uso de la memoria, por eso se colocan múltiples enteros pequeños en un solo entero de 32 bits. Por ejemplo:

Considere un entero **x** descompuesto en todos sus bits, desde el más significativo (x_{31}) hasta el menos significativo (x_0). **x** guarda un número pequeño de tan solo k bits, desde el bit x_i hasta el bit x_{i-k+1}. El problema consiste en extraer ese número pequeño, es decir, al llamar la función extraer(x,i,k), se debe entregar como resultado un entero que contiene x_i hasta x_{i-k+1} en los bits menos significativos y 0 en los más significativos.

Esto se logra por etapas.

1) Primero tenemos todos los bits de x

2) Lo transformamos borrando los bits que están antes de x_i

3) Desplazamos el resultado a la derecha en $i - k + 1$ bits

5.1 Máscaras de bits

Volvamos a la máscara de bits. Las máscaras de bits son enteros que se usan para borrar o encender bits en otro entero. El principio es que

- x & m: borra los bits de x que en m sean 0 (porque al hacer un *and* con cualquier bit con 0, se obtiene 0)
- x | m: enciende los bits de (porque al hacer un *or* de cualquier bit con 1, el resultado siempre es 1)

En el problema que queremos resolver se necesita borrar los 31-i primeros bits, necesitamos una máscara con los 31-i primeros bits en 0.

En general para construir una máscara con los j primeros bits en 0, se inicia con -1, que tiene todos sus bits en 1, luego se convierte al tipo sin signo.

```
((unsigned)-1) >> j
```

Luego se desplaza a la derecha en j bits, esto rellena por la izquierda con j bits en 0, es la máscara que necesitamos.

Para construir una máscara con los últimos j bits en 0, partimos con un -1 y lo desplazamos a la izquierda en j bits, que rellena por la derecha con j bits en 0. Esa es la máscara.

```
((unsigned)-1) << j
```

5.2 La función extraer

5.2.1 *Solución 1*

Esta es la primera solución a la función extraer: Entrega un entero sin signo, recibe como parámetros un entero x sin signo, un entero i con la posición del bit más significativo del número que se desea extraer, y el entero k que corresponde a su tamaño en bits.

El resultado de la función es x & la máscara que tiene el $31 - i$ primeros bits en cero, es decir, corresponde al primer caso de máscara. Con esto borramos los primeros $k - i$ bits, faltando una etapa.

Desplazamos en $i - k + 1$ bits, con lo que x_{i-k+1} quedó en el bit menos significativo.

5.2.2 *Solución 2*

Sin usar máscaras. Consiste en desplazar x a la izquierda en $31 - i$ bits, eliminando por completo los $31 - i$ bits de la izquierda, y luego desplazando en 31-i bits a la derecha, esto rellena con $31 - i$ bits en 0. Pero además debemos desplazar a la izquierda otros $i - k + 1$ bits. Simplificando nos queda $32 - k$bits. Es la solución más directa, pero en general para resolver los problemas típicos de manejos de bits, usted necesitará usar máscaras y desplazamientos.

V

CLASE 5

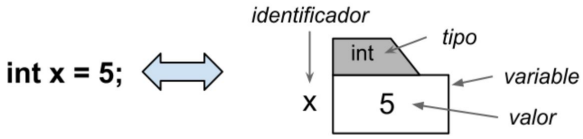
1 Agenda de la clase

- Variables
- Dirección de una variable
- Punteros
- Arreglos de variables
- Aritmética de punteros

2 Variables

Una variable almacena valores, se la conoce por su identificador (String) y su nombre es tal que le permita cambiar su valor. Otra cc es que todos los valores almacenados deben ser del mismo tipo. Gran diferencia con lenguajes dinámicamente tipados como Python (sus variables pueden almacenar valores de cualquier tipo). Tanto en C como en Java tenemos variables estáticamente tipadas.

Con las variables podemos tener operaciones. La primera de ellas es su declaración, seguida de la asignación y evaluación. Características únicas de C son las operaciones de Dirección, es decir, en donde se almacena dicha variable, porque las variables se almacenan en memoria y cada byte de la memoria tiene una dirección., así podemos saber la dirección donde comienza esta variable. Ora operación única en C es el Tamaño (sizeof(x)).



- Se conocen por su identificador
- Sirve para almacenar valores
- Todos los valores deben ser del mismo tipo
- La variable es la caja que almacena los valores
- Operaciones:

  Declaración int x;

  Asignación x=5;

  Evaluación x;

  Dirección: &x (Se suele llamar a & el operador de dirección)

  Tamaño sizeof(x)

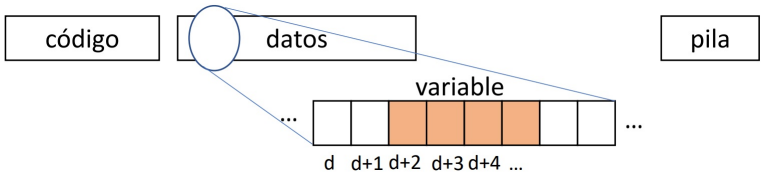
- Regla de substitución La evaluación de una variable equivale a substituir el identificador por el valor almacenado en la variable en el momento de la evaluación. Es decir, cuando llega el momento de ejecutar y hay una expresión que utilice la variable x, para evaluarla debemos ver el valor que tiene esta variable x (5 en el caso ejemplo) y substituir la variable por ese x.

2.1 Ejemplos

Visitemos algunos ejemplos.

```
int y = x + 10 ⇔ int y = 5 + 10
x = x + 1 ⇔ x = 5 + 1
```

3 La memoria de un programa



- El programa y sus datos se almacenan en memoria
- La memoria del programa se constituye de segmentos o áreas de memoria: código, datos, pila ...
- Cada segmento se constituye de bytes
- Cada byte tiene una dirección única
- Dentro del mismo segmento las direcciones de los bytes se asignan consecutivamente
- Una variable de tamaño X ocupa X bytes con direcciones consecutivas
- Alineamiento: Para variables de tipos primitivos (int, double, ...) la dirección de inicio es múltiplo de su tamaño

4 Punteros

- Son variables que almacenan direcciones de variables. Ejemplos:

```
int*ptr;
double *ptr_pi;
```

- Los usaremos para implementar los strings, la estructuras de datos y mucho más
- Si ptr almacena la dirección de x, se dice que ptr apunta a x
- Deben apuntar a variables del tipo declarado

```
int x;    ptr=&x;
int y;    ptr=&y;
double pi;    ptr_pi=&pi;
ptr_pi    // Incorrecto: puntero a double
          // Incorrecto: no puede apuntar a un int
```

- No es lo mismo que un entero

```
ptr = 0x7ffff8; // Incorrecto (se puede usar pero con un cast)
```

5 El operador de contenido *

- Si `x` se declaró como `int x` el tipo de `x` es `int`
- Si `s` se declaró como char * el tipo de s es char * (es para almacenar direcciones que hayan sido declaradas de tipo char)
- El símbolo * es la multiplicación cuando la operación es binaria: x * y
- Si la sintaxis es * expresión entonces es el operador de contenido y el tipo de expresión debe corresponder a un puntero como int *, double *, ...
- Usualmente la expresión es un puntero: ptr, ptr_pi
- La evaluación de *ptr en una expresión equivale a substituir *ptr por la variable a la cual apunta ptr en el momento de la evaluación

`ptr=&x;`

`*ptr=y;` ⇔ `x=y`

`ptr=&w;`

`*z=*ptr;` ⇔ `z=w`

- El tipo de *ptr es el tipo de la variable a la cual apunta ptr: si ptr es de tipo int* ⇔ *ptr es de tipo int

6 Arreglos de variables

- Una arreglo es un conjunto enumerado de variables del mismo tipo
- Se declaran con [ ]. Ejemplo:

```
int a[10]; // arreglo de 10 variables enteras
char s[20]; // 20 variables de tipo caracter
```

- En una expresión el operador binario de subindicación [ ] se usa para seleccionar una variable del arreglo
- Sintaxis: exp[exp-índice] Ejemplos: `a[7]` `s[i]`
- El tipo de exp-índice debe ser entero
- 0 es el índice de la primera variable del arreglo
- 1 es el índice de la segunda variable...
- 9 es el último índice válido para el arreglo a
- No se puede determinar el tamaño del arreglo
- No hay chequeo de índices ⇒ Segmentation fault

7 Aritmética de punteros

- double x[100];
- Para obtener la dirección de un elemento de x:

```
int i = 23;
double *p = &x[i];
```

- El identificador x representa la dirección del primer elemento:

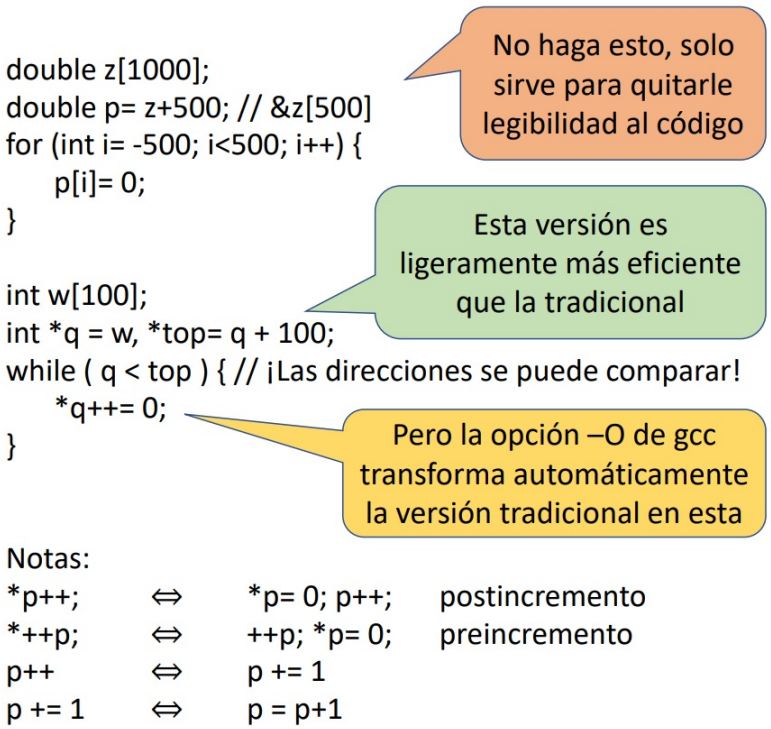
  `p = x;` ⇔ `p = &x[0];`
- En la expresión dir[índice]
  - índice puede ser cualquier expresión de tipo entero
  - dir puede ser cualquier expresión de tipo puntero (T *)
  - ¡También puede ser un puntero! Ejemplo: `p[i]`
  - Accede a la variable de dirección dir + índice * sizeof(T)
  - ¡El índice puede ser negativo!
  - Si la dirección cae fuera de un segmento del programa

    ⇒ Segmentation fault
- Equivalencias:

  & dir[índice] ⇔ dir + índice

  & dir[-índice] ⇔ dir - índice

8 Maneras rebuscadas de inicializar un arreglo en 0, pero correctas



9 Cuidado con los arreglos

La declaración: char s[10];

- Atribuye espacio en memoria para un arreglo de 10 caracteres
- s representa la dirección del primer elemento del arreglo
- Pero s no es una variable, representa un valor
- No se puede cambiar la dirección con una asignación:

```
s = ...; // ¡incorrecto!
```

- Por la misma razón que una constante no se puede cambiar con una asignación:

```
1000 = ...; // ¡incorrecto!
```

- Pero un puntero sí se puede asignar

```
char *p = s; // p almacena &s[0]
p = p + 5; // correcto porque p sí es una variable
```

- Pero la declaración solo atribuye espacio para el puntero, nunca para lo apuntado

```
char *p; p[4]= 0; // ¡incorrecto!
```

10 Más sobre arreglos y punteros

- Si p es un puntero o arreglo, las siguientes expresiones son inválidas: `p*5` `p + 2.5` `p/10`
- Solo tiene sentido p + o − una expresión entera
- Si p y q son punteros del mismo tipo: `p-q` es correcto
- Declaración con inicialización:

```
int a[ ] = { 2, 3, 5, 7, 11, 13 }; // Arreglo de 6 elementos
int b[100] = { 2, 3, 5 } ; // solo inicializa los 3 primeros
                           // de un arreglo de 100 elementos
b = { 4, 8, 12 }; // Incorrecto, no es una declaración
int c[ ]; // Incorrecto, no se puede inferir el tamaño
```

- Una función no puede recibir como parámetro un arreglo, pero sí puede recibir un puntero y operarlo como si fuese un arreglo
- Si se declara una función con este encabezado: `void fun(int arreglo[ ]);`
-
- El compilador silenciosamente lo cambia por: `void fun(int *arreglo);`

11 Resumen

- Una variable reside en la memoria
  - — Almacena valores de un tipo específico
  - — Posee un tamaño en bytes y una dirección
- Un puntero es una variable que almacena direcciones de variables
  - — Sirven para implementar los strings y las estructuras de datos
  - — El operador de contenido permite acceder a la variable a la cual apunta un puntero
- Un arreglo es un conjunto enumerado de variables del mismo tipo
  - — El operador de subindicación permite seleccionar una de esas variables por medio de un índice
  - — Punteros y arreglos son similares, pero hay diferencias
  - — Hay una equivalencia entre subindicación y aritmética de punteros
- Los programas son más eficientes en C porque no hay chequeo de índices, pero los errores de manejo de C son difíciles de depurar

12 Código usado en clases

```c
#include <stdio.h>

int main() {
  int x;                // declaracion
  x = 5;                // asignacion
  printf("%d\n", x);    // lectura
  printf("%p\n", &x);   // direccion
  printf("%ld\n", sizeof(x)); // Tamano
  int y = x + 10;       // declaracion con
                        // inicializacion
  x = x+1;              // asignacion
  double pi = 3.14159; // variable real
  pi = pi + x;
  printf("%ld\n", sizeof(pi));

  int *ptr = & x;        // decl. puntero
  printf("%p\n", ptr);
  ptr = &y;
  double *ptr_pi = &pi;
  printf("%p\n", ptr_pi);

  *ptr = x;      // operador de contenido
                 // en una asignacion
  x = 1;
  int z = *ptr; // operador de contenido
                 // en una expresion
  printf("%d\n", z);

  int a[10];
  printf("%p\n", a);
  for (int i= 0; i<10; i++) {
    a[i]= i*i;
    printf("%p: %d\n", &a[i], a[i]);
  }
  int k= 1000000000;
  printf("%d\n", a[k]);

  return 0;
}
```

VI
CLASE 6

1 Agenda de la Clase

- Tiempo de vida de una variable
- Variables locales
- La pila de registros de activación
- Variables dinámicas: malloc/free
- El heap de memoria
- Errores comunes: memory leaks y dangling referencese
- Sanitize y valgrind

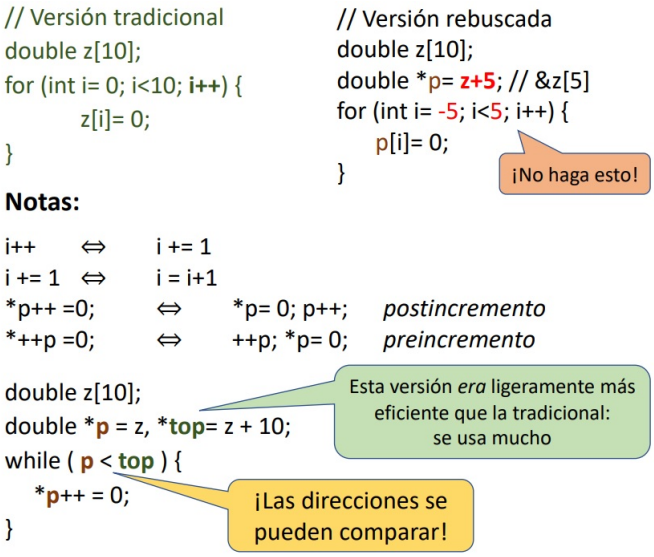
2 Resumen de la clase pasada: Variables y punteros

- Una variable reside en la memoria
  - Almacena valores de un tipo específico
  - Se puede declarar, asignar y evaluar
  - También se puede obtener su dirección
  - Y su tamaño en bytes
- Un puntero es una variable que almacena direcciones de variables
  - Sirven para implementar los strings, las estructuras de datos y mucho más
  - El operador de contenido permite acceder a la variable a la cual apunta un puntero
- Un arreglo es un conjunto enumerado de variables del mismo tipo
  - El operador de subindicación permite seleccionar una de esas variables por medio de un índice
  - Hay una equivalencia entre subindicación y aritmética de punteros

3 Aritmética de punteros

Visitar Clase 5, sección 7

4 Maneras rebuscadas de inicializar un arreglo en 0, pero correctas



5 Cuidado con los arreglos

Visitar Clase 5, sección 7

6 Más sobre aritmética de punteros

- Si p y q son punteros o arreglos, las siguientes expresiones son inválidas: `p*5` `p+2.5` `p/10` `p+q`
- Solo tiene sentido `p + o - una expresión entera`
- Si p y q son punteros del mismo tipo:

  `p - q` es correcto y es de tipo entero
- El valor de p − q satisface:

  `p - q = i` ⇔ `p = q + i`

7 Más sobre arreglos y punteros

Visitar Clase 5, sección 10

8 Strings

- Un string es un arreglo de caracteres que termina con un byte que almacena el valor 0: no `'0'`
- Cuidado: $48 = \texttt{'0'} \neq 0$
- Ejemplo: `char str[ ] = {'H', 'o', 'l', 'a', 0};`
- Se referencian por medio de la dirección de su primer caracter
- Ejemplo: `printf("%s\n", str );`
- Se puede asignar a un puntero: char *r = str;

Ejemplo: contar las letras mayúsculas:

```
char *r = str;
int cnt = 0;
while (*r != 0) {
if ('A' <= *r && *r <= 'Z')
cnt++;
```

```
r++;
}
printf("Mayúsculas: %d\n", cnt);
```

8.1 Strings constantes

- Todo lo que se escribe entre " ... "
- Ejemplo: `char *str2 = "Hola";`
- ¡No se pueden modificar! `*str2 = 'h'; // Seg. Fault`
- Se almacenan en un área de memoria de solo lectura

Sintaxis especial para declarar strings mutables:

```
char str3[ ] = "Hola"; // No estaba en el C original
*str3 = 'h'; // Correcto
printf("%d\n", str3); // Muestra hola, h minúscula
str3 [ ] = "Hello"; // Error sintáctico
```

8.2 Funciones para manipular strings

- `int strlen(char *s)`: calcula el largo de un strings, sin contar el 0 que lo termina
- `strlen("Hola")`: es 4, no entrega el tamaño de memoria atribuido
- `char *strcpy(char *d, char *s)`: copia el string s en el string d

```
char d[20];
strcpy(d, "Hola");
```

Código 4: El destino d debe ser la dirección de un área de tamaño suficiente (largo del string + 1)

```
char *p;
strcpy(p, "Hola"); // Incorrecto, ¿seg. Fault?
```

Código 5: (Incorrecto) Porque p no ha sido inicializado con ninguna dirección válida

```
char s[ ] = "Hola"; ⇔  char s[strlen("Hola")+1];
                       strcpy(s, "Hola")
```

8.3 Comparación de strings

`int strcmp(char *s, char *r)`: compara los strings s y r retornando 0 si son iguales, < 0 si s es lexicográficamente menor que r y > 0 si es mayor

```
char s[20]= "juan";
strcmp(s, "pedro") // es < 0
strcmp(s, "diego") // es > 0
strcmp(s, "juan") // es 0
strcmp(s, "Juan") // es > 0
```

- Cuidado con los operadores relacionales `== != < > <= >=` porque comparan direcciones, no contenidos. `s == "juan"` es $\neq 0$

9 Implementación de strlen y strcpy

```
int mistrlen(char *s) {
    char *r= s;
    while (*r++)
      ;
    return r-s-1;
}
```

```
char *mistrcpy(char *d, char *s) {
    char *t= d;
    while (*t++ = *s++)
      ;
    return d;
}
```

Nota de Mateu: No promuevo este estilo de código, pero deben aprender a entender este estilo porque hay mucho código en C escrito así.

10 Código usado en clases

```c
#include <stdio.h>
#include <string.h>

int mistrlen(char *s) {
  char *r= s;
  while (*r++)
    ;
  return r-s-1;
}

char *mistrcpy(char *d, char *s) {
  char *t= d;
  while (*t++ = *s++)
    ;
  return d;
}

int main() {
  double z[10];
  printf("%p\n", z);
  double *p = z, *top= z+10; // &z[0] y &z[10]
  while ( p < top ) {
    *p++ = 0;
    printf("%p\n", p);
  }

  char str[ ] = {'H', 'o', 'l', 'a', 0};
  printf("%s\n", str);
  char *r = str;
  int cnt = 0;
  while (*r != 0) {
    if ('A' <= *r && *r <= 'Z') {
      cnt++;
    }
    r++;
  }
  printf("Mayusculas: %d\n", cnt);

  char *str2= "Hello";
  // *str2= 'h';
  printf("%s\n", str2);

  char str3[] = "Salut";
  *str3= 's';
  printf("%s\n", str3);

  printf("%ld\n", sizeof(str3));
  printf("%ld\n", strlen(str3));
  r = str3;
  printf("%ld\n", sizeof(r));

  // Ejercicio: ejecutar paso a paso
  // las funciones strlen y strcpy
  printf("%d\n", mistrlen(str3));
  char str4[10];
  mistrcpy(str4, str3);

  return 0;
}
```

VII
CLASE 7

1 Esquema de la clase

- Tiempo de vida de una variable
- Variables locales
- La pila de registros de activación
- Variables dinámicas malloc/free
- El heap de memoria
- Errores comunes: Memory leak y dangling
- Sanitize y valgrind

2 Problema: Función que retorna una copia de un string

```c
char *copia(char*str) {
    char res[strlen(str)+1];
    return strcpy(res, str); // NO HAGA ESTO
}
```

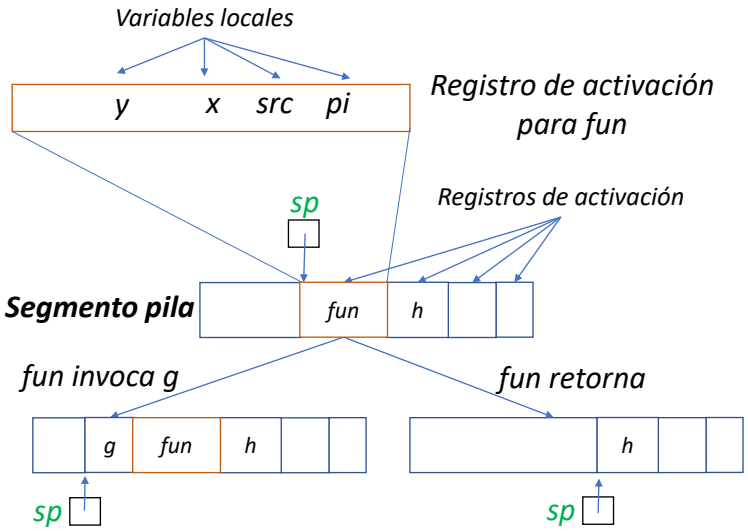
Código 6: ¡`strcpy` retorna `res`!

- Las variables declaradas dentro de una función se llaman variables locales o automáticas, por ejemplo `res` y `str`
- Los arreglos y estructuras también son variables
- Definición: tiempo de vida de una variable

  desde que se crea la variable

  hasta que se destruye
- La creación significa que la variable ya tiene atribuida una dirección en memoria y por lo tanto se puede usar
- La destrucción de una variable significa que la memoria que ocupaba se puede atribuir a otras variables y por lo tanto ya no se debe usar
- Las variables locales se crean en el momento de su declaración y se destruyen automáticamente cuando se sale del bloque en donde fueron declaradas

3 Implementación



- sp es el puntero al tope de la pila (stack pointer)
- Para apilar un registro de activación basta sumar su tamaño a sp
- Para desapilarlo hay que restar su tamaño a sp
- Crear/destruir variables locales tiene costo 0 en tiempo de ejecución

Sin la pila no sería posible la recursividad

- La creación y destrucción es solo conceptual
- Por razones de eficiencia se atribuye la memoria a las variables locales antes de su declaración
- El espacio de las variables locales se atribuye casi siempre al inicio de la función en un área de memoria denominada registro de activación o frame, casi siempre de tamaño fijo
- Las variables locales ocupan un lugar fijo en el registro de activación
- Al inicio de una función se apila su registro de activación en el segmento pila
- Se desapila en el momento del retorno
- La función copia es incorrecta porque se usa el arreglo local res después de su destrucción
- ¿Por qué no siempre? La excepción son las funciones que declaran arreglos de tamaño variable

4 Variables dinámicas

- Son variables que se crean al invocar la función malloc y se destruyen explícitamente con la función free
- Ejemplo: malloc(20)
- Crea una variable dinámica de 20 bytes
- La variable dinámica no tiene identificador
- No tiene tipo todavía
- La función malloc retorna su dirección que debe almacenarse en un puntero
- El tipo del puntero determina el tipo de la variable dinámica
- Ejemplo: char *str = malloc(20);
- La variable de 20 bytes será un arreglo de 20 caracteres que podría almacenar un string
- str es el identificador del puntero, no de la variable dinámica
- Si se destruye str, no se destruye la variable dinámica
- La función malloc es imprescindible para implementar las estructuras de datos recursivas como los ABBs

5 Implementación correcta de la función que retorna una copia de un string

```
// Versión legible
char *copia(char *str) {
    char *res= malloc(strlen(str)+1);
    strcpy(res, str);
    return res;
}
```

```
// Versión ilegible, pero correcta
char *copia(char *str) {
return strcpy( malloc(strlen(str)+1), str );
}
```

6 Destrucción de variables dinámicas

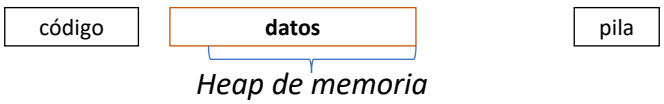
- La destrucción de las variables dinámicas es explícita invocando la función `free`
- Recibe como parámetro la dirección de la variable dinámica
- Ejemplo: `char *str=malloc(10); ... free(str);`
- Una variable dinámica se crea con malloc y vive hasta que se destruye con free
- La dirección que recibe free debe haber sido retornada previamente por malloc variable dinámica

```
int a; free(&a);
int *p=malloc(5*sizeof(int)); ... ; free(&p[4]);
free(p); // correcto
```

La destrucción es solo conceptual: significa que a partir de ese instante la función malloc puede atribuir esa misma memoria para una nueva

7 El heap de memoria

- Las variables dinámicas se alojan en el segmento de datos del programa, en un área que se denomina el memory heap
- Heap significa montón
- No tiene nada que ver con la estructura de datos heap que estudiaron en el curso de algoritmos



- La funciones malloc y free administran el heap de memoria
- `malloc(n)` **busca** un área de memoria contigua de n bytes que esté disponible: crear una variable dinámica tiene un sobrecosto en tiempo de ejecución
- Antes de entregarla, registra su tamaño y la marca como ocupada
- En `free(dir)` el área de memoria que comienza en dir debe estar marcada como ocupada todavía
- El área quedará marcada como disponible

```
free(dir); ...  free(dir); // incorrecto
```

- La función malloc de C cumple la misma función que el operador new de Java (y C++)
- No hay un recolector de basuras en C: **en C hay que destruir las variables dinámicas explícitamente con free**
- Cuando no se invoca free para destruir una variable dinámica que ya no se necesita, esa variable se transforma en un memory leak o goteo de memoria

- La función malloc nunca reutilizará esa memoria
- Cuando se agota la memoria disponible en el heap, malloc solicita al núcleo del sistema operativo la extensión del segmento de datos, haciendo crecer el heap
- En los programas que poseen memory leaks el heap no para de crecer, hasta que el núcleo ya no puede extender el segmento de datos y malloc retorna NULL: la dirección 0
- Típicamente el programa se cae por segmentation fault
- Cuando un programa termina, el núcleo destruye todos sus segmentos, liberando toda la memoria, incluyendo goteras de memoria

8 Referencias colgantes

- Cuando una variable es destruida, es válido que su dirección quede almacenada en un puntero
- Esa dirección se denomina *dangling reference* o referencia colgante
- Por ejemplo la dirección retornada por la función copiar del inicio de esta clase es una referencia colgante
- Acceder al contenido de una referencia colgante es un error porque tal vez esa memoria ya fue atribuida a otra variable y fue modificada
- Al liberar una variable dinámica con `free(ptr)`, la dirección almacenada en ptr no cambia y por lo tanto es una referencia colgante
- Mientras malloc no atribuya esa memoria a otra variable dinámica, el contenido de la variable dinámica no va a cambiar
- Es un error acceder la contenido de ptr o una copia de ptr

9 Herramientas para descubrir errores de manejo de memoria

9.1 Sanitize de gcc (y clang)

- la opción `-fsanitize=address` de gcc genera código adicional para detectar goteras de memoria y uso de referencias colgantes
- En contrapartida el programa corre mucho más lento
- No detecta todos los errores, ¡sea cuidadoso!

9.2 Valgrind

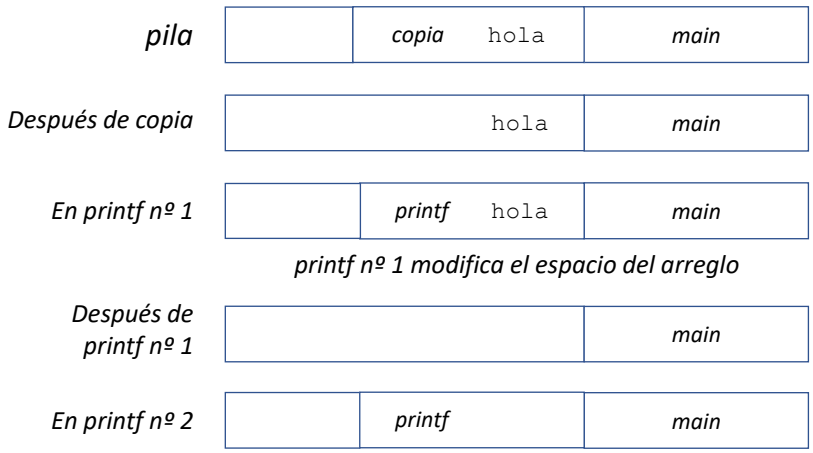
- El comando `valgrind` instrumenta un binario ejecutable compilado con gcc agregando mucho código para verificar el uso correcto de *malloc* y *free* detectando goteras de memoria y uso referencias colgantes asociadas a malloc
- En contrapartida el programa corre aún más lento que con sanitize
- Detecta menos errores que sanitize

10 El recolector de basuras

- Un recolector de basuras libera automáticamente las variables dinámicas que ya no son alcanzables por el programa
- C y C++ no posee recolector de basuras
- *Java* y *Python* sí poseen recolector de basuras
- No hay manera de liberar explícitamente la memoria
- La ventaja es que no pueden haber errores asociados a dangling references
- Casi no hay *memory leaks*
- No se pierde tiempo de desarrollo en descubrir en donde se debe liberar la memoria
- Pero el recolector de basuras introduce un sobrecosto importante en tiempo de ejecución y uso de memoria

- Las implementaciones más eficientes introducen pausas en la ejecución que son molestas en aplicaciones interactivas
- Hay recolectores de basuras que minimizan las pausas pero con un sobrecosto adicional en tiempo de ejecución

11 Explicación del error en copiar



VIII
CLASE 8

1 Agenda de la clase

- Ejemplo de un memory leak y un dangling reference
- Recolección de basuras vs. free
- Typedef, regla de sustitución para typedef
- Estructuras
- Punteros a estructuras
- Estructuras de datos recursivas
- El puntero nulo

2 Definición de alias para tipos: Typedef

- La sintaxis: `T id1, *id2, ...` declara que `id1, id2, ...` son variables que almacenan valores del tipo `T, T*, ...`
- Considere que a esta declaración se antepone typedef: `typedef T id1, *id2, ...`
- Ahora los identificadoresid1, `id2, ...` ya no son variables: ¡Son alias tipos!
- Por ejemplo: `typedef int Ent, *P_Ent;`
- `Ent` y `P_Ent` son tipos
- ¿Cuáles son los tipos concretos de x, y, z?

```
Ent x, y[10];

P_Ent z;
```

- Regla de substitución para `typedef`: para saber el tipo de `y[10]` substituya `Ent` en el `typedef` por `y[10]` y suprima el `typedef` y los otros tipos declarados
- Queda: `int y[10]; Ent` es un alias de `int`
- Por lo tanto y es un arreglo de 10 enteros
- `P_Ent` es un alias de int*

3 Estructuras

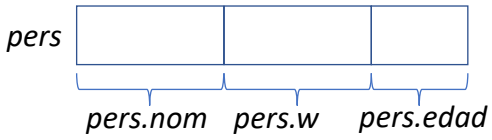
- Las estructuras son variables de tipo compuesto: almacenan múltiples valores a la vez
- Se declaran con struct etiqueta `{ decl1 decl2 ... } ...`
- Ej.:

```
struct persona {char *nom; double w; int edad;};

struct persona pers;
```

- La variable `pers` es una estructura de tipo `struct persona` que almacena 3 valores de tipos distintos
- `nom w edad` son los campos (fields) de la estructura
- Para seleccionar cada uno de los campos de pers se usa esta sintaxis: `pers.nom pers.w pers.edad`

- Son variable del tipo declarado en `struct` ...



- Con `pers.nom` se puede hacer todo lo que se puede hacer con una variable de tipo `char*` como `&pers.nom`

4 Estructuras y typedef

- Como usar struct persona es muy largo conviene usar typedef para crear un alias:

```
typedef struct persona Persona;
Persona pedro, ana; // tipo: struct persona
```

- También se puede usar directamente en la definición de struct persona:

```
typedef struct persona {
    char *nom;
    double w;
    int edad;
} Persona;
```

```
struct persona pedro; // o
Persona ana;
```

- Si nunca se va a usar persona otra vez se puede omitir:

```
typedef struct {
    char *nom;
    double w;
    int edad;
} Persona;
```

Código 7: La convención usual es que la primera letra de un alias correspondiente a un struct es mayúscula (`Persona`)

5 Ejemplo: números complejos

- Tipo: `typedef struct { double r, im; } Complejo;`
- Función que suma números complejos:

```
Complejo suma(Complejo zx, Complejo zy) {
    Complejo res;
    res.r= zx.r + zy.r;
    res.im= zx.im+zy.im;
    return res;
}
```

Código 8: Las estructuras se pueden retornar y pasar como parámetros

- O más corto

```
Complejo suma(Complejo zx, Complejo zy) {
    Complejo res = { zx.r + zy.r, zx.im + zy.im };
    return res;
}
```

Código 9: En la declaración se usa {...} para inicializar los campos de una estructura

- Uso:

```
Complejo a = { 1.3, -10 }, b = { -0.03, 0 };
Complejo c = suma(a, b);
```

- Las estructuras se pueden declarar con inicialización, asignar, pasar como parámetros a una función y ser retornadas, tienen sizeof y dirección.

6 Enfoque imperativo: punteros a estructuras

- Esto **No funciona**

```
void sumar(Complejo zx, Complejo zy) {
    zx.r += zy.r;
    zx.im += zy.im;
}
```

- Uso:

```
Complejo a = { 1.3, -10 }, b = { -0.03, 0 };
sumar(a, b); // ¡a sigue siendo { 1.3, -10 }!
```

- **Porque los parámetros se pasan por valor en C: zx es una copia de a, modificar zx no cambia a**

- Se debe usar puntero:

```
void sumar(Complejo *pz, Complejo zy) {
    (*pz).r += zy.r; // ~~*pz.r += ...~~
    (*pz).im += zy.im; // ~~*(pz.r) += ...~~
}
```

- Uso:

```
Complejo a = { 1.3, -10 }, b = { -0.03, 1 };
sumar(&a, b); // a es { 1.27, -9 }
```

7 Sabor sintáctico

- La expresión `(*p).campo` es tan necesitada que existe un sabor sintáctico equivalente más liviano y legible:

  `p->campo`

- Reescritura legible de la función sumar:

```
void sumar(Complejo *pz, Complejo zy) {
    pz->r += zy.r;
    pz->im += zy.im;
}
```

- Si `p` es un puntero a una estructura, acceda a sus campos con `p->campo`
- Si `e` es una estructura, acceda a sus campos con `e.campo`

```
expresión -> identificador
expresión.Identificador
suma(a, b).r // Correcto
zy."r" // Incorrecto
```

8 Estructuras de datos recursivas

- Estructuras de tipo T cuyos campos referencian variables del mismo tipo T son recursivas
- Ejemplo: un nodo de una lista simplemente enlazada es recursivo porque uno de sus campos es un puntero a otros nodos de la lista
- Declaración:

```
typedef struct nodo {
char *str;
struct nodo *prox;
} Nodo;
```

Código 10: `nodo` no es opcional.

- Forma incorrecta:

```
typedef struct {
    char *str;
    Nodo *prox; // Error tipo Nodo no existe aún
} Nodo;
```

- El alias Nodo se hace visible después del `typedef`

9 El puntero nulo

- Función que busca el string `pal` en la lista enlazada cuyo primer nodo es apuntado por cabeza:

```
int buscar(Nodo *cabeza, char *pal) {
    while (cabeza!=NULL) {
        if (strcmp(cabeza->str, pal)==0)
            return 1;
        cabeza= cabeza->prox;
    }
    return 0;
}
```

- NULL es la dirección 0
- Si cabeza es NULL, `cabeza->str` gatilla *segmentation fault*
- Uso: `int presente= buscar(L, "casa");`
- ¡L no cambia! Porque el paso de parámetros es por valor

IX
CLASE 9

1 Agenda de la clase

- Declaración de punteros con inicialización
- Cómo cambiar parámetros
- Punteros a punteros
- Punteros a punteros a estructuras
- Punteros a funciones: invocación y declaración
- Typedef para punteros a funciones

2 Declaración de punteros con inicialización

- Considerando esta declaración: `int a, *p = &a;`
- ¿Qué variable se está inicializando? ¿p o a?
- Respuesta: ¡p!
- La sintaxis es engañosa: ¿por qué?
- El significado de $*$ cambia según el contexto:
  a) como operador binario es la multiplicación: a $*$ b
  b) como operador unario es el operador de contenido: `*p`
  c) En una declaración, un $*$ antes del identificador que se está declarando es un modificador de tipo, **no es el operador de contenido**, señala que el identificador es un puntero
- La declaración `int *p` significa que cuando aparezca `*p` en una expresión, su tipo será `int`
- ¿Cuando vale c? `int a=10, *p= &a, c=*p*a;`

3 Cómo cambiar parámetros

- Función que intercambia valores de 2 variables

```c
// Versión incorrecta
void swap(int x, int y) {
    int tmp= x;
    x= y;
    y= tmp;
} // ¡No haga esto!

// Uso
int main() {
    int a= 1, b= 2;
    swap(a, b);
    printf("%d %d\n", a, b);
} // Resultado: 1 2
```

Código 11: Incorrecto

```c
// Versión correcta
void swap(int *px, int *py) {
    int tmp= *px;
    *px= *py;
    *py= tmp;
}

// Uso
int main() {
    int a= 1, b= 2;
    swap(&a, &b);
    printf("%d %d\n", a, b);
} // Resultado: 2 1
```

Código 12: Correcto

- ¿Función que intercambia valores de 2 punteros?

```c
int a= 1, b= 2, *p= &a, *q= &b;
swap_ptr(&p, &q);
printf("%d %d %d %d\n", a, b, *p, *q);
// Esperado:1 2 2 1
```

4 Punteros a punteros

- Función que intercambia valores de 2 variables

```c
// Versión correcta
void swap_ptr(int **px, int **py) {
    int *tmp= *px;
    *px= *py;
    *py= tmp;
}
```

- Un puntero a un puntero es una variable que contiene direcciones de punteros
- El modificador de tipo en la declaración es `**`
- ¿Cual es el tipo de la expresión `*px`?
- Respuesta: `int *` porque la declaración `int * *px` dice que en una expresión el tipo de `*px` es `int *`

5 Punteros a punteros a estructuras

- Función que elimina el primer nodo de una lista simplemente enlazada

```c
void elim(Nodo *cabeza) {
    Nodo *rem= cabeza;
    cabeza= cabeza->prox;
    free(rem);
} // Versión incorrecta

// Uso
Nodo *h= ...;
elim(h);
// h es dangling reference
```

Código 13: Incorrecto

```c
void elim(Nodo **pcabeza) {
    Nodo *cabeza= *pcabeza;
    *pcabeza= cabeza->prox;
    free(cabeza);
} // Versión correcta

// Uso
Nodo *h= ...;
elim(&h);
```

Código 14: Correcto

- Ejercicio: función que agrega un nodo al comienzo de una lista simplemente enlazada

```
// Uso
Nodo *h= ...;
agregar(&h, "gato");
```

6 Punteros a funciones: Motivación

- Función que calcula numéricamente la integral de f usando el método de los trapecios.
- Se puede aproximar el área de la curva como la suma de las áreas de n trapecios. Esta es la fórmula:

$$\int_{x_i}^{x_f} f(x)dx \approx h \cdot \left[\frac{f(x_i) + f(x_f)}{2} + \sum_{k=1}^{n-1} f(x_i + k \cdot h) \right] \tag{9.1}$$

Donde $h = \dfrac{x_f - x_i}{n}$

- Implementación:

```
double integral(double xi, double xf, int n) {
    double h= (xf-xi)/n;
    double sum= ( f(xi) + f(xf) ) / 2;
    for (int k= 1; k<n; k++)
        sum += f(xi + k*h);
    return sum * h;
}
```

- Si ahora se necesita la integral de g hay que reescribir la función integral cambiando f por g.
- Idea: pasar la función como parámetro

7 Punteros a funciones: invocación

- El encabezado para la función f es:

  ```
  double f(double x);
  ```

- El identificador `f` representa la dirección de la primera instrucción de máquina de la función
- ¿Cómo se declara un puntero `pf` que almacena la dirección de una función?
- Primero hay que pensar en cómo se invoca la función almacenada en `pf`
- Los diseñadores de C argumentaron que si se escribe `*p` para usar la variable a la cual apunta un puntero `p`, entonces debería ser lo mismo para invocar la función almacenada en `pf`:

  ```
  double x= 5.0, fx= *pf(x); // No sirve: * ( pf(x) )
  ```

- Problema: el operador `( )` tiene mayor precedencia que `*`
- Solución: parentizar

  — `double x= 5.0, fx= (*pf)(x); // ¡Forma correcta!`

8 Declaración de un puntero a una función

- El encabezado para la función f y g es:

  ```
  double f(double x), g(double x);
  ```

- Dice que en una expresión el tipo de la invocación $f(x)$ es `double`, en donde `x` es de tipo `double`

- La declaración del puntero `pf` es:

```
double (*pf)(double x);
```

- Dice que en una expresión el tipo de la invocación `(*pf)(x)` es double, en donde `x` es de tipo `double`
- Entonces:

```
pf= f; // pf apunta a f
double fx5= (*pf)(5.0); // Invoca f(5.0)
pf= g; // pf apunta a g
double gx2= (*pf)(2.0); // Invoca g(2.0)
```

- `pf` no debe apuntar a funciones con un encabezado distinto del de f y g

9 Función integral genérica

- Implementación

```
double integral(double (*pf)(double x),
                double xi, double xf, int n) {
    double h= (xf-xi)/n;
    double sum= ( (*pf)(xi) + (*pf)(xf) ) / 2;
    for (int k= 1; k<n; k++)
        sum += (*pf)(xi + k*h);
    return sum * h;
}
```

- Horrible:

```
double poli(x) { // Ejemplo de uso
    return 4.5*x*x-10*x+3.1;
}
int main( ) {
    printf("%f\n", integral(poli, 0.0, 10.0, 100));
    return 0;
}
```

10 Typedef para punteros a funciones

- ¿Qué pasa si agregamos typedef a la declaración de un puntero a una función?

```
typedef double (*Fun)(double x);
```

- `Fun` ya no es puntero a una función: es el tipo de los punteros a funciones que reciben un parámetro real y retornan un real
- La función integral se puede reescribir:

```
double integral(Fun pf,
                double xi, double xf, int n) {
...
}
```

11 Más integrales

- Se desea programar la función:

```
double h(double a, double b, double c,
         double xi, double xf, int n);
```

- tal que:

$$h(a,b,c,x_i,x_f,n) \approx \int_{x_i}^{x_f} ax^2 + bx + cdx \tag{9.2}$$

Aproximado con n trapecios

- ¿Se puede usar integral? ¿Hay manera de pasarle los valores de a, b y c?
- Se desea programar la función

```
double (*Fun2) (double x, double y);
double e(double xf, double yf, Fun2 g, int n);
```

- tal que:

$$e(x_f,y_f,g,n) \approx \int_0^{y_f} \int_0^{x_f} g(x,y)dxdy \tag{9.3}$$

Aproximado con n trapecios

- ¿Sirve integral?

12 Solución incorrecta para h

```
double poli2(double x) {
    return a*x*x+b*x+c; // MAL
}
double h(double a, double b, double c,
         double xi, double xf, int n) {
    return integral(poli2, xi, xf, n);
}
```

- Problema: No compila
- Las variables a, b y c no son visibles en `poli2`.

X

CLASE 10

1 Agenda de la clase

- Typedef para punteros a funciones
- Motivación: función integral genérica
- Variables globales
- Scope
- Cast de punteros
- Punteros opacos
- Tipo estático vs. tipo dinámico
- Programación de la función integral genérica
- Conclusiones

2 Scope

- Definición: Alcance, visibilidad o en inglés *scope* de un identificador de variable corresponde a la región del código fuente en donde ese identificador es conocido
- Usar un identificador fuera de su alcance es equivalente a usar un identificador que no está declarado
- La visibilidad de una variable local es el código que va inmediatamente después de su aparición en la declaración hasta que se cierra el bloque en que fue declarada
- El identificador de una variable local puede no ser visible, pero la variable sí podría estar viva
- Si se declara una variable con identificador id, pero ya existía otra variable declarada con el mismo identificador id en un bloque distinto, el identificador previo deja de ser visible hasta que termine la visibilidad del nuevo identificador
- Dado que una variable dinámica no tiene identificador, no tiene sentido hablar de su visibilidad

3 Variables globales

- **Definición**: Si una variable se declara fuera de toda función, es una variable global
- Ejemplo: esta solución para la función h es correcta

```
double g_a, g_b, g_c; // variables globales
double poli2(double x) {
    return g_a*x*x+g_b*x+g_c; // Check
}
double h(double a, double b, double c,
         double xi, double xf, int n) {
g_a = a; g_b = b; g_c = c;
return integral(poli2, xi, xf, n);
}
```

- `g_a`, `g_b`, `g_c` son variables globales
- Tiempo de vida: se crean al inicio de la ejecución del programa y se destruyen cuando este termina

- Scope: el código que va inmediatamente después de su aparición en la declaración hasta que termina el archivo en el que fue declarada
- Si es posible, evite usar variables globales

4 Más sobre scope de variables globales

- Si se antepone el atributo `static`, como en:

```
static double global_var;
```

- la variable no es visible desde otros archivos
- Si no lleva `static`, la variable sí puede ser visible desde otro archivo si en ese otro archivo se redeclara con el atributo `extern`

```
extern double global_var
```

- Típicamente el `extern` se usa en archivos de encabezados (.h)

5 Inicialización de variables globales

- Las variables globales se puede inicializar pero solo con valores constantes
- Por ejemplo:

```
int n= 100;
double pi= 3.14159;
```

- Más precisamente se puede inicializar con una expresión que el compilador pueda calcular

```
int m= 2*100;
int o= 100/sizeof(int);
```

- No se puede:

```
double c= sin(1.0); // Mal
double pi2= pi; // Mal
```

6 Cast de punteros

- Un cast de punteros permite cambiar el tipo de la variable a la que apunta un puntero
- Ejemplo:

```
double pi= 3.14159;
double *ptr_double= &pi;
int *ptr_int= (int*)ptr_double; // ✓ Cast de punteros
```

- Si no se coloca el `cast`, el compilador reclama con un warning y coloca automáticamente el cast
- Las direcciones almacenadas en `ptr_int` y `ptr_double` son idénticas
- No se realiza ningún tipo de conversión de los datos
- ¿Cuánto vale `*ptr_int`?
- Respuesta: Ni cerca de 3, es basura en realidad
- Es "legal" usar $ptr_int[0]$ y $ptr_int[1]$ pero rara vez útil

7 Tipo estático vs. tipo dinámico

- **Definición**: el tipo estático de un puntero es el tipo declarado en tiempo de compilación
- Ejemplos

  el tipo estático de `ptr_double` es `double*`

  el tipo estático de `ptr_int` es `int*`

- El tipo estático de un puntero no cambia jamás
- **Definición**: Si en un instante dado la última variable almacenada en la dirección `dir` fue de tipo T y un puntero `ptr` contiene la dirección `dir`, el tipo dinámico de `ptr` en ese instante es
- Ejemplo:

  el tipo dinámico de `ptr_int` es `double*`
- El tipo dinámico de un puntero sí cambia durante la ejecución
- Típicamente coincide con el tipo estático, pero puede diferir

8 Versión genérica de integral

- La función integral recibe un puntero `ptr` que incluirá como argumento en todas las invocaciones de `*pf`
- El usuario de `integral` suministra en `*ptr` cualquier otro parámetro que necesite para evaluar la función

```
typedef double (*Fun)(void *ptr, double x);
double integral(Fun pf, void *ptr,
                double xi, double xf, int n) {
    double h= (xf-xi)/n;
    double sum= ( (*pf)(ptr, xi) + (*pf)(ptr, xf) ) / 2;
    for (int k= 1; k<n; k++)
        sum += (*pf)(ptr, xi + k*h);
    return sum * h;
}
```

- Un puntero de tipo `void*` es un puntero opaco: no se sabe nada acerca del tipo de la variable a la que apunta
- El compilador no permite usar `*` o `->` con un puntero opaco
- ¡La función `malloc` retorna un puntero opaco!
- Cumple la misma función que Object en Java

9 Versión preferida para la función h

```
    typedef struct  double a, b, c;  Abc;
    double poli2(void *ptr, double x) {
        Abc *pabc= (Abc*)ptr; // Nota b)
        return pabc->a*x*x+pabc->b*x+pabc->c;
    }
    double h(double a, double b, double c,
             double xi, double xf, int n) {
    Abc abc= { a, b, c };
    return integral(poli2, &abc, xi, xf, n); // Nota a)
    }
```

Notas

a) Se permite asignar cualquier dirección a un puntero opaco, no es causa de errores

b) El cast es opcional cuando el puntero es `void*`

```
Abc *pabc= ptr; // ✓
// solo porque ptr es void *
```

Si el tipo dinámico de `ptr` no es Abc*, el resultado de pabc->a no está especificado

10 Conclusiones

- Cuando el tipo dinámico de `p` no coincide con su tipo estático, es un error leer la variable apuntada con `*p` o `p->`
- Durante la ejecución no se hace ningún chequeo para detectar este error

- Su ocurrencia se puede traducir en resultados incorrectos o segmentation fault
- **Es completa responsabilidad de los programadores evitar este tipo de errores**
- Es la lógica del programa la que ayuda a evitar estos errores
- Sí se puede escribir: ¡La asignación `*p = ...` sirve para cambiar el tipo dinámicos de p!
- Los casts de punteros junto a la aritmética de punteros le dan flexibilidad al lenguaje C, haciendo posible la programación orientada a objetos en C, la programación de colecciones genéricas, la programación en C de las funciones malloc y free, la programación de núcleos de sistemas operativos, etc.
- La gran desventaja es la ausencia del chequeo de tipos

11 Ejercicio desafiante

- Se desea programar la función:

```
typedef double (*Fun2) (double x, double y);
double e(double xf, double yf, Fun2 g, int n);
```

Aproximado con n trapecios

- ¡Usar integral!

$$e(x_f, y_f, g, n) \approx \int_0^{y_f} \int_0^{x_f} g(x, y) dx dy \tag{10.1}$$

XI
CLASE 11

1 Agenda de la clase

- Big endian vs. little endian
- Alineamiento
- Una cola genérica
- Cast entre enteros y punteros
- Unboxing vs. boxing

2 Resumen clase pasada

- Gracias a la aritmética de punteros y los cast de punteros, en C se puede almacenar datos de cualquier tipo en cualquier parte de la memoria del programa
- El tipo dinámico de una dirección de memoria es el tipo del último dato que se escribió en esa dirección
- Al leer ese dato por medio de un puntero es imprescindible que el tipo estático de su contenido sea idéntico al tipo dinámico de lo apuntado
- Pero abusar de los cast de punteros hace los programas muy frágiles y difíciles de depurar
- La mayoría del código no necesita los cast de punteros
- El sistema de tipos de C es un cinturón de seguridad que nos protege de errores accidentales en el manejo de punteros, pero no es una camisa de fuerza
- Los cast de punteros sirven en los pocos casos en donde el sistema de tipos de C es demasiado restrictivo, como en el caso de la función genérica para calcular integrales numéricamente, para las colecciones genéricas (mapas, colas, colas de prioridad, etc.), algoritmos genéricos (ordenamiento), etc.

3 Big endian vs. little endian

- ¿Qué valor retorna esta función?

```
int isLittleEndian() {
    int x= 1;
    char *p= (char*)&x;
    return p[0];
}
```

- Relación entre x y p:

- Depende del procesador: si es little endian significa que la dirección de una variable entera es la del byte menos significativo (little end) y por lo tanto retorna 1
- Si es big endian la dirección de una variable entera es la del byte más significativo (big end) y por lo tanto retorna 0
- X86, Arm y Risc-V son little endian
- Los procesador de IBM, Motorola y Sparc son big endian

4 Historia

- Viene de los viajes de Gulliver de Jonathan Swift (1726): los big endians son un grupo de gente de Lilliput que sostiene que los huevos duros se deben romper por el extremo más grande en vez del más pequeño, como lo ordenó el emperador de Lilliput (cita)
- Si un procesador es little o big endian es irrelevante desde el punto del área del chip, la frecuencia o el consumo: no es más caro ni más barato
- Pero sí es relevante si datos en formato binario se envían por la red o se almacenan en archivos: si por ejemplo un procesador little endian envía por la red un dato binario y lo recibe un procesador big endian, el valor será incorrecto si no se hace la conversión
- Sun Microsystems (ahora una división de Oracle) fue la empresa pionera de Internet y ordenó que todos los datos binarios transmitidos por Internet debían estar en formato big endian
- No hay un estándar para los archivos binarios

5 Alineamiento: ¿Qué muestra este programa?

```
typedef struct {
    char c;
    int n;
} U;

int main() {
    U *u= malloc(sizeof(U));
    printf("%p\n", u);
    printf("%ld\n", sizeof(U));
    printf("desplazamiento de c=%ld\n",
                            (char*)&u->c-(char*)u);
    printf("desplazamiento de n=%ld\n",
                            (char*)&u->n-(char*)u);
}
```

Salida:

```
0x56097c3262a0
8       // ¡Para garantizar el alineamiento de n!
desplazamiento de c=0
desplazamiento de n=4
```

6 Alineamiento

- Definición: Cuando las variables de tipos primitivos (int, short, double, etc.) y los punteros están alineados su dirección es múltiplo de su tamaño en bytes
- En algunas arquitecturas de procesadores acceder a datos no alineados gatilla un bus error
- En el resto de las arquitecturas, acceder a datos no alineados solo es menos eficiente porque requiere 2 accesos a la memoria, en vez de uno
- Los compiladores garantizan que variables locales y campos en estructuras locales que sean de tipo primitivo están alineados: el puntero a la pila siempre es una dirección múltiplo de 8 (al menos)
- Lo mismo ocurre con variables o arreglos dinámicos creados con malloc: las direcciones que retorna son siempre múltiplo de 8 (al menos)
- Si sizeof(U) fuese 5, no habría manera de alinear n
- Con el uso de cast de punteros y aritmética de punteros se puede hacer que un puntero direccione una variable no alineada

7 ¿Qué muestra este programa?

```
typedef struct {
    int n;
    char c;
} T;
```

```
    int main() {
        T *t= malloc(sizeof(T));
        printf("%p\n", t);
        printf("%ld\n", sizeof(T));
        printf("desplazamiento de c=%ld\n",
                                (char*)&t->c-(char*)t);
        printf("desplazamiento de n=%ld\n",
                                (char*)&t->n-(char*)t);
}
```

Salida:

```
0x56097c3262a0
8 // ¡Para garantizar el alineamiento
// de n en arreglos de T!
desplazamiento de c=4
desplazamiento de n=0
```

8 Más sobre cast de punteros

- En clase auxiliar se implemento el tipo Queue:

  ```
  Queue *makeQueue(); // Queue *q = makeQueue();
  void put(Queue* q, char *str); // put(q, "perro");
  char *get(Queue* q); // char *elem= get(q);
  void freeQueue(Queue *q); // freeQueue(q);
  ```

- ¿Encolar enteros?

```
int a= 5, b= 7, c= -3;
Queue *q= makeQueue();
put(q, &a); put(q, &b); put(q, &c); // × no compila
put(q, (char*)&a); put(q, (char*)&b); put(q, (char*)&c);
int *pa= (int*)get(q); // ✓
printf("%d\n", *pa); // ✓ ¡5!
```

- Funciona porque Queue nunca accede al contenido de los punteros depositados y el tipo estático de pa es consistente con su tipo dinámico

9 Una cola genérica

- Modificar Queue:
  - `Queue *makeQueue(); // Queue *q = makeQueue();`
  - `void put(Queue* q, void *ptr); // put(q, "perro");`
  - `void *get(Queue* q); // char *elem= get(q);`
  - `void freeQueue(Queue *q); // freeQueue(q);`

9.1 ¿Encolar enteros?

```
int a= 5, b= 7, c= -3;
Queue *q= makeQueue();
put(q, &a); put(q, &b); put(q, &c); // ✓ sí compila
int *pa= get(q); // ✓ ¡sin cast!
printf("%d\n", *pa); // ✓ ¡5!
```

- Se puede depositar cualquier cosa que sea un puntero
- Pero al extraer el contenido se debe usar un cast de puntero que coincida con el tipo dinámico de lo depositado
- Problema: ¿Qué pasa si se destruyen a, b o c?
- Los punteros almacenados en la cola se vuelven dangling references

10 ¡Horror!

```
    Queue *qabc(int a, int b, int c) { // ¡No haga esto!
        Queue *q= makeQueue(); // Es dinámica: se crea con malloc
        put(q, &a); put(q, &b); put(q, &c);
        return q;
    }
    int main() {
        Queue *q= qabc(5, 7, -3);
        int *pa= get(q); // ✓ Hasta aquí vamos bien
        printf("%d\n", *pa); // × pa es dangling reference
}
```

```
int *makeInt(int n) { // Crea un entero en el heap: Boxing
    int *p= malloc(sizeof(int));
    *p= n;
    return p;
}
Queue *qabc(int a, int b, int c) { // Versión correcta
    Queue *q= makeQueue();
    put(q, makeInt(a)); put(q, makeInt(b)); put(q, makeInt(c));
    return q;
}
```

11 Cast entre enteros y punteros

```
Queue *qabc( ) { // Unboxing
    int a= 5, b= 7, c= -3;
    Queue *q= makeQueue(); // Es dinámica: se crea con malloc
    put(q, (void*)a); put(q, (void*)b); put(q, (void*)c); // ?
    return q;
}
int main() {
Queue *q= qabc();
int a= (int)get(q); // ?
printf("%d\n", a); // ✓ 5
...
}
```

- Unboxing: El cast (`void*`) disfraza un entero como si fuese un puntero
- warning: cast from pointer to integer of different size
- Hay warning solo si los punteros son de 64 bits
- No hay warning si los punteros son de 32 bits
- Pero ejecuta correctamente

12 Manera correcta de casts entre enteros y punteros para el unboxing

```
#include <stdint.h>
Queue *qabc( ) {
    intptr_t a= 5, b= 7, c= -3;
    Queue *q= makeQueue(); // Es dinámica: se crea con malloc
    put(q, (void*)a); put(q, (void*)b); put(q, (void*)c); // ✓
    return q;
}
int main() {
    Queue *q= qabc();
    int a= (intptr_t)get(q); //  ✓
    printf("%d\n", a); //  ✓5
}
```

- Se garantiza que el tipo `intptr_t` es del mismo tamaño que un puntero
- Está definido en stdint.h
- No hay warning ya sea que los punteros sean de 16, 32 o 64 bits
- No hay una manera estándar de disfrazar el tipo double

13 Comparación de unboxing vs boxing

- Unboxing es más cómodo y eficiente porque se evita el malloc y el free
- Pero es menos legible: no todos los programadores van a entender qué se está haciendo
- En los lenguajes con tipos dinámicos, como Python, las variables pueden almacenar valores de cualquier tipo
- Todas las variables son de tipo `void*` u Object (Java)
- La implementación estándar es que una variable es un puntero a una estructura en el heap de memoria, cuyo primer campo es una etiqueta con el tipo del valor almacenado y el siguiente campo es el valor
- En cada operación se chequea el tipo de los operandos
- Boxing: los enteros también se representan con punteros al heap, lo que es muy ineficiente
- Unboxing: los enteros (y reales opcionalmente) se almancenan directamente en el puntero de la variable
- Hay trucos para etiquetar los punteros que almacenan enteros y así distinguirlos del resto de los tipos de datos

XI
CLASE 11-2: CLASE EXTRA

1 Aclaración

"La explicación acerca de cómo funciona sort está en este video, pero no lo recomiendo porque es bastante complicado. Es la razón por la que este semestre decidí usar la integral como ejemplo de algoritmo genérico, en vez de sort." -Luis Mateu

2 Agenda de la clase

- Repaso: punteros a funciones y punteros opacos
- Repaso: tipo estático vs tipo dinámico
- Función genérica para ordenar arreglos de punteros
- Ordenar arreglo de punteros a personas descendentemente por edad
- Ejercicio: ordenar arreglo de punteros a personas ascendentemente por rut
- Ordenar arreglo de personas (no punteros a personas)
- Código de función genérica que ordena arreglos de cualquier tipo y criterio.

XII
CLASE 12

- Temario

  Funciones de C para manipular archivos: abrir y cerrar archivos (fopen), lectura y escritura de archvos de texto (fgetc, fgets, fputc, fputs, fprintf, fscanf) y otras (feof, ferror, fseek), entrada estándar, salida estándar y salida estándar de errores, archivos binarios (fread, fwrite), acceso directo a archivos (fseek).

1 Código usado en clases

Disponible aquí

XIII
CLASE 13

Aclaración: "para entender esta clase se requiere considerar lo siguiente. Hay 2 problemas con la clase grabada. El primero es que hace referencia a una versión genérica de quicksort que ordena cualquier tipo de arreglo/archivo, que se vio en semestres anteriores, pero no se vio este semestre. No se preocupen por esto, no tienen que entender como funciona este quicksort. Lo único que tienen que entender es cómo se usa. Este es el encabezado de la función:"

```
typedef int (*Comparator)(void *ptr, int i, int j);
typedef void (*Swapper)(void *ptr, int i, int j);

void sort(void *ptr, int left, int right, Comparator compare, Swapper swap) {
}
```

La función recibe un puntero `ptr` a un "objeto" de cualquier tipo que se pueda subindicar, un índice inicial left, un índice final right, una función que compara el i-ésimo elemento del objeto con el j-ésimo (retorna < 0, 0, > 0) y otra función que intercambia los elementos i y j. La función sort usa quicksort para ordenar el "objeto" entre los índices left y right. En el ejemplo de esta clase, el "objeto" es un archivo de personas y el i-ésimo elemento es la i-ésima persona en el archivo.

El otro problema con esta clase grabada es que cuando se invoca finalmente sort, se usan cast de funciones:

```
sort(archPers, 0, numPers-1,  (Comparator)cmpPers,  (Swapper)swapPers );
```

Es muy complicado explicar por qué es correcto en este caso específico. Pero su uso puede llevar a errores inesperados, así es que recomiendo no usar los cast de funciones. Por esa razón en la solución incluida en los archivos adjuntos no se usan estos cast de funciones. La invocación de sort es:

```
sort(archPers, 0, numPers-1, cmpPers, swapPers);
```

pero para que no reclame el compilador tuve que cambiar los encabezados de `cmpPers` y `swapPers` a:

```
void swapPers(void *ptr, int i, int j);
int cmpPers(void *ptr, int i, int j);
```

La función sort es un ejemplo de algoritmo genérico, que depende de operaciones que se reciben como parámetros, al igual que la función integral vista en cátedra o los tipos de datos abstractos Queue y Map de la última clase auxiliar.

La explicación acerca de cómo funciona sort está en este video, pero no lo recomiendo porque es bastante complicado. Es la razón por la que este semestre decidí usar la integral como ejemplo de algoritmo genérico, en vez de sort.

Con esta aclaración deberían entender la clase grabada: `https://www.youtube.com/watch?v=x1WrvknGoIc&feature=youtu.be`

1 Código usado en clases

Disponible aquí

AUXILIARES

En adelante se tomarán notas de las clases auxiliares.

1 AUXILIAR 1

1 Introducción

Este auxiliar introductorio nos responderá algunas preguntas básicas.

2 Diferencias importantes entre Python y C

C, junto a Java y otros lenguajes son algunos de varios lenguajes compilados, no interpretados.

3 ¿Cómo se usa el compilador de C? (gcc)

```
gcc -std=c99 helloworld.c -o ejecutable
ls
./ejecutable
```

Recordemos tener los warnings activados, por ejemplo:

```
gcc -Wall helloworld.c
```

4 ¿Qué se puede hacer con un debugger?

Un debugger ejecutará un problema paso a paso.

5 Torpedo GDB

```
gdb <archivo binario>
```

Código 15: Invocar gdb

```
help <comando>
```

Código 16: Obtener ayuda

```
(gdb) b main
```

Código 17: Poner breakpoints en funciones

```
(gdb) del <nro del break>
```

Código 18: Borrar breakpoints

```
(gdb) del <nro del break>
```

Código 19: Correr el programa

6 Nuestro primer Hello World!

```
#include <stdio.h>

int main() {
    print("Hello world\n")
}
```

Código 20: Nuestro primer código en C

7 Convertidor de temperaturas

```
#include <stdio.h>

int main() {
    int grados_celsius = 23;
    int grados_fahrenheit = (grados_celsius * 9)/5 + 32;
    printf("son %d grados fahrenheit\n", grados_fahrenheit);
    return 0;
}
```

8 Factorial 1

```
#include <stdio.h>

int factorial(int num) {
    if (num <= 1) {
        return 1;
    }
    return num * factorial(num-1);
}

int main() {
    int resultado = factorial(9);
    printf("%d\n", resultado);
}
```

9 Factorial 2

```c
#include <stdio.h>
#include <stdlib.h>

int factorial(int num);

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Error: debe dar el número para calcular el factorial\n");
        return 1;
    }
    int num = atoi(argv[1]);
    int resultado;
    resultado = factorial(num);
    printf("la primera vez, da %d\n", resultado);
    // calculémoslo de nuevo, solo por si acaso
    int resultado_2 = factorial(num);
    printf("la segunda vez, fact(%d) da %d\n", num, resultado_2);
}

int factorial(int num) {
    if (num <= 1) {
        return 1;
    } else {
        int factorial_prev;
        factorial_prev = factorial(num - 1);
        return num * factorial_prev;
    }
}
```

2

AUXILIAR 2: BITS

1 ¿Bits?

- Dentro de un computador, todo se representa en binario
- Unos y ceros
- Cada dígito (uno o cero) se llama bit

2 Operaciones de bits

Recordando la Clase 4 en la sección 1

- AND lógico &
- OR lógico |
- NOT lógico ∼
- Desplazamiento a la derecha >>
- Desplazamiento a la izuierda <<

Símbolo	Operación	Significado	Ejemplo (1100 op 1010)
&	and	**y** bit a bit	1000
|	or	**o** bit a bit	1110
∧	xor	**o exclusivo** bit a bit	0110

Símbolo	Operación	Significado	Ejemplo (10110111)
∼	not	negación bit a bit	01001000

Símbolo	Operación	Significado	Ejemplo (10111011 op 2)	Equivalencia
<<	shift left	desp. de bits a la izquierda	11101100	$x<<y \;\; x \cdot 2^y$
>>	unsigned shift right	desp. de bits a la derecha	00101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$
>>	signed shift right	desp. de bits a la derecha	11101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$

3 Hexadecimal

- Ejemplo: Convertir 473 a binario

  Respuesta: 110101011
- Necesitamos una forma más cómoda de escribir constantes, la respuesta es hexadecimal
- Se traduce literalmente (símbolo a símbolo)
- Un dígito hex equivale a 4 dígitos en binario
- Los caracteres de hex son 1-9 y A-F

3.1 Hexadecimal: Ejemplo

- ¿Cómo se escribe en binario en número hexadecimal `0x38C6`?. Veamos

$$\underset{}{\texttt{0x}}\ \underbrace{\texttt{3}}_{0011}\ \underbrace{\texttt{8}}_{1000}\ \underbrace{\texttt{C}}_{1100}\ \underbrace{\texttt{6}}_{0100}$$

El `0x` significa "Lo que sigue es un número hexadecimal" por convención, en C y otros lenguajes

4 Ejercicios (Preguntas 1-3)

P1. Cree una función llamada `bits1` que dado un número `n` , retorna el número de bits en 1 que tiene `n`. Como pista, su declaración debería ser `int bits1(unsigned int n)`.

P2. Programe la siguiente función: `int posicionBits(unsigned x, unsigned p, int n);` [1]

Esta función busca en `x` el patrón `p` de `n` bits entregando su posición expresada como la posición del bit menos significativo de `p` en `x` , o -1 si no se encuentra

P3. Programe la siguiente función: [2]

```
unsigned repBits(unsigned x, int i, int k, unsigned val);
```

Esta función retorna el resultado de reemplazar `k` bits en `x` , a partir del `i`-ésimo bit de `x` hacia la izquierda, por el valor `val`.

[1] (P2a C1 Otoño 2014)
[2] (P2b C1 Otoño 2013)

Solución. (**P1**, **P2**, **P3**)

```c
#include <stdio.h>

int bits1(int n);
int posicionBits(int x, int p, int n);
unsigned repBits(unsigned x, int i, int k, unsigned val);

int main() {
        int bits1_n = 0x2784; // 1 + 3 + 1 + 1 = 6
        int bits1_esperado = 6;
        int bits1_resultado = bits1(bits1_n);
        if (bits1_resultado != bits1_esperado) {
                printf("error en bits1\n");
                return 1;
        }

        int posicionBits_x = 0x540; // 5:0101 4:0100 0:0000
        int posicionBits_p = 0xA; // 1010
        int posicionBits_n = 4;
        int posicionBits_esperado = 5;
        int posicionBits_resultado = posicionBits(posicionBits_x, posicionBits_p, posicionBits_n);
        if (posicionBits_resultado != posicionBits_esperado) {
                printf("error en posicionBits\n");
                return 1;
        }

        unsigned repBits_x = 0x4CD; // 0100 1100 1101
        int repBits_i = 5;
        int repBits_k = 3;
        unsigned repBits_val = 1; // 0001
        unsigned repBits_esperado = 0x42D; // 0100 0010 1101
        unsigned repBits_resultado = repBits(repBits_x, repBits_i, repBits_k, repBits_val);
        if (repBits_resultado != repBits_esperado) {
                printf("error en repBits\n");
                return 1;
        }

        printf("éxito!\n");
}

int bits1(int n) {
        int resultado = 0;
        for (int i = 0; i < sizeof(int)*8; i++) {
                if ((n >> i) & 1) {
                        resultado++;
                }
        }
        return resultado;
}

int posicionBits(int x, int p, int n) {
        unsigned mask = ~((-1)<<n); // 000...011...1 } n unos
        for (int i = 0; i < sizeof(int) * 8 - n + 1; i++) {
                if (((x >> i) & mask) == p) {
                        return i;
                }
        }
        return -1;
}

unsigned repBits(unsigned x, int i, int k, unsigned val) {
        unsigned mask1 = ~((-1)<<k); // 000...011...1 } k unos
        unsigned mask2 = mask1 << i; // 0...0 1^{k} 0^{i}
        val <<= i;
        x &= ~mask2;
        x |= val;
        return x;
}
```

Solución. (**P3 EXTRA**. Auxiliar de Alexandra Ibarra)

- **Versión 1:** Bits se cuentan de derecha a izquierda.

```c
#include <stdio.h>

unsigned repBits(unsigned x, int i, int k, unsigned val) {
    /* mask1 es una máscara con i 1s desde la primera posición */
    unsigned mask1 = (1 << i) - 1; // equivalente a ~(-1<<i)

    /* mask2 es una máscara k+i 0s desde la primera posición */
    unsigned mask2 = (-1<< (k+i));
    unsigned valDespl = val << i;
    return (x & mask2) | valDespl | (x & mask1)
}
```

- **Versión 2:** Bits se cuentan de izquierda a derecha.

```c
#include <stdio.h>

unsigned repBits(unsigned x, int i, int k, unsigned val) {
    /* m es una máscara con k 0s desde la posición i, el resto son 1s */
    unsigned m = ~((1 << (32 - i)) - 1 ) | ((1 << (32 - i - k)) - 1 );
    unsigned valDespl = val << (32 - i - k);
    return (x & m) | valDespl;
}
```

3
AUXILIAR 3: STRINGS

Auxiliar de Blaz Korecic

P1. Programe las siguientes funciones

```
void to lower(char *s);
void to upper(char *s);
```

Estas funciones reciben un string `s`. La primera convierte todas las letras del string mayúculas en minúsculas y la segunda hace lo contrario.

P2. Escriba una función que retorne `1` si el string `s` es un palíndromo y `0` en el caso contrario.

P3. Escriba una función que reciba un string `s` y lo invierta en el lugar, es decir, usando memoria adicional $O(1)$.

```
void reverse(char *s);
```

P4. Escriba una función que retorne el caracter de `s` que tiene el mayor número de repeticiones en el string. Si hay varios con la mayor cantidad, retorna cualquiera.

```
char mas repetido(char *s);
```

P0. Preliminares.

Nos preparamos a hacer las funciones que nos piden.

```
#include <stdio.h>

void to_lower(char *s) {
...
}

void to_upper(char *s) {
...
}

int main() {
...
}
```

Dado un string "hola", en la memoria del computador se guardará como varios caracteres consecutivos. Entonces tenemos bloques del estilo

'h'	'o'	'l'	'a'	'\0'

O bien

'h'	'o'	'l'	'a'	0

Volviendo a la Tabla ASCII (2.5) y tomamos de ejemplo la letra `A` con su equivalencia a 65. Podemos probar la siguiente función

```
#include <stdio.h>

int main() {
    char c = 'A';
    int x = c;
    printf("%d\n", x); // out ---> 65

    return 0;
}
```

Notando que los caracteres simplemente son números. Anotemos ideas

```
si(*s == 0) estamos al final del string
```

¿Cómo aprovechamos la tabla ASCII para convertir de minúscula a mayúscula o viceversa? Podemos usar como pista que las letras están contiguas. Usando operaciones de suma y resta tenemos respuesta. Pero siquiera es necesario tener una tabla a mano, ¿Qué pasa si aprovechamos la diferencia en el mismo código para convertir? Veamos.

```
#include <stdio.h>

int main() {
    char x = 'a';
    x += ('A' - 'a'); // Diferencia fija
    printf("%d\n", x); // out ---> A
    return 0;
}
```

P1. (Solución)

Retomamos la plantilla preliminar.

```c
#include <stdio.h>

void to_lower(char *s) {
    while(*s != 0) { // Aprovechando el cero del final del string
    if ('A' <= *s && *s <= 'Z') { // if (*s es mayuscula)
        *s += 'a' - 'A';
    }
    s++;
    }
}

void to_upper(char *s) {
    while(*s != 0) { // Aprovechando el cero del final del string
    if ('a' <= *s && *s <= 'z') { // if (*s es mayuscula)
        *s += 'A' - 'a';
    }
    s++;
    }
}

int main() {
    char s[] = "hOlAqeuEtal";
    to_lower(s);
    printf("%s\n", s);
    to_upper(s);
    printf("%s\n", s);
    return 0;
}
```

P2. (Solución)

```c
#include <stdio.h>
#include <string.h>

/*
 *  l       r
 *  |       |
 *  HOLAHOLA  largo=8
 *
 *   l     r
 *   |     |
 *  HOLAHOLA  largo=8
 *
 *     l  r
 *     |  |
 *  HOLAHOLA  largo=8
 *
 *      lr
 *      ||
 *  HOLAHOLA  largo=8
 */

int palindromo(char *s){
  char *l = s;
  char *r = l+strlen(s)-1;
  while(l < r){
    if(*l != *r){
      return 0;
    }
    l++;
    r--;
  }
  return 1;
}

int main(int argc, char* argv[]){
  if(argc == 2){
    char *p = argv[1];
    if(palindromo(p)){
      printf("YES\n");
    }
    else{
      printf("NO\n");
    }
  }
  else{
    printf("Usage:\n");
    printf("[prog] string\n");
  }
  return 0;
}
```

P3. (Solución)

```c
#include <stdio.h>
#include <string.h>

/*
 *  l       r
 *  |       |
 *  HOLAHOLA  largo=8
 *
 *   l     r
 *   |     |
 *  HOLAHOLA  largo=8
 *
 *     l  r
 *     |  |
 *  HOLAHOLA  largo=8
 *
 *      lr
 *      ||
 *  HOLAHOLA  largo=8
 */

void reverse(char *s){
  char *l = s;
  char *r = l+strlen(s)-1;
  while(l < r){
    char tmp = *l;
    *l = *r;
    *r = tmp;
    l++;
    r--;
  }
}

int main(int argc, char* argv[]){
  if(argc == 2){
    char *p = argv[1];
    reverse(p);
    printf("%s\n", p);
  }
  else{
    printf("Usage:\n");
    printf("[prog] string\n");
  }
  return 0;
}
```

P4. (Solución)

```c
#include <stdio.h>
#include <string.h>

char mas_repetido(char *s){
  int frec[256] = {0};
  while(*s != 0){
    char c = *s;
    frec[(int)c]++;
    s++;
  }
  char ans = 0;
  int ans_frec=0;
  for(int i=0; i<256; i++){
    if(frec[i] > ans_frec){
      ans = i;
      ans_frec = frec[i];
    }
  }
  return ans;
}

int main(int argc, char* argv[]){
  if(argc == 2){
    char *p = argv[1];
    printf("%c\n", mas_repetido(p));
  }
  else{
    printf("Usage:\n");
    printf("[prog] string\n");
  }
  return 0;
}
```

4

AUXILIAR 4: ESTRUCTURAS Y MEMORIA

P1. (Memoria (malloc) en strings) Dada la función `to_lower` del auxiliar anterior, escriba la función `char *lowerCase(char *str)` que retorne una copia de `str` con las letras convertidas a minúscula, sin editar el string original.

P2. (Estructuras: Cola con array) Implemente una cola FIFO que permita hacer las siguientes operaciones:

```
Cola *creaCola(int capacidad);
void put(Cola* q, char *str);
char *get(Cola* q);
void freeCola(Cola *q);
```

Las operaciones principales (`put` y `set`) se deben ejecutar en tiempo $O(1)$. Use un array guardado en el Heap para implementarla.

P3. (Estructuras: Cola con lista enlazada) Modifique su cola para que use una lista enlazada en lugar de un array. Observe que esto simplifica la posibilidad de hacer que la capacidad sea arbitrariamente grande.

P1. (Solución)

Si se trata de no editar el string original, usaremos `strcpy(destino, fuente)`. Pero primero necesitamos `malloc` (`malloc` obtiene memoria en el heap) y debemos indicar la memoria que quermeos pedir, en este caso, esa cantidad es la longitud de `str` (que queremos copiar) pero sumando +1 (recordando el `0` que finaliza los strings).

Esto es `char *dst = malloc(strlen(str) + 1);`

```
char *lowerCase(char *str) {
    char *dst = malloc(strlen(str) + 1);
    strcpy(dst, str);
    to_lower(dst);
    return dst;
}
```

P2. (Solución)

La cola First in First Out quiere decir que el primer elemento que se pone es el primero en retirarse. Debemos contar con las siguientes operaciones:

- Crear la cola
- Podemos poner elementos (`strings`)
- Podemos obtener elementos de la cola
- y podemos liberar la cola

Notar que si las operaciones `put` y `get` ocurran en tiempo $O(1)$.

Lo que queremos es

```
Queue *makeQueue() {
    Queue *q = malloc(sizeof(Queue));
    q->primero = NULL;
    q->ultimo = NULL;
    return q;
}

void put(Queue *q, char *string) {
    QueueNode *nuevo = malloc(sizeof(QueueNode));
    nuevo->val = string;
    nuevo->next = NULL;
    if (q->ultimo == NULL) {
        q->primero = nuevo;
        q->ultimo = nuevo;
    } else {
        QueueNode *penultimo = q->ultimo;
        penultimo->next = nuevo;
        q->ultimo = nuevo;
    }
}

char *get(Queue *q) {
    if (q->primero == NULL) {
        return NULL;
    }
    QueueNode porDestruir = *q->primero;
    free(q->primero);
    if (q->primero == q->ultimo) { // comparamos las direcciones
        q->ultimo = NULL;
    }
    q->primero = porDestruir.next;
    return porDestruir.val;
}

void freeQueue(Queue *q) {
    QueueNode *qn = q->primero;
    while (qn) {
        QueueNode tmp = *qn; // para no usar qn después de liberarlo
        free(qn);
        qn = tmp.next;
    }
    free(q);
}
```

Versión comentada

```c
typedef struc QueueNode {
    struc QueueNode *next;
    char *val;
} QueueNode;

Queue *makeQueue() {
    Queue *q = malloc(sizeof(Queue));
    q->primero = NULL;
    q->ultimo = NULL;
    return q;
}

void put(Queue *q, char *string) {
    QueueNode *nuevo = malloc(sizeof(QueueNode)); // creamos un QueueNode
    // si nuevo en lugar de *nuevo estaríamos creando copias y no trabajando con lo mismo
    nuevo->val = string; // llenamos el valor con el string que nos pasaron
    nuevo->next = NULL; // lo que uno obtiene de malloc podría no ser nulo, aseguramos
    if (q->ultimo == NULL) {
        // Si el último fuera nulo, el nuevo debe ser q->primero y q->ultimo
        q->primero = nuevo;
        q->ultimo = nuevo;
    } else {
        QueueNode *penultimo = q->ultimo;
        penultimo->next = nuevo;
        q->ultimo = nuevo;
    }
}

char *get(Queue *q) {
    if (q->primero == NULL) {
        return NULL;
    }
    QueueNode porDestruir = *q->primero;
    free(q->primero);
    if (q->primero == q->ultimo) { // comparamos las direcciones
        q->ultimo = NULL;
    }
    q->primero = porDestruir.next;
    return porDestruir.val;
}

void freeQueue(Queue *q) {
    QueueNode *qn = q->primero;
    while (qn) {
        QueueNode tmp = *qn; // para no usar qn después de liberarlo
        free(qn);
        qn = tmp.next;
    }
    free(q);
}
```

5

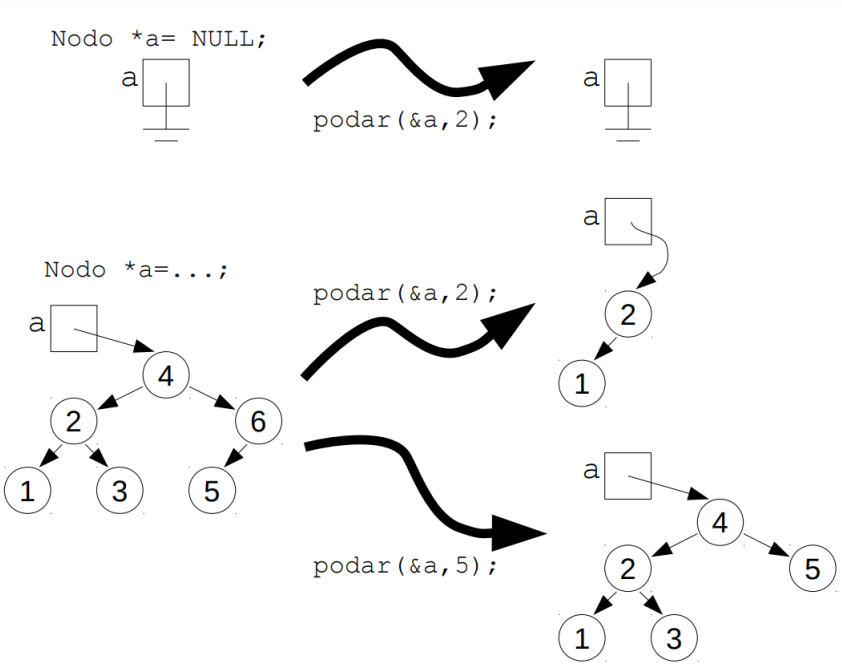
AUXILIAR 5: ESTRUCTURAS DE DATOS RECURSIVAS

P1. (P2 C1 Primavera 2017). Sea la estructura

```
typedef struct nodo {
    int x;
    struct nodo *izq, *der;
} Nodo;
```

Programe la función: `void podar(Nodo **pa, int y);`

Esta función debe modificar un árbol de búsqueda binaria `*pa` eliminando todos los nodos etiquetados con valores mayores que y. No necesita liberar la memoria de los nodos eliminados. Estudie los 3 ejemplos de uso de la siguiente figura:



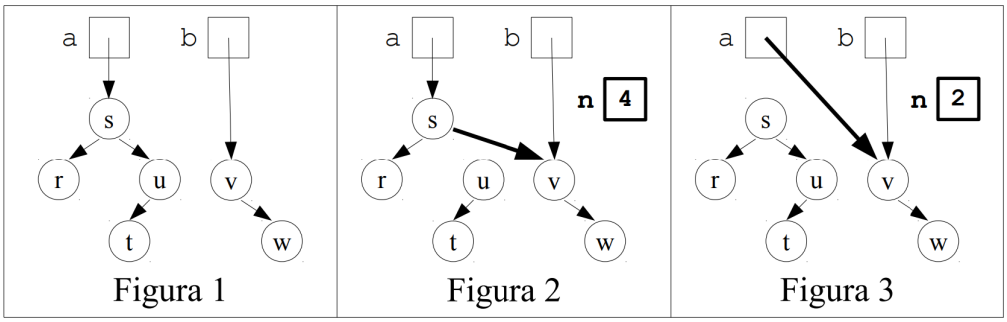
Restricciones: Sea eficiente, el tiempo de ejecuci´on debe ser proporcional a la altura del árbol en el peor caso. No puede usar ciclos (como `while` o `for`). Debe usar recursión. **No puede usar malloc**.

P2. (P2 C1 Otoño 2017). Sea la estructura

```
typedef struct nodo {
    char c;
    struct nodo *izq, *der;
} Nodo;
```

Programe la función: `int reemplazarNodoK(Nodo **pa, int k, Nodo *b);`

Sea `a = *pa`. Esta función reemplaza el k-ésimo nodo del árbol a por el nodo b. El k-ésimo nodo de a es el k-ésimo nodo al enumerar los nodos de a en inorden. Por ejemplo en la Figura 1, r es el nodo 1, s el 2, t el 3 y u el 4. Esta función retorna k si se hizo el reemplazo, es decir cuando el árbol a tenía al menos k nodos. Si no, entrega el número de nodos encontrados en a, que será inferior a k. Las siguientes figuras sirven para explicar algunos ejemplos de uso. Los punteros a y b**E** son de tipo `Nodo*`.



La figura 2 se obtiene cuando a partir de la figura 1 se llama: `int n = reemplazarNodoK(&a, 4, b);`

Se reemplazó u por v y la función retornó 4. La figura 3 se obtiene cuando a partir de la figura 1 se llama: `int n = reemplazarNodoK(&a, 2, b);`

Acá se reemplazó la raíz s del árbol por v. Por eso se requiere que el puntero a la raíz del árbol se pase por referencia (`&a`). Por último si a partir de la figura 1 se intentara reemplazar el quinto nodo de a que no existe, no se haría ningún reemplazo y la función retornaría 4.

P1. (Solución)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct nodo{
  int x;
  struct nodo *izq, *der;
} Nodo;

void podar(Nodo **pa, int y){
  Nodo *a = *pa;
  if(a == NULL){
    return;
  }
  if(a->x <= y){
    podar(&a->der, y);
  }
  else{
    podar(&a->izq, y);
    *pa = a->izq;
  }
}

Nodo *createNodo(int x, Nodo *izq, Nodo *der){
  Nodo *a = (Nodo*)malloc(sizeof(Nodo));
  a->x = x;
  a->izq = izq;
  a->der = der;
  return a;
}

void printArbol(Nodo *a){
  if(a == NULL) return;
  printf("%i ", a->x);
  printArbol(a->izq);
  printArbol(a->der);
}

int main(){
  Nodo *arbol =
    createNodo(4,
        createNodo(2,
          createNodo(1, NULL, NULL),
          createNodo(3, NULL, NULL)),
        createNodo(6,
          createNodo(5, NULL, NULL),
          NULL));
  printArbol(arbol);
  printf("\n");
  podar(&arbol, 5);
  printArbol(arbol);
  printf("\n");
  return 0;
}
```

P2. (Solución)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct nodo {
  char c;
  struct nodo *izq, *der;
} Nodo;

int reemplazarNodoK(Nodo **pa, int k, Nodo *b){
  Nodo *a = *pa;
  if(a == NULL) return 0;
  int cnt_left = reemplazarNodoK(&a->izq, k, b);
  if(cnt_left == k){
    return k;
  }
  else if(cnt_left == k-1){
    *pa = b;
    return k;
  }
  int cnt_right = reemplazarNodoK(&a->der, k-cnt_left-1, b);
  return cnt_left+1+cnt_right;
}

Nodo *createNodo(char c, Nodo *izq, Nodo *der){
  Nodo *a = (Nodo*)malloc(sizeof(Nodo));
  a->c = c;
  a->izq = izq;
  a->der = der;
  return a;
}

void printArbol(Nodo *a){
  if(a == NULL) return;
  printArbol(a->izq);
  printf("%c ", a->c);
  printArbol(a->der);
}

int main(){
  Nodo *arbol = createNodo('s',
      createNodo('r', NULL, NULL),
      createNodo('u',
        createNodo('t', NULL, NULL),
        NULL));
  Nodo *b = createNodo('v', NULL, createNodo('w', NULL, NULL));

  printArbol(arbol);
  printf("\n");

  printf("resultado = %d\n", reemplazarNodoK(&arbol, 2, b));

  printArbol(arbol);
  printf("\n");

  return 0;
}
```

6

AUXILIAR 6: TIPOS DE DATOS ABSTRACTOS

APÉNDICE

En lo que sigue del texto se encontrará material suplementario para el curso.

A
APUNTE OFICIAL DEL CURSO - LUIS MATEU

A continuación puede encontrar atajos directos a los apuntes del curso. (Mateu, 2021).

1. Introducción al curso
2. El Lenguaje C
   I. Principios básicos
   II. El sistema de tipos
   III. Operaciones con bits
   IV. Variables
   V. Punteros
   VI. Arreglos
   VII. Strings
   VIII. Estructuras
   IX. Punteros a funciones
   X. Archivos
   XI. setjmp y longjmp
   XII. Funciones con número variable de argumentos
   XIII. Etapas de la compilación
3. Obsoleto: Threads (ya no es parte del contenido de PSS)
4. Sistema Operativo Unix
   I. Historia
   II. Manejo de archivos
   III. Procesos
   IV. Señales
5. Obsoleto: Sockets (ya no es parte del contenido de PSS)
6. Obsoleto: Herramientas para programación de sistemas (ya no es parte del contenido de PSS)
   I. programación en el shell
   II. programación en perl

B

INSTALACIÓN Y USO BÁSICO DE DEBIAN 11 EN VIRTUALBOX (WINDOWS)

Futuramente se introducirán instrucciones precisas y preguntas frecuentes

`https://users.dcc.uchile.cl/~lmateu/CC3301/#virtualbox`

IMPORTANTE: Ud. podrá resolver las tareas de este curso en la distribución basada en Linux de su preferencia. Por ejemplo en Ubuntu bajo WSL de Windows o también en OS X de Apple. Sin embargo, Ud. deberá probar sus tareas en Debian 11.

C

INSTALACIÓN Y USO BÁSICO DE DEBIAN 11 EN VMWARE (WINDOWS)

Esta sección está en desarrollo. Pero se hará una guía breve.

El archivo *debian-11-2-mate.ova* está disponible en la misma ubicación de Drive que para el tutorial de VirtualBox: `https://drive.google.com/file/d/1eNut2ErXjPIDlAGyMeRtj1R96F9HEvyK/view?usp=sharing`

Referencia rápida a VMWare Workstation 16 Player: `https://www.vmware.com/cl/products/workstation-player/workstation-player-evaluation.html`

How to install VMware tools in Debian 10

D

INSTALACIÓN DE KALI LINUX EN VMWARE (WINDOWS)

Referencia: `https://www.youtube.com/watch?v=DpJ-dDX3uVk`

IMPORTANTE: Ud. podrá resolver las tareas de este curso en la distribución basada en Linux de su preferencia. Por ejemplo en Ubuntu bajo WSL de Windows o también en OS X de Apple. Sin embargo, Ud. deberá probar sus tareas en Debian 11.

E

INSTALACIÓN DE KUBUNTU EN VMWARE (WINDOWS)

Referencia: `https://kubuntu.org/getkubuntu/`

IMPORTANTE: Ud. podrá resolver las tareas de este curso en la distribución basada en Linux de su preferencia. Por ejemplo en Ubuntu bajo WSL de Windows o también en OS X de Apple. Sin embargo, Ud. deberá probar sus tareas en Debian 11.

F

CÓMO USAR DDD

Próximamente. Por ahora se puede consultar el video de Luis Mateu: `https://www.youtube.com/watch?v=FtHZy7UkTT4`

- Tutorial Alternativo: (Aguayo, 2022) `https://www.youtube.com/watch?v=Cwxe2NksJIo`

G

README - DOCUMENTACIÓN PARA COMPILAR Y EJECUTAR TAREAS

La mayoría de las tareas tiene un formato bastante similar. Aquí se incluye una copia del archivo README.TXT que usualmente si incluye en las instrucciones de estas.

```
============================================================
Esta es la documentación para compilar y ejecutar su tarea
============================================================

Ud. debe crear el archivo pedido con tipo de archivo .c
Usualmente se encuentra una plantilla.

Pruebe su tarea bajo Debian 11 de 64 bits nativo o virtualizado.  Quedan
excluidos WSL 1 y WSL 2 para hacer las pruebas.  Estos son los requerimientos
para aprobar su tarea:

+ make run-san debe felicitarlo y no debe reportar ningún problema como
  gotera de memoria, referencias colgantes, etc.
+ make run-g debe felicitarlo.
+ make run debe felicitarlo por aprobar este modo de ejecución.  Esta
  prueba será rechazada si su solución es 50% más lenta que la solución
  del profesor.

Cuando pruebe su tarea con make run en su computador asegúrese de que
que está configurado en modo alto rendimiento y que no estén corriendo
otros procesos intensivos en uso de CPU al mismo tiempo.  De otro modo
podría no lograr la eficiencia solicitado.

Invoque el comando make zip para ejecutar todos los tests y generar un
archivo elim.zip que contiene elim.c, con su solución,
y resultados.txt, con la salida de make run, make run-g y make run-san.

Para depurar use: make ddd

Video con ejemplos de uso de ddd: https://youtu.be/FtHZy7UkTT4
Archivos con los ejemplos:
https://www.u-cursos.cl/ingenieria/2020/2/CC3301/1/novedades/r/demo-ddd.zip

...
```

```
-----------------------------------------------

Entrega de la tarea

Ejecute: make zip

Entregue por U-cursos el archivo elim.zip

A continuación es muy importante que descargue de U-cursos el mismo
archivo que subió.  Luego examine el archivo elim.c revisando
que es el correcto.  Es frecuente que no lo sea producto de un defecto
de U-cursos.

-----------------------------------------------

Limpieza de archivos

make clean

Hace limpieza borrando todos los archivos que se pueden volver
a reconstruir a partir de los fuentes: *.o binarios etc.

-----------------------------------------------

Acerca del comando make

El comando make sirve para automatizar el proceso de compilación asegurando
recompilar el archivo binario ejecutable cuando cambió uno de los archivos
fuentes de los cuales depende.

A veces es útil usar make con la opción -n para que solo muestre
exactamente qué comandos va a ejecutar, sin ejecutarlos de verdad.
Por ejemplo:

   make -n ddd

También es útil usar make con la opción -B para forzar la recompilación
de los fuentes a pesar de que no han cambiado desde la última compilación.
Por ejemplo:

   make -B run
```

H

MATERIAL DOCENTE DEL CURSO DISPONIBLE

Exclusivamente material de U-Cursos

- CC3301-1 - Primavera 2021
- CC3301-1 - Otoño 2021
- CC3301-1 - Primavera 2020
- CC3301-1 - Primavera 2016
- CC3301-1 - Primavera 2014
- CC3301-1 - Otoño 2014
- CC3301-1 - Primavera 2012
- CC3301-1 - Otoño 2012
- CC3301-1 - Primavera 2011
- CC3301-1 - Otoño 2011
- CC3301-1 - Primavera 2010
- CC3301-1 - Otoño 2010
- CC3301-1 - Primavera 2009

I

CONTROLES DE SEMESTRES ANTERIORES

- Controles de mitad de semestre durante la pandemia:

  Semestre Otoño 2021 (tests de prueba). No resuelva la pregunta 5 porque ya no es materia de este curso.

  Semestre Primavera 2020 (tests de prueba, lea README.txt, luego de resolver una pregunta, puede revisar la solución en soluciones). No resuelva la pregunta 5 porque ya no es materia de este curso.

  Semestre Otoño 2020

- Control 1:

  Semestre Primavera 2019

  Semestre Otoño 2019

  Semestre Primavera 2018

  Semestre Otoño 2018

  Semestre Primavera 2017

  Semestre Otoño 2017

  Semestre Primavera 2016

  Semestre Otoño 2016

  Semestre Primavera 2015

  Semestre Otoño 2015

  Semestre Primavera 2014

  Semestre Otoño 2014

  Semestre Primavera 2013

  Semestre Otoño 2013

  Semestre Primavera 2012

- Control 2:

  Semestre Primavera 2019

  Semestre Otoño 2019

  Semestre Primavera 2018

  Semestre Otoño 2018

  Semestre Primavera 2017

  Semestre Otoño 2017

  Semestre Primavera 2016

  Semestre Otoño 2016

  Semestre Primavera 2015

  Semestre Otoño 2015

  Semestre Primavera 2014

  Semestre Otoño 2014

  Semestre Primavera 2013

Semestre Otoño 2013

Semestre Primavera 2012

- Control 3:

  Semestre Otoño 2019

  Semestre Primavera 2018

  Semestre Otoño 2018

  Semestre Primavera 2017

  Semestre Otoño 2017

  Semestre Primavera 2016

  Semestre Otoño 2016

  Semestre Primavera 2015

  Semestre Otoño 2015

  Semestre Primavera 2014

  Semestre Otoño 2014

  Semestre Primavera 2013

  Semestre Otoño 2013

  Semestre Primavera 2012

- Exámen:

  Semestre Otoño 2021 (tests de prueba)

  Semestre Primavera 2020 (tests de prueba)

  Semestre Otoño 2020

  Semestre Otoño 2019

  Semestre Primavera 2018

  Semestre Otoño 2018

  Semestre Primavera 2017

  Semestre Otoño 2017

  Semestre Primavera 2016 (tests de prueba)

  Semestre Otoño 2016

  Semestre Primavera 2015

  Semestre Otoño 2015

  Semestre Primavera 2014

  Semestre Otoño 2014

  Semestre Primavera 2013

  Semestre Otoño 2013

  Semestre Primavera 2012

J

APUNTES DEL DOCENTE - PATRICIO POBLETE

Los apuntes del profesor de cátedra Patricio Poblete están listados en la página de Luis Mateu (Poblete, 2002).

I Lenguaje C

II Entrada/salida en Unix

III Procesos y señales en Unix

IV Entrada/salida en el Terminal

V Sockets

VI Perl

VII Shell

K

EDITAR Y COMPILAR C/C++ DESDE VISUAL STUDIO CODE (WINDOWS)

Sección en construcción.

Link temporal: `https://code.visualstudio.com/docs/cpp/config-mingw`

a

EJERCICIOS PARA ESTUDIAR (ORDEN CRONOLÓGICO)

Se listarán ejercicios que sean de utilidad para el curso. Estos cumplirán el siguiente formato:

Título + Contenido + Fecha

Donde una breve descripción de ellos podrá ser encontrada en la sección respectiva de cada ejercicio.

1 Ejercicio ejemplo - Ejemplo - Fecha ejemplo

WEB 1: GITHUB

WEB 2: REPLIT

BIBLIOGRAFÍA

Aguayo, A. (2022). *Ddd en 10 minutos.* Descargado de `https://www.youtube.com/watch?v=Cwxe2NksJIo` (Última vez visitado 16 de Abril 2022)

Mateu, L. (2021). *Cc3301 programación de software de sistemas - apuntes elaborados por luis mateu.* Descargado de `https://wiki.dcc.uchile.cl/cc3301/temario` (Última vez visitado 27 de Marzo 2022)

Mateu, L. (2021). *Clase 3, programación de software de sistemas.* Descargado de `https://www.youtube.com/watch?v=R6fWcz1GDXs` (Última vez visitado 15 de Marzo 2022)

Mateu, L. (2022). *Clase 1, programación de software de sistemas.* Descargado de `https://www.u-cursos.cl/ingenieria/2022/1/CC3301/1/novedades/r/20220308143308F5485D0D3A536D__Intro.pdf` (Última vez visitado 8 de Marzo 2022)

Mateu, L. (2022). *Cátedras del martes 15 y jueves 17 de marzo.* Descargado de `https://www.u-cursos.cl/ingenieria/2022/1/CC3301/1/novedades/detalle?id=370773` (Última vez visitado 15 de Marzo 2022)

Poblete, P. (2002). *Index of /lmateu/cc3301/apuntes.* Descargado de `https://users.dcc.uchile.cl/~lmateu/CC3301/apuntes/` (Última vez visitado 10 de Abril 2022)