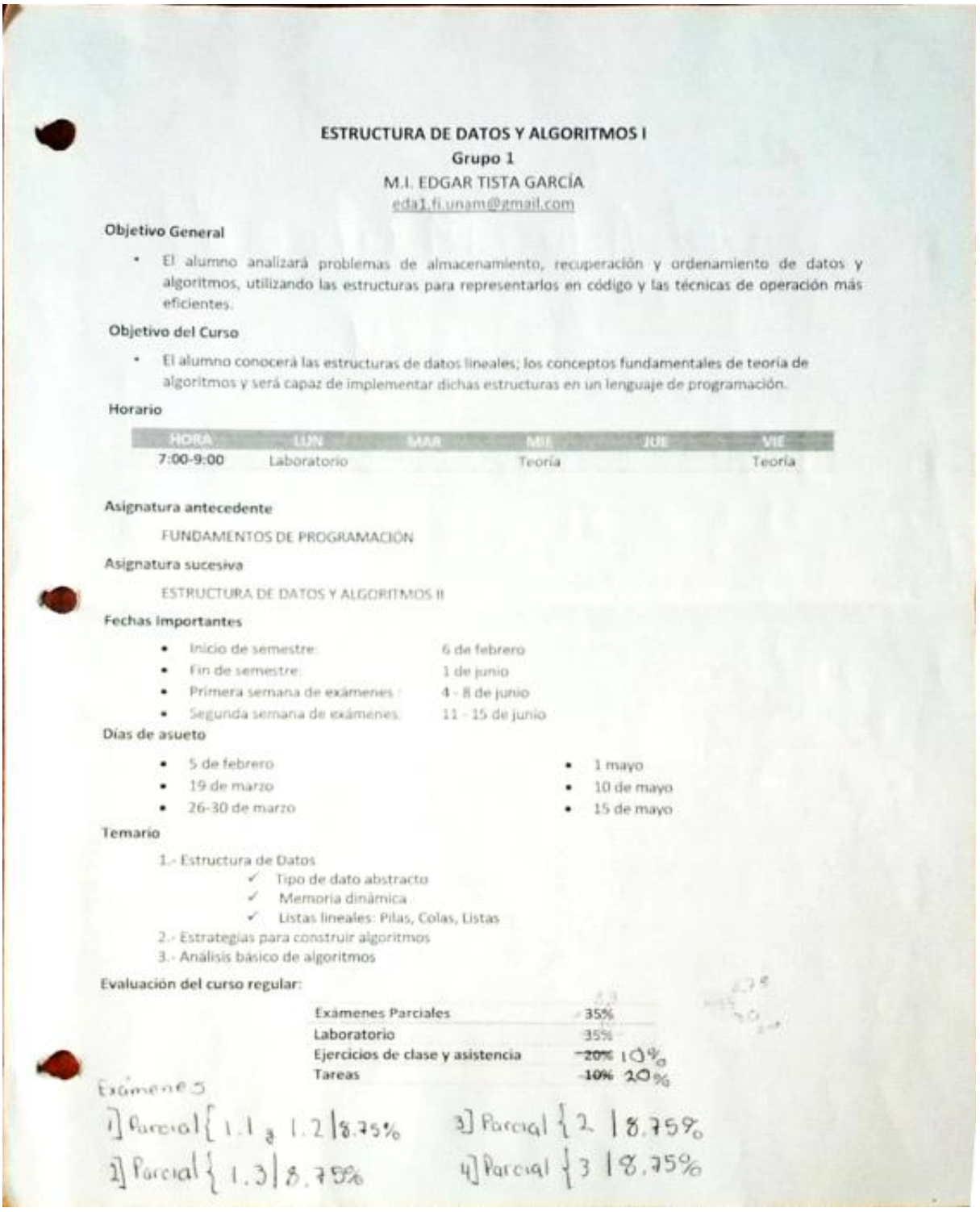
FORMA DE EVALUACIÓN

Saturday, April 7, 2018 4:49 PM



ESTRUCTURA DE DATOS Y ALGORITMOS I

Grupo 1

M.I. EDGAR TISTA GARCÍA

eda1.fi.unam@gmail.com

Objetivo General

- El alumno analizará problemas de almacenamiento, recuperación y ordenamiento de datos y algoritmos, utilizando las estructuras para representarlos en código y las técnicas de operación más eficientes.

Objetivo del Curso

- El alumno conocerá las estructuras de datos lineales; los conceptos fundamentales de teoría de algoritmos y será capaz de implementar dichas estructuras en un lenguaje de programación.

Horario

HORA	LUN	MAR	MIE	JUE	VIE
7:00-9:00	Laboratorio		Teoría		Teoría

Asignatura antecedente

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura sucesiva

ESTRUCTURA DE DATOS Y ALGORITMOS II

Fechas importantes

- Inicio de semestre: 6 de febrero
- Fin de semestre: 1 de junio
- Primera semana de exámenes: 4 - 8 de junio
- Segunda semana de exámenes: 11 - 15 de junio

Días de asueto

- 5 de febrero
- 19 de marzo
- 26-30 de marzo
- 1 mayo
- 10 de mayo
- 15 de mayo

Temario

1.- Estructura de Datos
   - ✓ Tipo de dato abstracto
   - ✓ Memoria dinámica
   - ✓ Listas lineales: Pilas, Colas, Listas

2.- Estrategias para construir algoritmos

3.- Análisis básico de algoritmos

Evaluación del curso regular:

Exámenes Parciales	35%
Laboratorio	35%
Ejercicios de clase y asistencia	~~20%~~ 10%
Tareas	~~10%~~ 20%

Exámenes

1] Parcial { 1.1 y 1.2 | 8.75%

2] Parcial { 1.3 | 8.75%

3] Parcial { 2 | 8.75%

4] Parcial { 3 | 8.75%

Consideraciones de la evaluación

- No se podrá acreditar el curso si no se acredita el laboratorio.
- Calificaciones .5 se redondean a la siguiente calificación superior.
- Para poder justificar una falta se debe dar aviso antes de la clase.
- Después de 15 min de retardo se considera falta.
- Ningún alumno que presente al menos un examen parcial o al menos 3 prácticas de laboratorio podrá tener "NP" como calificación en actas.

Consideraciones sobre el examen final

- El examen final es, en principio, obligatorio para todos.
- Para tener derecho a presentar examen final se requiere haber obtenido al menos 5 de calificación en el curso y no haber causado baja de la asignatura.
- Los alumnos que obtengan una calificación del curso mayor o igual a 7.5 y que tengan calificación aprobatoria en todos los rubros de evaluación tienen derecho de exentar el examen final.
- Es posible presentar reposición de algún examen parcial o realizar el examen final para subir calificación siempre y cuando se cumpla el punto anterior (*únicamente en el primer examen final*).
- La calificación de los alumnos que presenten examen final se obtendrá de la siguiente manera:

Examen final	70%
Calificación del curso	30%

NOTA: Para aprobar el curso, es necesario aprobar el examen final.

Causales de baja de la asignatura

- 6 faltas no justificadas a lo largo del semestre
- 3 faltas no justificadas consecutivas
- 4 faltas no justificadas en un mes
- Infringir alguna regla critica del reglamento interno de la clase.

Bibliografía

- AHO, Alfred, ULLMAN, Jeffrey, et al.
Data Structures and Algorithms
New Jersey
Addison-Wesley, 1983

- BAASE, Sara, VAN GELDER, Allen
Algorithms: Introduction to Design and Analysis
3rd edition
San Diego
Addison-Wesley, 1999

- KNUTH, Donald
The Art of Computer Programming
New Jersey
Addison-Wesley Professional, 2011
Volumes 1-4A

- KERNIGHAN, Brian, RITCHIE, Dennis
C Programming Language
2nd edition
New Jersey
Prentice Hall, 1988

- SZNAJDLEDER, Pablo
Algoritmos a fondo: con implementación en C y JAVA
Buenos Aires
Alfaomega, 2012

1. **Estructura de datos**
   Objetivo: El alumno resolverá problemas de almacenamiento, recuperación y ordenamiento de datos y las técnicas de representación más eficientes, utilizando las estructuras para representarlos.
   Contenido:
   1.1. Representación de datos en memoria.
      1.1.1. Tipos primitivos.
      1.1.2. Arreglos.
      1.1.3. Apuntadores.
      1.1.4. Estructuras.

   1.2. Administración del almacenamiento en tiempo de ejecución.

   1.3. Estructura de datos.
      1.3.1. Pila: almacenamiento contiguo y ligado, y operaciones.
      1.3.2. Cola: almacenamiento contiguo y ligado, y operaciones.
      1.3.3. Cola doble: almacenamiento contiguo y ligado, y operaciones.
      1.3.4. Listas circular: almacenamiento contiguo y ligado, y operaciones.
      1.3.5. Listas doblemente ligadas: almacenamiento contiguo y ligado, y operaciones.

2. **Estrategia para construir algoritmos**
   Objetivo: El alumno aplicará diversas técnicas como la recursividad para construir algoritmos.
   Contenido:
   2.1. Algoritmos de búsqueda exhaustiva y fuerza bruta.
   2.2. Top-down y bottom-up.
   2.3. Algoritmos ávidos o voraces (Greedy).
   2.4. Divide y vencerás (Divide and conquer).
   2.5. Recursividad.
      2.5.1. El concepto de recursividad.
      2.5.2. Funciones matemáticas de recursividad.
      2.5.3. Uso de relaciones de recurrencia para analizar algoritmos recursivos.
      2.5.4. Retroceso recursivo.
      2.5.5. Implementación de la recursividad.
   2.6. Backtrack.

3. **Análisis básico de algoritmos**
   Objetivo: El alumno analizará algoritmos mediante medidas de rendimiento, espacio y tiempo para conocer su complejidad y generar programas usando los mismos.
   Contenido
   3.1. Fundamentos de algoritmica.
   3.2. Análisis asintótico de los límites superior y media.
   3.3. Notación O, omega y teta.
   3.4. Medidas empíricas de rendimiento.
   3.5. Compensación espacio y tiempo en los algoritmos.
   3.6. Complejidad.
      3.6.1. P.
      3.6.2. NP.
      3.6.3. NP completos.

Laboratorio

- Página web { · Reglamento · Prácticas · servicios

- Evaluación Laboratorio
  - Prácticas
  - Evaluación habilidades (Examen final) → Aprobarlo o baja 1 punto
  - Asistencia mínima del 80% (11 prácticas)

- Reportes de las prácticas
  - Los reportes se entregan el mismo día
  - Reportes de calidad
  -

→ reporte → PDF

- Portada
- Objetivo de la práctica
- Desarrollo
  - * Breve introducción
  - Análisis de los problemas { · ejemplos propuestos en las práctica · Ejercicios propuestos
- Evidencia de la implementación
- Conclusiones de la práctica { Indicar si se cumplió el objetivo y por qué

Adjuntar el código | Programas

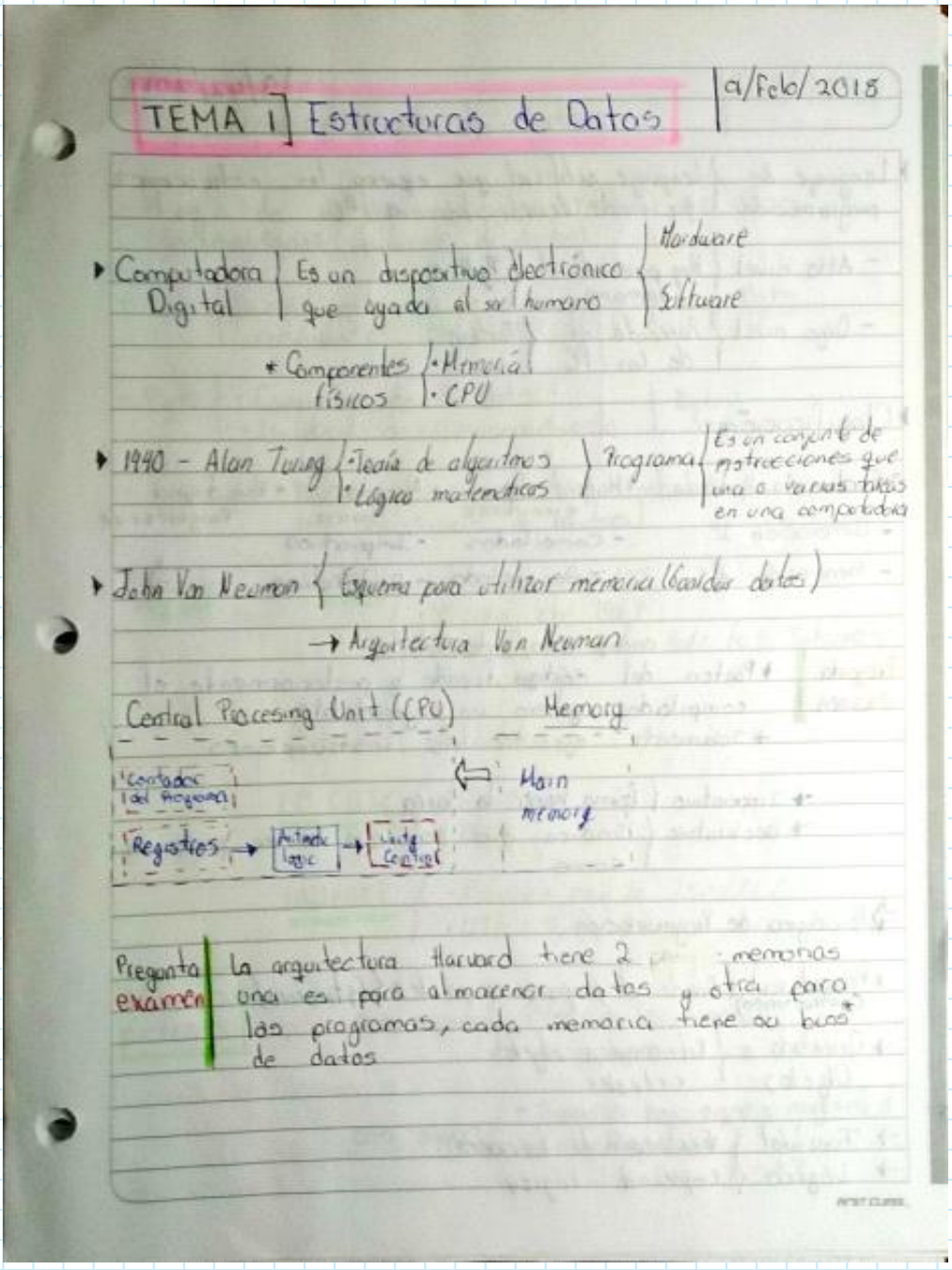
Formato de envío | MurrietaVillegasAlfonso G1 P_ V_ } Correo (espacios; grupo; Número de práctica; versión de la práctica)

MurrietaVillegasAlfonsoG1P V } Archivo
* punto

edaI01alu32
313048937

INTRODUCCIÓN

miércoles, 4 de abril de 2018 12:52 p. m.



9/02/2017

▸ Lenguaje de programación { Lenguaje artificial que expresa las instrucciones que puede llevarse en una PC

- Alto nivel { Muy parecido al humano { Python
- Bajo nivel { Parecido al de las PC { Fortran

▸ Clasificaciones

• Evolución Histórica	• Manera de ejecutarse	• Abordar tarea	• Paradigma Programación
- Generación 1°	- Compilados	- Imperativos	
- Generación 2°	- Interpretado	- Declarativos	

compilado/interpretado

Pregunta examen → Parten del código fuente y posteriormente el compilador genera un ejecutable
→ Solamente ejecuta las instrucciones

→ Imperativo { Cómo hacer la tarea
→ Declarativo { Indica que hay que hacer pero no como

⇒ Paradigma de Programación

→ Procedural (Estructurado) { Divide el programa en partes { Estructural
→ Orientado a Objetos { Referencia a objetos virtuales
→ Funcional { Evaluación de funciones
→ Lógico { Empleando lógica

REPRESENTACIÓN DE DATOS

miércoles, 4 de abril de 2018 01:24 p. m.

1.1 Representación de datos — 9/02/2017

- Manejo de datos mediante PATRÓN DE BITS (Representación uniforme de datos)
- Bit | Unidad mínima de almacenamiento mediante 0 y 1 | Bits
- Byte | - Conjunto de 8 Bits - Unidad de almacenamiento | Bytes
- Codifica: Proceso de representar símbolos (De los tipos de datos)

Códigos

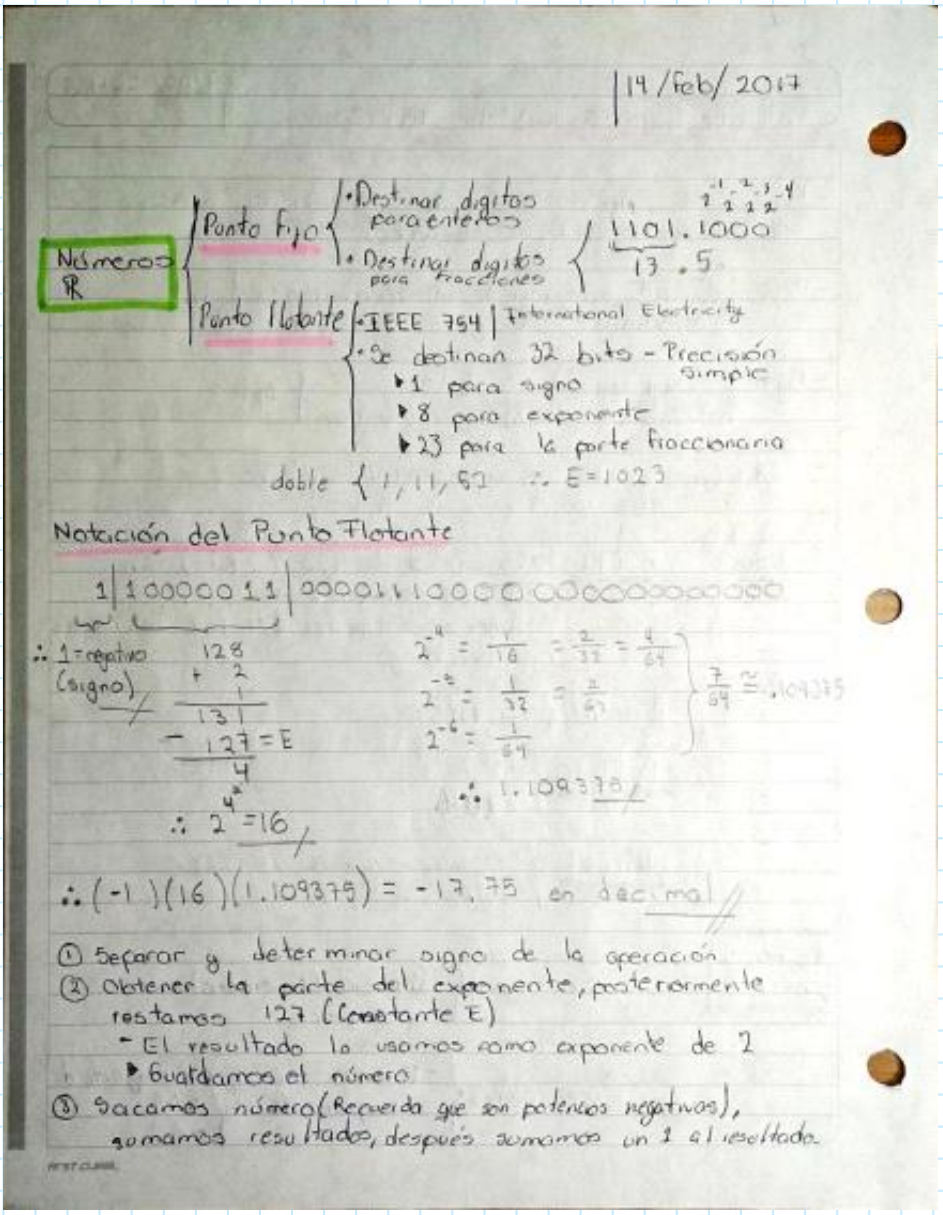
- **ASCII**
  - Es capaz de 128 (7 bits)
  - Se creó en 1967
  - American standar Code for Information Interchange
- (Extended)
  - 8 bits
- **EB CDIC**
  - 8 bits
  - IBM
- **UNICODE**
  - Relación con la ISO/IEC
  - UTF 8, 16, 32

Números Enteros

- sin signo
  - Binaria Convencional o pura
- con signo
  - Binario con signo y magnitud
  - Complementario
    - A1
    - A2

PUNTO FIJO Y PUNTO FLOTANTE

miércoles, 4 de abril de 2018 01:25 p. m.



14/feb/2017

- 340.475 Decimal → Binario

- Verificar signo
- Convertir la parte entera a binario
- Recorrer hasta el último 1
- A la constante c = 127 se le suma n
- Convertir el num. obtenido del paso anterior a binario | Exponente
- Concadenar 1 (signo), los obtenidos del anterior y después los saltados
- La parte real se debe multiplicar por 2

1 | 1.0000111 | 1 0 10 10 100 0111100110000

Signo (1) | Exponente (8) | Parte entera (23)

340 / 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1
1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 → 1.0101010.0
n = 8

E n
127 + 8 = 135

135 / 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | Exponente
1 | 0 | 0 | 0 | 0 | 1 | 1 | 1

.475 x 2 = 0.95
.95 x 2 = 1.90
.9 x 2 = 1.8
.8 x 2 = 1.6
.6 x 2 = 1.2
.2 x 2 = 0.4
.4 x 2 = 0.8

.8 x 2 = 1.6
.6 x 2 = 1.2
⋮

① se repite
② se corta en 0
③ ajustadas

Ejercicios de Clase | 14/02/2018

a) 1 | 10001011 | 0011 0000 0000 0000 0000 000

▶ Negativo

128	64	32	16	8	4	2	1
1	0	0	0	1	0	1	1

139
-127
012

$2^{-3} = \frac{1}{8} = \frac{2}{16}$ | $\frac{3}{16} = .1875$

$2^{-4} = \frac{1}{16} =$

▶ $2^{12} = 4096$

se agrega 1

$\therefore (-1)(4096)(1.1875) = -4.864$

---

b) 35.375

I) Signo positivo { 0

II) Parte entera 35 { 32 | 16 | 8 | 4 | 2 | 1 ; 1 | 0 | 0 | 0 | 1 | 1 { 1.0001 ; n=5

III) 127 + 5 = 132 { 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 ; 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | Exponente

IV) .375 x 2 = 0.75
.75 x 2 = 1.5
.5 x 2 = 1
: 0

8 | 15 ceros

$\therefore$ 35.375 = 0 | 10000100 | 00011011 | 00000000000000

n | M

c) -145.44

I) Signo { 1

II) 145 { 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 / 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 { 1.00100001 n=7

III) 127 + 7 = 134

128	64	32	16	8	4	2	1	
1	0	0	0	0	1	1	0	Exponente

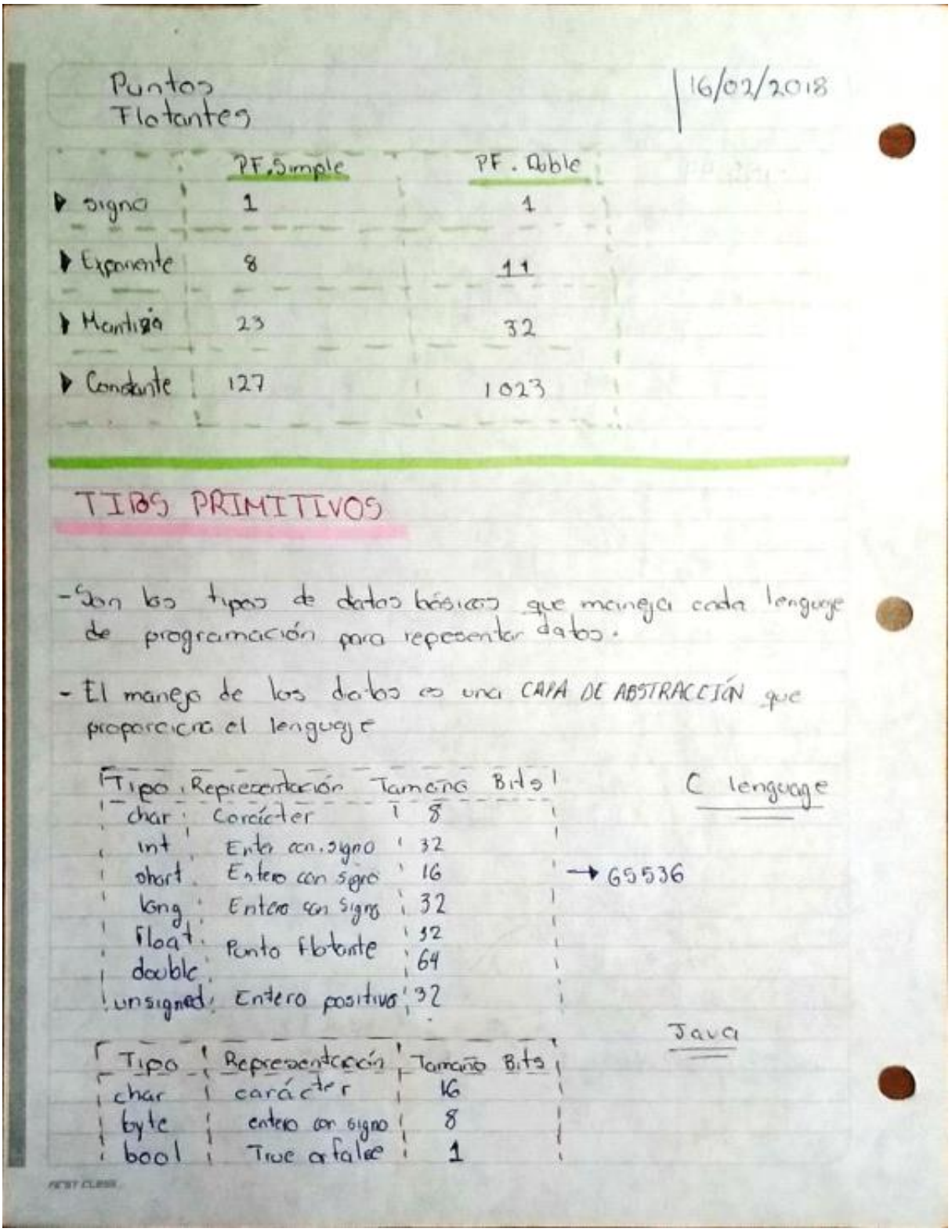
IV)
.44 x 2 = 0.88
.88 x 2 = 1.76
.76 x 2 = 1.52
.52 x 2 = 1.04
.04 x 2 = 0.08
.08 x 2 = 0.16
.16 x 2 = 0.32
.32 x 2 = 0.64
.64 x 2 = 1.28
.28 x 2 = 0.56
.56 x 2 = 1.12
.12 x 2 = 0.24
.24 x 2 = 0.48
.48 x 2 = 0.96
.96 x 2 = 1.92
.92 x 2 = 1.84

16 números

-145.44 = 1 | 10000110 | 00100001011100001010011

TIPOS PRIMITIVOS

Wednesday, April 4, 2018 6:01 PM



Puntos Flotantes — 16/02/2018

	PF. Simple	PF. Doble
Signo	1	1
Exponente	8	11
Mantisa	23	32
Constante	127	1023

TIPOS PRIMITIVOS

- Son los tipos de datos básicos que maneja cada lenguaje de programación para representar datos.
- El manejo de los datos es una CAPA DE ABSTRACCIÓN que proporciona el lenguaje

C lenguage

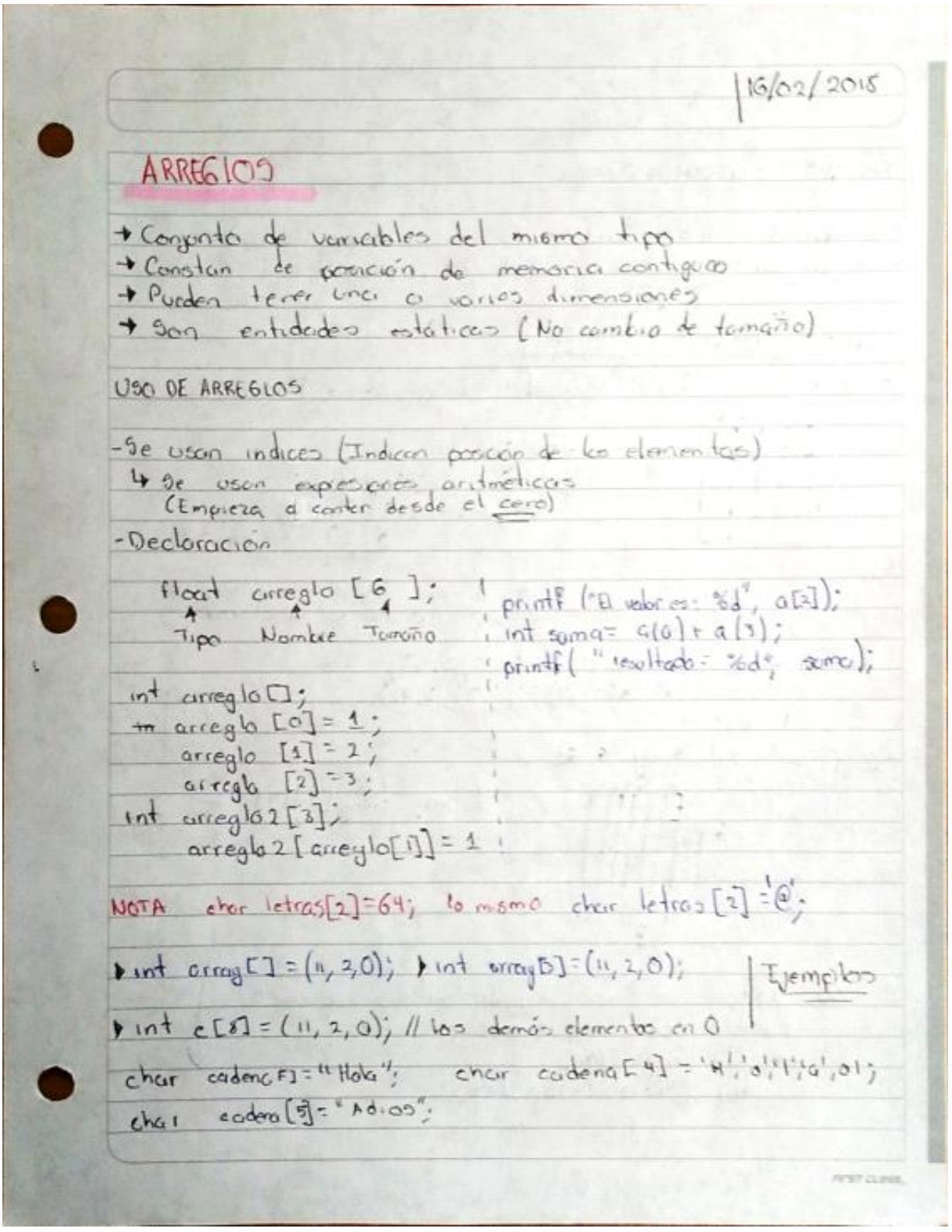
Tipo	Representación	Tamaño Bits
char	Carácter	8
int	Entero con signo	32
short	Entero con signo	16 → 65536
long	Entero con signo	32
float	Punto flotante	32
double		64
unsigned	Entero positivo	32

Java

Tipo	Representación	Tamaño Bits
char	carácter	16
byte	entero con signo	8
bool	True or false	1

ARREGLOS

Wednesday, April 4, 2018 6:02 PM



16/02/2018

ARREGLOS

- Conjunto de variables del mismo tipo
- Constan de porción de memoria contigua
- Pueden tener una o varias dimensiones
- Son entidades estáticas (No cambio de tamaño)

USO DE ARREGLOS

-Se usan indices (Indican posición de los elementos)
  - Se usan expresiones aritméticas
    (Empieza a contar desde el cero)

-Declaración

float arreglo [6];
Tipo Nombre Tamaño

```
int arreglo[];
arreglo [0] = 1;
arreglo [1] = 2;
arreglo [2] = 3;
int arreglo2 [3];
arreglo2 [arreglo[1]] = 1
```

```
printf ("El valor es: %d", a[2]);
int suma= a[0] + a[3];
printf( "resultado= %d", suma);
```

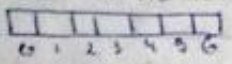
NOTA char letras[2]=64; lo mismo char letras [2] ='@';

Ejemplos

```
int arrag[] = (11, 2, 0);   int arrag[5] = (11, 2, 0);
int c[8] = (11, 2, 0); // los demás elementos en 0
char cadena[] = "Hola";   char cadena[4] = 'H','o','l','a',0;
char cadena[5] = "Adios";
```

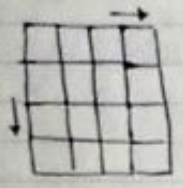
ARREGLOS UNIDIMENSIONALES

- Constan de un solo índice (se conoce como Vector)
- Se almacenan de forma adyacente, representación simple



ARREGLOS BIDIMENSIONALES

- Tienen 2 dimensiones (se conocen como Matrices)



- Declaración

```
int c [6][8][1];
int matriz[8][3];
```

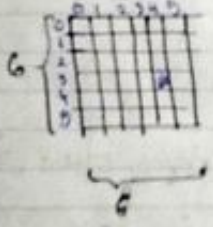
(3 ← columna, 8 ← renglón)

▶ Las dimensiones agregadas son los primeros en declarar

Declarar:

```
int a[2][2] = {1,2,3,4};
int a[2][2] = {{1,2},{3,4}};
```

Ejemplo: `int array [6][6]`



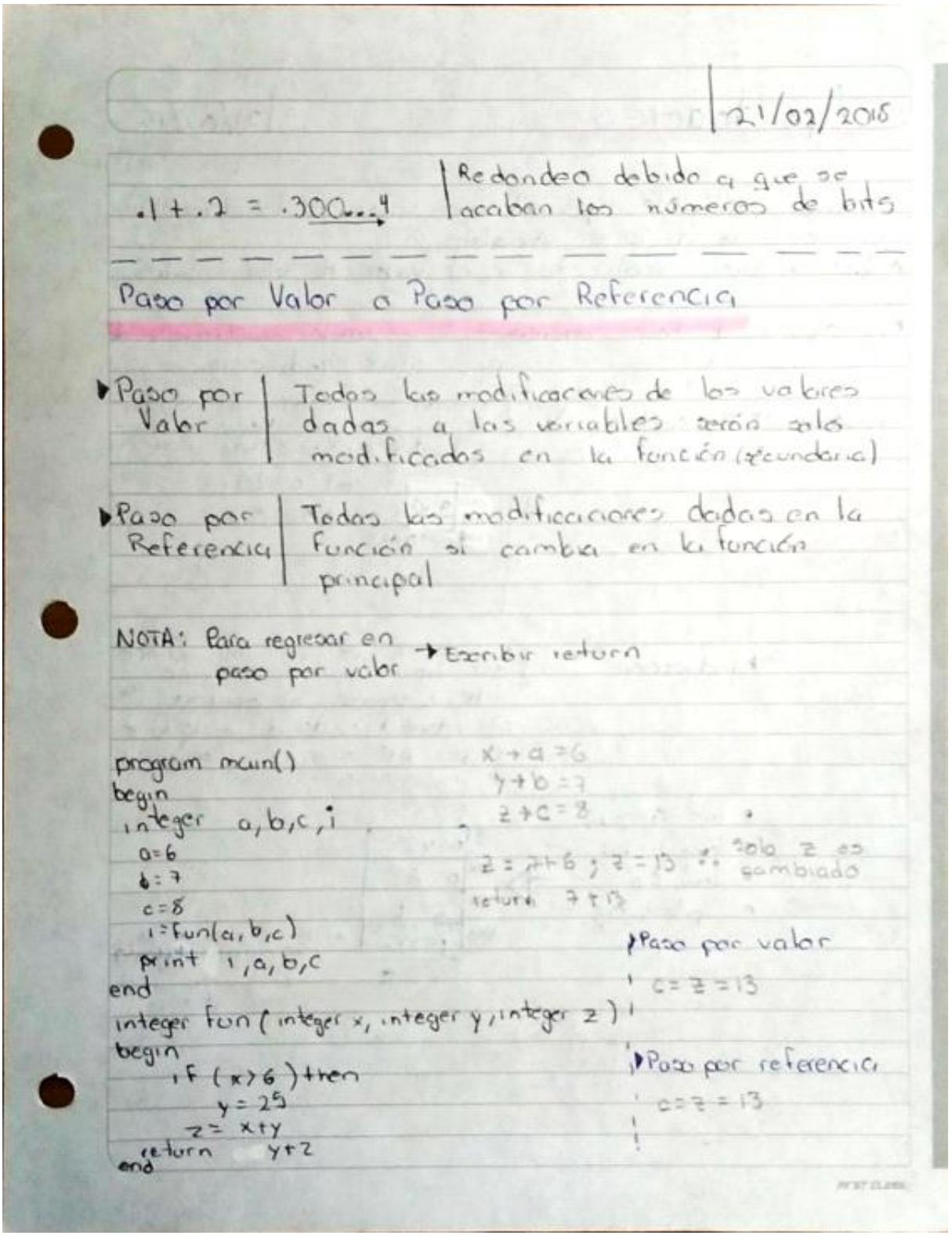
∴ `int array[3][4] = 2;`

* Los datos de los arreglos unidimensionales se guardan de manera lineal

```
int i, arreglo[100];
for ( i=0, i<100, i++)
```

PASO POR VALOR Y REFERENCIA

Wednesday, April 4, 2018 6:02 PM



21/02/2018

.1 + .2 = .300...4 | Redondeo debido a que se acaban los números de bits

Paso por Valor a Paso por Referencia

- Paso por Valor | Todas las modificaciones de los valores dadas a las variables serán solo modificadas en la función (secundaria)
- Paso por Referencia | Todas las modificaciones dadas en la función si cambia en la función principal

NOTA: Para regresar en paso por valor → Escribir return

```
program main()
begin
 integer  a,b,c,i
  a=6
  b=7
  c=8
  i=fun(a,b,c)
  print i,a,b,c
end
integer fun( integer x, integer y, integer z )
begin
    if (x>6) then
       y=25
     z= x+y
  return    y+z
end
```

x → a = 6
y → b = 7
z → c = 8

z = z+6 ; z = 13 ∴ solo z es cambiado

return 7+13

Paso por valor
c = z = 13

Paso por referencia
c = z = 13

APUNTADORES

Wednesday, April 4, 2018 6:03 PM

Apuntadores

21/02/18

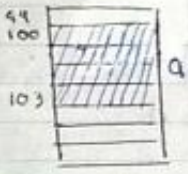
Es una variable especial cuyo valor es la dirección de memoria de una variable

- Hace mención indirecta a el valor de una variable

- El proceso de hacer referencia a un valor a través de un apuntador se le conoce como indirección

Operadores → Dirección "&" | cuando se antepone al nombre de la variable hace referencia a la dirección de memoria

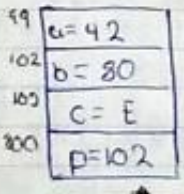
int a = 50;
&a = 100



→ Indirección "*" | cuando se utiliza una variable del tipo apuntador se debe anteponer el operador "*" al identificador, de esta forma es posible acceder al espacio de memoria

NOTA: El apuntado debe ser del mismo tipo

- int a=42;
- long b=80;
- long *p;
- p = &b;



← Dirección de la variable guarda se guarda en alguna parte

1] Declarar el apuntador → 2] Asociar a una variable → 3] utilizar

21/02/2018

Ejemplo ①

```
int a=80; // Declaración variable
int *p; // Declaración apuntador
p = &a; // Asignar a una variable
*p= 12; // Se cambia el valor indirectamente
printf("a = %d", a);
```

→ Salida a= 12

Ejemplo ②

```
int a=5, b=20, c = 10, d;
int *p1, *p2;
p1 = &c; // Asigna a "c" = 10
p2 = &b; // Asigna a "b" = 20
d= *p1 + *p2;
printf ("d = %a", d);
```

→ Salida d = 30

Operador "*=" { *px *= x*2;
(14) * (14*2); (14)*(28); 392 //

Apuntadores 3 elementos necesarios
- Declaración de la variable
- Asociación con una dirección de memoria
- Asignación con valores

27/02/2018

Corrección → EJERCICIO-4

```
Procedure mystery
    a: integer;
    b: integer;
        procedure enigma(x,y)
        begin
                y = y+b;
                x = b+x;
                b = x+b;
                a = y
        end enigma
begin
    a=2 ; b=7;
    enigma (a,b);
    write (a); write (b);
end mistery
```

SALIDAS

▸ Pasa por Valor
*No se modifican "a" y "b" (Variables globales)

a = 2
b = 7

y = y+b; y = 7+7; y = 14
x = b+x; x = 7+2; x = 9
b = 9+7; b = 16
a = y = 14

//Al finalizar * x y y se mueren*
y por lo tanto la salida de a y b
a = 14 y b = 16

* Cuando variable es retornada a global (Ahí si se cambia)

23/02/2018

▸Paso por referencia

a = 2 ; b = 7;
∴ a = x;
b = y;
// En paso por referencia se cambian valores

y = y + b ; y = 7 + 7 ; y = 14;
x = b + x = x = 14 + 2 ; x = 14 + 2 ; x = 16;
b = x + b ; b = 16 + 14 ; b = 30;
a = y = 30;

Recordar que se cambian los valores

SALIDA
a = 30 // b = 30 //

PASOS DE APUNTADORES

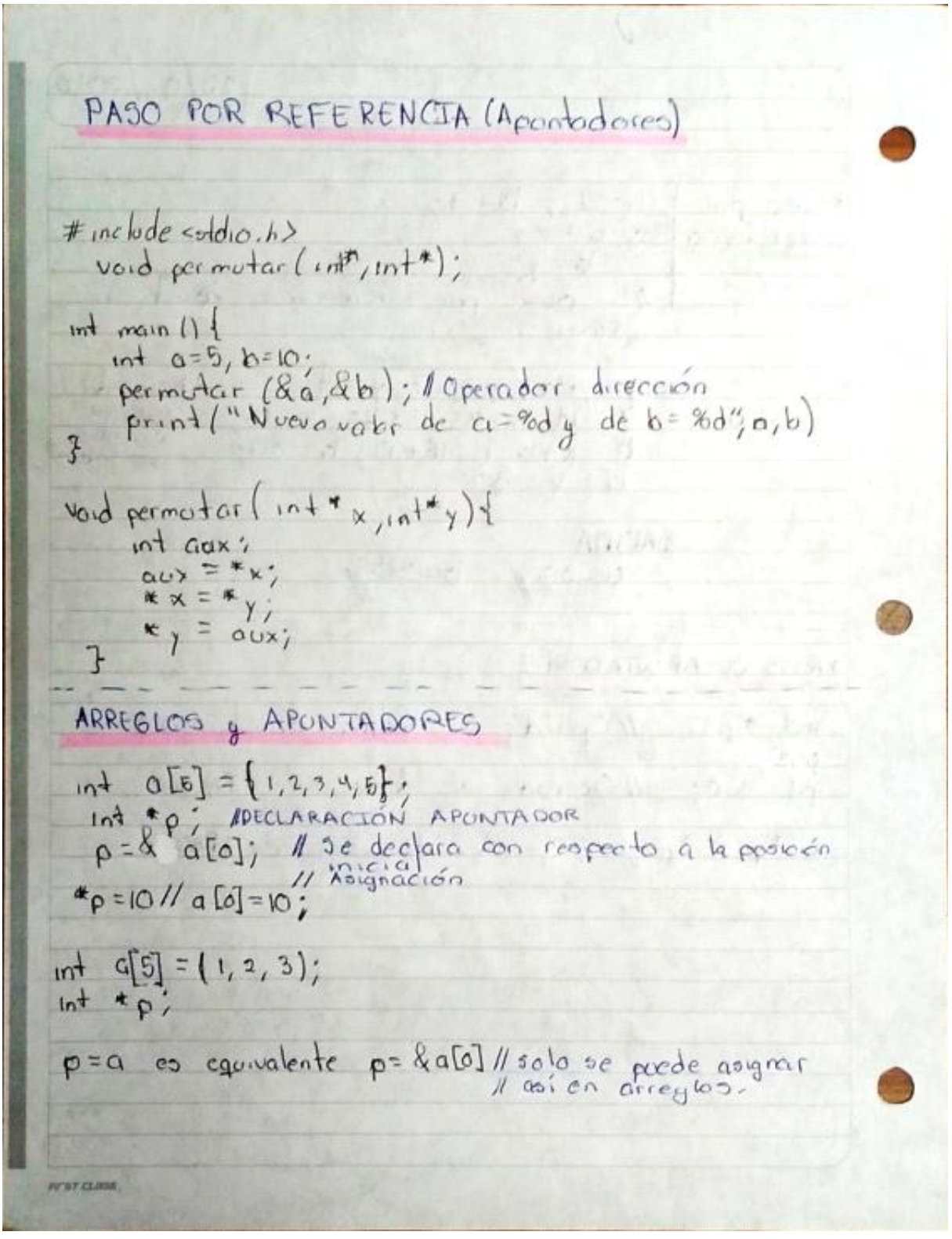
int *p1; // Operador de indirección | PASO 1

p1 = &c; // Operador de dirección | PASO 2

d = *p1 + *p2; // Operador de indirección | PASO 3

PASO POR REFERENCIA

Wednesday, April 4, 2018 6:04 PM



PASO POR REFERENCIA (Apuntadores)

```
#include <stdio.h>
  void permutar(int*, int*);

int main(){
   int a=5, b=10;
   permutar(&a,&b); // Operador dirección
   print("Nuevo valor de a=%d y de b= %d", a, b)
}

void permutar( int *x, int*y){
    int aux;
    aux = *x;
    *x = *y;
    *y = aux;
 }
```

ARREGLOS y APUNTADORES

```
int a[5] = {1,2,3,4,5};
 int *p; //DECLARACIÓN APUNTADOR
 p=& a[0]; // Se declara con respecto a la posición
                 // inicial
*p=10 // a[0]=10; Asignación

int a[5] = (1, 2, 3);
int *p;

p=a es equivalente p= &a[0] // solo se puede asignar
                            // así en arreglos.
```

23/02/2018

Modificar los elementos

▶ Precedencia de operadores

```
*p = 10   // a[0] = 10;

*(p+3) = 5 // a[3] = 5;
```

cambiar valor de la ~~tercera~~ cuarta celda del arreglo

`*p+3 = ?`

Jerarquía de operaciones
∴ Primero apuntador

```
#Include <stdio.h>
int  arreglo [ ] ={3,25,19,6,5,300};
int  *qp;

int main (void) {
int i;
 qp = arreglo;
 printf ("\n\n");
 for ( i=0 ; i<6; i++) {
      printf ("arreglo[%od]= %d ", i, arreglo[
      prin

 }
return 0;
}
```

CAPAS DE ABSTRACCIÓN

Wednesday, April 4, 2018 6:05 PM

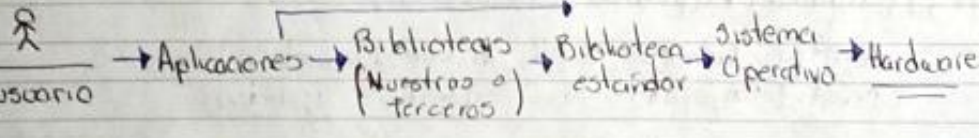
28/02/2018

Operador de estructura (->)
Tiene acceso a un miembro de estructura vía un apuntador a la estructura
opAlu -> edad = 36;

opAlu = &var;
var.nombre = "Alfonso";
var.edad = 35
opAlu -> Alu = 36
var2.nombre = var.nombre

CAPAS DE ABSTRACCIÓN

- El programador no asume cuestiones de plataforma
  - Cada función se influye como parte de la biblioteca estándar <stdlib>

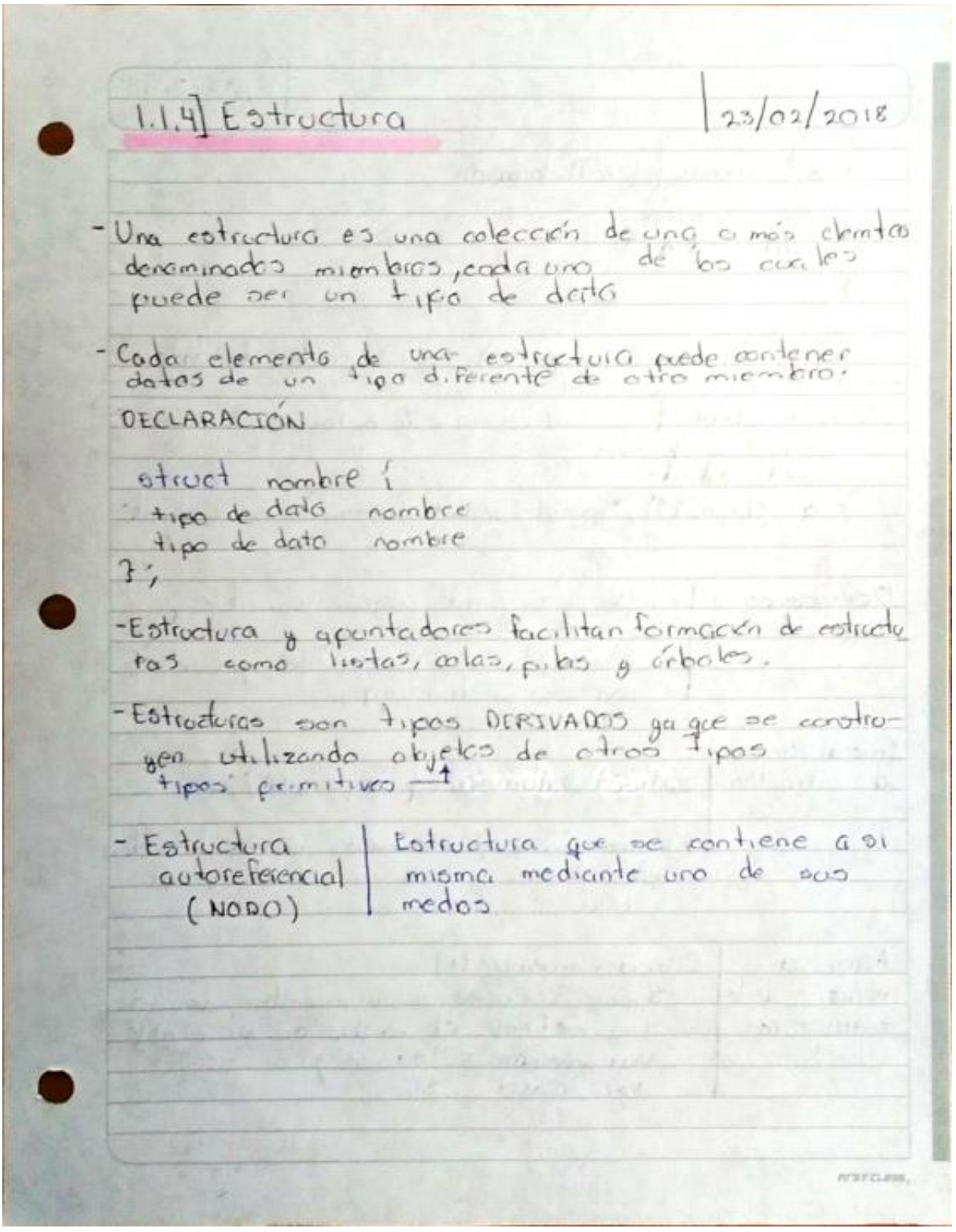


Tipo Abstracto de DATO (TAD)

- Tipo de dato abstracto es una extención de Tipos de Datos donde el programador puede crear sus propios datos
- fopen, fclose, fread, fwrite componen un TAD

ESTRUCTURAS

Wednesday, April 4, 2018 6:05 PM



1.1.4] Estructura | 23/02/2018

- Una estructura es una colección de una o más variables denominados miembros, cada uno de los cuales puede ser un tipo de dato
- Cada elemento de una estructura puede contener datos de un tipo diferente de otro miembro.

DECLARACIÓN

```
struct nombre {
  tipo de dato  nombre
  tipo de dato  nombre
};
```

- Estructura y apuntadores facilitan formación de estructuras como listas, colas, pilas y árboles.
- Estructuras son tipos DERIVADOS ya que se construyen utilizando objetos de otros tipos (tipos primitivos)

- Estructura autoreferencial (NODO)	Estructura que se contiene a sí misma mediante uno de sus medios

28/02/2018

```
struct alumno {  // Declaración
    char *nombre;
    int  edad;
};

struct alumnoa, grupo(45), *point; //Uso de datos

Struct alumno{      // Versión 2 o forma 2
    char *nombre;
    int edad;
} a, grupo[45], *point; // Esta forma no es conveniente
```

Operaciones con estructuras

- Solo se se puede asignar variables de estructuras
- La estructura no se puede operar con otra estructura

Inicialización de estructuras

```
struct alumno alu1={"pepe", 29};
```

- Si no aparecen elementos en la estructura estos tendrá o será inicializado en 0 o NULL

Acceso a miembros de la estructura

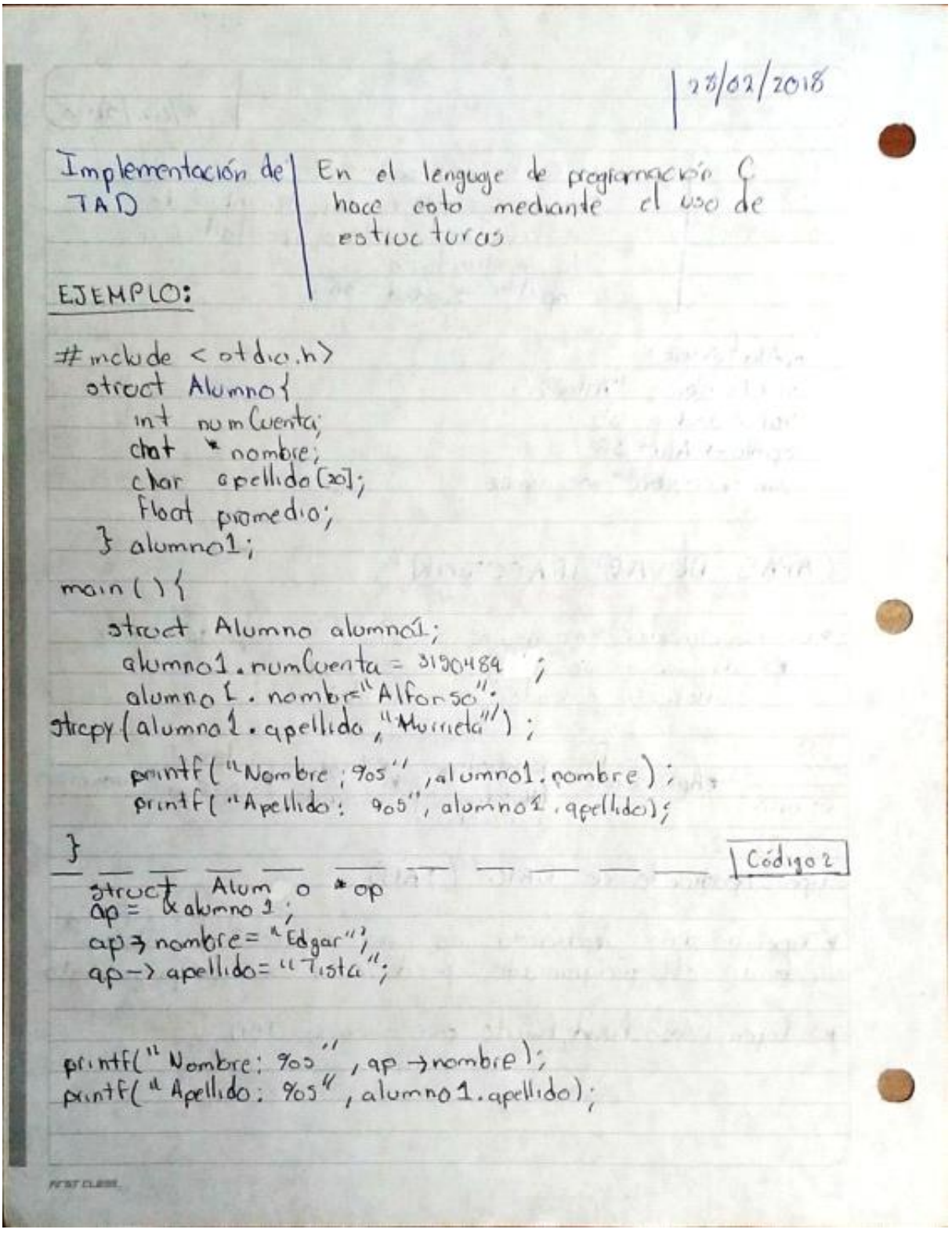
Operador miembro (.)

Tiene acceso a un miembro de estructura mediante el nombre de la variable

```
var.nombre = "Alfonso";
var.edad = 35
```

EJEMPLOS ESTRUCUTRAS

Wednesday, April 4, 2018 6:05 PM



28/02/2018

Implementación de TAD

En el lenguaje de programación C hace esto mediante el uso de estructuras

EJEMPLO:

```
#include <stdio.h>
  struct Alumno{
      int numCuenta;
      char *nombre;
      char apellido[20];
      float promedio;
   } alumno1;
main(){
    struct Alumno alumno1;
     alumno1.numCuenta = 3130484 ;
     alumno1.nombre="Alfonso";
strcpy(alumno1.apellido,"Murrieta");
     printf("Nombre: %s",alumno1.nombre);
     printf("Apellido: %s", alumno1.apellido);
}
```

Código 2

```
struct Alumno *ap
ap = &alumno1;
ap->nombre="Edgar";
ap->apellido="Tista";

printf("Nombre: %s", ap->nombre);
printf("Apellido: %s", alumno1.apellido);
```

typedef | Proporciona un mecanismo para la creación de sinónimos o alias al tipo de dato

```
typedef int Enteros;
Entero a = 10;
Entero b = 20;
```

EJEMPLO

```
#include <stdio.h>
struct alumno{
  ...
};
typedef struct alumno Alumno;
```

Tipo, Nombre, Nuevo Nombre

Caso 1

```
main(){
  Alumno alumno1;
  Alumno alumno2;
  alumno1.numCuenta = 31904;
  alumno1.nombre = "Alfonso";
  printf("Nombre: %s", alumno1.nombre);
```

```
typedef struct{
  int numCuenta;
  char *nombre;
} Alumno;
```

Caso 2 | Asignamos a la estructura Alumno

Ejercicio: (Trabajo) 12/03/2018

```
typedef struct{
    char* calle;
    int numero;
    char* colonia;
}Direccion;
// Las variables son despues
```

①

```
struct Alumno{
    ...
    struct Direccion domicilio;
};// Tipo y identificador
```

```
struct Alumno{
    ...
    Direccion domicilio
};// En caso de usar typedef
```

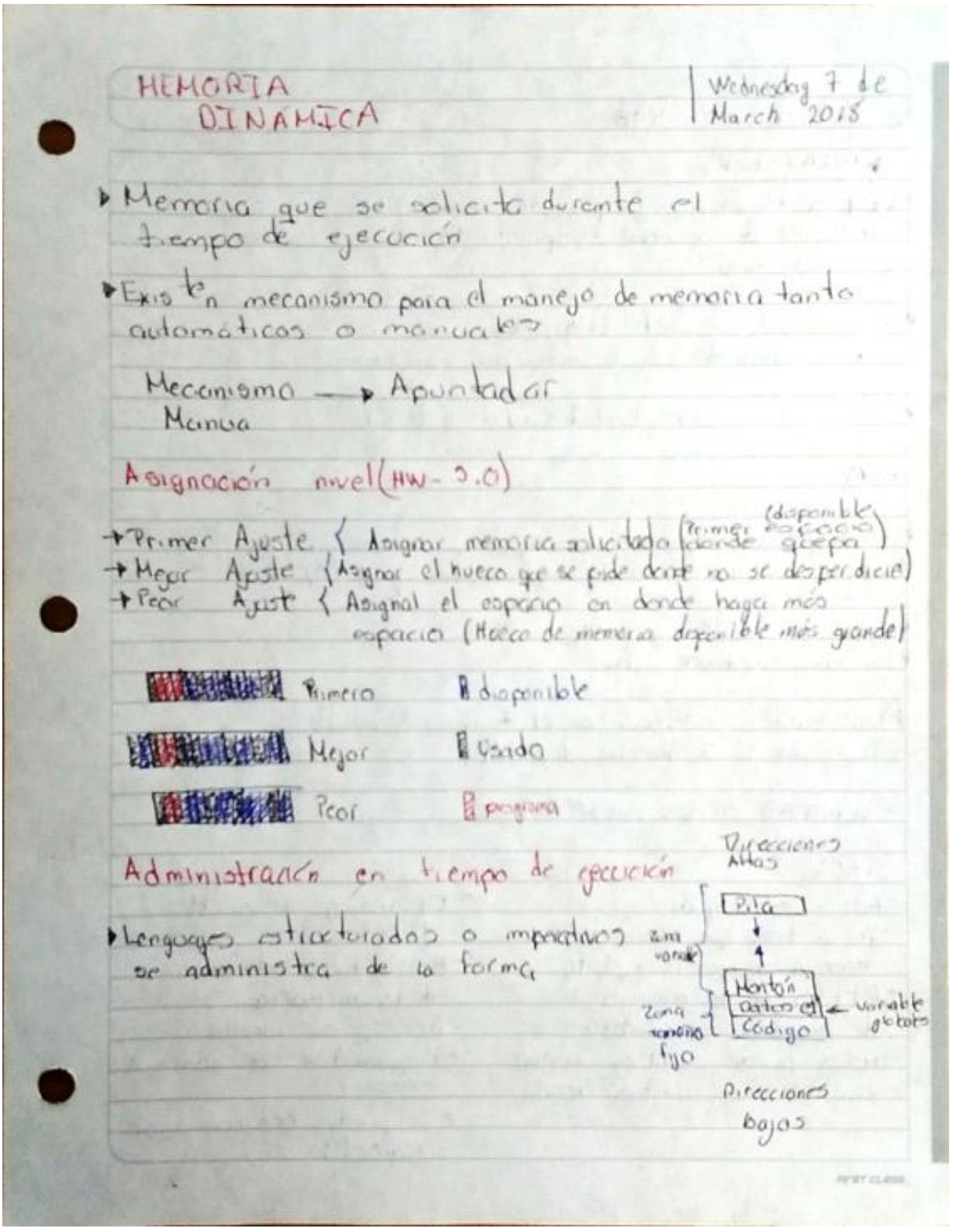
```
main(){
    struct Alumno *apAlumno, alumno1;
    apAlumno = &alumno1;
    alumno1.nombre = "Tiota";
    apAlumno->apellido = "Tiota"; // manera indirecta
    alumno1.domicilio.calle = "calle falsa";
    ...
    apAlumno->domicilio.calle = "Benito"; // para cambiar dato mediante apuntador */
    // apuntador  ogro  struct  elemento
}
```

NOTA: La mejor práctica es usar funciones para optimizar cosas

TAREA 4
- Investigar como se almacenan y distribuyen en memoria
- Ejemplo mediante un esquema

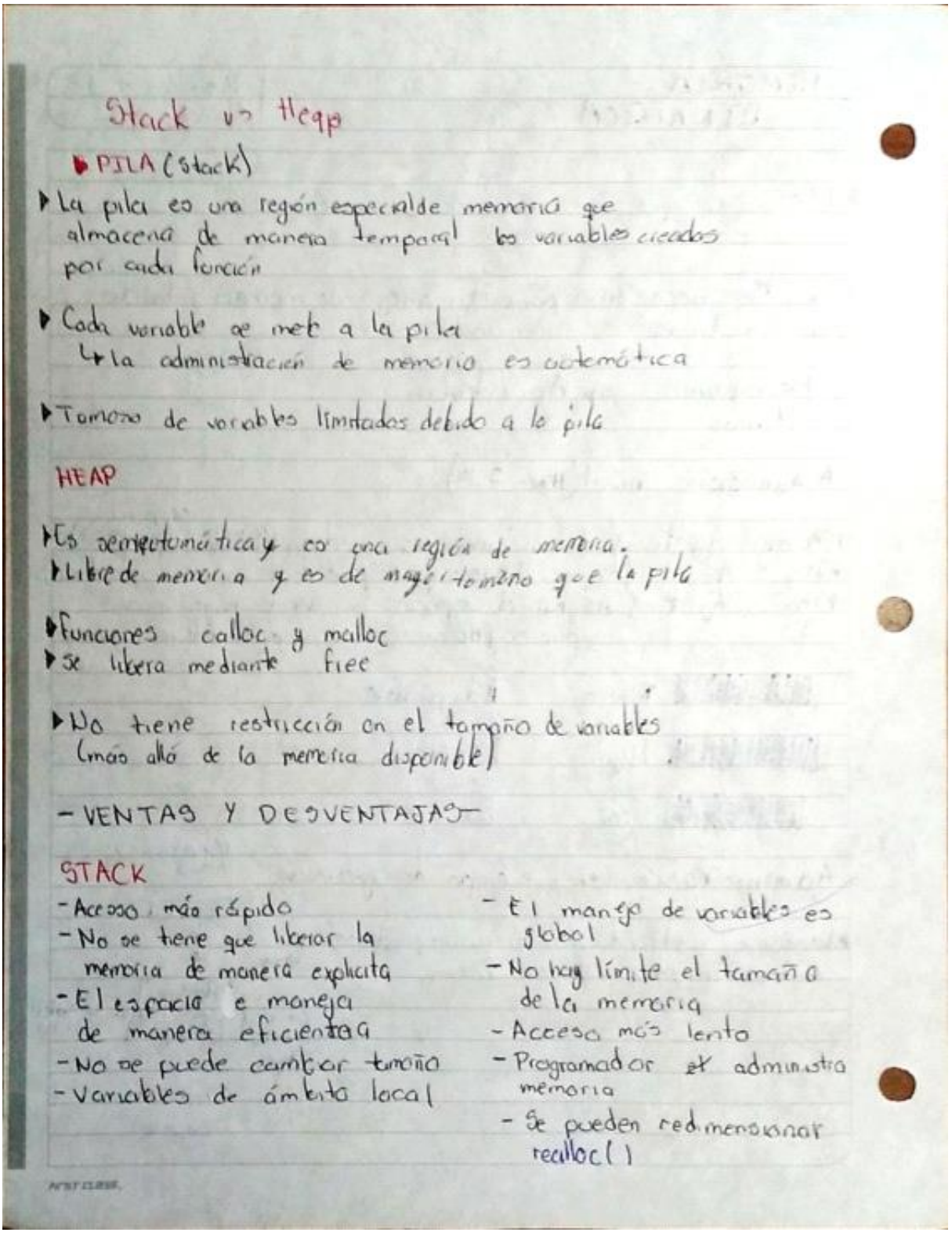
MEMORIA DINÁMICA

Wednesday, April 4, 2018 6:06 PM



STACK 7 HEAP

Wednesday, April 4, 2018 6:06 PM



Stack vs Heap

PILA (stack)

- La pila es una región especial de memoria que almacena de manera temporal las variables creadas por cada función
- Cada variable se mete a la pila
  - La administración de memoria es automática
- Tamaño de variables limitados debido a la pila

HEAP

- Es semiautomática y es una región de memoria.
- Libre de memoria y es de mayor tamaño que la pila
- Funciones calloc y malloc
- Se libera mediante free
- No tiene restricción en el tamaño de variables (más allá de la memoria disponible)

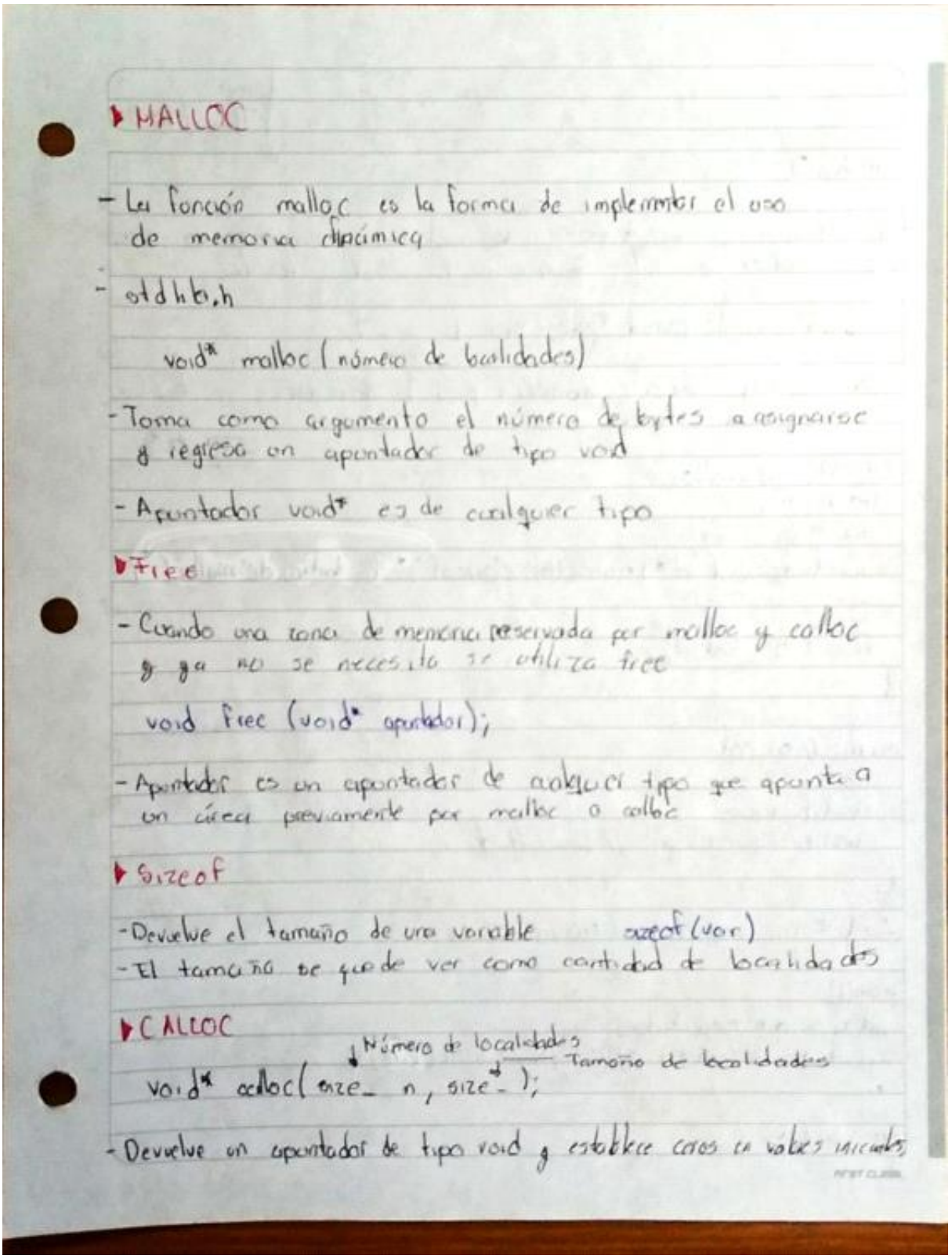
- VENTAS Y DESVENTAJAS -

STACK

- Acceso más rápido
- No se tiene que liberar la memoria de manera explícita
- El espacio e maneja de manera eficiente
- No se puede cambiar tamaño
- Variables de ámbito local
- El manejo de variables es global
- No hay límite el tamaño de la memoria
- Acceso más lento
- Programador el administra memoria
- Se pueden redimensionar realloc()

MALLOC/ CALLOC / REALLOC

Wednesday, April 4, 2018 6:07 PM



MALLOC

- La función malloc es la forma de implementar el uso de memoria dinámica
- stdlib.h

  void* malloc (número de localidades)

- Toma como argumento el número de bytes a asignarse y regresa un apuntador de tipo void
- Apuntador void* es de cualquier tipo

Free

- Cuando una zona de memoria reservada por malloc y calloc ya no se necesita se utiliza free

  void free (void* apuntador);

- Apuntador es un apuntador de cualquier tipo que apunta a un área previamente por malloc o calloc

Sizeof

- Devuelve el tamaño de una variable sizeof(var)
- El tamaño se puede ver como cantidad de localidades

CALLOC

Número de localidades / Tamaño de localidades

void* calloc(size_t n, size_t);

- Devuelve un apuntador de tipo void y establece ceros en todos los bits inicializados

17/03/2016

REALLOC

- redistribuye un área de memoria la cual fue previamente asignada por malloc y calloc y que no ha sido liberada

```
void* realloc(void *ptr, size_t size);
```

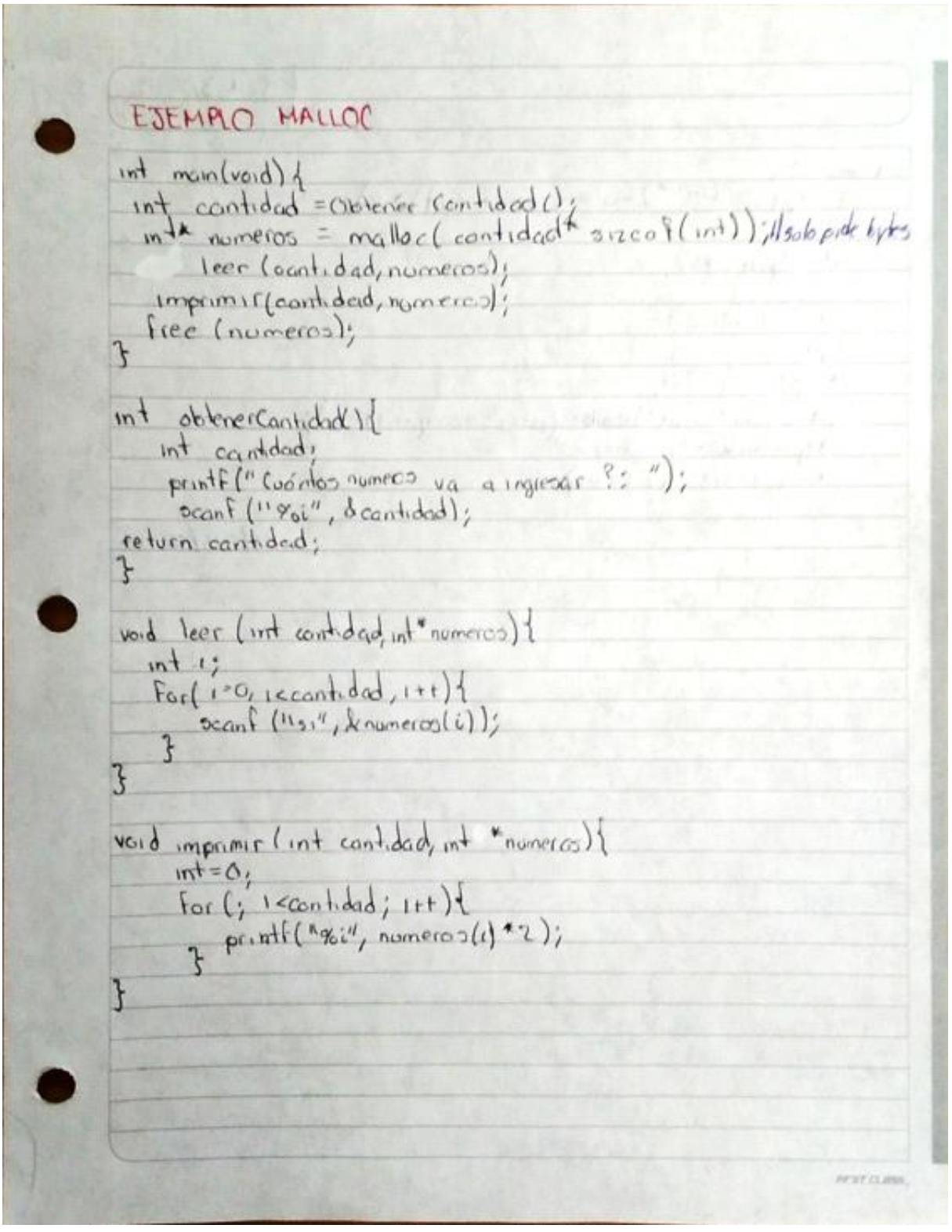
- Si no hay suficiente memoria para la distribución se devuelve un apuntador nulo

EJEMPLO

```
main(){ // malloc
 int a=10;
 int *apuntador;
 apuntador = (int*) malloc(4); //(sizeof)
  ...
  ...
 free (apuntador);
}

main(){ // sizeof
  int a=10;
  float f=100;
  int b= sizeof(a); // El resultado es entero
}

:.float b = sizeof(a); //NO es válido

main(){
int a=10;int *apuntador;
apuntador = (int*) calloc (4, sizeof(int));
 ...
 ...
free apuntador;
}
```

EJEM.1

Wednesday, April 4, 2018 6:07 PM



EJEMPLO MALLOC

```
int main(void){
 int cantidad =ObtenerCantidad();
 int* numeros = malloc( cantidad* sizeof(int));//solo pide bytes
      leer (cantidad, numeros);
   imprimir(cantidad, numeros);
  free (numeros);
}

int obtenerCantidad(){
   int cantidad;
   printf(" Cuántos numeros va a ingresar ?: ");
    scanf ("%d", &cantidad);
return cantidad;
}

void leer (int cantidad, int* numeros){
  int i;
  for( i=0; i<cantidad, i++){
     scanf ("%d", &numeros(i));
   }
}

void imprimir (int cantidad, int *numeros){
    int=0;
    for (; i<cantidad; i++){
       printf("%d", numeros(i)*2);
     }
}
```