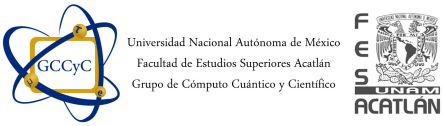
Programación Avanzada en C

Edificio de la división de
Matemáticas e Ingeniería
Oficina 10.



Proyecto PAPIME PE104919

Actualizado el: 5 de mayo de 2020

jaorduz@comunidad.unam.mx
Version 1.0

Tabla de Contenido

Licencia

Este documento está creado con fines educativos. La información contenida está sometida a cambios y a revisiones constantes, por lo que se sugiere no imprimirlo.

Puedes compartir el documento con quien desees; sin embargo debes respetar la autoría original. Puedes citar el material y considerarlo como referencias en tus proyectos.

Para citar este documento:

1. Bibtex

```
@manual{introCAdGCCyC,
title  ="{Programaci\'on avanzada con C}",
author ="{Orduz-Ducuara, J.A.}",
organization ="{Grupo de C\'omputo Cu\'antico y Cient\'ifico}",
month ="{Febrero}",
year  = "{2020}",
address ="{FESAC-UNAM}",
note ="Los miembros del GCCyC aporta constantemente al desarrollo
de este proyecto}",
}
```

2. bibitem

```
\bibitem{introCAdGCCyC}
Orduz-Ducuara, J.A.
\textit{Programaci\'on avanzada con C}.
Los miembros del GCCyC aporta constantemente al desarrollo
de este proyecto
GCCyC, FESAC-UNAM, 2020.
```

Miembros activos del GCCyC:
Salvador Uriel Aguirre Andrade
Miguel Aguilar Hilario
Miguel González Briones
Alfonso Flores Zenteno
Ana Karen Del Castillo
Diana De Luna
Leslie Valeria Vivas Laurrabaquio
Zitlalli Nayeli Avilés Palacios
Eledtih Andrea González Sánchez
Edgar Ivan Martínez Villafañe
Oscar Jair Vargas Palacios

Dr. Javier A. Orduz-Ducuara [a]
Profesor de Carrera Asociado C.
Edificio de la división de Matemáticas e Ingeniería
Oficina 10.
FES Acatlán-UNAM

[a] Miembro que contribuyó al desarrollo de este material

Parte I: Introducción

Introducción general del curso

Aprender un lenguaje de programación es similar, en cierto modo, a aprender un nuevo idioma diferente al nativo. Un lenguaje de programación, se parece a un idioma, tiene su sintaxis (reglas del lenguaje), asimismo, en computación, existe el concepto de semántica: en pocas palabras la matemática detrás de la programación. Este conjunto de características son propias de cada lenguaje. Al final, un lenguaje de programación y un idioma son formas de expresión y de comunicación. Este aprendizaje puede implicar tanta pasión y compromiso como el discente quiera: la satisfacción por una nueva forma de expresión es propia de cada individuo. Después de incluir nuevas palabras, conceptos y reglas a la forma de expresión, seguramente, el lector se sentirá satisfecho y optará por aprender un nuevo lenguaje.

En estas notas se dejan las bases de la programación en C. Por cuestiones de fluidez, este documento omite la semántica de la programación y expone los principios de la sintaxis del lenguaje. El propósito es compartir el conocimiento con aquellos interesados en el autoaprendizaje y el desarrollo de habilidades computacionales, más que fortalecer impartir la lógica detrás de la programación. Sin embargo, es importante que el lector comprenda que la programación tiene profundos conceptos matemáticos que no se discuten en este trabajo.

En este manual, el lector encuentra ejercicios que puede mantener ocultos hasta que dé clic sobre la palabra "solución" para mostrar el resultado y comparar su respuesta. Se aníma al lector que mantener oculta la respuesta hasta proponer una solución. Además, el estudiante debe considerar que la respuesta puede tener diferentes soluciones. No se desanime.

Por otro lado, para mi (Javier Orduz) es un orgullo comentar que este trabajo ha sido desarrollado por los miembros del GCCyC (grupo que yo inicié), un grupo muy joven en la FESAc-UNAM, con miembros muy activos, que ven en sus compañeros: un equipo y un amigo de quien aprender; y con quien compartir. Esta generación aporta bastante al desarrollo del institución y del país. Es un grupo que ha nacido por iniciativa de jóvenes inquietos que me dieron la oportunidad de acompañarlos en una parte de su etapa de formación y que me brindaron su confianza para pensar en un ejercicio académico como el Grupo de Cómputo Cuántico y Científico. Espero aportar al grupo y continuar con la recepción de las enseñanzas que me brindan cada día. Agradezco mucho su apoyo.

Posiblemente, algunos de los nombres de los miembros que contribuyeron al inicio de este proyecto, por algún error, no estén en este documento, así que agradeceré que me contacten para dar el respectivo reconocimiento.

Además de las siguiente división del contenido se sugiere que el usuario revise los códigos que se dejan en Github, GitLab, Jupyter y otros repositorios que están asociados a este curso, y que pueden ser consultados desde el sitio: yeyecoa[1]. El contenido de este documento es: capítulo 1, hay revisión de conceptos básicos sobe la p rogramación; capítulo 4, se discute el tema de tipos de datos derivados, capítulo 9, se exponen los apuntadores y la implementación de memoria dinámica se discuten los operadores las fases para solución de problemas a través de la computadora; capítulo **??**, se introducen los temas de programación. Capítulo **??**, se presentan los tipos de datos básicos de C. Finalmente (cap. **??**), se exponen los temas básicos de programación y se realizan ejercicios.

[1] La dirección es http://www.yeyecoa.acatlan.unam.mx/

Generalidades

- **G-2202** de 9 a 11 horas
  Horario de la clase:
  - Lunes (A-425),
  - Miércoles (A-723) y
  - viernes (A-425).
- **G-2251** de 14 a 16 horas
  Horario de la clase:
  - Lunes (A-422).
  - Miércoles (A-422).
  - viernes (A-723).

  La hora máxima de llegada es $t + 15$ minutos, donde t es la hora de inicio de clase.

  Fin de ciclo escolar 22 de mayo 2020

  Días inhábiles y asueto académico:
  - 03 de febrero, 2020
  - 16 de marzo, 2020
  - 01, 10, 15 de mayo, 2020
  - 06 al 10 de abril, 2020
- Más información sobre mi horario aparece en: http://www.mac.acatlan.unam.mx/portada/profesores/0/.
- Además en el sitio: SEA (http://sea.acatlan.unam.mx/), encontrarán más información sobre el curso para matricularse deben usar: **u8l5Tgb**.
  Consultarlo para tareas, material, bibliografía, información y preguntas.
- En el sitio https://www.codechef.com/getting-started encontrarán ejercicios y más material relacionado a la programación.
- Les dejo un google classroom (código de clase **n2f37gz**)
- Además se deben realizar los ejercicios que se dejan en www.code.org con el código: **YRFBNL**. Debes realizar el 70 % del curso para tener derecho a presentar examen y proyecto.
- Tenemos un sitio en Github: https://github.com/UNAM-FESAc/ y http://www.yeyecoa.acatlan.unam.mx donde encontrarán: algunos materiales e información (código en C)

y otras cosas más. Además pueden revisar el repositorio del **curso anterior** (http://www.yeyecoa.acatlan.unam.mx) o en github (https://github.com/UNAM-FESAc/).

Es posible que encuentren detalles por cambiar; agradeceré todos los comentarios y sugerencias.

Es deber de cada estudiante revisar constantemente el sitio SAE, google classroom, code.org, Github y los demás materiales que se comparten en clase.

Usaremos mucho material en inglés, así que los anímo a que estudien, lean y se comuniquen en este idioma.

Ética UNAM.

Legislación Universitaria.

- Todo el material (escrito) debe ser original, ¡cuidado con el plagio! Cuidado en los exámenes. Pueden revisar:
  - El código de ética de la UNAM: http://www.ifc.unam.mx/pdf/codigo-etica-unam.pdf y http://eticaacademica.unam.mx/
  - La legislación universitaria https://goo.gl/M9G3cC y defensoría de los derechos universitarios http://tinyurl.com/v2jyr2s

En la siguiente sección veremos algunos detalles sobre las calificaciones y los criterios de evaluación.

1. Criterios generales de evaluación

Estos son los criterios de evaluación. El alumno debe

- Comprender mucho material bibliográfico que estará en inglés, por lo tanto, es importante se capacite constantemente en este idioma.
- El material escrito debe estar presentado adecuadamente; es decir, limpio, escrito en computador y, además, debe respetar las normas de ortografía. Todos los documentos deben entregarse en documento **PDF** (excepto: algunos de mis *scripts* que no respetan acentos)
- Enviar los documentos finales (productos), tomando las sugerencias dadas: fechas y horas acordadas.
- Realizar y mantener ordenado el repositorio: Drobox, Drive, OneDrive, etc.. Ejemplos de repositorios:

Figura 1: Ejemplos de repositorios.

- Guardar el 🔊 en modo avión.
- Nota mínima aprobatoria 7.0.

  El porcentaje de la calificación total está dada por:

Primer examen (Semana del 16 al 20 de marzo):	30 %
Segundo examen (Semana del 04 de 08 de mayo):	30 %
Proyecto final (Semana del 11 de 15 de mayo):	20 %
Dropbox: Ejercicios y tareas (en o de laboratorio o en clase):	10 %
Material, información, tareas, ejercicios en SAE y en code.org	10 %

Respecto a la actitud y otros detalles

- Tomar una postura de responsabilidad y cumplimiento con la materia.
- Solamente el

- Es necesaria mucha comunicación.

Matengamos un ambiente de respeto, tolerancia y cordialidad.

2. Sobre los exámenes de primera y segunda vuelta

Estos exámenes se llevan a cabo durante la semana del 25-29 de mayo de 2020 (primera vuelta) y la semana del 01-05 de junio de 2020 (segunda vuelta). Estas fechas las envía el programa de MAC.

Estos exámenes se llevarán a cabo en las fechas acordadas por el programa de MAC, su evaluación máxima de $6 \leq x_f \leq 8$ y se evualará todo el material revisado en el curso.

3. Actividades

- GCCyC y Robocop . JAOD
- Seminario de investigación en CTIM. JAOD *et. al.*
- MAC-Day. Mtra. G. Eslava.

Tarea: Manifiesto. Programación II. 2020-II

A quien interese:

Por este medio, yo, _____con matrícula _____,del grupo _____, de la carrera de _____, de la FES Acatlán, declaro que estoy enterado del temario de Programación II, fechas de inicio y fin del curso, y de exámenes; tiempos de llegada a cada clase, sitios web de consulta (Github, code, SEA, yeyecoa, google classroom, entre otros); de los materiales, criterios y formas de calificar; y de los porcentajes de calificación a través de exámenes y proyecto. Además, conozco los requerimientos del curso: respeto, compromiso y disciplina. Por lo tanto firmo, de manera legible, coloco mi nombre y subo este documento en el sitio SEA.

_____	_____	_____
Nombre	Firma	Fecha

Tarea: Manifiesto. Programación II. 2020-II

A quien interese:

Por este medio, yo, _____con matrícula _____,del grupo _____, de la carrera de _____, de la FES Acatlán, declaro que estoy enterado del temario de Programación II, fechas de inicio y fin del curso, y de exámenes; tiempos de llegada a cada clase, sitios web de consulta (Github, code, SEA, yeyecoa, google classroom, entre otros); de los materiales, criterios y formas de calificar; y de los porcentajes de calificación a través de exámenes y proyecto. Además, conozco los requerimientos del curso: respeto, compromiso y disciplina. Por lo tanto firmo, de manera legible, coloco mi nombre y subo este documento en el sitio SEA.

_____	_____	_____
Nombre	Firma	Fecha

Parte II: Repaso de conceptos

Capítulo 1
Actividad de ensamble

Capítulo 2
Repaso

En esta parte realizaremos algunas actividades de repaso e introducción al nuevo curso.

1. Discutir los conceptos de...
    - tipos de datos
    - operadores
    - Funciones (lectura recomendada (Molina-López, 2006))
    - definición, declaración y llamada de una función.
    - Algunos lenguajes de programación
    - Forma general de un archivo fuente en C.

Capítulo 3
¿Qué es un arreglo?

Es muy comun almacenar muchos datos en valores de datos especificados: Puntajes de un juego.

¿Cómo se puede declarar un arreglo?

- Los arreglos tienen elementos
- Los elementos deben ser del mismo tipo.
- Se usan los [ ]. Dentro va el número de elementos. Esto es el tamaño o dimensión del arreglo.

Se puede acceder a los elementos del arreglo.

- Los índices son enteros secuencias que inicial desde cero.
- Es común usar un loop para acceder a cada elemento del arreglo: `for(i){printf("Arreglo %d", arr`
- ¿Qué pasa si un valor de índice está fuera de rango; entonces, el programa se "crashea" o el arreglo puede contener datos basura. Incluso el compilador puede compilar el archivo fuente.
- Es importante asegurarse sobre los límites del arreglo.
- Se pueden asignar valores a un elemento de un arreglo.

> Un arreglo (*arrays*, lista o tabla) es una secuencia de datos del mismo tipo. Consta de **elementos** y pueden ser, incluso, estructuras definidas por el usuario.

Cada elemento tiene un índice o subíndice que numera consecutivamente de izquierda a derecha, iniciando desde cero (0). Estos índices sirven permitir acceso directo al elemento del arreglo.

Ejemplo 3.1. *Ejemplo de un arreglo en C*

```
#include <stdio.h>
#define n 100

void main ()
{
char nombre[n];
char apellido[n];

printf("Como te gusta que te llamen? ");
scanf("%[^\n]%*c", nombre);
printf("Cual es tu apellido (sin espacios)? ");
scanf("%[^\n]", apellido);

printf("\v\tHola %s tu apellido es %s\n", nombre, apellido);
}
```

Ejercicio 3.1. *Escriba un ejemplo de un código que implemente un arreglo unidimensional.*

1. Inicializando un arreglo

Es importante inicializar un arreglo, porque nos ayuda detectar los errores. A continuación mostramos algunas razones por las que conviene inicializar un arreglo.

- Asignar un valor a un arreglo es fácil, solamente se deben colocar valores en cero, por ejemplo.
- No es necesario inicializar todos los valores de un arreglo. Ejemplo: `float datas[100] = {1.0, 2.5` De esta manera se inicializan los primeros tres valores; los demás son cero.
- Existe los inicializadores designados: `float data[100]= {[2]=100.1,[1]=20.3, [0]=103,9};` hemos inicializado los primeros tres valores del arreglo data.
- Un problema se presenta cuando hay muchos elementos por introducir uno por uno. Pero podemos usar *loops*.

2. ¿Cómo se escriben arreglos en C?

En esta parte retomamos algunos conceptos vistos en el curso anterior.

Un arreglo se numera consecutivamente de izquierda a derecha iniciando en "0". Así `a[0]` es el nombre del elemento que está en la primera posición de la iquierda (indexación basada en cero).

Algunas características de los arreglos:

- Un arreglo se declara como sigue: `TIPO Nombre[Numero_elementos]={ele_1, ele_2,...,ele_n}`
- En general el $i-$ésimo elemento estará en la posición $i-1$.
- Normalmente, un arreglo se usa para almacenar variables de tipo `char, int, float`.

- Un arreglo se almacena con sus elementos en una secuencia de posiciones de memoria contigua.

Una cadena (de texto) es un conjunto de caracteres, e.g. `BCDEF` y un arreglo de caracteres es una secuencia de caracteres: `char cadena[]="BCDEF";`

> Nota que existe las cadenas de caracteres y arreglos de caracteres. La primeras contienen un caractere nulo al final del arreglo de caracteres. Además las cadenas se deben almacenar en arreglos de caracteres, pero no todos los arreglos de caracteres contienen cadenas.

Veamos un ejemplo de una cadena de caracteres:

```
cadena[0] = 'B';
cadena[1] = 'C';
cadena[2] = 'D';
cadena[3] = 'E';
cadena[4] = '\0';
```

Existen los arreglos multidimensionales, los más comunes son los bidimensionales. Estos se pueden visualizar como arreglos rectangulares con filas y columnas.

Se declaran como: `int matrix[i][j];` Un arreglo bidimensional se inicializa como `int matrix[2][2] = {{1,2},{2,3}};` Aunque no se requiere inicializar completamente; pero en pares internos de *braquets* es necesario inicializar para una correcta implementación.

Otra forma es `int matrix[100][10] = {[12][2]=5,[22][3]=4,[42][4]=3};` Los demás valores serán cero.

Más dimensiones implica más *loops* anidados para procesar más arreglos. Los veremos más adelante.

Ejercicio 3.2. *Escriba un código que implemente arreglos bidimensionales.*

Tarea 3.1. *Escribe un programa que encuentre los números primos del 3 al 100 usando arreglos.*

Las condiciones son las siguientes:

1. *No habrá dato de entrada*
2. *la salida será un número primo separado por un espacio, todos en una sola línea horizontal.*
3. *Debes usar un ciclo for para encontrar los números primos hasta el 100 e imprimirlo en un arreglo de salida.*
4. *puede usar la siguiente condición de salida:* `p / primes[i]>=primes[i]`, *para asegurarse que el valor de p no excede la raíz cuadrada de los* `primes[i]`.
5. *Puedes saltar los números pares (ya que no son primos) y hacer más eficiente el código.*
6. *Puedes usar* `#include <stdbool.h>` *para declarar variables de tipo booleano.*

Los datos de tipo *bool* se pueden implementar de la siguiente manera.

Ejemplo 3.2. *Datos de tipo bool.*

```
#include <stdio.h>
#include <stdbool.h>
int main()
{
    int p=3;
    bool isEven = true;
puts("1 = true and 0 = false\n\n");
if(p==2){
    printf("p=%i es Par %i\n",p ,isEven);
 }else{
    printf("p=%i es Impar %i\n",p ,isEven);
 }
    return 0;
}
```

3. Areglos Multidimensionales

Tenemos arreglos lineales o unidimensionales; sin embargo, existen los arreglos multidimensionales, los más comunes son los bidimensionales. Estos se pueden visualizar como arreglos rectangulares con filas y columnas.

1. Se declaran como: `int matrix[i][j];`
2. Un arreglo bidimensional se inicializa como `int matrix[2][2] = {{1,2},{2,3}};`
3. Aunque no se requiere inicializar completamente; pero en pares internos de braquets es necesario inicializar para una correcta implementación. Otra forma es `int matrix[100][10] = {[12`
   Los demás valores serán cero.
4. Arreglos de más dimensiones: `int box[10][20][30];` Más dimensiones implica más *loops* anidados para procesar más arreglos. Ejemplo:

```
int numbers[2][3][4] = {
    {  // primer bloque
{1,2,3,4},
{5,6,7,8},
{9,10,11,12}
    },
    { //segundo bloque
{10,20,30,40},
{50,60,70,80},
{90,100,110,120}
}
}
```

Ejemplo del loop que itera sobre cada dimensión del arreglo:

```
int i, j k,  sum = 0;
for(i=0; i = 2; ++i){
```

```
for(j=0; j = 3; ++j){
for(k=0; k = 4; ++k){
sum + = numbers[i][j][k];
}
}
}
```

Los arreglos son muy usados en diferentes lenguajes de programación.

4. Concatenación

Concatenación es proceso de agregar una cadena a otra cadena.

Capítulo 4
Tipos de datos derivados

Anteriormente, se discutieron otros tipos de datos básicos. Se realizaron diferentes ejercicios para conocer el funcionamiento de los tipos de datos, además de las características.

En esta parte conoceremos los tipos de datos derivados. Nuestro objetivo general es:

Objetivo

Desarrollar programas utilizando tipos de datos derivados.

En este capítulo hablaremos sobre la declaración `struct`. Esta palabra reservada del lenguaje nos sirve para definir un conjunto de elementos de diferente tipo.

Capítulo 5
Estructuras (struct)

En este capítulo estudiaremos el concepto de estructura, su uso y cualidades, además veremos algunos ejemplos.

1. Definición

Las estructuras en C brindan otra forma de agrupar objetos. Por ejemplo, si quieremos almacenar datos como mes, día, año, RFC, CURP; y que requiere mantener el camino separado de cada una de las variables, que están relacionadas. Considere que los objetos son de diferente tipo.

Estructura

Una estructura es una colección de una o más variables, agrupadas bajo un solo nombre para un manejo conveniente (Kernighan y Ritchie, 1991). Esta colección puede contener información de diferente tipo.

Reflexiona

¿Cuál es la diferencias con los arreglos?

Las estructuras tienen atributos que a su vez pueden ser otras estructuras. Además, las estructuras pueden usarse en las funciones.

La forma general de una estructura es:

```
struct Nombre_o_rotulo{
Lista de declaraciones 1;
Lista de declaraciones 2;
...;
Lista de declaraciones n;
}
```

dentro de la lista de declaraciones puede aparecer las variables nombradas, llamadas `miembros` o campos. Puede suceder que un `miembros` y una `variable (ordinaria)` (no miembro) pueden tener el mismo nombre; sin embargo no se recomienda a menos que haya una estrecha relación.

Una estructura se puede declarar de la siguiente forma:

```
struct Nombre_o_rotulo{};
```

Algunas veces el rotulo se reemplaza por marca como identificador de la estructura.

Definición y declaración

Algunos autores comentan que no hay una definición formal entre definición y declaración. En nuestro caso hemos aclarado que hay diferencia.

1.1. Ejemplo

Pensemos en un ejemplo gráfico. Un punto en el plano cartesiano de coordenadas x y y (fig. 5.1).

Figura 5.1: Supongamos que tenemos una estructura con un punto cuyas coordenadas son x y y.

Para la implementación correcta de una estructura debemos retomar los conceptos **definición (declaración) y llamamiento**. En estructuras, los conceptos definición y declaración no presentan distinción.

La forma general para declarar una estructura es:

```
struct Nombre_o_rotulo{
Lista de declaraciones 1
Lista de declaraciones 2
...
Lista de declaraciones n
}[lista de una o mas variables de estructura];
```

Otros ejemplos de estructuras:

Ejemplo 5.1. *Bancos:*

```
1 struct cuenta{
2  int nUffi_cuenta;
3  char tipo_cuenta;
4  char nombre[80];
5  float saldo;
6 };
```

Ejemplo 5.2. *Venta-compra:*

```
struct venta
{
 char vendedor[30];
 unsigned int codigo;
 int inids_articulos;
 float precio_unit;
};
```

Ejemplo 5.3. *Organización:*

```
struct coleccion_CD
{
 char titulo[30];
 char artista[25];
 int num_canciones;
 float precio;
 char fecha_compra[81];
};
```

Preguntas y respuestas

1. ¿Cuántos miembros tiene cada uno de los ejemplos anteriores?
2. ¿Cuántos tipos básicos identificas?,
3. ¿cuántos y cuáles cualificadores?

Una forma esquemática para las estructuras es:

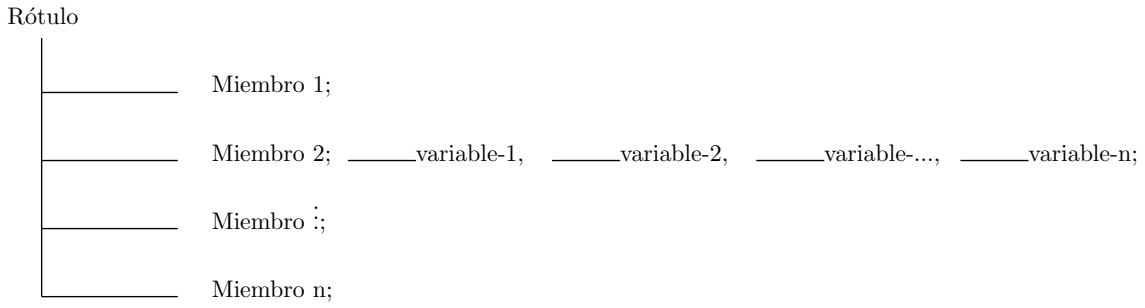


Figura 5.2: En este esquema mostramos la forma general de una estructura.

La fig. 5.2 muestra un esquema de la forma general para la declaración de las variables indetificadas con `variable-1,..,variable-n` y `variable-2`, que son variables (de tipo Rótulo) de estructura identificadas mediante la palabra `Rótulo`.

vamos a colocar un ejemplo del código de una estructura con su esquema:

Ejemplo 5.4. *Código en C y esquema de estructura.*

```
struct corredor
{
 char nombre[30];
```

```
 char sexo;
 int edad;
 char categoria[10];
 float tiempo;
};

// La declaracion de las variables de estructura

struct corredor c1, c2, c3
```

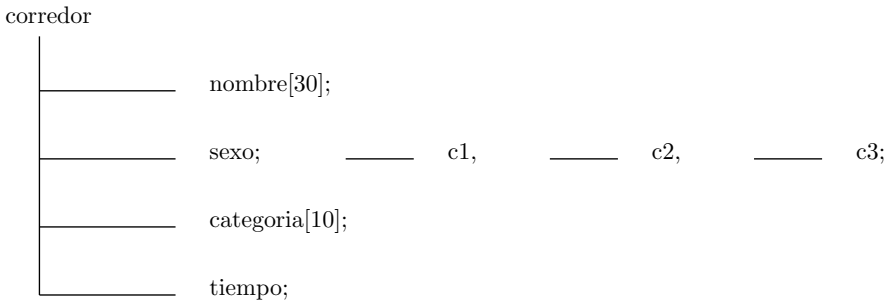


Figura 5.3: Ejemplo de una estructura. donde c1, c2, c3 son variables de estructura; es decir son estructuras del tipo **corredor**.

En la siguiente sección veremos como inicizalizar una estructura.

2. Inicialización de estructuras

Las estructuras se pueden inicializar de dos formas (L. y Zahonero Martínez, 2001):

1. dentro de la sección de código de su programa
2. como parte de la definición se deben especificar los valores iniciales, entre corchetes, después de la definición de las variables de estructura.

Ejemplo 5.5. *Incialización de una estructura:*

```
struct coleccion_CD
{
 char titulo[30];
 char artista[25];
 int num_canciones;
 float precio;
 char fecha-compra[8];
} cd1 = {
 "El humo nubla tus ojos",
 "col Porter",
 15,
 500,
 "02/6/99"
};
```

Veamos otro ejemplo:

Ejemplo 5.6. *Inicialización de una estructura:*

```
struct corredor
{
 char nombre[30];
 char sexo;
 int edad;
 char categoria[10];
 float tiempo;
};



struct corredor c1 = {
"Orduz, J.",
29,
"Senior",
"Independiente",
0.0
}
```

Ejercicio en clase:

1. Declare una estructura de para almacenar algún tipo de información.
2. Use el operador `sizeof` para conocer el tamaño en memoria de la estructura. que contenga datos de diferente tipo.

En la siguiente sección veremos cómo acceder a los datos de una estructura.

3. Usando una estructura

Para usar una estructura es necesario **definir** el nuevo tipo: `struct rotulo`, esto define un nuevo tipo en el lenguaje y ahora las variables puede ser **declaradas** del tipo `struct rotulo VariableDeEstructura`.

4. Acceso a campos

En las estructuras se puede almacenar información desde la inicialización, asignación directa o lectura de teclado. Para acceder a los miembros de una estructura debemos usar operadores que realicen un trabajo sobre los elementos del código.

4.1. Operadores

Para acceder a los miembros de la estructura usamos los operadores:

Operador de miembro (·):
Este operador proporciona un camino directo al miembro correspondiente.

Operador puntero ($->$):

Este operador sirve para acceder a los datos de la estructura a partir de un puntero. Por lo que primero se debe definir un operador puntero. Lo dejaremos para otro momento.

La forma general de implementar el operador de miembro para hacer referencia de un miemrbo de una estructura en particular es de la forma `nombre_estructura.miembro_estructura` Este operador de miembro conecta al nombre de la estructura con el nombre del miembro.

Ejemplo 5.7. *Uso del operador de miembro de estructura:*

```
#include<stdio.h>

struct corredor{
 char nombre[30];
 short unsigned int edad;
 float altura;
 float califi;
} crrdr1={
 "Pepe Perez",
 20,
 1.8,
 9.0},
  crrdr2={
 "Paco Diaz",
 23,
 1.7,
 8.0};


int main()
{

printf("\n");
printf("Hola %s tienes %d anios, tu estatura es %.1f m y\n tu
   calificacion en programacion II es %.1f\n", crrdr1.nombre,
   crrdr1.edad,crrdr1.altura, crrdr1.califi);
printf("\n");
printf("Hola %s tienes %d anios, tu estatura es %.1f m y\n tu
   calificacion en programacion II es %.1f\n", crrdr2.nombre,
   crrdr2.edad,crrdr2.altura, crrdr2.califi);
printf("\n");
printf("Promedio de calificacion es %.1f\n",
   (crrdr1.califi+crrdr2.califi)/2.0);

return 0;
}
```

Una vez se implementa el operador de miembro, se observa su función sobre la etiqueta del miembro.

Operador de miembro

El operador punto o de miembro proporciona una camino directo al miembro respectivo.

La forma general para usar el este operador es: `nombre_estructura.miembro`

Ejercicio en clase:
Realiza un código donde muestres la declaración de una estructura con cuatro miembros de diferentes tipos: dos `char`, un `int` y un `float`. Luego crea la función principal con tres declaraciones de la estructura anterior. Ingresa datos para cada miembro de cada variable de estructura, finalmente, Imprime los datos en pantalla.

Hablemos sobre el operador puntero.

Operador de puntero

El operador puntero sirve para acceder a los datos de la estructura a partir de un puntero.

Para usar el operador puntero primero se debe definir una variable puntero dirigido a la estructura.

La forma general es: `puntero_estructura -> nombre_miembro = datos;`

Ejercicio en clase:
Realiza un código donde implementes una estructura con cuatro miembros, un puntero y, finalmente, imprime los datos en pantalla los datos.

En la siguiente sección veremos otras operaciones con estructuras.

5. Funciones y estructuras

En términos generales las estructuras no se puede comparar.

Operaciones con estructuras

Una estructura se puede copiar, asignar completamente (como un todo) y se puede tomar su dirección de memoria.

Una forma de ayudar con las diferentes operaciones entre estructuras es usar funciones. La forma general de una función con una estructura es: `struct rotulo Nombre_funcion(parametros){miem`

Vamos a ver cómo se crea una función con una estructura:

Ejemplo 5.8. *Una función y una estructura:*

```
1 // Realizamos la declaracion rotulada de la estructura
2 struct punto{
3
4  int x;
5  int y;
6 };
7
8
```

```
// Realizamos la declaracion de la funcion que toma dos enteros
// y regresa una estructura punto.
struct punto coordenadas(int x, int y)
{

// Declaramos la variable de estructura temp para...
struct punto temp;

// ...asignar los parametros de la funcion a
// los miembros de la estructura.
 temp.x = x;
 temp.y = y;

return temp;
}
```

Ejercicio en clase:
Observamos el código y discute en clase las partes y la estructura del ejemplo 5.8.

Nótese que se puede usar una función para inicializar dinámicamente cualquier estructura. Esto quiere decir que podemos asignar valores a una estructura declarada (definida) previamente. También se puede usar para proporcionar los miembros de la estructura a una función.

Veamos otros ejemplos:

Ejemplo 5.9. *Una función y una estructura: usando las coordenadas de un punto*

```
#include<stdio.h>

// Declaracion de la estructura
struct pto
{

 float x;
 float y;
};

// Declaracion de la funcion
void sumapuntos(struct pto p1);

// Funcion principal
int main(){

struct pto pto1;

printf("Escriba el valor de x ");
scanf("%f", &pto1.x);
printf("Escriba el valor de y ");
scanf("%f", &pto1.y);
```

```
23
24// Paso de la variable de estructura como argumento
25sumapuntos(pto1);
26
27return 0;
28}
29
30// Definicion de la funcion con argumento de tipo struc
31void sumapuntos(struct pto p1)
32{
33float ZZ;
34
35ZZ = p1.x*p1.x + p1.y*p1.y;
36
37printf("ZZ=%.1f\n", ZZ);
38}
```

En el ejemplo anterior (5.9), observamos diferentes conceptos discutidos en clase. Es importante que el estudiante los identifique.

El siguiente ejemplo contiene campos de diferente tipo, observa y analiza, e identifica los conceptos discutidos en clase.

Ejemplo 5.10. *Una función y una estructura: usando diferentes tipos de datos.*

```
1#include<stdio.h>
2
3// Declaracion de la estructura
4struct estudiante
5{
6    char nombre[50];
7    int matricula;
8};
9
10// Declaracion de la funcion
11void display(struct estudiante estdnt);
12
13
14// Funcion principal
15int main()
16{
17    struct estudiante std;
18
19    printf("Nombre del estudiante: ");
20    scanf("%s", std.nombre);
21    printf("Matricula: ");
22    scanf("%d", &std.matricula);
23// Paso de la variable de estructura como argumento
24    display(std);
25    return 0;
26}
27
28// Definicion de la funcion con argumento de tipo struc
```

```
29 void display(struct estudiante estdnt){
30   printf("\nNombre del estudiante: %s", estdnt.nombre);
31   printf("\nMatricula: %d\n", estdnt.matricula);
32 }
```

Ejercicios en clase:

1. Usa el ejemplo 5.9, ajusta el código para que tome cuatro puntos (las coordenadas de los vértices de un rectángulo) y calcule:
   a) la diagonal,
   b) el perímetro y
   c) el área.
2. Usa el ejemplo 5.10, implementa otros tipos de datos y concatena la información, usando funciones de bilbioteca, para que muestre un mensaje de salida.

En la siguiente sección implementaremos arreglos de estructuras.

6. Arreglos de tipo struct

Este tipo de arreglos permiten almacenar diversos valores de diferente tipo, organizados como estructuras Paul Deitel (2016).

Ejemplo 5.11. *Arreglo de estructuras.*

```
1 #include<stdio.h>
2
3 // Declaracion de un arreglo de estructuras
4 // de 100 elementos denominado libros
5 struct Todos_Libros
6 {
7 char titulo[30];
8 char autor[30];
9 char editorial[30];
10 int anio;
11 };
12
13 // Declaracion una variable-arreglo de estructuras.
14 struct Todos_Libros libros[100];
```

Veamos un ejemplo implementado en un código.

Ejemplo 5.12. *Arreglo de estructuras para organizar libros.*

```
1 #include<stdio.h>
2
3 struct Todos_Libros
```

```
{
char titulo[30];
char autor[30];
char editorial[30];
int anio;
};

// Funcion principal
int main()
{
// Declaracion de una variable-arreglo de estructura
struct Todos_Libros libros[100];

// Inicializacion (Definicion) de los miembros
strcpy(libros[0].titulo,"Programacion I");
strcpy(libros[0].autor,"Perez, P.");
strcpy(libros[0].editorial,"Patito-Hill");
libros[0].anio = 2018;
return 0;
}
```

El estudiante debe revisar los código colocados en www.github.com.

7. Estructuras anidadas

Los miembros de las estructuras pueden ser estructuras, a esto le llamamos estructuras anidadas. Este tipo de objetos se pueden usar para: reducir la cantidad de código escrito, tener códigos más organizados, entre otras.

Ejemplo 5.13. *Estructuras anidadas.*

```
#include <string.h>

struct StrFruits{
    char name[30];
 int account;
};

struct StrPerson{
    char name[30];
 char email[30];
};
```

En el siguientes capítulo implementaremos la declaración `typdef`, esta es una declaración similar `struct`.

Ejercicios en clase:

1. Usa un ejemplo anterior (5.9 o 5.10) e implementa un ciclo `for` para realizar una operación aritmética sobre el mismo miembro de diferentes variables de estructura.
2. Implementa una estructura anidada para almacenar datos de una tienda de abarrotes.
3. Implementa una estructura de, al menos, cuatro miembros con, al menos, cinco variables de estructura y estructuras de control selectivas o repetitivas (`if`, `for`, entre otras) para realizar una tarea específica. Prepara un documento (ver documento ejemplo 1) donde describas el trabajo realizado. No olvides colocar el archivo fuente y el PDF a tu repositorio de tareas.

Capítulo 6
Typedef

Esta es una llamada que funciona para crear nuevos tipos de datos. La forma general es: `typedef TIPO ETIQUETA;`

Lo cual significa que la palabra `ETIQUETA` es un sinónimo de `TIPO`. Por convención se promueve el uso de las letras mayúsculas.

typedef

Esta palabra clave funciona muy bien para dar nombre a un tipo de dato definido por el usuario.

Estrictamente, `typedef` no crea un nuevo tipo de dato; simplemente agrega un nuevo nombre para algún tipo ya existente.

Las diferencias principales de `typedef` con `#define` son:

1. `typedef` se limita a nombres simbólicos a tipos mientras `#define` puede definir alias para valores numéricos, también.
2. La interpretación de `typedef` la realiza el compilador mientras el pre-procesador se encarga de `#define`.

Veamos un ejemplo:

Ejemplo 6.1. *Palabra clave* `typedef`.

```
/* ********************* TEXTO SIN ACENTOS **********************/
// Recuerda documentar tus codigos
#include <stdio.h>
#include <string.h>
// Creamos un tipo de dato: struct Books{Miembros} y lo
   etiquetamos como BOOK
typedef struct Books{
    char title[50], author[50], subject[100];
```

```
 int book_id;
} BOOK;

int main( ) {

// Delaramos book del tipo BOOK
   BOOK book[2];

// Inicializamos la estructura
   strcpy( book[0].title, "C Programming");
   strcpy( book[0].author, "Nuha Ali");
   strcpy( book[0].subject, "C Programming Tutorial");
   book[0].book_id = 6495407;

   printf( "Book title : %s\n", book[0].title);
   printf( "Book author : %s\n", book[0].author);
   printf( "Book subject : %s\n", book[0].subject);
   printf( "Book book_id : %d\n", book[0].book_id);

   return 0;
}
```

Capítulo 7
Union

Una unión es una variable que puede tener objetos de diferentes tipo y tamaños.

union

Es un tipo de dato especial para guardar tipos de datos diferentes en la misma ubicación de memoria.

A través de este concepto podemos manipular diferentes tipos de datos dentro de una sola área de almacenamiento.

Ejemplo 7.1. *La forma general de esta declaración* `union`.

```
/* ********************* TEXTO SIN ACENTOS *********************/
// Recuerda documentar tus codigos


union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

El concepto `union` es diferente al de `struct` en que la esta última almacena variables relacionadas en posiciones contiguas en memoria. En la `union` el tamaño se determina usando la mayor tamaño de la variables después de analizarlas todas.

Ejemplo 7.2. *Ejemplo para conocer el tamaño de una* `union`

El tamaño es 8 bytes.

En el siguiente capítulo veremos un tipo de datos definido por el usuario.

Capítulo 8
Enumeración

Algunas veces es necesario crear un tipo de dato definido por el usuario con constantes de nombre de tipo entero. Para este caso existe la palabra reservada `enum`.

enum

Es un tipo de dato usado para escribir un lista de identificadoes que internamente se asocian con las constantes enteras.

Ejemplo 8.1. *Un tipo enum*

```
enum NOMBRE{enumerador1, numerador2,..,numeradorn };
```

Donde NOMBRE se refiere al nombre de la `enum` y `enumeradori` son los valores de tipo NOMBRE, estos valores se inicializan en 0, 1,... En el ejemplo 8.2 Si desea cambiar los valores debe implementar:

Ejemplo 8.2. *Un tipo enum inicializado por el usuario con valores constantes asignados.*

```
enum valores_numer{
 enumerador1=0,
 numerador2=1,
 numerador=12 };
```

Ejercicio en clase:
Observa los código de los ejemplos anteriores y comenta la mayor diferencia entre los tipos `enum`, `struct` y `union`.

Capítulo 9
Apuntadores y memoria dinámica

En este capítulo se abordan temas importantes para la optimización de los programas en C. Por ejemplo, el apuntador/puntero o *pointer* requiere análisis y práctica.

Objetivo

Utilizar la memoria de manera dinámica con el uso de apuntadores.

Un apuntador es una variable cuyo valor es la dirección de alguna otra variable, i. e., la dirección directa de la ubicación de la memoria. Como cualquier variable o constante, se debe declarar un puntero antes de usarlo para almacenar la dirección variable. La forma general de la declaración variable apuntador es: `tipo *nombre-de-variable`. Donde `tipo` es uno de los tipos de datos (*int, float, struct*, entre otros.) y `*nombre-de-variable` será el nombre dado. L. y Zahonero Martínez (2001).

En las siguientes páginas definimos algunos conceptos relacionados a los apuntadores y sus operadores.

1. Operadores para operar con la memoria en C

En C, tenemos dos operadores utilizados para acceder o revisar los valores en la dirección de memoria de una variable. Estos operadores son: & y *.

Veamos un ejemplo de un apuntador con el operador de dirección de memoria.

Ejemplo 9.1. *Código para obtener la dirección de memoria usando el operador &.*

```
#define N 5
int main () {
   int  var1, i;
   char var2[N];

   printf("Direccion de la variable var1: %p\n", &var1  );
   printf("Direccion de la variable var2: %p\n", &var2  );

for(i=0; i<N; i++){
   printf("Direccion de la variable var2[%i]: %p\n", i, &var2[i]
   );
   }

   return 0;
}
```

Podemos usar el operador $*$. Veremos algunos ejemplos.

Ejemplo 9.2. *Código para obtener la dirección de memoria usando referenciación.*

```
#include <stdio.h>

int main(){
    int var1=6, var2=4;
    int * var1_ap, * var2_ap;

    var1_ap = &var1;
    var2_ap = &var2;

    printf("%p %d\n", var1_ap, var1);
    printf("%p %d\n", var2_ap, var2);
return 0;
}
```

Veamos otro ejemplo.

Ejemplo 9.3. *Código para obtener la dirección de memoria usando indirección (dereferenciación).*

```
int main(){
    int var1=6, var2=4, var3=0;
    int * var1_ap, * var2_ap;

```

```
var1_ap = &var3;
var3 = var1 + var2;
printf("%p %d %p %d\n", var1_ap, var3, &var3, *var1_ap);
return 0;
}
```

Un apuntador en C se usa para alojar la memoria dinámicamente; es decir, en tiempo de ejecución. Estos conceptos y otros los estaremos revisando en esta parte del curso.

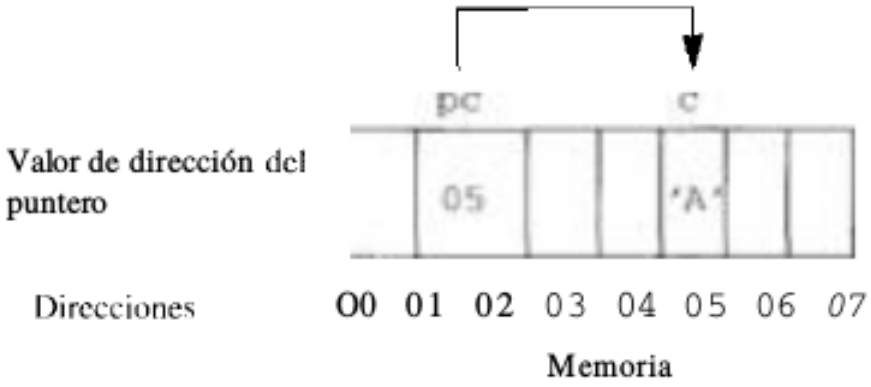
Operadores unarios

Nótese que hemos implementado los operadores & y $*$ los cuales son llamados operador de referencia y operador de indirección (dereferencia)

Algunos textos diferencian entre el símbolo $*$ en `int * a` y `printf ("imprime valor ", * a)`, ya que la primera notación se usa para crear un puntero, mientras que en el segundo contexto, el caracter se puede pensar como un operador de dereferencia o indirección. En este documento, seguiremos esta idea.

2. Operaciones

Consideremos el siguiente diagrama:



Ahora observa el siguiente código:

Ejemplo 9.4. *Código para apuntar a la dirección de memoria de una variable. ¿Está correcto?*

```
#include<stdio.h>

int main(){

int c, * pc

pc = &c;

*pc = c;

return 0;
}
```

Podemos realizar diferentes operaciones con los punteros, la primera que debemos aprender es a inicializar los punteros.

Tenemos dos formas de inicializar los punteros:

1. Estáticamente

   Asignar memoria :
   Para realizar esta forma es necesario definir una variables y a continuación hacer que el puntero apunte al valore de la variable. `int i, *p;`, declarar la variable y el puntero `p = &i;`, asignar la dirección de memoria de i a p.

   Asignar un valor a dirección de memoria:
   `* p = 40;` asignar un valor a dirección de memoria. Esta línea quiere decir `el contenido de p es igual a` o `el contenido de memoria de p es del tipo`, debido a que ya se declaró el puntero.

   Error

   Es un error asignar un valor a un contenido de una variable puntero si previamente no se ha inicializado con la dirección de una variable.

   ```
   float * px
   * px = 25.0
   ```

2. Dinámicamente. Usando funciones

   malloc(): `malloc (number *sizeof(int));`

   calloc(): `calloc (number, sizeof(int));`

   realloc(): `realloc (pointer_name, number * sizeof(int));`

   free(): `free (pointer_name);`

   Más adelante entraremos en detalle sobre los tópicos anteriores.

Hablemos de la indirección de punteros.

Indirección (Dereferenciar o Desreferenciar)

es el uso de un puntero para obtner el valor al que apunta y no la dirección de memoria. Se usa el operador $*$.

Ejemplo 9.5. *Inicialización y la indirección de un puntero.*

```
1 #include<stdio.h>
2 int main(){
3
4 int edad, *p_edad;
5
6 p_edad=&edad;
7
8 printf("Cual es tu edad? \n");
9 scanf("%d", &(*p_edad));
10
```

```
11 printf("(Valor de la variable) Tu edad es %d anios\n", edad);
12 printf("(Indireccion  p_edad) Tu edad es %d\n", * p_edad);
13
14 return 0;
15 }
```

En el ejemplo 9.13 la variable p_edad apunta la dirección de memoria de edad; por otro lado, sucede que p_edad apunta (indirección) a la variable edad.

Verificación de tipos

Es necesario que las variables puntero direccionen variables del mismo tipo de dato al que se liga a los punteros.
`int *ptr_a, a; ptr_a = &a;` de otra manera surgiran problemas.

Ejemplo 9.6. *Apuntadores, referenciación y caracteres.*

```
1 #include<stdio.h>
2 int main(){
3 char * pcM, c;
4
5 pcM = &c;
6
7 for (c = 'A'; c<= 'Z'; c++){
8   printf("%c\n", * pcM);
9 }
10
11 return 0;
12 }
```

A fin de mantener el concepto de indirección presentamos un ejemplo que muestra algunos detalles diferentes.

Ejemplo 9.7. *Indireccionar o dereferenciar.*

```
1 #include <stdio.h>
2 int main()
3 {
4     int v1, v2;
5     int * ptr_v1, * ptr_v2;
6
7     ptr_v1 = &v1;
8     ptr_v2 = &v2;
9
10    v1 = 12;
11    v2 = 2;
12
13    * ptr_v1 = 3;
14    * ptr_v2 = 4;
15
16    * ptr_v2 = * ptr_v1;
17 printf("Valor final para * ptr2 = %i\n", * ptr_v2);
```

```
18
19      return 0;
20  }
```

En las líneas 16 y 17, el código muestra la inicialización del contenido de los apuntadores a través del operador de indirección. Hemos platicado sobre los diferentes usos del caracter asterisco (∗): en los comentarios multilínea, como operador producto, para declarar un apuntador y como operador de indirección. Veamos otro ejemplo de uso como operador de indirección.

> **Ejercicio en clase:**
> Implementa documentación interna completa del código del ejemplo 9.7, (es importante que el estudiante discuta la diferencia entre el caracter ∗).

En las siguientes secciones veremos apuntadores y arreglos, entre otros temas.

3. Apuntadores y arreglos

Este tema es importante porque nos permite implementar cadenas de caracteres ya que en C no existe el tipo de dato CADENA.

> **Ejercicio en clase:**
> Discuta en clase sobre diferencia que encuentra entre un arreglo y un puntero. Ayudas: Use un arreglo como ejemplo e imprima la salida. Use notación de subíndices. Nótese que los elementos de un arreglo (constante puntero) no se pueden modificar.

Ejemplo 9.8. *Punteros y arreglos.*

```
1  #include<stdio.h>
2
3  int main(){
4
5  float v[6]={16, 15, 8, 55, 12, 13};
6
7  printf("salida %f\n", v[0]);
8
9  printf("salida %f\n", * v);
10
11 return 0;
12 }
```

Con el anterior programa se demuestra que un arreglo se comporta como un puntero.

Ejemplo 9.9. *Punteros contra arreglos (L. y Zahonero Martínez, 2001, pág. 336).*

```
#include <stdio.h>

int longitud(char cadena[]);


void main()
{

char cad[] = "Universidad
    Nacional Autonoma de
    Mexico";

printf("La longitud de %s es
    %d \n", cad,
    longitud(cad));

}




int longitud(char cadena[])
{
 int cuenta=0;

while (cadena[cuenta] !=
    '\0')
 {
  cuenta++;
 }
return cuenta;
}
```

```
#include <stdio.h>

int longitud(char * cadena);
void main()
{
char cad[] = "Universidad
    Nacional Autonoma de
    Mexico";
printf("La longitud de %s es
    %d \n", cad,
    longitud(cad));

}
int longitud(char * cadena)
{
 int cuenta=0;

while (* cadena++)
 {
  cuenta++;
 }
return cuenta;
}
```

En el ejemplo del lado izquierdo, línea 21, usamos `'\0'`, llamado condición de terminación de nulo o byte cero, esto sirve para hacer que el bucle termine.

> **Ejercicio en clase:**
> Compare los códigos anteriores (ejemplo 9.9) y reflexione sobre las diferencias que hay entre los archivos.

Veamos un ejemplo en el que implementa un puntero para referencias valores de diferentes variables (Paul Deitel, 2016).

Ejemplo 9.10. *Punteros y arreglos.*

```
#include<stdio.h>

int main(){
int i, sum = 0, dt;

```

```
printf("Ingrese el numero de datos a sumar\n");
scanf("%d", &dt);

int clases[dt];


printf("Ingrese %d numeros\n", dt);

for(i = 0; i < dt; ++i)
{
 scanf("%d",  &clases[i]);
 sum += clases[i];
}

printf("La suma es %d\n", sum);
```

En el ejemplo 9.10, es posible demostrar que hay equivalencia entre `&clases[i]` y `clases+i`; y que `clases[i]` es equivalente a `*(clases + i)`.

Ejemplo 9.11. *Acceder a los elementos de un arreglo usando apuntadores*

```
#include <stdio.h>

int main()
{

  char * data[5] = {"enero1","enero2","enero3","enero4","enero5"};
  int i;

 printf("Valores guardados: \n");

 for(i = 0; i < 5; i++)
{
     printf("%s\n", * (data + i));
}
```

Más adelante abordaramos un arreglo de apuntadores.

3.1. Arreglos de punteros

Los arreglos pueden contener apuntadores con este concepto podemos implementar un arreglo de cadenas basadas en apuntadores. Esto se conoce como arreglos de cadenas.

Ejemplo 9.12. *Arreglos de punteros.*

```
#include <stdio.h>

const int MAX = 5;

int main () {
   int i = 0, j=0;

   char * names[] = {
```

```
9       "Simpson, H.",
10      "Simpson, M.",
11      "Simpson, L.",
12      "Simpson, B.",
13      "Simpson, M.",
14   };
15
16for ( i = 0; i < MAX; i++) {
17      printf("Value of names[%d] = %s La direccion de memoria es
    %p\n", i, names[i], &(names[i]));
18}
19   return 0;
20}
```

Veamos un ejemplo de una función creada por el programador con argumentos como apuntadores.

Ejemplo 9.13. *Función con un apuntador como argumento.*

```
1  float num;
2
3printf("De un valor: ");
4scanf("%f", &num);
5Func_Cubo(&num);
6}
7
8void Func_Cubo(float * numero){
9 * numero = (* numero) * (* numero) * (* numero);
10 printf("El cubo es: %2.2f \n", * numero);
11}
```

Veamos otro ejemplo.

Ejemplo 9.14. *Funciones con apuntadores como argumentos.*

```
1#include <stdio.h>
2
3void funcionA(int a, int b);
4void funcionB(int * a, int * b);
5
6void main(){
7  int aa, bb;
8  printf("De el valor de dos numeros enteros a, b:");
9  scanf("%d, %d", &aa, &bb);
10    funcionA(aa, bb);
11    funcionB(&aa, &bb);
12
13}
14
15void funcionA(int a, int b)
16{
17  printf("Los valores de a=%d y b=%d, usando paso por copia\n",
     a, b);
```

```
}

void funcionB(int * a, int * b){
  printf("Los valores de a=%d y b=%d, usando paso por
   referencia\n", * a, * b);
}
```

4. Apuntadores de tipo estructura

En esta sección mostraremos cómo pasar a una función un apuntador que apunta a una estructura y cómo usarlo dentro de una función.

Ejemplo 9.15. *Estructuras, funciones, apuntadores y operadores de acceso.*

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

struct data{
 char nombre[20];
    int edad;
    float salario;
}   my_data;

void sh_name(struct data * p);

int main(){

 struct data * st_ptr_data;

 st_ptr_data = &my_data;
 strcpy(my_data.nombre, "Javier");
 my_data.edad=29;

   printf("Dato nombre: %s\n", my_data.nombre);
   printf("Dato edad: %i\n", my_data.edad);

 sh_name(st_ptr_data);
   return 0;
}

void sh_name(struct data * p){
 strcpy(p->nombre, "Jose");
 p->edad=35;
printf("%s\n", p -> nombre);
printf("%i\n", p -> edad);
}
```

Ejercicio en clase:

- Use el código del ejemplo anterior (ejemplo 9.15) e implemente el operador dereferenciar par imprimir en pantalla la misma información.
- Realice los cambios pertinentes para que capture el nombre por terminal y lo imprima.

Es importante que el estudiante observe qué sucede cuando se reemplaza `(* identificador_puntero)` por `identificador_puntero -> miembro_struct`.

Lo anterior sucede porque el operador `->` apunta al miembro de la estructura.

Ejercicio en clase:
Realiza un código que contenga una estructura con rótulo `coordenadas` cuyos miembros sean `x, y, z` de tipo `float`; declara dos variables de estructura normales y dos como apuntadores, referencia las dos direcciones de las estructuras normales a las dos estructuras-apuntadores; luego asigna valores a los miembros de una estructura y (usando el operador `->`), finalmente, copia los valores a la otra estructura.

4.1. Punteros a punteros

Podemos tener puntero de un puntero la respuesta es: sí.

Ejemplo 9.16. *Apuntadores a apuntadores.*

```
#include <stdio.h>

int main()
{
    int i, * ptr_i, ** ptr_ptr_i;
    i = 10;
    ptr_i = &i;
    ptr_ptr_i = &ptr_i;
    printf("ptr_ptr_i = %p\n ptr_1= %p\n i =%i\n",\
    &ptr_ptr_i, &ptr_i, i);
```

Figura 9.1: Esta figura ilustra la asignación de diferentes variables. Ilustración inspirada en *Notación de caja-flecha* (s.f.).

Ejercicio en clase:
Realiza un diagrama similar al de la figura 9.1 y su respectivo código para explicar la asignación de `ptr_ptr_i = &ptr_j` y `ptr_ptr_j = &ptr_i` donde `i` y `j` son enteros.

Ejemplo 9.17. *Apuntadores a apuntadores (dos niveles de apuntadores).*

```
#include <stdio.h>

int main()
{

 float a = 2.0;
 float * ptr_a = &a;
 float ** ptr_ptr_a = &ptr_a;

printf("El apuntador de un apuntador = %f\n",  ** ptr_ptr_a);

return 0;
}
```

Ejercicio en clase:
Implementa documentacion interna sobre el código mostrado en 9.17.

Ejercicio en SAE:
Realiza una investigación sobre punteros y formaciones unidimensionales. Escribe un código de ejemplo y súbelo junto con tu reporte, a tu repositorio; posteriormente coloca un comentario corto referente al tema y, también, la liga en el foro reservado para tal fin. Recuerda que todos tus compañeros podrán ver tu documento final.

Capítulo 10
Memoria dinámica

Las computadoras tienen características importantes para la programación entre los que se cuentan: tarjeta de red, tarjetas de memoria, procesadores (que requieren almacenamiento) y tiempos de respuesta entre otras. En esta sección nos centraremos en dos características importantes: almacenamiento y tiempos de ejecución.

Antes de iniciar veamos los tipos de memoria de un programa en C:

1. **Memoria global** zona de memoria donde se almacenan todos aquellos datos que están presentes desde el comienzo del programa hasta que termina.

2. **La pila** donde se encuentran todas las variables que tienen un ámbito reducido (por ejemplo variables locales de funciones), y por tanto, en esa zona se está continuamente insertando y borrando variables nuevas.

3. **El** *heap* (montón), contiene memoria disponible para que se reserve y libere en cualquier momento durante la ejecución de un programa.

En la **La pila** y en **El** *heap* están los datos cuyo tamaño nse asigna hasta que se ejecuta el programa.

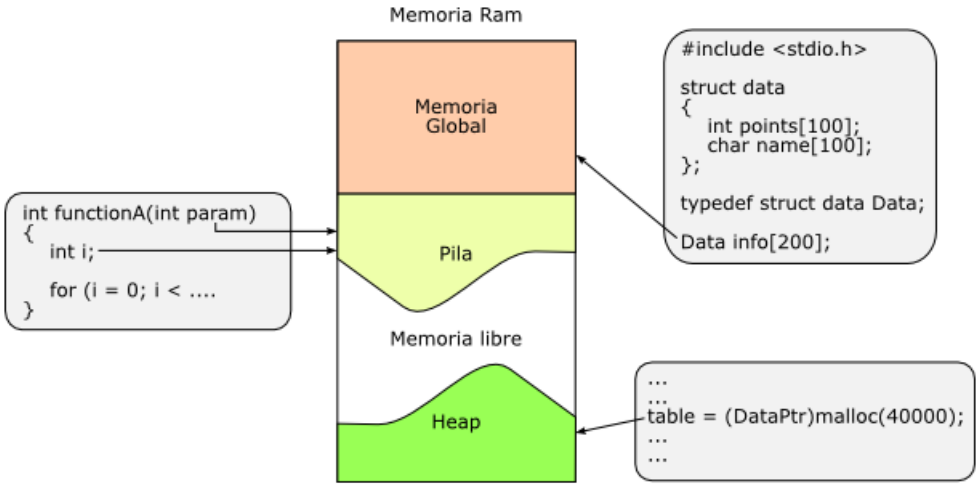


Figura 10.1: Zonas de memoria. Tomada de *Memoria dinámica* (s.f.).

Concepto

El proceso de asignación de memoria durante la ejecución del programa se llama asignación de memoria dinámica.

Este proceso permite reservar la memoria en tiempo de ejecución. Esto lo haremos a través de las funciones de biblioteca. Vamos a retomar los temas mencionados en la parte 2. Veamos los conceptos y algunos ejemplos de las funciones para gestionar la memoria.

Concepto: Gestión explícita de memoria

Es el esquema que consiste en dos tipos de operaciones: la petición y liberación de memoria. El ciclo consiste en precisar el tamaño del dato a almacenar, se solicita tanta memoria en bytes como sea necesaria, y una vez que ese dato ya no se necesita la memoria se devuelve para poder ser reutilizarda.

1. Funciones para asignación de memoria

1.1. malloc()

Suponga que al inicio definimos un arreglo de un tamaño específico, n; Sin embargo, posteriormente, requerimos una asignación de memoria diferente, $n+m$. Entonces es necesario definir un puntero sin tamaño fijo para que, considerando los requerimientos, se asigne el espacio de memoria en el tiempo de la ejecución.

Concepto

`malloc()` se usa para asignar espacio de memoria durante la ejecución del programa.

malloc(): `(TYPE *) malloc (n));` o *memory allocation*

Es una función (método) usado para asignar dinámicamente la memoria a un bloque de memoria con un tamaño especificado. Ejemplos de uso:

1. Asignación para un arreglo de n entero:
   `int * ptr = malloc (n*sizeof(int));`
2. Asignación para un arreglo con el tipo explícito
   `int * ptr = malloc (sizeof(int[n]));`
3. Asignación para un arreglo con el tipo implícito
   `int * ptr = malloc (n*sizeof(* ptr));`

Donde en `n` es un entero. Algunas veces se coloca la conversión explícita en la definición.

1.2. calloc()

Reserva espacio para tantos elementos como indica el primer argumento y cada uno de ellos con un tamaño que se indica por el segundo argumento.

Concepto

`calloc()` inicializa la asiganación de memoria en cero.

calloc(): `(TYPE *)calloc (n, tamElemento);` o *contiguos allocation*

Es una función (método) usado para asignar dinámicamente la memoria a un número especificado de bloques de memoria de un tipo particular e inicializar cada bloque en el valor cero (0).

Ejemplos de uso:

1. Asignación para un arreglo de n entero:
   `int * ptr = calloc(n, sizeof(int));`
2. Asignación para un arreglo con el tipo explícito
   `int * ptr = calloc(1, sizeof(int[n]));`
3. Asignación para un arreglo con el tipo implícito
   `int * ptr = calloc(n, sizeof(* ptr));`

Donde en `n` es un entero. Algunas veces se coloca la conversión explícita en la definición.

1.3. realloc()

Es una buena práctica liberar aquella memoria que no necesitas para esto podemos usar la función `free()`; también podemos usar `realloc()`.

Concepto

`realloc()` modifica (incrementa o decrementa el tamaño de un bloque de memoria asignado) la asignación de memoria a un nuevo tamaño.

realloc(): `(TYPE *)realloc(ptr, nuevoTam);` o *re-allocation*

Este método se usa para cambiar dinámicamente la asignación de memoria de un objeto con una memoria asignada inicialmente. Aquellos casos en los que requerimos ajustar debido, por ejemplo, a memoria insuficiente.

Donde en `n` es un entero. Algunas veces se coloca la conversión explícita en la definición.
Si no hay suficiente espacio existente en la memoria del bloque actual para extenderse, el nuevo bloque se asigna para el tamaño completo de reasignación, entonces copia los datos existentes al nuevo bloque y luego libera el bloque antiguo.

1.4. free()

Esta función libera la memoria asignada por las otras funciones y se la regresa al sistema.

Concepto

`free()` libera la memoria asignada por las otras funciones y regresa la memoria al sistema.

Como su traducción lo indica, libera memoria para regresarla al sistema.

Asignación estática de memoria	Asignación dinámica de memoria
La memoria se asigna mientras se escribe el programa. El usuario pide la memoria que será asignada.	La memoria se asigna mientras se ejecuta el programa.
El tamaño de memoria no puede ser modificado mientras se realiza la ejecución.	El tamaño de memoria puede ser modificado mientras se realiza la ejecución.

Veamos algunas diferencias entre las un par de funciones:

malloc()	calloc()
Asigna solo un bloque simple de la memoria requerida. No inicializa la memoria asignada. Contiene valores basura.	Asigna solo un bloque múltiples de la memoria requerida. Inicializa la memoria asignada al valor cero.

Ejemplo 10.1. *Ejemplos de usos de las funciones de gestión de memoria:* `malloc()`.

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 int main () {
6    char *str;
7
8    str = (char *) malloc(1);
9    strcpy(str, "Universidad"); //Nacional  Autonoma de Mexico
10   printf("String = %s y %zu\n", str,sizeof(1));
11
12   free(str);
13
14   return(0);
15 }
```

Veamos otro ejemplo. Ahora solicitamos la información desde terminal.

Ejemplo 10.2. *Ejemplos de usos de las funciones de gestión de memoria:* `malloc()`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define n 10
int main() {

   char name[n*5];
   char * description;

puts("Escriba algo");
scanf("%[^\n]", name);

description = name;

 description = (char *) malloc(n* sizeof(char));
strcpy(description, "estas en clase de Programacion II");

   printf(" %s %s\n",name, description );
}
```

IMPORTANTE: Se aconseja implementar un condicional para avisar si el sistema operativo no puede asignar más memoria para el programa, `malloc()` fallará y regresará un valor `NULL`. De esta manera se asegura que la funcion `malloc()` es exitosa.

Ejemplo 10.3. *Ejemplos de usos de las funciones de gestión de memoria:* `malloc()`.

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num, i, *ptr, sum = 0;

    printf("Ingrese el numero de elementos: ");
    scanf("%d", &num);

    ptr = (int*) malloc(num * sizeof(int));

    if(ptr == NULL)
    {
        printf("Error! Memoria no asignada.");
        exit(0);
    }

    for(i = 0; i < num; ++i)
    {
      printf("escriba el %d-numero\n",i+1);
      scanf("%d", ptr + i);
      sum += *(ptr + i);
    }
```

```
24     printf("Suma = %d. \n \tEl tamanio es %zu B.\n", sum, num *
       sizeof(int));
25
26     free(ptr);
27     return 0;
28 }
```

Ejemplo 10.4. *Ejemplos de usos de las funciones de gestión de memoria:* `calloc()`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6 int m=4, n=0;
7 int * p2;
8  p2 = calloc(m, sizeof(int[m]));
9 for(n=0; n<m; ++n){
10     *(p2+n)+=n*n;
11 printf("p2[%d] = %d y %p Bytes, %zu\n", n, *(p2+n), (p2+n),
       sizeof(p2[n]);
12 }
13     free(p2);
14 return 0;
15
16 }
```

Ejemplo 10.5. *Ejemplos de usos de las funciones de gestión de memoria:* `malloc()` *y* `realloc()`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int * ptr, i , n1, n2;
7
8  printf("Ingrese el tamanio del arreglo: \n");
9     scanf("%d", &n1);
10
11    ptr = (int *) malloc(n1 * sizeof(int));
12
13 printf("Direccion de memoria asignada previamente: \n");
14
15  for(i = 0; i < n1; ++i){
16          printf("%i> %p\t",i+1, ptr + i);
17       printf("\n");
18  }
19
20 printf("Ingrese el nuevo tamanio del arreglo: \n");
21     scanf("%d", &n2);
22     ptr = realloc(ptr, n2);
```

```
23      for(i = 0; i < n2; ++i){
24        printf("%i> %p\t",i+1, ptr + i);
25        printf("\n");
26      }
27
28      free(ptr);
29      return 0;
```

Ejercicio en clase:
Trabajo con un compañero y elabora un documento de dos cuartillas donde muestres y explíques un ejemplo con las funciones `malloc()`, `calloc()`, `realloc()` y `free()`. Además implementa estructuras (struct, union, enum u otros conceptos) Sube el archivo (PDF) a tu repositorio, adjunta el código.

2. Apuntadores y arreglos multidimensionales

Ejemplo 10.6. *Ejemplos de usos de las funciones de gestión de memoria.*

```
1 #include<stdio.h>
2
3 int main(){
4   int * ptr_x, * ptr_y;
5   int a[6] = {50, 40, 30, 20, 10, 4};
6   ptr_x = &a[0];
7   ptr_y = &a[5];
8
9   printf("ptr_x = %p; ptr_y = %p\n", ptr_x, ptr_y);
10  printf("ptr_y - ptr_x = %d \n", (int)(ptr_y - ptr_x));
11
12  return 0;
13 }
```

Un arreglo unidimensional puede ser representado en términos de un apuntador. Vamos a considerar la siguiente afirmación: "Un arreglo multidimensional puede ser expresado en notación de apuntadores"

Observa la fig.10.2, que muestra un arreglo entero bi-dimensional que tiene 10 fila y 4 (o más) columnas.

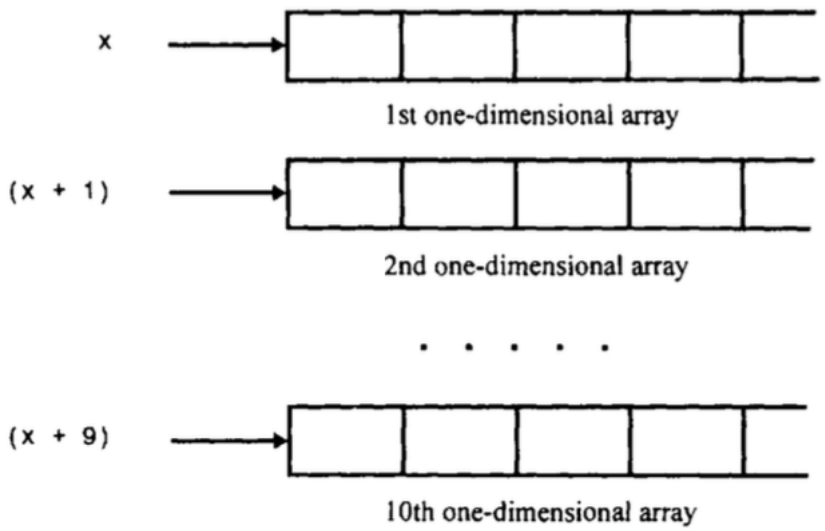


Figura 10.2: Figurada tomada de Gottfried (1996).

La forma de declarar este apuntador es: `int (*x)[4]`. En la parte superior de la figura observamos la fila, `x`, de los elementos a los que el apuntador se dirige (la fila 0), el segundo diagrama `(x+1)`, muestra la fila a la que el apuntador se dirige (la fila 1).

Para acceder a un elemento de un arreglo declarado a través de apuntadores, usamos el operador de indirección. Supongamos que deseamos acceder al elemento `x[1, 2]`; lo podemos hacer de la siguiente `*(*(x+1)+2)`. La notación `*(x+a)` apunta al primer elemento de la fila `a`

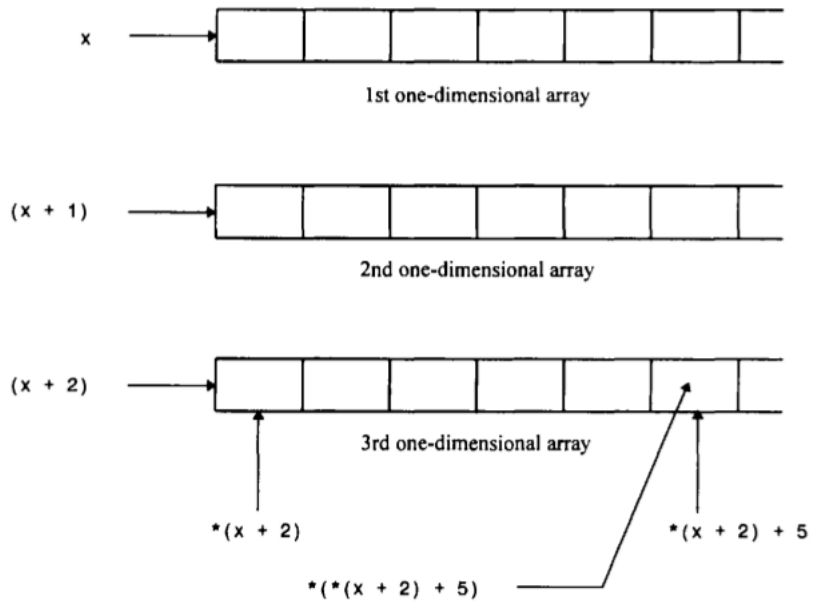


Figura 10.3: Figurada tomada de (Gottfried, 1996).

Ejemplo 10.7. *Un arreglo bi-dimensional como un arreglo de apuntadores a un conjunto de arreglos enteros unidimensionales Gottfried (1996).*

```
1 #include<stdio.h>
2 #include<stdlib.h>
```

```
#define MAXROWS 20

void leer_entrada(int * a[MAXROWS], int nrows, int ncols);
void cal_suma(int * a[MAXROWS], int * b[MAXROWS], int *
   c[MAXROWS], int nrows, int ncols);
void escri_salida(int *a[MAXROWS], int nrows, int ncols);


int main(){

int row, nrows, ncols;
int * a[MAXROWS], * b[MAXROWS], * c[MAXROWS];

puts("\t\t\tCuantas filas?");
scanf("%d", &nrows);
puts("\t\t\tCuantas columnas?");
scanf("%d", &ncols);

 for(row = 0; row < nrows; ++row)
 {
 a[row] = (int *) malloc (ncols * sizeof(int));
 b[row] = (int *) malloc (ncols * sizeof(int));
 c[row] = (int *) malloc (ncols * sizeof(int));
 }


puts("La primer tabla");
leer_entrada(a, nrows, ncols);

puts("La segunda tabla");
leer_entrada(b, nrows, ncols);

cal_suma(a, b, c, nrows, ncols);
puts("Suma de los elementos");
escri_salida(c, nrows, ncols);

return 0;
}


void leer_entrada(int * a[MAXROWS], int m, int n)
{
int row, col;

for(row = 0; row < m; ++row)
{
 printf("De la forma a b c; ingrese datos para la fila %2d\n",
   row + 1);
 for(col = 0; col < n; ++col)
 {
 scanf("%d", (*(a + row) + col));
```

```
52  }
53 }
54 }
55
56 void cal_suma(int * a[MAXROWS], int * b[MAXROWS], int *
       c[MAXROWS], int m, int n){
57 int row, col;
58
59 for (row = 0; row < m; ++row){
60  for(col = 0; col < n; ++col){
61   *(*(c + row) + col) = *(*(a + row) + col) + *(*(b + row) + col);
62   }
63  }
64 }
65
66
67 void escri_salida(int *a[MAXROWS], int m, int n){
68
69 int row, col;
70
71 for(row = 0; row < m; ++row){
72  for(col = 0; col < n; ++col){
73  printf("%4d", *(*(a + row) + col));
74     }
75  printf("\n");
76    }
77  return;
78 }
```

Observando el ejemplo 10.7 podemos decir que un arreglo bi-dimensional puede ser una colección de arreglos unidimensionales; y que un arreglo multidimensional puede ser declarado como (observa detalladamente, al omitir los paréntesis expresarás un arreglo de punteros) `TIPO (*ptr_arr)[Num_elem_2];` y no como lo escribimos en términos de arreglos: `TIPO arr[Num_Fil][N`
Al generalizar lo anterior en términos de apuntadores; un arreglo `N`-dimensional obtenemos:
Hemos usado `ptr_arr` para el identificador (nombre) de la variable apuntador y las definiciones son claras.

Algunos comentarios relacionados con el código 10.7:

1. `a, b, c` son definidos como un arreglo de apuntadores a enteros. Por lo que debemos dar a cada apuntador suficiente memoria para cada fila de cantidades enteras (`malloc()`).

2. Las funciones (prototipo) y los argumentos, también representan arreglos.

Ejercicio en clase:
Use el código 10.7 para explicar las líneas 22-24. Es decir, explíque `a[0],a[1], ...`, ¿qué puede concluir?

¿Esto es cierto?

Cada elemento del arreglo apunta a un bloque de memoria demasiado grande para almacenar una fila de cantidades enteras.

3. Resumen

Recordemos algunas cosas importantes sobre los apuntadores:

- Una variable apuntador guarda la dirección de memoria de la variable, mientras las otras variables guardan valores.
- El contenido de un apuntador en C siempre será un número total: dirección de memoria.
- Un apuntador siempre se iniciazaliza con valor NULL (nulo (= 0)).
- EL símbolo (operador) & se usa para obtener la dirección de memoria de la variable.
- el símbolo (operador) $*$ se usa para obtener el valor de la variable al que el apuntador está apuntando.
- Si el apuntador en C se asigna como NULL, esto quiere decir que está apuntando a nada.
- Dos apuntadores pueden ser substraidos para conocer cuántos elementos están disponibles entre esos dos punteros.
- Sin embargo, la operación suma de punteros, la multiplicación y la división no están permitidas.
- El tamaño de cualquier apuntador es 2 byte (para un compilador de 16 bit).
- Algunas relaciones permitidas entre arreglos y punteros:
  - `TIPO * Identificador[];` Arreglo de punteros
  - `TIPO (* Identificador)[];` Puntero a una arreglo de elementos
  - `TIPO * (* Identificador)[];` Puntero a un arreglo de punteros
  - `TIPO ** Identificador;` Apuntador de un apuntador (dos niveles de apuntadores).
- El operador de indirección significa "accede al contenido al que apunta el puntero".
- Un apuntador a una función contiene la dirección de la función en memoria.
- Algunas relaciones permitidas entre funciones y punteros:
  1. `TIPO * Identificador(TIPO id_1, TIPO id_2);` declaración de una función que recibe dos enteros como parámetros y regresa un apuntador a un enetero.
  2. `TIPO1 *I_1(TIPO2 I_2,.., TIPO3 (*I_3)(TIPO4 I_4, TIPO5 I_5), TIPO6 I_6);` Esto le dice a la función `I_1` que espere un parámetro (`I_3`) que es un apuntador a una función que recibe dos parámetros de tipos `TIPO4` y `TIPO5`y regresa un resultado de tipo `TIPO3`. Nota que implementamos los paréntesis en `(*I_3)` para indicar que `I_3` es un apuntador, si no se usamos los paréntesis obtendríamo la declaración 1.

Parte III: Manejo de archivos con C

Objetivo

Derrollar aplicaciones en lenguaje C que requieran almacenar y recuperar datos en archivos.

Capítulo 11
Introducción a archivos

El manejo de archivos en C se hace mediante el concepto de flujo, también llamada secuencia. Un flujo puede estar abierto o cerrado y conducen los datos entre el programa y los dispositivos externos L. y Zahonero Martínez (2001).

Concepto: Flujo

Un flujo (*stream*) es un concepto que se refiere a la corriente de datos que fluyen entre un origen (fuente) y un destino (sumidero).

Concepto: Canal

El objeto por donde circulan los datos se llama **canal** (*pipe*) es la conexión que existe entre el origen y el destino.

El hecho de abrir un archivo significa establecer una conexión del programa con dispositivo que contiene el archivo; por el canal que comunica el archivo con el programa fluirán las secuencias de datos.

El acceso a los archivo se hace a través de un *buffer*. Este almacena datos dirigidos al archivo o desde el archivo.

Buffer

Cuando se llama una función para leer una cadena del archivo, la función lee tantos caracteres como quepan en el buffer. Se obtiene la cadena del buffer. Llamadas posteriores obtendrán la siguiente cadena y así sucesivamente hasta que el buffer esté vacío y se llene con una llamada posterior a la función de lectura.

En C existen los archivos secuenciales de datos (estándar) y los los archivos orientados al sistema (de bajo nivel). Los más usados son los primeros. Estos archivos están divididos en dos:

- Archivos de texto:
  son los archivos que contienen caracteres consecutivos (números, letras, componentes de una cadena, etc.).
- Archivos sin formato:
  son los archivos que organizan los datos en bloques de bytes contiguos de información. Los bloques representan estructuras de datos más complejas.

Vamos a explorar diferentes conceptos sobre los archivos y su implementación en C, centrados en archivos secuenciales.

Capítulo 12
Declaración, apertura y cierre de archivos

En nuestro programa, el archivo tendrá un nombre interno que será un apuntador a una estructura predefinida (puntero a archivo). Esta estructura contiene información sobre el archivo tal como la dirección del buffer que utiliza. El tipo de identificador de la estructura es es `FILE` (pertenece a `stdio.h`).

En nuestro caso, tenemos: COMPILADOR: `gcc (MacPorts gcc48 4.8.5_2) 4.8.5` y `FILE (definido en /usr/include/stdio.h)`

Ejemplo 12.1. *información sobre el* `FILE`

```
/* ******************** TEXTO SIN ACENTOS ********************/
// Recuerda documentar tus codigos
typedef struct __sFILE {
 unsigned char *_p; /* current position in (some) buffer */
 int _r;  /* read space left for getc() */
   int _w;  /* write space left for putc() */
   short _flags;  /* flags, below; this FILE is free if 0 */
  short _file;  /* fileno, if Unix descriptor, else -1 */
   struct __sbuf _bf; /* the buffer (at least 1 byte, if !NULL) */
   int _lbfsize; /* 0 or -_bf._size, for inline putc */
     /* operations */
  void *_cookie; /* cookie passed to io functions */
  int (*_close)(void *);
  int (*_read) (void *, char *, int);
   fpos_t (*_seek) (void *, fpos_t, int);
   int (*_write)(void *, const char *, int);

   /* separate buffer for long sequences of ungetc() */
  struct __sbuf _ub; /* ungetc buffer */
   struct __sFILEX *_extra; /* additions to FILE to not break ABI
   */
    int _ur;  /* saved _r when _r is counting ungetc data */
    /* tricks to meet minimum requirements even when malloc()
   fails */
  unsigned char _ubuf[3]; /* guarantee an ungetc() buffer */
    unsigned char _nbuf[1]; /* guarantee a getc() buffer */

   /* separate buffer for fgetln() when line crosses buffer
   boundary */
   struct __sbuf _lb; /* buffer for fgetln() */

   /* Unix stdio files get aligned to block boundaries on fseek()
   */
    int _blksize; /* stat.st_blksize (may be != _bf._size) */
   fpos_t _offset; /* current lseek offset (see WARNING) */
  } FILE;
```

El detalle de los campos del FILE puede cambiar entre compiladores (mostrar en otra computadora).

Es necesario definir un apuntador a `FILE` por cada archivo a procesar. Esta variable de tipo `FILE *` representa un flujo de datos que se asocia a un dispositivo físico de entrada/salida. Algunos ejemplos de flujos de datos estándar predefinidos son:

stdin : representa la entrada estándar del sistema (teclado).

stdout : representa la salida estándar del sistema (pantalla).

stderr : representa la salida de error estŕandar (pantalla).

A continuación, revisaremos algunos conceptos de apertura y cierre de archivos.

1. Apertura y cierre de un archivo

Esta parte es importante porque supone conectar el archivo externo con el programa e indicar cómo será tratado el archivo.

En C, accedemos a los archivos a través de un puntero a la estructura `FILE`. Además usaremos algunas funciones de biblioteca como:

`fopen(nombre_archivo, parametros)`

fopen()

Esta función devuelve un apuntador a `FILE`; a través de dicho apuntador, el programa hace referencia al archivo.

`nombre_archivo` es el identificador externo. `parametros` contiene el modo en que se tratará el archivo. Más adelante discutimos otra información respecto al parámetro de los modos de archivo.

Ejemplo 12.2. *Abrir un archivo.*

```
#include <stdio.h>
int main ()
{
  FILE * archivo;
  archivo = fopen ("prueba.txt", "a");
  fclose ( archivo );
  return 0;
}
```

La función `fopen()` devuelve un apuntador a un archivo. Al usar esta función para abrir un archivo para escritura, entonces cualquier archivo existente con el mismo nombre se borrará y se crea uno nuevo.

Ejemplo 12.3. *Ejemplo para abrir un archivo.*

```
#include <stdio.h>
int main ()
{
  FILE * archivo;
  char nom[] = "licencia0.est";
  archivo = fopen ( nom, "a" );

  fclose ( archivo );
```

2. ¿Cómo abrir y escribir en un archivo?

La escritura en archivos es importante para mantener almacenada la información y no dejarla en el búfer o la caché. En las siguientes líneas exploramos la forma de crear y escribir información en archivos externos.

Lo primero que debemos considerar es que usaremos el estándar `<stdio.h>` que nos permite usar varias funciones para la edición de archivos.

De forma general debes considerar las siguientes lineas:

1. Agregar un flujo `FILE *Nom_apuntador;`
2. Abrir el archivo con `fopen("nombre_archivo.extension", "parametros");`

   `nombre_archivo.extension`, debes darle un nombre al archivo.

   `parametros` puede ser (archivos de texto):
   - `r` abre para leer.
   - `w` abre para crear nuevo archivo y escribir (pierde información, en caso de existir)
   - `a` abre para escribir (crear) al final del archivo.
   - `r+` abre un archivo para lectura y escritura de un archivo existente.
   - `w+` abre un archivo para actualizar (leer y escribir).
   - `a+` abre un archivo para actualizar (leer y escribir) al final del archivo.

3. Cerrar el archivo con `fclose (Nom_apuntador);`
4. Escribir algo en el archivo: Podemos realizarlo de diferentes formas:

   `fputc(variables_introducir, Nom_archi_apuntador);` para poner caracteres en el archivo.

Algunas veces se implementa una letra `t` para indicar que es texto; ejemplo: `rt`; en otros casos se implementa `rb` para indicar que el archivo es binario.

El cierre de los archivos se trabajan con memoria intermedia (*buffer). Es posible que al terminar la ejecución del programa queden datos en el buffer. Por lo que es importante cerrarlos para que se limpie el buffer.*

Ejemplo 12.4. *Un ejemplo de cierre de archivos en C.*

```
1 #include <stdio.h>
2
3 int main ()
4 {
5   char * archivo[2]={"04-files01.dat", "04-files01.txt"};//, *
     archivo2;
6   FILE * archivo1, * archivo2;
7
8   archivo1 = fopen ( archivo[0], "at" );
9   archivo2 = fopen ( archivo[1], "at" );
10
11  fclose(archivo1);
```

La función `fclose()` *cierra una secuencia que fue abierta mediante una llamada a* `fopen()`*. Escribe toda la información, que todavía se encuentre en el buffer, en el disco y realiza un cierre formal del archivo a nivel del sistema operativo.*

Buffer

Un error en el cierre de una secuencia puede generar problemas como: la pérdida de datos, destrucción de archivos y entre otros.

Evita retirar un periférico de almacenamiento antes de tiempo.
En la siguiente sección, veremos algunas funciones de para la escritura y lectura de datos en un archivo.

Capítulo 13
Escritura y lectura de archivos

funciones de entrada y salida

Las funciones
`printf()`, `scanf()`,`getc()`, `putchar()`, `gets()`, `puts()` *tienen su versión para archivos*
`fprintf()`, `fscanf()`, `fgetc()`, `fputchar()`, `fgets()`, `fputs()`.

Nota que las funciones que operan sobre archivos empiezan con `f`.
Mostraremos algunos prototipos de las funciones de biblioteca.

- `int fgetc(FILE * f_ptr);`

- `int fgets(char * s, int n, FILE * f_ptr);`
  lee una serie de caracteres de un fichero y los asigna a una cadena. Primer argumento, valor devuelto; segundo, tamaño `n -1`*; y, tercero, el apuntador.*

- `int fputs(char * s, FILE * f_ptr);`
  primer argumento, caracteres que forman una cadena; segundo argumento, el apuntador.

- `fscanf(FILE * fptr, "%i %c", int a, char b);`
  primer argumento, el apuntador; segundo, datos a imprimir; y, tercero, los datos con sus tipos (ver 13.3).

Ejemplo 13.1. *Veamos un ejemplo de algunas funciones de biblioteca.*

```
1 #include <stdio.h>
2 #include <ctype.h>
3
4 int main ()
5 {
6   FILE * f_ptr;
7 char c;
8
9 f_ptr = fopen("ejemplo.dat", "w");
10
```

```
printf("Escribe algo (terminal) \n");

fprintf(f_ptr,"Escribe algo (fichero)\n");
do{
putc(toupper(c = getchar()), f_ptr);
}while(c !='\n');

  fclose (f_ptr);
  return 0;
}
```

Recordemos la forma de intercambiar información entre archivos de C.

Ejemplo 13.2. *Intercambio de información entre archivo en C.*

```
float suma(float x, float y){

return x+y;

}
```

```
/*
Ejemplo de un archivo llamando a otro.
gcc 06-fun-02.c
*/

#include<stdio.h>
float suma(float x, float y);

int main(){
float a = 2;
float b = 4;
printf("suma a + b = %f\n",suma(a, b));

return 0;
}
```

Ahora veamos cómo intercambiar información entre archivos.

Ejemplo 13.3. *Intercambio de información entre archivo en C.*

```
#include<stdio.h>
#include<stdlib.h>
int main(){

FILE * fptr1, *fptr2;
char c1, c2;


fptr1 = fopen("07-files01.txt","r"); //r == read

 if(fptr1 == NULL)
   {
```

```
13      printf("Error al abrir el archivo original");
14      exit(1);
15 }
16
17 fptr2 = fopen("07-files02.txt","w");
18  fscanf(fptr1,"%c %c", &c1, &c2);
19  fprintf(fptr2, "%c %c",c1, c2);
20
21 fclose(fptr1);
22 fclose(fptr2);
23
24
25 return 0;
```

Ejercicio en clase:
Modifique el código del ejemplo 13.3 para que lea la cadena completa de caracteres que se encuentra en el primer archivo y la copie al segundo archivo.

Ahora veremos más ejemplos, usando las funciones de biblioteca.

Ejemplo 13.4. *Para escribir un caracter a un fichero.*

```
1 #include<stdio.h>
2 int main(){
3
4   FILE * fptr1;
5   char c1;
6
7   printf("Introduce un caracter \n");
8   scanf("%c", &c1);
9
10  fptr1 = fopen("08-files.txt","w");
11  fputc(c1, fptr1);
12
13  fclose(fptr1);
14
15 return 0;
16 }
```

Ejemplo 13.5. *Para tomar un caracter a un fichero.*

```
1 #include<stdio.h>
2
3 int main(){
4   FILE * fptr;
5   char cr;
6
7   fptr = fopen("09-files.txt","r");
8   cr = fgetc(fptr);
```

```
  fclose(fptr);

  printf("El caracter del archivo es...%c\n", cr);
  return 0;
}
```

La función `fgetc()` *lee un caracter de un archivo. Esta función recibe como un argumento a el apuntador de* `FILE` *para el archivo desde el cual el caracter se lee. Cuando la función está de la forma* `fgetc(stdin)` *esto lee un caracter desde* `stdin` *(que es el estándar de entrada), equivalente, a* `getchar()`.

Ejercicio en clase:
Modifique el código del ejemplo 13.5 para que lea la cadena completa de caracteres que se encuentra en el archivo.

Ejemplo 13.6. *Para escribir usando* `fputs()` *y* `fgets()`.

```
#include<stdio.h>

int main(){
  FILE * fptr;

char cad[160] = "Bienvenidos al curso de programacion estudiantes
   de MAC G2251";

fptr = fopen("10-files00.txt","w");
fputs(cad, fptr);

fclose(fptr);
return 0;
}
```

Vamos a usar la función `fprintf` *para escribir a un archivo. Esta función se usa para escribir en el archivo. Es equivalente a* `printf`*, excepto por los argumentos. La forma general de la función es:*
`fprintf(FILE * fptr, "%d %s %.2f\n", int a, char b, double c);` *donde los argumentos del tercer al quinto son ejemplos.*

Ejemplo 13.7. *Implementación de* `fprintf`.

```
int main(){
  FILE * fptr;
 char cadr;


fptr = fopen("11-fles00.txt","a");

puts("Escribe algo (un caracter)\n");

scanf("%c", &cadr);
```

```
11
12 fprintf(fptr,"El caracter es...%c\n", cadr);
13
14 fclose(fptr);
15   return 0;
16 }
```

Ejercicio en clase:
Realice un código que contenga el historial de los empleados de una empresa, cuyos datos serán ingresados desde la terminal. Implemente: `struc`, `enum`, `typedef`, `FILE` *y las funciones de biblioteca necesarias para que la información se guarde en un archivo externo. Presente un reporte con el archivo de salida de, al menos, cinco (5) empleados con, al menos, siete (7) datos. Implemente un formato adecuado de presentación como el que se muestra en la siguiente figura*.*

```
CUSTOMER BILLING SYSTEM - INITIALIZATION

Please enter today's date (mm/dd/yyyy): 5/24/1998

Name (enter 'END' when finished): Steve Johnson
Street: 123 Mountainview Drive
City: Denver, CO
Account number: 4208
Current Balance: 247.88
```

Figura 13.1: Ejemplo de la salida implementada en un archivo.

** sin el subrayado.*

Las funciones `fprintf()` *y* `fscanf()` *permiten escribir o leer variables de cualquier tipo de dato estándar, ambas tienen como primer argumento el puntero a* `FILE` *asociado.*

Ejercicio en clase:
Agregue una función del ejercicio relacionada a la figura 13.1 para que le permita actualizar los datos de alguno de los empleados.

Capítulo 14
Actualización de archivos

Ejercicio en clase:
El estudiante debe realizar la actividad que aparece en el sitio: github/partIII/readme.md

1. Archivos sin formato

Almacenar bloques de datos consiste en almacenar un número fijo de bytes contiguos. Estos bloques representan una estructura de datos que pueden ser: `struct` *o arreglos. Por ejemplo, arreglos del mismo tamaño pueden ser escritos o leídos como un bloque en vez de hacerlo de manera separada. Para esto usamos las funciones* `fread` *o* `fwrite`*. Algunas veces llamadas funciones de lectura o escritura sin formato y a los archivos creados se les conoce como archivos de datos sin formato. La forma general de estas funciones son:*

- `fwrite(&bd, tam, nbt, fpt)`
- `fread(&bd, tam, nbt, fpt)`

donde,

1. `bd` *es un puntero al bloque de datos,*
2. `tam` *es el tamaño del bloque de datos (cada registro),*
3. `nbt` *es el número de bloques a transferir y*
4. `fpt` *es el puntero a un archivo.*

Nótese que tienen la misma estructura.

Ejemplo 14.1. *fwrite y fread*

Ejercicio en clase:

1. *Implementa documentación interna de los archivos anteriores.*
2. *Realiza tus códigos con un mensaje "secreto" (de más de 50 caracteres), elige un compañero al que le compartas tu archivo binario y un archivo de lectura para que pueda ver la información.*
3. *Comparte tu experiencia.*

Los archivos[1] *binarios consisten de una secuencia de bytes que representan números, arreglos o estructuras.*

Las funciones `fwrite` *y* `fread` *se usan para leer un bloque de caracteres (bytes) a diferencia de otras funciones (*`fgetc`*, etc.) que leen byte a byte. Los datos en modo binario se almacenan en la misma forma que en la memoria, por lo que no se lleva a cabo ninguna transformación de datos, por lo que en este modo es más rápido que en modo texto.*

2. Resumen general

1. *la función* `eof()` *(end-of-file), es un indicador, que informa al programa que no hay más datos para ser procesados. la función* `feof()` *es su equivalente para archivos.*
2. *La macro* `NULL` *está definida en* `STDIO.H`*. Este método detecta cualquier error al abrir un archivo; e.g. disco lleno o protegido contra escritura antes de comenzar la escritura.*

[1] La palabra correcta es flujoss que es un conjunto de datos o unaserie de bytes.

Parte IV: Manipulación de Bits

Objetivo

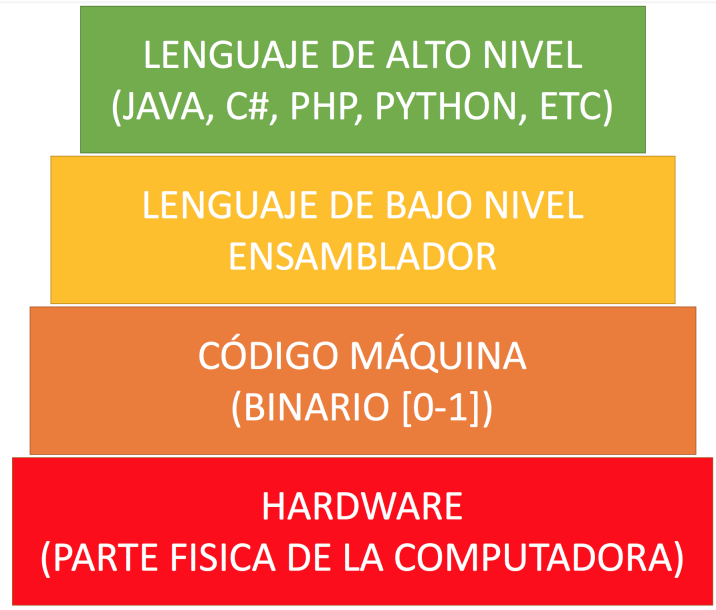
Desarrollar programas en lenguaje C que requieran el manejo de bits.

Ya hemos revisado conceptos del lenguaje de programación (C), ahora vamos a estudiar la manipulación de bits (desplazar bits individuales dentro de una palabra, por ejemplo) y otros conceptos en C. Debemos retomar los tipos de almacenamiento del curso anterior (ver sección 16.4 del guión del curso anterior). En particular, retomemos el el tipo de almacenamiento `register`*. Las variables con este tipo de almacenamiento tienen ventajas de rapidez de manipulación. Esto sucede porque se guardan en áreas especiales de almacenamiento dentro de la unidad central de procesamiento (CPU).*

Capítulo 15
Programación de bajo nivel

C posee ciertas características de un lenguaje de programación de **bajo nivel***. Sin embargo, para algunos autores es claro que C es un lenguaje de alto nivel y otros que difieren (auth.) (2017). A veces se describe como un lenguaje ensamblador de alto nivel o un lenguaje con soporte de programación de bajo nivel, o, simplemente, lenguaje de nivel medio.*

Que un lenguaje de programación sea da **bajo nivel** *quiere decir que hay instrucciones con las que se puede ejercer un control directo sobre el hardware (características propias de lenguajes como ensamblador o máquina); es decir un lenguaje más cercano al código de máquina. Por otro lador, un lenguaje es de* **alto nivel** *(Java, Python, C#) cuanto más lejano se encuentre del lenguaje de máquina y más cercano al lenguaje de humano.*



Recordemos (mencionemos) algunas características de C. . .

1. *Permite manipulación de memoria.*
2. *Paradigma de programación imperativa-estructurado, que permite trabajar de forma cercana con el hardware*

3. *Permite programar desde aplicaciones sencillas hasta sistemas operativos.*

Algunas desventajas...

1. *No muy usado, actualmente.*

2. *Pocas cosas hechas (implementadas).*

En esta sección exploramos algunas características de C que nos permite manipular bits y las operaciones para manipularlos.

Capítulo 16
Operadores y operaciones con bits

En esta capítulo revisaremos algunos operadores y su significado en C. Estos operadores son útiles para manipular bits, los cuales son importantes para la programación de microcontroladores, por ejemplo.

Símbol	Nombre	Explicación
&	AND	Simula una compuerta lógica and.
|	OR	Simula una compuerta or.
^	XOR	Simula una compueta xor.
~	Complemento	Complemento a uno (A1).
<<	Desplazamiento izq.	Desplazamiento a la izquierda.
>>	Desplazamiento der.	Desplazamiento a la derecha

Cuadro 16.1: Esta tabla muestra los operadores que se pueden usar en *bitwise.*

1. Las tablas de verdad y sus códigos

Veamos algunas tablas de verdad:

&	0	1
0	0	0
1	0	1

Ejemplo 16.1. *Código para implementar AND.*

```
#include <stdio.h>

int main ()
{
  int a= 23;
  int b = 90;
  int resultado;
  printf("a = %d y b = %d\n", a, b);
  resultado = a & b;
  printf("a & b  = %d\n",resultado);
  return 0;
}
```

Ejercicio en clase:

1. *Convierta a binarios y compruebe el resultado del código 16.1.*
2. *Realice un programa que tome un número en decimal, conviértalo a binario y averigüe si algún bit es cero o uno.*

Vamos a revisar la operación OR.

|	0	1
0	0	1
1	1	1

Ejemplo 16.2. *Código para implementar la operación OR.*

```
#include <stdio.h>

int main ()
{
  int a= 23;
  int b = 90;
  int resultado;
  printf("a = %d y b = %d\n", a, b);
  resultado = a | b;
  printf("a | b = %d\n",resultado);
  return 0;
}
```

Ejercicio en clase:
Confirme los números y resultados en una operación binaria.

Implementemos la operación binaria OR exclusiva.

^	0	1
0	0	1
1	1	0

Ejemplo 16.3. *Código para implementar la operación XOR.*

```
#include <stdio.h>

int main ()
{
  int a= 23;
  int b = 90;
  int resultado;
  printf("a = %d y b = %d\n", a, b);
  resultado = a ^ b;
  printf("a ^ b = %d\n",resultado);
  return 0;
}
```

Ejercicio en clase:
Confirme los números y resultados en una operación binaria.

La operación complemento A1 a la base.

Es necesario recordar que un byte está compuesto por 8 bits. El bit más significativo (MSB, siglas en inglés), se encuentra a la izquierda y el El bit menos significativo (LSB, siglas en inglés), se encuentra a la derecha. El MSB se considera el bit de signo si implementamos la representación módulo y bit de signo. Para esto usaremos la siguiente tabla:

$\sim$	*signo*
0	-
1	+

Básicamente, el resultado se comporta como el negativo del número original con adición binaria -1 *(*$0x + 01$ *con* x *expresado en sistema binario). En sistema decimal, un número* n *el complemento a uno se construye usando* $-(n + 1)$.

Ejemplo 16.4. *Código para implementar la operación complemento A1.*

```
#include <stdio.h>

int main ()
{
  int a= 4;
  int resultado;
  printf("a = %d\n", a);
  resultado = ~a;
  printf("~a = %d\n",resultado);
  return 0;
}
```

2. Desplazamiento de bits

El desplazamiento de bits es importante para modificar un número a niveles de bits. Veamos un ejemplo:

Número binario	bits	Resultado	Número Decimal
01010110	0	01010110	86
01010110	1	00101011	43
00101011	2	00010101	21
00010101	3	00001010	10
00001010	4	00000101	5
00000101	5	00000010	2

Cuadro 16.2: Sistema binario, sistema decimal y el corrimiento a la derecha.

Ejemplo 16.5. *Código para desplazar bits a la derecha*

```c
#include <stdio.h>

int main ()
{
  int a= 86;
  int resultado;
  printf("a = %d\n", a);
resultado = a>>1;
 printf("a>>1 =  %d\n",resultado);

resultado = a>>2;
 printf("a>>2 =  %d\n",resultado);

resultado = a>>3;
 printf("a>>3 =  %d\n",resultado);

resultado = a>>4;
 printf("a>>4 =  %d\n",resultado);

 resultado = a>>5;
 printf("a>>5 =  %d\n",resultado);
  return 0;
}
```

Ejercicio en clase:

1. *Implementa el código anterior en el sistema decimal y binario.*
2. *Implementa el desplazamiento a la izquierda con los números en sistemas decimal y binario. Implementa un cuadro similar al 16.2 de 10 filas.*

Los operadores que aparecen en la tabla 16.1 puede usarse con variables en otros sistemas, veamos un ejemplo:

Vamos a realizar operaciones con hexadecimales:

Ejemplo 16.6. *Código que realiza operaciones en el sistema hexadecimal.*

```c
#include <stdio.h>
int main ()
{
  int a=0XF5;
  int b = 0XFA, resultado;
  printf("Los numeros son: a = 0x%X y b = 0x%X\n", a, b);
  printf("Los numeros son: a = %d y b = %d\n", a, b);

resultado = a & b;

  printf("Los numeros son: a & b = 0x%X  \n", resultado);
  printf("Los numeros son: a & b = %d  \n", resultado);
```

```
13   return 0;
14 }
```

donde observa el prefijo `0X` *que implementa C. Recuerda que para representar cualquier número binario a hexadecimal el procedimiento es:*

1. *Hacer grupos de 4 bits, iniciando desde la derecha.*
2. *Sustituir cada grupo por su valor hexadecimal.*

Recuerda el prefijo `0X`*.*

Ejercicio en clase:
Convierte 01 1010 1111 0100 1111 0101 0101 a hexadecimal[a].

[a] 0X1AF4F55

Capítulo 17
Campos de bits

Concepto

Una estructura compuesta por bits se denomina campo de bits.

Un campo de bits es una manera de utilizar el espacio de memoria de situaciones donde se almacenan ceros y unos.

La forma general de un campo de bits es:

Ejemplo 17.1. *Parte del código que muestra un campo de bits.*

```
1 typedef struct str_bts{
2 TIPO nombre: <cantidad_bits>;
3 TIPO nombre: <cantidad_bits>;
4 TIPO nombre: <cantidad_bits>;
5 } cam_bits;
6
7 /*
8 Veamos un ejemplo
9 */
10
11 typedef struct str_bts{
12 unsigned int mostrar:1;
13 unsigned int transparencia:1;
14 } cam_bits;
```

Los números indican el número de bits en el campo de bits que usarán, en total 8. Aunque la estructura tendrá un valor diferente de 4 bytes. De esta manera usaremos hasta 32 variables cada uno con un 1 bit reservado. Sin embargo, cuando se necesiten una más (33 variables) se iniciará un espacio de memoria extra con 4 bytes.

Ejemplo 17.2. *Código que muestra la comparación entre una estructura y una estructura con campo de bits.*

```
#include <stdio.h>
#include <string.h>
/* Definicion de una estructura simple */
struct {
   unsigned int var1;
   unsigned int var2;
   unsigned int var3;
   unsigned int var4;
   unsigned int var5;
   unsigned int var6;
} st1;
/* Definicion de una campo de bits */
struct {
   unsigned int var1 : 15;
   unsigned int var2 : 1;
   unsigned int var3 : 21;
   unsigned int var4 : 10;
   unsigned int var5 : 10;
   unsigned int var6 : 15;
} st2;

int main(){

   printf("Tamanio de memoria ocupada por (st1) la estructura=
   %d\n", (int)sizeof(st1));
   printf("Tamanio de memoria ocupada por (st2) el campo de bits=
   %d\n", (int)sizeof(st2));
   return 0;
}
```

En el siguiente código:

1. *El número indica cuantos bits se permiten 00000000. Inicia desde LSb al MSb.*
2. *Cambia el número y observa lo que sucede.*
3. *¿Qué debes hacer para que se imprima el valor correcto?*

Ejemplo 17.3. *Código que muestra el uso del campo de bits.*

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

struct {
   unsigned int age : 4;
} Age;

int main( ) {

   Age.age = 4;
```

```
11    printf( "Sizeof( Age ) : %d\n", (int)(sizeof(Age)) );
12
13    printf( "Age.age : %d\n", Age.age );
14
15    Age.age = 7;
16    printf( "Age.age : %d\n", Age.age );
17
18    Age.age = 8;
19    printf( "Age.age : %d\n", Age.age );
20
21    Age.age = 63;
22    printf( "Age.age : %d\n", Age.age );
23
24    return 0;
25 }
```

Ejercicio en clase:

1. *Realiza un programa que convierta cantidades entre los diferentes sistemas de unidades.*
2. *Realiza un programa que realice operaciones aritméticas y con operadores booleanos.*

Las salida debe implementarse en un archivo de salida externo (.txt).

Este método tiene algunas ventajas y desventajas:

1. *Mayor velocidad de procesamiento al momento de ejecutar el programa.*
2. *Ahorro de memoria en caso de modificar variables.*
3. *Ocupan poco espacio.*
4. *Sirven para programar microcontroladores.*

1. *Código poco legible.*
2. *Poca portabilidad.*
3. *Dependen de la arquitectura.*
4. *no se puede usar arreglos dentro de un miembro de un campo de bits; es decir, no es posible escribir:* `struct sDatos{unsigned int b[5]: 2;};`

Ejercicio en clase:
Suponga que tiene una tarjeta de adquisición de datos que proporciona una señal en una posición de valor conocido; por ejemplo, `unsigned short`*. Active (los siguientes estados están en sistema decimal):*

- *el estado 1 para obtener su nombre completo,*
- *el estado 2 para obtener su primer nombre,*
- *el estado 3 para obtener su segundo nombre,*
- *el estado 4 para obtener su primer apellido,*
- *el estado 5 para obtener su segundo apellido,*
- *el estado 6 para obtener sus dos apellidos,*
- *el estado 7 para obtener sus dos nombres,*
- *el estado 8 para obtener su primer nombre y primer apellido,*
- *el estado 9 para obtener su segundo nombre y segundo apellido,*
- *el estado 10 para obtener un alias.*
- *use otros estados, al menos cuatro, para imprimir su edad, fecha de nacimiento y otros datos.*

Vamos a complementar las operaciones con otras relaciones a nivel de bits.

Ejemplo 17.4. *Otras operaciones a nivel de bits (bit a bit o Bitwise)*

```
1  #include <stdio.h>
2
3  int main(){
4  int a = 27;
5  int b = 19;
6  int c = 22;
7
8  printf("!(!a) = %d\n",!(!a));
9
10 printf("!a = %d\n",!a);
11
12 printf("a | a = %d\n",a | a);
13
14 printf("a | !a = %d\n",a | !a);
15
16 printf("!a | !b = %d = %d =!(a & b)\n", !a | !b, !(a & b));
17
18 printf("!a & !b = %d = %d =!(a | b)\n", !a & !b, !(a | b));
19
20 printf("a & b = %d \n", a & b);
21
```

```
22 printf("a & !a = %d \n", a & !a);
23
24 printf("a | b = %d = b | a \n", b | a);
25
26 printf("a & b = %d = b & a \n", b & a);
27
28 printf("(a | b) | c = %d = %d = a | (b | c)\n", (a | b) | c, a |
       (b | c));
29
30 printf("(a & b) & c = %d = %d = a & (b & c)\n", (a & b) & c, a &
       (b & c));
31
32 printf("a & b | a & c = %d = %d = a & (b | c)\n", a & b | a & c,
       a & (b | c));
33
34 printf("(a | b) & (a | c) = %d = %d = a | (b & c)\n", (a | b) &
       (a | c), a | (b & c));
35
36 printf("a | a & b = %d = %d = a\n", a | a & b, a);
37
38 printf("a & b | a & !b = %d = %d = b\n", a & b | a & !b, b);
39
40 printf("(a & b) | (!a & c) | (b & c) = %d = %d = (a & c) | (!a &
       c)\n", (a & b) | (!a & c) | (b & c), (a & b) | (!a & c));
41
42 printf("(a | b) & (!a | c) & (b | c) = %d = %d = (a | b) & (!a |
       c)\n", (a | b) & (!a | c) & (b | c), (a | b) & (!a | c));
43
44 printf("a & (a | b) = %d = %d = a\n", a & (a | b), a);
45
46 printf("(a | b) & (a | !b) = %d = %d = a\n", (a | b) & (a | !b),
       a);
47
48 return 0;
49 }
```

Ejercicio en clase:

1. *Implementa documentación interna.*
2. *Realiza las operaciones en tu cuaderno con tres números aleatorios (dados por el usuario desde la terminal). Comprueba las operaciones convirtiendo de sistema decimal a sistema binario.*
3. *Implementa la salida en un archivo externo.*

Parte V: Graficación básica con C, LaTeX, Python, Gnuplot y Mathematica Wolfram (y Swift)

En esta parte se entregan recursos básicos para la graficación. Usaremos diferentes lenguajes de programación, esto implica más trabajo continuo y personal.

Objetivo

Desarrollar aplicaciones que utilicen funciones gráficas.

Esta parte tendrá una amplia riqueza de material, ya que usaremos diferentes paquetes y lenguajes de programación. Es importante que el estudiante encuentre un alto grado de concentración. Además debe desarrollar habilidades para buscar material de apoyo en diferentes idiomas.

Capítulo 18
Recursos para la graficación

Usaremos diferentes ideas y conceptos para realizar graficación. El estudiante notará que hay semejanza a pesar de considerar diferentes lenguajes de programación.

1. Graficación con C

La graficación en este lenguaje de programación depende mucho del sistema operativo. Por ejemplo:

1. *En windows, la biblioteca* `#include<graphics.h>` *(posiblemente) requiere de instalación a "mano". posible solución dada por miembros del GCCyC* [1].

2. *en linux (18.04 LTS)* [2],

   a) *Puedes descarlo de: libgraph-1.0.2.tar.gz*

   b) *sudo apt(-get) install build-essential*

   c) *Instalar los paquetes:*

   ```
   sudo apt-get install libsdl-image1.2 libsdl-image1.2-dev guile-1.8 \
   guile-1.8-dev libsdl1.2debian libart-2.0-dev libaudiofile-dev \
   libesd0-dev libdirectfb-dev libdirectfb-extra libfreetype6-dev \
   libxext-dev x11proto-xext-dev libfreetype6 libaa1 libaa1-dev \
   libslang2-dev libasound2 libasound2-dev
   ```

   d) *Extraer* `libgraph-1.0.2.tar.gz` *y copiarlo a la ruta:* `/usr/local/lib/`*. Requiere permisos de administrador.*

   e) *Ir a la ruta anterior y en la terminal escribir los siguientes comandos:*

   ```
   ./configure
   make
   sudo make install
   sudo cp /usr/local/lib/libgraph.* /usr/lib
   ```

[1] Miguel Briones comparte la siguiente solución Otras soclución ver Aquí. o una más Geeks. Descarga la carpeta con los archivos desde aquí.

[2] Ver Aquí.

Ahora puedes graficar en C, usando:

```
gcc nombreArchivo.c [-o archivoSalida] -lgraph
```

No olvides que requieres:

```
int gd=DETECT,gm;
initgraph(&gd,&gm,NULL);
```

Es momento de realizar diferentes ejercicios usando la biblioteca `graphics.h`. *Revisa los ejercicios que aparecen en el repositorio. Usa las referencias que se dejan para investigar las funciones y los conceptos relacionados con esta y otras bibliotecas que sirven para graficar.*

2. Graficación con LaTeX

En caso de que quieras graficar usando LaTeX, requieres:

1. *Instalar LaTeX*
2. *Usar los paquetes requeriodos:*
   a) `\usepackage{pstricks}, \usepackage{pst-plot}` *y* `\usepackage{xcolor}`
   b) *Crear algunos ámbitos (ambientes) para colocar las imágenes.*
   c) *Python y LaTeX*

3. Graficación con Python

Debe usar los recursos que se encuentran en el sitio SAE y GitHub Usaremos algunos módulos de python como:

1. `Tkinter`
2. `wx`
3. `visual`

Implementaremos funciones de graficación, combinación de texto y gráficas y amimación en 2D y 3D.

Es importante leer el material que aparece en el GitHub y SAE.

Parte VI: Introducción a la programación orientada a objetos: Conceptos básicos

Objetivo

Describir las características del paradigma Orientado a Objetos

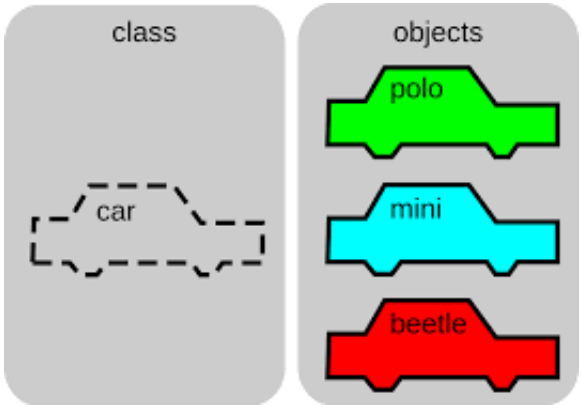
Capítulo 19
Paradigma de Programación Orientada a Objetos

Este paradigma de programación tuvo un gran avance al introducir conceptos importantes (instancias). El primer lenguaje de programación para OOP [1] *fue* `simula` *Hubo diferentes contribuciones al lenguaje de programación, Los estudiantes deben leer el artículo Black (2013) que se encuentra en OOP historia y más. En caso de tener restricciones de descarga, debes asistir a la biblioteca para solicitar tu acceso remoto.*

Ejercicio en clase:
Realizar una lectura y discusión del artículo Black (2013). Realiza un esquema que contenga información global del surgimiento de OOP.

Capítulo 20
Introducción al análisis y diseño Orientado a Objetos

El mundo está creado por objetos. En los programas de OOP tomaños como **objetos** *a las características (color, sabor, olor, forma), eventos (interrupción, conexión), estados (movimiento, calma), y todo lo que podamos llamar con un nombre, incluso objetos "abstractos" como la belleza (Pecinovsky, 2013).*



0.1. Creando un ejemplo

Pensemos en crear una aplicación que que calcule las y los perímetros de (Hillar, 2015):

- *Cuadrados*
- *Rectángulos*

[1] Usaremos estas siglas para referirnos a la programación orientada a objetos.

- *Círculos*
- *Elipses.*

Podemos crear cuatro funciones diferentes para calcular el área; y cuatro para calcular el perímetro de las figuras mencionadas.

Ejercicio en clase:
Dar ejemplos de las etiquetas que identifiquen los Objetos[a]*. E.g.:* `AreaCirculo`.

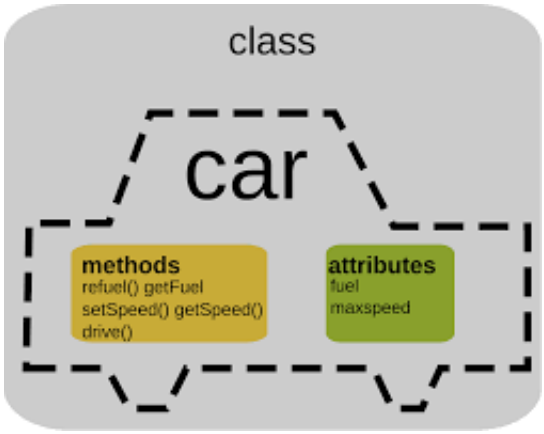
[a] Estos Objetos se refieren a la primer "O" de OOP

Estos Objetos representan el estado y el comportamiento de las figuras geométricas. A estos Objetos les asociamos atributos (datos requeridos) para calcular las áreas y los perímetros.

Concepto: Atributos

Las variables definidas para encapsular datos se conocen como atributos o campos.

A continuación mostramos un esquema para mostrar el concepto de atributo.



En la figura anterior mostramos un ejemplo de las clases, métodos y atributos.

Ejercicio en clase:
Haz una lista de los datos requeridos para calcular las áres y los perímetros de cada una de las figuras.

En OOP se requiere del concepto de: modelo, esquema o plan de trabajo(framework) que nos ayuda a simplificar el proceso completo. A esto se le llama **clase**. *Con esta clase se define el estado y el comportamiento del Objeto, y la podemos usar para generar objetos que representan el estado y el comportamiento de un objeto del mundo real.*

Concepto: Clase

Son grupos de elementos que tienen caracterśticas en común.

Usando Objetos podemos crear (en la computadora) objetos. Cada uno de estos objetos tendrán diferentes anchos y altos: datos requeridos para crear diferentes instancias.

Concepto: Objeto

Un Objeto (instancia) es una unidad, que pertenece a cierta clase, que consta de un estado y un comportamiento.

Ejemplo: mi computador es una `instancia` *de la* `clase` *computadores personales.*

Ejercicio en clase:
Vamos a crear seis (6) cuadrados: podemos usar una clase (`class`*) llamada* `cuadrado` *que sea un esquema para generar las seis diferentes instancias.*

OOP nos permite descubrir el modelo usado para generar un objeto específico. Por ejemplo, usando una `class` *podemos inferir que cada objeto es una instancia de esa clase.*

Es importante asegurarnos que cada clase tiene las variables necesarias que encapsulan todos los datos requeridos para los objetos que realizan todas las tareas. Esto lo haremos usando atributos. Regresando al ejemplo de los cuadrados, debemos conocer el lado de cada uno de los seis cuadrados. Necesitamos una variable encapsulada que permita que Objeto de la clase (`cuadrado`*) especifíque el valor de la longitud de cada lado.*

Concepto: Encapsulamiento

El encapsulamiento se refiere a guardar los datos requeridos.

El encapsulamiento de datos es uno de los conceptos más importantes de OOP.
Existen algunas sutilezas que se deben tener en cuenta en la OOP, e.g.:

1. *Si bien consideramos que* `instancia` *y* `objeto` *son sinónimos; los últimos se usan cuando se habla de* `objetos` *en general; mientras que la primera se usa para resaltar que una instancia pertenece a una* `clase`*; por ejemplo: El* `objeto` *XXX es una instancia de la clase* `YYY`*.*

2. *una* `clase` *también puede ser un objeto. Es un objeto que puede crear instancias, entre otras cosas. Sin embargo, este punto puede ser discutido dependiendo del lenguajes de programación.*

Concepto: Herencia

Se refiere a la facilidad de que una clase le otorgue la las operaciones y atributos a otra clase como si hubieran definido en la segunda clase.

Concepto: Atributo

Se refiere a las características que tiene una clase.

Concepto: Método

Se refiere a los algoritmos asociado a un objeto. Es lo que el Objeto puede hacer (comportamiento).

Parte VII: Anexos

Título Título Título Título

Autor Autor Autor Autor

February 12, 2018

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

2 Models, methods or materials

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum

libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

3 Results

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

4 Discussion or Conclusiones

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

5 References

Referencias

(auth.), I. Z. (2017). *Low-level programming: C, assembly, and program execution on intel® 64 architecture*. Apress.

Black, A. P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, *231*, 3 - 20. (Fundamentals of Computation Theory)

Gottfried, B. (1996). *Schaum's outline of programming with c* (2.ª ed.). McGraw-Hill.

Hillar, G. C. (2015). *Learning object-oriented programming: Explore and crack the oop code in python, javascript, and c*. Packt Publishing.

Kernighan, B., y Ritchie, D. (1991). *El lenguaje de programación c*. Pearson Educación.

L., J. A., y Zahonero Martínez, I. (2001). *Programación en c: Metodología, estructura de datos y objetos*. McGraw-Hill.

Memoria dinámica. (s.f.). Descargado de http://www.it.uc3m.es/abel/as/MMC/M1/MallocFreeExplained_es.html

Molina-López, J. (2006). *Fundamentos de programación. grado superior*. McGraw-Hill Interamericana de España S.L.

Notación de caja-flecha. (s.f.). Descargado de https://www.eskimo.com/~scs/cclass/int/sx8.html

Paul Deitel, H. D. (2016). *C how to program. with an introduction to c++* (Global Edition ed.). Pearson International.

Pecinovsky, R. (2013). *Oop - learn object oriented thinking and programming*. Tomas Bruckner.