ELECTRÓNICA DIGITAL III

CLASE 1-A

Introducción al Lenguaje C

Ing. Emiliano Migliore

UNIVERSIDAD NACIONAL DE CÓRDOBA - Facultad de Ciencias Exactas, Físicas y Naturales

2025

Introducción al Lenguaje C: Lenguajes de Programación

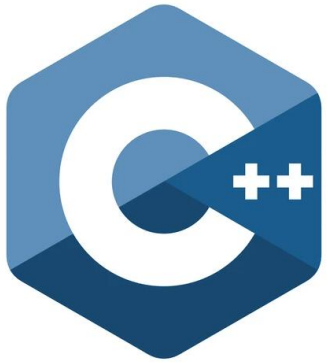
Los procesadores digitales comprenden instrucciones representadas en forma de **unos y ceros**, es decir, en **lenguaje de máquina**. Para facilitar la programación, existen distintos niveles de lenguajes que se usan como intermediarios entre el ser humano y el hardware:

Lenguajes de programación de bajo nivel:

- **Lenguaje de máquina (Binario):** Se compone exclusivamente de secuencias de bits (1 y 0) que la CPU interpreta directamente para ejecutar operaciones. Es difícil de leer y escribir para los humanos, pero es el único lenguaje que entiende el hardware sin intermediarios.
- **Ensamblador (ASM):** Usa códigos mnemotécnicos que representan instrucciones básicas del procesador. Para que el procesador las ejecute, un ensamblador convierte estas instrucciones en código binario.

Lenguajes de programación de alto nivel: Permiten escribir programas más comprensibles para los humanos, y son independientes del hardware específico. Estos lenguajes ofrecen estructuras más abstractas y legibles, facilitando el desarrollo de software complejo.

- **C**
- **C++**
- **Java**
- **Python**


Introducción al Lenguaje C: Tipo de Datos Primitivos

Los **datos primitivos** son aquellos que están definidos por el compilador. Se definen como un conjunto de n bits de longitud con o sin representación en complemento a 2 (C_2):

bool: $n = 1 \Rightarrow 2^1 = 2 \equiv [0\ a\ (2^1 - 1)]$

char: $n = 8\ \&\ signed\ \&\ \mathbb{L} \Rightarrow 2^8 = 256 \equiv \left[-\frac{2^8}{2}\ a\ \left(\frac{2^8}{2} - 1\right)\right]$

unsigned char: $n = 8\ \&\ unsigned\ \&\ \mathbb{L} \Rightarrow 2^8 = 256 \equiv [0\ a\ (2^8 - 1)]$

short: $n = 16\ \&\ signed\ \&\ \mathbb{Z} \Rightarrow 2^{16} = 65536 \equiv \left[-\frac{2^{16}}{2}\ a\ \left(\frac{2^{16}}{2} - 1\right)\right]$

unsigned short: $n = 16\ \&\ unsigned\ \&\ \mathbb{Z} \Rightarrow 2^{16} = 65536 \equiv [0\ a\ (2^{16} - 1)]$

int: $n = 32\ \&\ signed\ \&\ \mathbb{Z} \Rightarrow 2^{32} = 4294967296 \equiv \left[-\frac{2^{32}}{2}\ a\ \left(\frac{2^{32}}{2} - 1\right)\right]$

unsigned int: $n = 32\ \&\ unsigned\ \&\ \mathbb{Z} \Rightarrow 2^{32} = 4294967296 \equiv [0\ a\ (2^{32} - 1)]$

Dependiendo de la arquitectura del procesador, la longitud de un int podría ser de 32 o 16 bits.

long: $n = 32\ \&\ signed\ \&\ \mathbb{Z} \Rightarrow 2^{32} = 4294967296 \equiv \left[-\frac{2^{32}}{2}\ a\ \left(\frac{2^{32}}{2} - 1\right)\right]$

unsigned long: $n = 32\ \&\ unsigned\ \&\ \mathbb{Z} \Rightarrow 2^{32} = 4294967296 \equiv [0\ a\ (2^{32} - 1)]$

Dependiendo de la arquitectura del procesador, la longitud de un int podría ser de 32 o 64 bits.

float: $n = 32\ en\ IEEE\ 754\ \&\ \mathbb{R}\ (1\ signo\ + 8\ exponente\ + 23\ mantisa\ )\ \&\ \mathbb{R} \Rightarrow [\mp 3.4x10^{38}]$

double: $n = 64\ en\ IEEE\ 754\ \&\ \mathbb{R}\ (1\ signo\ + 11\ exponente\ + 52\ mantisa\ )\&\ \mathbb{R} \Rightarrow [\mp 1.7x10^{308}]$

long double: $n = 80\ en\ IEEE\ 754\ \&\ \mathbb{R}\ (1\ signo\ + 15\ exponente\ + 64\ mantisa\ )\&\ \mathbb{R} \Rightarrow [\mp 1.18973x10^{4932}]$

Introducción al Lenguaje C: Tipo de Datos Primitivos

```c
#include <stdio.h>

int main() {
    printf("sizeof(char)   = %zu byte\n", sizeof(char));
    printf("sizeof(int)    = %zu byte\n", sizeof(int));
    printf("sizeof(float)  = %zu byte\n", sizeof(float));
    printf("sizeof(double) = %zu byte\n", sizeof(double));
    return 0;
}
```

OUTPUT

```
sizeof(char)   = 1 byte
sizeof(int)    = 4 byte
sizeof(float)  = 4 byte
sizeof(double) = 8 byte
```

Introducción al Lenguaje C: Tipo de Datos Definidos

La palabra reservada ***typedef*** declara nuevos nombres de tipos de datos (alias), es decir, sinónimos de otro tipo ya sean primitivos o derivados, los cuales pueden ser utilizados para declarar variables de dichos tipos. Las declaraciones ***typedef*** permiten parametrizar el programa para evitar problemas de portabilidad.

typedef dataType nameVariable;

Las siguientes definiciones se encuentran ubicadas en la libreria standard de C **stdint.h**:

Definición de Tipo de Dato	Declaración de Variable
typedef ***signed char*** **int8_t**;	⟶ int8_t var_x; ≡ signed char
typedef ***unsigned char*** **uint8_t**;	⟶ uint8_t var_x; ≡ unsigned char
typedef ***signed short*** **int16_t**;	⟶ int16_t var_x; ≡ signed short
typedef ***unsigned short*** **uint16_t**;	⟶ uint16_t var_x; ≡ unsigned short
typedef ***signed int*** **int32_t**;	⟶ int32_t var_x; ≡ signed int
typedef ***unsigned int*** **uint32_t**;	⟶ uint32_t var_x; ≡ unsigned int
typedef ***signed long long*** **int64_t**;	⟶ int64_t var_x; ≡ signed long long
typedef ***unsigned long long*** **uint64_t**;	⟶ uint64_t var_x; ≡ unsigned long long

Introducción al Lenguaje C: Tipo de Datos Definidos

```c
#include <stdio.h>

typedef unsigned char uint8_t;
typedef unsigned int uint32_t;


int main() {
    printf("sizeof(uint8_t)   = %zu byte\n", sizeof(uint8_t));
    printf("sizeof(uint32_t)  = %zu byte\n", sizeof(uint32_t));

    return 0;
}
```

OUTPUT

```
sizeof(uint8_t)   = 1 byte
sizeof(uint32_t)  = 4 byte
```

Introducción al Lenguaje C: Operadores

Operadores Aritméticos:

+ (suma): contempla operandos $\mathbb{Z}, \mathbb{R}, \mathbb{C}$.
- (resta): contempla operandos $\mathbb{Z}, \mathbb{R}, \mathbb{C}$.
*** (multiplicación):** contempla operandos $\mathbb{Z}, \mathbb{R}, \mathbb{C}$.
/ (división): contempla operandos $\mathbb{Z}, \mathbb{R}, \mathbb{C}$.
% (módulo): contempla operandos $\mathbb{Z}$.

Operadores Relacionales:

< (menor): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
> (mayor): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
<= (menor o igual): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
>= (mayor o igual): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
!= (distinto): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
== (igual): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.

Operadores Lógicos:

&& (AND): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
|| (OR): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.
! (NOT): contempla resultados $\mathbb{Z}$ $(true = 1\ o\ false = 0)$.

Introducción al Lenguaje C: Operadores

Operadores a Nivel de Bits:

& (AND sobre bits): contempla operandos $\mathbb{Z}$.
| (OR sobre bits): contempla operandos $\mathbb{Z}$.
^ (XOR sobre bits): contempla operandos $\mathbb{Z}$.
<< (Desplazamiento a la Izquierda sobre bits): contempla operandos $\mathbb{Z}$.
>> (Desplazamiento a la Derecha sobre bits): contempla operandos $\mathbb{Z}$.

Operadores de Asignación Simple:

++ (Incremento): variable_x++ ≡ ++ variable_x ≡ variable_x + 1
-- (Decremento): variable_x-- ≡ -- variable_x ≡ variable_x – 1
+= (Suma más Asignación): variable_x += variable_y ≡ variable_x = variable_x + variable_y
-= (Resta más Asignación): variable_x -= variable_y ≡ variable_x = variable_x - variable_y
***= (Multiplicación más Asignación):** variable_x *= variable_y ≡ variable_x = variable_x * variable_y
/= (División más Asignación): variable_x /= variable_y ≡ variable_x = variable_x / variable_y
%= (Módulo más Asignación): variable_x %= variable_y ≡ variable_x = variable_x % variable_y
<<= (Desplazamiento a la Izquierda más Asignación): variable_x <<= variable_y ≡ variable_x = variable_x << variable_y
>>= (Desplazamiento a la Derecha más Asignación): variable_x <<= variable_y ≡ variable_x = variable_x << variable_y
&= (AND sobre bits más Asignación): variable_x &= variable_y ≡ variable_x = variable_x & variable_y
|= (OR sobre bits más Asignación): variable_x |= variable_y ≡ variable_x = variable_x | variable_y
^= (XOR sobre bits más Asignación): variable_x ^= variable_y ≡ variable_x = variable_x ^ variable_y

Introducción al Lenguaje C: Operadores

Operador de Tamaño:
sizeof (Tamaño en Bytes): contempla resultados en bytes del operando.

Operador de Dirección:
& (Dirección de): contempla como resultado la dirección en memoria del operando.

Operador de Indirección:
*** (Indirección de):** contempla como resultado el contenido de la dirección en memoria del operando.

Introducción al Lenguaje C: Cualificadores

La palabra reservada **const** declara un cualificador, e indica que el código no debe cambiar el valor de la variable declarada. Cuando defines una variable constante, se reserva memoria para ella y se garantiza que su valor no cambie durante la ejecución del programa.

const dataType NAME_VARIABLE = value;

```c
#include <stdio.h>

int main() {
    const float PI = 3.14159f;      // PI es constante
    float radio = 5.0f;

    // Cálculo del área del círculo usando la constante PI
    float area = PI * radio * radio;

    printf("Radio = %.2f\n", radio);
    printf("Área del círculo = %.2f\n", area);

    // PI = 3.14f;  // Descomenta: ERROR de compilación, PI es de sólo lectura

    return 0;
}
```

OUTPUT

```
Radio = 5.00
Área del círculo = 78.54
```

OUTPUT

```
./Playground/file0.c: In function 'main':
./Playground/file0.c:13:9: error: assignment of read-only variable 'PI'
   13 |       PI = 3.14f;
      |          ^
```

Introducción al Lenguaje C: Constantes Simbólicas y Macros

La directriz de sustitución ***#define*** sirve para declarar constantes simbólicas mediante un identificador que representa una constante. Cuando se define una constante con #define, no se reserva memoria para ella. En su lugar, en cada lugar donde se utiliza esa constante, el preprocesador simplemente reemplaza el identificador **CONST** de la constante por su valor **value** en el código fuente antes de que se compile.

#define CONST value

La directriz de sustitución ***#define*** también sirve para declarar macros mediante un identificador que representa sentencias o expresiones.

#define MACRO (expresion)

```c
#include <stdio.h>

// Constante simbólica
#define PI 3.14159

// Macro para calcular el cuadrado de x
#define SQUARE(x)   ((x) * (x))

int main() {
    float radio = 2.5f;
    float area = PI * SQUARE(radio);

    printf("Radio  = %.2f\n", radio);
    printf("Área   = %.2f\n", area);

    return 0;
}
```

OUTPUT

```
Radio  = 2.50
Área   = 19.63
```

Introducción al Lenguaje C: Funciones

Declaración de Prototipo de Funciones

El **prototipo de una función** es una plantilla que se utiliza para asegurar que una sentencia de invocación escrita antes de la definición de la función es correcta; esto es, que son pasador los argumentos adecuados para los parámetros especificados en la función y que el valor retornado se trata correctamente.
En el prototipo de función no hace falta los identificadores, solo los tipos de los parámetros.
Para indicar que la función no retorna nada, o no presenta parámetros, se utiliza la palabra reservada **void**.

Definición de Funciones

La ***definición explicita* de una función** específica el número y el tipo de los parámetros de la función, así como el valor retornado. Los identificadores utilizados en la lista de parámetros de la declaración del prototipo de función, no necesariamente tiene que nombrarse igual que los identificadores en la lista de parámetros en la definición de la función.

Llamada de Funciones

La ***llamada* o *ejecución*** de una función se realiza desde otra función, o incluso desde ella misma.
Una función puede retornar cualquier valor de un tipo primitivo o estructurado, excepto un arreglo (no dinámico) o una función.

Introducción al Lenguaje C: Transmisión por Valor y Referencia

Transmisión por Valor:

La **transmisión por valor** implica que la función llamada recibe valores de la función que llama, almacena y manipula los valores transmitidos y devuelve en forma directa cuando mucho un valor único. En ningún momento a función llamada tiene acceso directo a cualquier variable definida en la función que llama, aun si la variable se usa como un argumento en la llamada a la función.

El resultado de una función es devuelto por medio de la sentencia **return** al punto donde se realizó la llamada. Esta sentencia puede ser o no la última y puede aparecer más de una vez en el cuerpo de la función; cuando se ejecuta se da por finalizada la ejecución de la función. En caso de que la función no retorne un valor (void), se puede especificar simplemente return, o bien, si se trata de la última sentencia, omitir.

Por defecto, todos los argumentos de la función, excepto las matrices, son *transmitidos por valor.* Esto significa que a la función se le pasa una copia del valor del argumento, para evitar alterar las variables de donde proceden los valores pasados.

Transmisión por Referencia:

La **transmisión por referencia** implica darle a la función llamada acceso directo a las variables de su función que llama. Esto le permite a la función, la cual es la función llamada, usen y cambien el valor de variables que se han definido en la función que llama. Para hacer esto se requiere que la dirección de la variable se transmita a la función llamada.

Introducción al Lenguaje C: Transmisión por Valor y Referencia

```c
#include <stdio.h>

// Prototipos de Funciones
void funcForValue(int x);
void funcForReference(int *x);

int main() {
    int a = 10;
    int b = 10;

    // Llamada de Funciones
    // Transmisión por valor
    funcForValue(a);
    printf("Después de funcForValue: a = %d\n", a);  // No cambia

    // Transmisión por referencia
    funcForReference(&b);
    printf("Después de funcForReference: b = %d\n", b);  // Sí cambia

    return 0;
}

// Definición de Funciones
void funcForValue(int x) {
    x = x + 5;
    printf("[funcForValue] x = %d\n", x);
}

void funcForReference(int *x) {
    *x = *x + 5;
    printf("[funcForReference] *x = %d\n", *x);
}
```

OUTPUT

```
[funcForValue] x = 15
Después de funcForValue: a = 10
[funcForReference] *x = 15
Después de funcForReference: b = 15
```

Introducción al Lenguaje C: Variables Globales

Variables Globales

Se declaran fuera de las funciones, y son accesibles desde todo el archivo donde se declaran.

```c
#include <stdio.h>

int globalVar = 100;  // Variable global

void printGlobal() {
    printf("Global: %d\n", globalVar);
}

int main() {
    printGlobal();
    globalVar = 200;
    printGlobal();
    return 0;
}
```

OUTPUT

```
Global: 100
Global: 200
```

Variables Globales Estáticas

También global, pero solo visible en el archivo donde se declara. Otras unidades de código no la pueden usar, aunque se llame igual.

```c
// En sourceFile1.c
static int globalVar = 100;

// En sourceFile2.c
// No puedes usar globalVar aquí.
```

Variables Globales Externas

Permite usar variables globales de otros archivos.

```c
// En sourceFile1.c
int globalVar = 100;

// En sourceFile2.c
extern int globalVar;
void useVar() {
    printf("%d", globalVar);
}
```

Variables Globales Volátiles

Su valor puede cambiar en cualquier momento, como por una interrupción o hardware externo. El compilador no optimiza su lectura.

```c
volatile int flag = 0;

void ISR() {
    flag = 1;  // Cambia en una interrupción
}
```

Introducción al Lenguaje C: Variables Locales

Variables Locales

Se declaran dentro de funciones o bloques, y solo son accesibles dentro de ese bloque.

```c
#include <stdio.h>

int main() {
    int localVar = 10;
    printf("Local: %d\n", localVar);
    return 0;
}
```

OUTPUT

```
Local: 10
```

Variables Locales Estáticas

Se mantiene entre ejecuciones del bloque, y no se reinicia cada vez.

```c
#include <stdio.h>

void countCalls() {
    static int counter = 0;
    counter++;
    printf("Call #%d\n", counter);
}

int main() {
    countCalls();
    countCalls();
    countCalls();
    return 0;
}
```

OUTPUT

```
Call #1
Call #2
Call #3
```

Introducción al Lenguaje C: Estructuras de Control

Estructura if-else

La estructura de selección **if-else** permite tomar una decisión para ejecutar una sentencia u otra, basándose en el resultado booleano de una condición.

```
if (condición 1) {
    sentencia 1;
}
else {
    sentencia 2;
}
```

La estructura de selección **if-else anidada** permite tomar una decisión para ejecutar una sentencia u otra decisión, basándose en el resultado booleano de una condición.

```
if (condición 1){
    sentencia 1;
}
else if (condición 2){
    sentencia 2;
}
   ⋮
else {
    sentencia n;
}
```

Introducción al Lenguaje C: Estructuras de Control

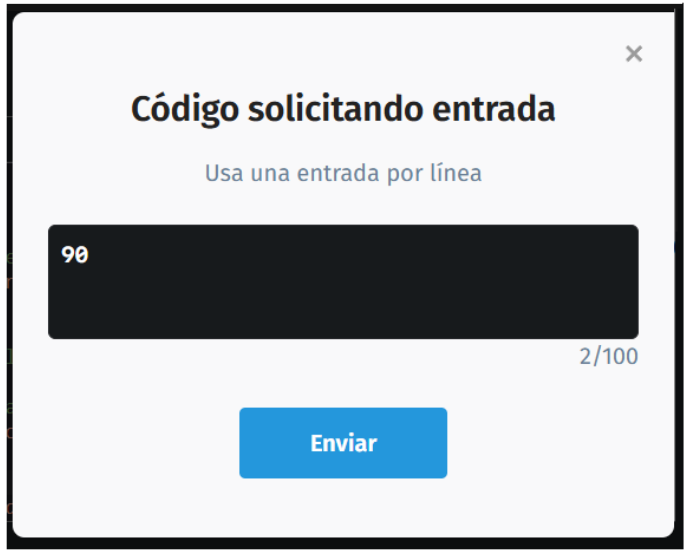
```c
#include <stdio.h>

int main() {
    int nota;

    // Pedimos al usuario que ingrese una nota
    printf("Ingrese la nota del alumno (0 a 100): ");
    scanf("%d", &nota);

    // Evaluamos la nota usando if-else
    if (nota >= 60) {
        // Si la nota es mayor o igual a 60, aprueba
        printf("El alumno ha aprobado.\n");
    } else {
        // Si la nota es menor a 60, no aprueba
        printf("El alumno ha reprobado.\n");
    }

    return 0;
}
```



OUTPUT

```
Ingrese la nota del alumno (0 a 100): El alumno ha aprobado.
```

Introducción al Lenguaje C: Estructuras de Control

Estructura switch

La estructura de selección **switch** permite ejecutar una de varias sentencias, en función del valor de una condición.

```
switch (condición) {
    case n_1:
        sentencia 1;
        break;
    case n_2:
        sentencia 2;
        break;
            ⋮
    default:
        sentencia n;
        break;
}
```

Introducción al Lenguaje C: Estructuras de Control

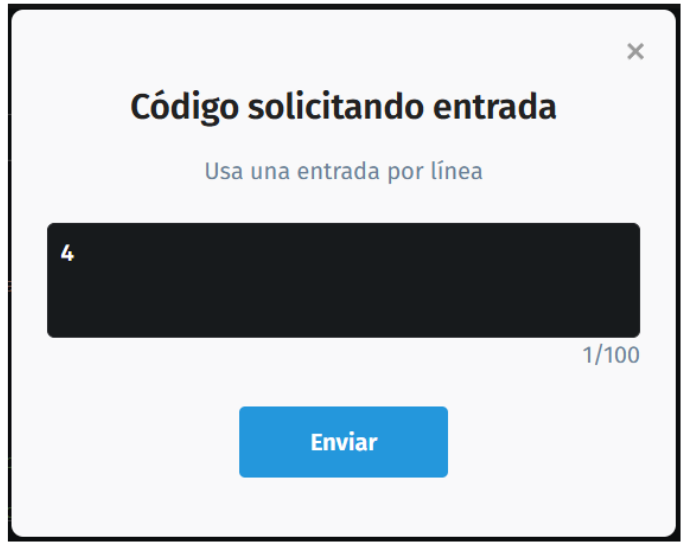
```c
#include <stdio.h>

int main() {
    int dia;

    // Mostrar opciones al usuario
    printf("Ingresa un número del 1 al 5 para saber el día:\n");
    printf("1. Lunes\n");
    printf("2. Martes\n");
    printf("3. Miércoles\n");
    printf("4. Jueves\n");
    printf("5. Viernes\n");
    printf("Tu elección: ");
    scanf("%d", &dia);  // Leer la opción ingresada

    // Evaluar la opción usando switch-case
    switch (dia) {
        case 1:
            printf("Hoy es Lunes.\n");
            break;
        case 2:
            printf("Hoy es Martes.\n");
            break;
        case 3:
            printf("Hoy es Miércoles.\n");
            break;
        case 4:
            printf("Hoy es Jueves.\n");
            break;
        case 5:
            printf("Hoy es Viernes.\n");
            break;
        default:
            // Si no se ingresa un número entre 1 y 5
            printf("Opción inválida.\n");
            break;
    }

    return 0;
```



OUTPUT

```
Ingresa un número del 1 al 5 para saber el día:
1. Lunes
2. Martes
3. Miércoles
4. Jueves
5. Viernes
Tu elección: Hoy es Jueves.
```

Introducción al Lenguaje C: Estructuras de Control

Estructura for

La estructura de repetición **for** permite ejecutar una sentencia, repetidamente un número de veces n.

for (int i = n_{init}; i <= n_{end}; i++) {
**   sentencia;**
}

//Repetición con incremento unitario

```c
#include <stdio.h>

int main() {
    // Declaramos una variable de control i
    int i;

    // Estructura for: desde i = 1 hasta i <= 10,
    //incrementando i de uno en uno
    for (i = 1; i <= 10; i++) {
        printf("Número: %d\n", i);
    }

    return 0;
}

```

OUTPUT

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
Número: 6
Número: 7
Número: 8
Número: 9
Número: 10
```

Introducción al Lenguaje C: Estructuras de Control

Estructura while

La estructura de repetición **while** permite ejecutar una sentencia, cero o más veces, dependiendo del resultado numérico de una condición.

```
while (condición) {
    sentencia;
}
```

```c
#include <stdio.h>

int main() {
    int contador = 1;  // Variable de control

    // Mientras contador sea menor o igual a 5, ejecuta el bloque
    while (contador <= 5) {
        printf("Contador: %d\n", contador);
        contador++;  // Incrementa el valor en 1
    }

    return 0;
}
```

OUTPUT

```
Contador: 1
Contador: 2
Contador: 3
Contador: 4
Contador: 5
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

Enumeraciones

Un enumerado **enum** representa una lista de valores numéricos secuenciales $\in \mathbb{Z}^+$ que pueden ser tomador por una variable del tipo enumerado. Los tipos enumerados solo admiten una cantidad finita de posibles valores.

La **definición** de una variable de tipo enumerado tiene la siguiente sintaxis:

```
enum nameEnum
{
        variable_1,          // variable_1 = 0
        ⋮
        variable_n           // variable_n = n-1
};
```

La **declaración** de una variable de tipo enumerado tiene la siguiente sintaxis:

```
enum nameEnum nameVariable;
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

```c
#include <stdio.h>

// Enumeración
enum DiaSemana {
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES,
    SABADO,
    DOMINGO
};

int main() {
    enum DiaSemana hoy = JUEVES;

    if (hoy == JUEVES) {
        printf("Hoy es jueves: se cursa EDIII\n");
    }
    else {
        printf("Me pongo a estudiar con tiempo\n");
    }

    return 0;
}
```

```
OUTPUT

Hoy es jueves: se cursa EDIII
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

Arreglos

Un **arreglo estático** implica que el compilador es el encargado de realizar el recuento de memoria necesaria para almacenar sus elementos, y la mantiene constante durante la ejecución del programa; por lo cual se requiere que la dimensión del arreglo sea expresada como una contante entera en el código.

```c
#include <stdio.h>

int main() {
    // Arreglo estático de 5 enteros
    int dataArray[5] = {10, 20, 30, 40, 50};

    printf("Contenido del arreglo estático:\n");

    for (int i = 0; i < 5; i++) {
        printf("elemento[%d] = %d\n", i, dataArray[i]);
    }

    return 0;
}
```

OUTPUT

```
Contenido del arreglo estático:
elemento[0] = 10
elemento[1] = 20
elemento[2] = 30
elemento[3] = 40
elemento[4] = 50
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

Punteros

Un **puntero** contiene la dirección de memoria de un dato o de otro objeto que contiene al dato. Es decir, el puntero apunta al espacio físico donde está el dato o el objeto en memoria. Los punteros son variables que no contienen datos sino posiciones de memoria, por tanto, permiten acceder y modificar datos de forma indirecta.

```c
#include <stdio.h>

int main() {
    int numero = 42;          // variable normal que contiene un dato
    int *puntero = NULL;      // puntero que apunta a un entero, inicializado a NULL

    puntero = &numero;        // puntero apunta a la dirección de 'numero'

    printf("Valor de numero: %d\n", numero);
    printf("Valor apuntado por el puntero: %d\n", *puntero);
    printf("Dirección de numero: %p\n", &numero);
    printf("Valor del puntero: %p\n", puntero);

    // Incremento de puntero (avanza al siguiente entero en memoria)
    puntero++; // siguiente posición de int en memoria (4 bytes adelante)
    printf("Dirección del puntero después de incrementarlo: %p\n", puntero);

    return 0;
}
```

OUTPUT

```
Valor de numero: 42
Valor apuntado por el puntero: 42
Dirección de numero: 0x7ffc71ecc534
Valor del puntero: 0x7ffc71ecc534
Dirección del puntero después de incrementarlo: 0x7ffc71ecc538
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

Estructuras

Una **estructura** permite agrupar una o más variables de diferentes tipos de datos, bajo un mismo nombre para facilitar su manejo. La declaración de un miembro de una estructura no puede contener calificadores de clase de almacenamiento como extern, static, auto, o register y no se los puede inicializar con un valor.

Una estructura, al igual que otro tipo de variables, puede ser pasado por valor o por referencia. Por valor significa que el valor almacenado (una dirección) en el parámetro actual (argumento especificado en la llamada a la función) se copia en el parámetro formal correspondiente (parámetro declarado en la cabecera de la función); si ahora modificamos el valor de este parámetro formal, el valor actual correspondiente no se verá afectado.

La palabra struct es la abreviatura de (structured data type). Los datos se acceden mediante un dot(.)

```
typedef struct
{
        dataType variable_1;
                ⋮
        dataType variable_n;
} nameStruct;
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

```c
#include <stdio.h>

// Definición de la estructura con typedef
typedef struct {
    char nombre[50];
    int nota;
} Alumno;

// Función que recibe un Alumno por valor (copia)
void mostrarPorValor(Alumno p) {
    printf("Mostrar por valor:\n");
    printf("Nombre: %s\n", p.nombre);
    printf("Nota: %d\n", p.nota);
    // Cambiamos la nota, pero no afecta el original
    p.nota = 0;
}

// Función que recibe un puntero a Alumno (por referencia)
void mostrarPorReferencia(Alumno *p) {
    printf("Mostrar por referencia:\n");
    printf("Nombre: %s\n", p->nombre);
    printf("Nota: %d\n", p->nota);
    // Cambiamos la nota, afecta al original
    p->nota = 100;
}

int main() {
    // Declaramos una variable de tipo Alumno sin usar 'struct'
    Alumno alumno1 = {"Juan", 90};

    printf("Datos originales:\n");
    printf("Nombre: %s\n", alumno1.nombre);
    printf("Nota: %d\n\n", alumno1.nota);

    mostrarPorValor(alumno1);
    printf("Después de mostrarPorValor, nota: %d\n\n", alumno1.nota);

    mostrarPorReferencia(&alumno1);
    printf("Después de mostrarPorReferencia, nota: %d\n", alumno1.nota);

    return 0;
}
```

OUTPUT

```
Datos originales:
Nombre: Juan
Nota: 90

Mostrar por valor:
Nombre: Juan
Nota: 90
Después de mostrarPorValor, nota: 90

Mostrar por referencia:
Nombre: Juan
Nota: 90
Después de mostrarPorReferencia, nota: 100
```

Introducción al Lenguaje C: Tipos de Datos Estructurados

Uniones

Una **unión** es un tipo de dato compuesto que permite almacenar diferentes tipos de datos en una misma zona de memoria, pero solo uno de ellos a la vez. Todos los miembros de una unión comparten el mismo espacio de memoria, que es igual al tamaño del miembro más grande. Esto permite reutilizar memoria eficientemente cuando una variable puede tomar diferentes formas o tipos, pero nunca al mismo tiempo.

- *Diferencia con struct:* A diferencia de una estructura, donde cada miembro tiene su propia memoria, en una unión todos los miembros se solapan en la misma ubicación de memoria.

- *Uso típico:* Útil para optimizar el uso de memoria en sistemas embebidos o cuando un dato puede variar de tipo en distintos momentos, pero no necesita almacenarlos simultáneamente.

```
union
{
        dataType var_1;
                ⋮
        dataType var_n;
} nameUnion;
```