



# Resumen TDL - Mod I

## Características Importantes de C

- Es un lenguaje muy flexible que soporta la programación estructurada (permitiendo ciertas licencias de ruptura).
- Usa un lenguaje de preprocesado con posibilidades para definir macros e incluir múltiples archivos de código fuente.
- Acceso a memoria de bajo nivel mediante el uso de punteros.
- Pasaje de parámetros **por valor**.
- Tipos de datos agregados (struct) equivalentes a los registros de Pascal.

## Orden de operaciones

- `()`: Se calculan primero (izq a der).
- `*`, `/`, `%`: Se evalúan en 2do lugar (izq a der).
- `+`, `-`: Se calculan al final (izq a der).

## Operadores Logicos

- `&&` AND
- `||` OR
- `!` NOT

## Sizeof(Var)

- `Char` 1 byte.
- `Int` 4 bytes.
- `Float` 4 bytes.
- `Double` 8 bytes.

## Tipo Real

```
float: %f. sizeof(4)
float: %e. sizeof(4)
double: %lf sizeof(8)
double: %le sizeof(8)
```

## Tipo INT

- `short int`: -32.768 a 32767 (2 bytes).
- `unsigned short int`: positivos < 65.535 (2 bytes).
- `unsigned int`: naturales (4 bytes).

## Dato logico

En C no existe el tipo de dato logico. Para ello se utilizan numeros en representacion del mismo.

```
/*Ejemplo de conversion para dividir*/
int suma = 10;
int divisor = 4;
float res = (float) suma/divisor;
printf("Resultado: %f", res);
```

## Estructura de selección

```
if (condición != 0) { /* Acción bloque de acciones a realizarse si la condición es verdadera */
}
```

```
else { /* Acción bloque de acciones a realizarse si la condición es falsa */
}
```

## Estructura iterativa

```
while(condición) /* acción bloque de acciones a realizarse mientras la condición sea verdadera */
```

## Estructura de repetición

```
for (inicialización; condición; acciones_posteriores) {
/* acción o bloque de acciones pertenecientes al cuerpo del for */
}
```

- **inicialización:** es una acción o una secuencia de acciones separadas por comas que se ejecuta *ANTES* de iniciar el for.
- **condición:** es una expresión lógica cuyo valor se evalúa *ANTES* de iniciar el for y debe ser verdadera para que el for se ejecute. Que la condición sea verdadera quiere decir que sea  $\neq 0$
- **acciones\_posteriores:** es una o más acciones separadas por comas que se ejecutan *LUEGO* de las instrucciones del for.

## Operador Ternario

```
Expresión lógica ? valor1 : valor2
/*Evalúa la expresión y si es !=0 devuelve valor1, sino devuelve valor2.*/
```

## Sentencia switch

```
switch (variable) {
case valor1:
/* acción o acciones a realizar*/
break;
case valor2:
/* acción o acciones a realizar*/
break;
...
default:
/* acción o acciones por defecto*/
}
```

## Sentencia do-while

```
do
/* acción o bloque de acciones*/
while (cond)
```

## Operadores de asignación

```
var = var + x --> var += x
var = var - x --> var -= x
var = var * x --> var *= x
var = var / x --> var /= x
var = var % x --> var %= x
```

## Operadores de incremento

```
++ ++a /*Se incrementa a en 1 y luego se utiliza el nuevo valor de a en la expresión
++ a++ /*Utilizar el valor actual de a en la expresión en la cual reside a y después
-- --b /*Se decrementa b en 1 y a continuación se utiliza el nuevo valor de b en la
-- b-- /*Se utiliza el valor actual de b en la expresión en la cual reside b y después
```

## Break y continue

- **break:** al ejecutarla, la iteración termina y la ejecución del programa continúa en la próxima línea a la estructura iterativa.
- **continue:** al ejecutarla se saltan las instrucciones que siguen hasta terminar la iteración actual y el loop continúa por la siguiente iteración.

## Funciones

```
TipoValorRetornado NombreFuncion(TipoParametros) /*Prototipo*/
int main(){
    printf("%TipoValorRetornado"NombreFuncion);
}
TipoValorRetornado NombreFuncion(parametros){
    static vec[10]={}; /*hace que no se pierda el contenido de esta variable cada vez que se ingresa al modulo*/
    /*Acciones a ejecutar*/
    return expresion; /*opcional*/
}
```

## Define vs const

- **#define** TEXTO constante
- **const** tipo\_var var
- #define es una directiva para el precompilador que reemplaza el identificador por el texto correspondiente **ANTES** de compilar.
- La palabra clave const evita que el nombre de la variable se modifique en su alcance. Este chequeo se hace en la compilación.

```
#define TRUE 1
#define FALSE 0

int main( ) {
    const int valor = 1;
    if(valor == 1){
        printf( "Value of TRUE : %d\n", TRUE);
    }
    else printf( "Value of FALSE : %d\n", FALSE);
    return 0;
}
```

## Vectores

```
tipo_base nombre[TEXT0];
tipo_base nombre[cant_elem] = {0};
tipo_base nombre[cant_elem] = {valor1,valor2,...,valorN};
tipo_base nombre[] = {1,2,3};

/*Para mandar un vector a una funcion*/
void reciboVector(nombre,cant_elem)
//Cualquier cambio que se produzca se vera reflejado en el main
```

## Matrices

```
int matriz1[2][3] = {{1,2,3}, {4,5,6}};
int matriz2[2][3] = {1,2,4,6};
int matriz3[2][3] = {{1,2}, {6}};

//Para mandar una matriz a una funcion
void reciboMatriz(matriz[][3], int filas)
```

## Vector de caracteres

```
charpalabra[] = "Ejemplo";
charpalabra2[8] = { 'E', 'j', 'e', 'm', 'p', 'l', 'o', '\0' };
printf("%s",charpalabra2); //%s va hasta el primer blanco
printf("%c",charpalabra2[i]);
```

## Funciones para cadenas de caracteres

- **strlen(c1):** Retorna el número de chars hasta el carácter nulo (el cual no se incluye)

- **strcpy(c1,c2):** Copia la cadena c2 en la cadena c1. La cadena c1 debe ser lo suficientemente grande para almacenar la cadena c2 y su caracter nulo (que tambien se copia).
- **strcat(c1,c2):** Agrega la cadena c2 al arreglo c1.
- **strcmp(c1,c2):** Compara c1 con c2 y devuelve. (resta longitudes)

## Punteros

- Permiten simular el pasaje de parámetros por referencia.
- Permiten crear y manipular estructuras de datos dinámicas.
- Un puntero es una variable que contiene una dirección de memoria.
- Por lo general, una variable contiene un valor y un puntero a ella contiene la dirección de dicha variable.

```
int *nomPtr, num /*lo primero es un puntero a un entero*/
nomPtr = &num; /*guardo la dir de memoria de num*/
printf("%d", *nomPtr);

tipo_dato * const p = "X" //el valor de P ya no puede cambiar
const char *p = "X" //el puntero puede señalar otra direccion de memoria
```



Desde C no es posible indicar numéricamente una dirección de memoria para guardar información (esto se hace a través de funciones específicas).

## Pasaje de parametros por referencia utilizando punteros

```
void cuadrado(int *);
int main(){
    cuadrado(&a);
}
void cuadrado(int * nro);
*nro *= *nro;
}

void cuadrado(const int * nro) //no permite cambiar el valor del dato
```

## Uso de arreglos como punteros

```
int b[]={1, 2, 3, 4, 5}, *bPtr;
bPtr=&b[0];
printf("b[3]= %d o %d",
      b[3], bPtr[3]);
```

Los punteros pueden tener subíndices como los arreglos.

```
int b[]={1, 2, 3, 4, 5}, *bPtr;
bPtr=&b[0];
printf("b[3]= %d o %d",
      b[3], *(b + 3));
```

El mismo arreglo puede ser tratado como un puntero y utilizado en aritmética de punteros.

## Uso de matrices como punteros

### MATRICES Y PUNTEROS

- Si la matriz se declara de la siguiente forma

```
int nros[5][15];
```

sus elementos se almacenarán en forma consecutiva por filas.

- Por lo tanto, puede accederse a sus elementos utilizando
  - `nros[fila][col]`
  - `*(nros + (15 * fila) + col)`

- Una función que espera recibir como parámetro una matriz declarada de la siguiente forma

```
int nros[5][15];
```

puede utilizar cualquiera de las siguientes notaciones

```
function F (int M[ ][15], int FIL)
```

```
function F (int *M, int FIL, int COL)
```

## Punteros void

- Es un puntero generico que puede recibir el valor de cualquier otro puntero, incluso NULL.

```
#include <stdio.h>
int main () {
    int x = 1;
    float r = 1.0;
    void* vptr = &x;

    *(int *) vptr = 2;
    printf("x = %d\n", x);

    vptr = &r;
    *(float *)vptr = 1.1;
    printf("r = %1.1f\n", r);
}
```

Un puntero a void puede recibir el valor de cualquier tipo de puntero

```
#include <stdio.h>
int main () {
    int x = 1;
    float r = 1.0;
    void* vptr = &x;

    *(int *) vptr = 2;
    printf("x = %d\n", x);

    vptr = &r;
    *(float *)vptr = 1.1;
    printf("r = %1.1f\n", r);
}
```

Un puntero a void no puede ser desreferenciado, sin ser convertido previamente

## Estructuras

Son equivalentes a los registros en pascal

```
struct Nom_Tipo{
    tipo_campo_1  nom_campo1;
    tipo_campo_2  nom_campo1;
    ...
    tipo_campo_nnom_campo1;
};

/*Opcional: declara nombre variable ahi mismo*/
struct Nom_Tipo{
    tipo_campo_1  nom_campo1;
    tipo_campo_2  nom_campo1;
    ...
    tipo_campo_nnom_campo1;
} array[TOTAL];
struct Nom_Tipo var = {valor1,valor2};
```

## Operaciones

- Asignar variables de estructura a variables de estructuras del mismo tipo.
- Obtener la dirección de una variable estructura mediante el operador &.
- Acceder a los elementos de la estructura.
- Las estructuras no pueden compararse entre sí porque sus campos no necesariamente están almacenados en posiciones consecutivas de memoria. Puede haber "huecos".

## Typedef

```
typedef struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;

int main( ) {
    Book book;
```

## Tipo Union

Al igual que una estructura, una unión también es un tipo de dato compuesto heterogéneo, pero con miembros que comparten el mismo espacio de almacenamiento.

```
union numero {
    int x;
    double y;
};
union numero valor = {10}; //Correcto ya que x, es el primer campo y es
union numero valor = {10.234}; //El valor asignado se trunca en 10, por
```

```
union numero {
    int x;
    double y;
};
union numero valor;

valor.x = 10.34;
printf("Asigno 10.34 en el campo (int)\n");
printf("    (int) valor.x = %d\n", valor.x);
printf("(double) valor.y = %f\n", valor.y);

valor.y = 10.34;
printf("Asigno 10.34 en el campo (double)\n");
printf("    (int) valor.x = %d\n", valor.x);
printf("(double) valor.y = %f\n", valor.y);
```

Asigno 10.34 en el campo (int)  
(int) valor.x = 10  
(double) valor.y = 0.000000

Asigno 10.34 en el campo (double)  
(int) valor.x = 2861584382  
(double) valor.y = 10.340000

Así se vería el truncamiento.

## Operadores a nivel de bits

| Operador | Descripción   |
|----------|---|
| &        | AND a nivel de bits. Compara sus dos operandos bit a bit. Los bits en el resultado son setados a 1 si los bits correspondientes a <b>ambos</b> operandos valen 1.                                   |
|          | OR a nivel de bits. Compara sus dos operandos bit a bit. Los bits en el resultado son setados a 1 si <b>al menos uno</b> de los bits correspondientes a los operandos valen 1.                      |
| ^        | XOR a nivel de bits. Compara sus dos operandos bit a bit. Los bits del resultado se establecen en 1, si <b>exactamente uno</b> de los bits correspondientes a los dos operandos es 1.               |
| <<       | Desplazamiento a la izquierda. Desplaza hacia la izquierda los bits del 1er. operando, el número de bits indicados por el 2do. operando; desde la derecha completa con bits en 0.                   |
| >>       | Desplazamiento a la derecha. Desplaza hacia la derecha los bits del 1er. operando, el número de bits indicados por el 2do. operando; el método de llenado desde la izquierda depende de la máquina. |
| ~        | Complemento a uno. Todos los bits en 0 se cambian a 1 y viceversa.  |

```
int a=1, b=2, c, d;
c = a & b; //0001 & 0010 = 0000
d = 3 & b; //0011 & 0010 = 0010

printf("1&2 = %d \n", c);
printf("3&2 = %d \n", d);

int a=1, b=2, c, d;
c = a | b; //0001 | 0010 = 0011
d = 3 | b; //0011 | 0010 = 0011

printf("1|2 = %d \n", c);
printf("3|2 = %d \n", d);
```

1 & 2 = 0  
3 & 2 = 2

1 | 2 = 3  
3 | 2 = 3

```
int a=1, b=2, c, d;
c = a ^ b; //0001 ^ 0010 = 0011
d = 3 ^ b; //0011 ^ 0010 = 0001

printf("1^2 = %d \n", c);
printf("3^2 = %d \n", d);

int b=2;
unsigned char c;
c = ~b; // 00000010 = 11111101
printf("~2 = %d \n", c);
```

1 ^ 2 = 3  
3 ^ 2 = 1

~2 = 253

| Operador | Descripción   |
|----------|---|
| &=       | Operador de asignación AND a nivel de bits              |
| =        | Operador de asignación OR a nivel de bits               |
| ^=       | Operador de asignación XOR a nivel de bits              |
| <<=      | Operador de asignación de desplazamiento a la izquierda |
| >>=      | Operador de asignación de desplazamiento a la derecha   |

```
int a, b;
a = 64;
b = a >> 3; //01000000 --> 00001000
printf("64 >> 3 = %d \n", b);

int a, b;
a = 1;
b = a << 3; //0001 --> 1000
printf("1 << 3 = %d \n", b);
```

64 >> 3 = 8

1 << 3 = 8

## Campos de bits

- C permite a los programadores especificar el número de bits en el que un campo unsigned int de una estructura o unión se almacena. A esto se le conoce como campo de bits.
- Los campos de bits permiten hacer un mejor uso de la memoria almacenando los datos en el número mínimo de bits necesario.
- Los miembros de un campo de bits deben declararse como int o unsigned.
- La manipulación de los campos de bits depende de la implementación.
- Aunque los campos de bits ahorran espacio, utilizarlos puede ocasionar que el compilador genere código en lenguaje máquina de ejecución lenta.

```

struct datetime {
    unsigned int second : 6;
    unsigned int minute : 6;
    unsigned int hour : 5;
    unsigned int day : 5;
    unsigned int month : 4;
    unsigned int year : 6;
};

int main() {
    struct datetime dt = {30, 25, 11, 10, 4, 23};
    printf("sizeof(struct datetime) = %d bytes\n", sizeof(struct datetime));
    printf("Fecha y hora: %d/%d/%d %d:%d:%d\n",
        dt.day, dt.month, dt.year + 2000,
        dt.hour, dt.minute, dt.second * 2);
    return 0;
}

```

```

struct cartaBit {
    unsigned cara : 4;
    unsigned palo : 2;
    unsigned color : 1;
};

```

↑  
Cantidad de bits  
utilizados para  
almacenar el campo

## Errores comunes con el campo de bits

- Intentar acceder a bits individuales de un campo de bits, como si fueran elementos de un arreglo, es un error de sintaxis. Los campos de bits no son "arreglos de bits".
- Intentar tomar la dirección de un campo de bits (el operador & no debe utilizarse con campos de bits, ya que éstos no tienen direcciones).

## Constantes de enumeración

Una enumeración es un conjunto de constantes de enumeración enteras representadas por identificadores. Los valores de una enumeración empiezan por 0 a menos que se especifique lo contrario, y se incrementan en 1. Una constante de enumeración o un valor de tipo enumerado se pueden usar en cualquier lugar donde el lenguaje C permita una expresión de tipo entero.

```
enum meses {ENE = 1, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC};
```

- Los identificadores de una enumeración deben ser únicos (incluye también nombres de variables).
- El valor de cada constante de enumeración puede establecerse explícitamente en la definición, asignándole un valor al identificador.
- Varios miembros de una enumeración pueden tener el mismo valor constante.

## Enum vs Define

Las enumeraciones proporcionan una alternativa a la directiva de preprocesador #define con las ventajas de que los valores se pueden generar automáticamente.

```
enum boolean {false, true};
```

- Las enumeraciones siguen las reglas de alcance.
- A las variables enum se les asignan valores automáticamente.

```

#define Working 0
#define Failed 1
#define Freezed 2

```

```

enum state {Working,
            Failed,
            Freezed};

```