

# TDL Modulo 1 - (Bautista Garcia)

## Conceptos Basicos

**Detalles:** SCANF("mascara", puntero a lo que vamos a leer).

- Cuando tenemos mas de un scanf seguidos para que no interpreten la tecla **enter** tenemos dos opciones. Una es poner un espacio luego de la mascara y la otra es limpiar el buffer de datos con la siguiente sentencia.

```
fflush(stdin);
```

## Mascaras

- Las mascarar pueden llevar un numero acompañandolas(indican longitud de dato) o pueden no llevar nada(por defecto puede representar hasta **maxRep**). Acortar la longitud es util para ocasiones donde se desea **redondear**.

**Mascaras:**

Tipo	printf	scanf
long double	%Lf	%Lf
double	%lf	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short int	%hu	%hu
short int	%hd	%hd
unsigned char	%u	%u
short	%hd	%hd
char	%c	%c

## Operadores Logicos

- Cualquier numero que sea  $\neq$  de 0 se interpreta como **true**. Estos operadores retornan o un 1 o un 0, indicando **true or false** respectivamente.

Operador	Operación lógica
&&	AND
	OR
!	NOT

## Tamanios de representacion

- Operador **sizeof** (tipo int) devuelve cantidad de bytes reservados en memoria por una variable(unicamente declarada) no hace falta que se le asigne nada.
- **Representacion de numeros:** En casos donde el numero a representar es mayor al rango de representacion el compilador le resta a ese numero **maxRep** y luego el sobrante es lo que va a representar si entra en rango (sino se repite el mismo proceso hasta que pueda ser representado).

```
//Rango f 0..5
f variable = 7;
//internamente se guarda 1
```

## Estructuras Iterativas

- Instrucciones **break** y **continue**.
  - **Break:** Sale de la estructura iterativa.
  - **Continue:** Pasa a la siguiente iteracion.

# Funciones

## Prototype

Se enuncia antes del main, sin necesidad de definir la funcion. Sirve para hacer validaciones. En la parte de parametros solo enunciamos los tipos(no hace falta varName).

- Si se omite se usa primer invocacion como prototipo. Por esto va a pedir que retorne tipo de dato int(default).
- Si no coinciden tipos de prototipo con los de la definicion no compila.
- Si coinciden y pasamos como argumento un tipo de dato distinto al del prototipo se hace la **coercion**.

## Pasaje de parametros

Los pasajes de parametros son unicamente **por valor** aunque en caso de querer simular un **pasaje por referencia** lo podemos hacer con punteros.

- **Detalles P x Referencia:** En casos de arreglos(strings incluidas) no hace falta pasar punteros para leerlas ya que internamente estan representados como **punteros a la primer posicion**.

## Clases de almacenamiento

### Persistencia Automatica:

- Identificadores aplicados a variables para que estas se creen al comenzar el bloque donde estan definidas y se destruyan al salir.
  - **Auto:** Por defecto
  - **Register:** Se usa para colocar la variable en uno de los registros de memoria.

### Persistencia estatica:

- **Static:** Son conocidas unicamente dentro del bloque donde fueron creadas pero conservan su valor al terminar el bloque, por lo que en una proxima ejecucion su valor es el mismo.
- **Extern:** funciones o variables globales por default.

## Arreglos

- Si se inicializa una cantidad menor a **DIMF** el resto de elementos se inician en 0.
- Es util declarar la dimension del vector usando **#define** constName value. La cual hara un intercambio de constName por value en el **pre compilador/pre procesamiento**. En caso de que DIMF no haya sido declarada la dimension pasa a ser la cantidad de elementos inicializados. No se pueden omitir ambas.

```
static int valores[30];
```

Util en caso de ser variables locales a funciones

- Si nos queremos asegurar de que una funcion no modifique los valores de un arreglo podemos usar:

```
funcion(const dataType arrName[])
```

### Arreglos bidimensionales/Matrices:

Se mantienen las reglas de los arreglos unidimensionales con una excepcion en el pasaje de parametros.

```
int mini(int M[][COL], int f)
```

Es necesario especificar **DIMCOL** para que el compilador sepa cuantas posiciones de **memoria secuencial** tiene que saltar para llegar a x fila. Internamente **fila \* dimCol + col**.

### Strings

- Vector de caracteres con un caracter '\0' para indicar **fin**(este caracter ocupa lugar).

### Imprimir:

```
printf("%s\n", texto);
```

- Imprime hasta encontrar caracter nulo.

## Punteros

Asignacion

```
PtrDato = &Dato;
```

Declaracion:

```
int *countPtr,
```

El puntero puede ser tratado como un arreglo en el sentido del desplazamiento por memoria.

### Nota:

El 0 es el único valor que  
puede asignarse a un  
puntero.  
La conversión a (int \*) es  
automática.

### Puntero Void

- Puntero void sirve como ptr generico. Solo sirve como un almacenador. No puedo operar sobre el, por ejemplo con indireccion

- El ptr void aloca memoria y almacena cualquier puntero. Luego en caso de querer operar sobre el le asigno el ptr Void otro ptr que no sea void.

## Punteros Constantes

```
char * const p = "Ejemplo de ptr.";
```

## Structs

### Declaracion e inicializacion:

```
typedef struct fechaStruct
{
    short int anio;
    char mes;
    char dia;
} fecha;

main(){
    fecha varName = {2000, '1', '31'}; //La inicializacion puede omitirse
}
//Si se omite typedef y un nombre de struct las variables solo pueden ser declaradas
//donde se declaro fecha
```

### Operaciones Validas

- ◉ Las operaciones válidas son
  - > 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.

### Acceso a campos

En caso de tener un **PTR** a un Struct, podemos acceder a sus campos como:

```
PTRSTRUCT -> campo = "Something"
//es equivalente a
```

```
(*PTRSTRUCT).campo = "Something"
```

## Uniones

### Declaracion

El uso de **typedef** es el mismo que para los structs.

- Cuando se declara una union se reserva espacio para el campo que ocupe mayor cantidad de bits.
- El acceso y las operaciones validas son iguales a los **structs**.

```
union Nom_Tipo {  
    tipo_campo_1 nom_campo_1;  
    tipo_campo_2 nom_campo_2;  
    ...  
    tipo_campo_n nom_campo_n;  
};
```

## Operadores BIT a BIT

&	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.

- Si a estos operadores la agregamos la directiva **=** asigna el resultado de la operacion binaria a la variable apuntada.

## Campos de bits

## Declaracion

```
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;  
};
```

- Los miembros de un campo de bits deben declararse como int o unsigned.

### Notas:

- No podemos leer directamente de campos. Debemos leer con **aux** y luego asignárselo al campo. Consecuencia de que los campos de bits no tienen direcciones por lo que no se les puede aplicar &.

## Constantes de enumeracion

### Declaracion

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

- Por defecto arranca desde 0 excepto que yo le asigne a ENE otro numero.
- Los identificadores deben ser unicos.
- Diferencias con **#define**: los **enums** siguen reglas de alcance por lo que son utiles para ser declaradas en multiples bloques de codigo manteniendo localidad.

```
enum { buffersize = 256 };  
static unsigned char buffer[buffersize] = {0};
```

Caso de uso parecido a **#define**.

## Librerias Utiles

### STDIO.H

- Librerías necesarias para printf y scanf.

## STDLIB.H

- Útiles por funciones rand() y srand(time(NULL)).

## TIME.H

- Vienen con un puntero a un struct definido con campos de tiempo.

### struct tm

Campo	Descripción
int tm_hour	hora (0 - 23)
int tm_isdst	Horario de verano enabled/disabled
int tm_mday	día del mes (1 - 31)
int tm_min	minutos (0 - 59)
int tm_mon	mes (0 - 11, 0 = Enero)
int tm_sec	segundos (0 - 60)
int tm_wday	día de la semana (0 - 6, 0 = domingo)
int tm_yday	día del año (0 - 365)
int tm_year	año desde 1900

Declaración:

```
struct tm *info;
```

```
strftime(fechaHora, sizeof(fechaHora), "%d-%m-%Y", struct_tm);
```

- Almacena en fechaHora los datos de struct\_tm con el formato indicado por la máscara.

## STRING.H

**strlen(c1):** Retorna el número de caracteres de la cadena **c1** 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 como para almacenar la cadena **c2** y su carácter de terminación NULL (que también se copia).



| **strcat( c1, c2)** : Agrega la cadena **c2** al arreglo **c1**.

El primer carácter de **c2** sobrescribe el carácter de terminación NULL de **c1**.

**strcmp( c1, c2)** : Compara **c1** con **c2** y devuelve

$$\text{strcmp}(c1, c2) = \begin{cases} < 0 & \text{si } c1 < c2 \\ 0 & \text{si } c1 = c2 \\ > 0 & \text{si } c1 > c2 \end{cases}$$

```
puts(strName) //Imprime string hasta caracter nulo  
gets(strName) //Lee secuencia hasta leer enter
```