



# TDL - Mod I

📅 Age 2023

## Operadores Logicos

Estos operadores admiten operandos lógicos, esto es, de valor verdadero o falso según el criterio habitual en C (nulo es falso, no nulo es verdadero). No se puede afirmar nada acerca de la estructura de bits del resultado; solo se sabe que será falso o verdadero en los términos anteriores.

- && AND
- || OR
- ! NOT

## Sizeof(Var)

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

## Tipo INT

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

Tipos de datos (de mayor a menor)

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

## Estructura de selección

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

## Estructura iterativa

```
while(condición){ /* acción o bloque de acciones a realizar 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. Puede no existir si la variable ya esta inicializada antes.
- **condicion:** es una expresion logica cuyo valor se evalua *ANTES* de iniciar el for y debe ser verdadera para que el for se ejecute. Que la condicion sea verdadera quiere decir que sea  $\neq 0$ .
- **acciones\_posteriores:** es una o mas acciones separadas por comas que se ejecutan *LUEGO* de las instrucciones del for.

## Operador Ternario

```
Expresion logica ? valor1 : valor2
/*Evalua la expresion 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 y se utiliza*/
++ a++ /*Se utiliza y después incrementa*/
-- --b /*Se decrementa y se utiliza*/
-- b-- /*Se utiliza y después decrementa*/

```

## 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("%s",NombreFuncion(parametros));
}
TipoValorRetornado NombreFuncion(parametros){
    static TipoVar var; /*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 e
```

## Matrices

```
int matriz1[2][3] = {{1,2,3}, {4,5,6}};
int matriz2[2][3] = {1,2,4,6}; //completa con ceros
int matriz3[2][3] = {{1,2}, {6}}; //busca 1er forma

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

## Vector de caracteres

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

## Funciones para cadenas de caracteres

- **strlen(c1)**: Retorna el numero de chars hasta el caracter 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.

## 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); /*imprime lo contenido en el puntero (num)*/
```

```
tipo_dato * const p; //el valor de P ya no puede cambiar pero si su contenido
const tipo_dato *p; //el puntero puede señalar otra direccion de memoria pero no puede cambiar contenido
```

! 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 contenido por el ptr
```

## 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[filas][col]`
  - `*(nros + (15 * filas) + 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};

```

/\*Acceso con punteros\*/

```

struct cartas mazo = {1,2};
ptr = &mazo;
printf("%d",ptr->mazo.campo);

```

## 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; //creo var

```

## 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 pr
union numero valor = {10.234}; //El valor asignado se tr

```

```

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\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\n", valor.y);

```

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 señalados 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 señalados 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.

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=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);

1 & 2 = 0
3 & 2 = 2

```

```

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);

1 ^ 2 = 3
3 ^ 2 = 1

```

```

int a, b;

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

64 >> 3 = 8

```

```

int a, b;

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

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 enumeracion

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};
```





## TDL - Mod II

📅 Age	2023
-------	------

### Asignacion de memoria dinamica

- **Malloc:** `void * malloc(size)`
  - Retorna un puntero de tipo `void *` el cual es el inicio en memoria de la porcion reservada.
  - Si no puede reservar esa cantidad de memoria la funcion regresa un puntero `NULL`.
  - Para arreglos dinamicos: `= void * malloc (N*sizeof(type))`
    - Se puede acceder con `ptr[x]` o `*(ptr+x)`
- **Calloc:** `void * calloc(N,size)`
  - Retorna un puntero de tipo `void *` el cual es el inicio en memoria de la porcion reservada.
  - Reserva un espacio para un arreglo de N objetos.
  - El espacio es inicializado con todos los bits en cero. Por esta razon, consume un poco mas de memoria y demora un poco mas con respecto a `malloc`.
- **Realloc:** `void * realloc(ptr, size)`
  - Cambia el tamaño del objeto apuntado por `ptr` al tamaño especificado por `size`.
  - Busca memoria → Reserva memoria → Copia el objeto → Libera la conexion anterior.
  - Si `size` es cero y `ptr` no es nulo, el objeto al que apunta es liberado.
- **Free:** `void free(ptr)`
  - Libera el espacio apuntado por `ptr` (elimina conexión con la memoria reservada).
  - Si `ptr` es `NULL` no se realiza ninguna accion.



Importante antes de hacer cosas con el puntero creado, fijarse que no esté en nulo...

## Declaracion de una lista o pila

```
#include <stdio.h>
int main(){
    struct nodo {
        int valor;
        struct nodo * ptr;
    }
    struct nodo *Pila = NULL, *aux;
}
```

## Matrices dinamicas

```
/* Programa asignar una matriz dinámica */
#include <stdlib.h>
int main(void)
{
    const int f=3; /* número de filas */
    const int c=1; /* número de columnas */
    double **mat; /* puntero de puntero */
    int i; /* contador */

    /* Asigno memoria (falta el control de errores) */
    mat=(double**)calloc(f,sizeof(double*));

    for(i=0; i<f; i++){
        mat[i]=(double*)calloc(c,sizeof(double));
    }
    < ... sigue ... >

    f  3
    c  1
    mat 26F3
    i    2
```

```
//Liberar memoria
for(int i=0;i<f;i++){
    free(mat[i]);
}
free(mat); //solo libero la conexion al primer indice
```

Para la reserva de memoria, siempre es necesario pasar la direccion del puntero \*\* creado.



Copiar datos de un bloque de memoria a otro: `memcpy(destino,origen,size)`

## Archivos de Texto

```
FILE * arch; //crea un puntero a la direccion de memori
arch = fopen("nombre.txt","modo_apertura");
if(arch==NULL) printf("El archivo no abrio correctamer
fclose(arch); //retorna cero si fue cerrado con exito
```

### Operaciones entrada/salida

### MODOS DE APERTURA DE ARCHIVO

Modo	Descripción
<b>r</b>	Abrir un archivo para lectura.
<b>w</b>	Crear un archivo para escritura. Si el archivo ya existe, se descarta el contenido actual.
<b>a</b>	Abrir o crear un archivo para escribir al final del mismo
<b>r+</b>	Abrir un archivo para lectura y escritura.
<b>w+</b>	Genera un archivo para lectura y escritura. Si el archivo ya existe, se descarta el contenido actual.
<b>a+</b>	Abrir o crear un archivo para actualizar. La escritura se efectuará al final del archivo.

- **fprintf(arch o stdout, \*cadena):** escribe en el archivo lo apuntado por cadena

- **fscanf(arch, tipo de datos a recuperar, dir variables para guardar datos (opcional))**

- **int feof(arch)**

- La función feof retorna 1 si el archivo terminó y 0 en otro caso.
- Note que la función feof indica si ya se realizó una operación fuera del límite del archivo; no si se encuentra posicionado en el límite del archivo.

```
while (! feof( arch )){
    /* procesamiento de los datos */
    /* operación de lectura */
}
```

- **int fgetc(arch):** lee un caracter desde el archivo y lo convierte a int.

- fgetc(stdin) equivale a getchar().

- **int fputc(int c, arch):** escribe el caracter leído desde el archivo en la variable c.

- fputc('a', stdout) equivale a putchar('a')

- **fgets(\*cadena, n, arch)**

- Lee n-1 caracteres para agregar un carácter nulo inmediatamente después del último carácter leído en el array.

- **fputs(\*cadena, arch)**

- Escribe lo apuntado por cadena en el archivo.
- No escribe el \0.

```
while ((caracter = fgetc(fuente)) != EOF) {
    fputc(caracter, destino);
}
//*****//
while (fgets(linea, sizeof(linea), fuente) != NULL) {
    fputs(linea, destino);
}
```

## Desplazamiento en el archivo

Al abrir un archivo en modo de acceso "r" o "w" el desplazamiento se inicializa en 0 (comienzo del archivo), en cambio, si se utiliza el modo "a" el desplazamiento comienza al final del archivo.

- **long ftell(arch);**

- **int fseek(arch, desplazamiento, origen);**

- Reubica la posición del puntero al archivo.
- La nueva posición, medida en caracteres, es obtenida mediante la suma de desplazamiento y la posición especificada por origen.
- Los valores para origen son: SEEK\_SET (inicio del archivo), SEEK\_CUR (actual), SEEK\_END (final del archivo).

## Archivos de texto binarios

## Operacion E/S

- `fwrite(&num,sizeof(int),1,arch);`
  - Envía desde el arreglo apuntado por puntero, la cantidad de elementos indicada en cuantos cuyo tamaño es especificado por tamaño, al dispositivo apuntado por stream.
- `fread(vector, sizeof(int),5, arch);`
  - Recibe en el arreglo apuntado por puntero, la cantidad de elementos indicada en cuantos cuyo tamaño es especificado por tamaño, desde dispositivo apuntado por stream.

## Modo de apertura de los Archivos Binarios

Modo	Descripción
<b>rb</b>	Abre un archivo binario para lectura
<b>wb</b>	Crea un archivo binario para escritura; si el archivo ya existe se descarta el contenido actual.
<b>ab</b>	Abre o crea un archivo binario para escribir al final del mismo
<b>rb+ ó r+b</b>	Abre un archivo para lectura y escritura.
<b>wb+ ó w+b</b>	Crea un archivo binario para lectura y escritura. Si el archivo existe, se descarta el contenido actual.
<b>ab+ ó a+b</b>	Abre o crea un archivo binario para actualizar. La escritura se realizará al final del archivo.

### Concepto de archivo binario

Un archivo binario es un tipo de archivo que permite almacenar un bloque de datos de cualquier tipo. Los archivos binarios los puede crear únicamente el programa y el acceso a sus elementos sólo es posible a través del programa. El contenido de un archivo binario es ilegible ya que utiliza un esquema de representación binario interno. Este esquema depende de la computadora que se use, por lo que no puede ser visualizado mediante un editor de textos.

### Diferencia entre archivo binario y archivo de texto

Las componentes de un archivo de texto son de tipo *char* y están organizados en líneas mientras que las componentes de un archivo binario pueden ser de cualquier tipo predefinido o definido por el usuario (excepto de tipo archivo).

## Preprocesador

El preprocesamiento es el primer paso en la etapa de compilación de un programa. Es una característica del compilador de C. Todas las directivas del preprocesador o comandos inician con un `#`.

Ventajas de usar el preprocesador:

- El código C es más portable entre diferentes arquitecturas de máquinas.
- Programas más fáciles de desarrollar, de leer y de modificar.

## Directivas

- **include:** sirve para insertar archivos externos dentro de nuestro archivo de código fuente.
  - `#include <archivo>`
    - Busca el archivo en la librería estándar.
    - Se utiliza para los arch de la librería estándar.
  - `#include "archivo"`
    - Busca primero en el directorio actual y luego en la librería estándar.
    - Se utiliza para archivos definidos por el usuario.



A estos dos comandos se los llama prototipos de cabecera

- **define identificador valor:** Si un valor es provisto, el identificador será reemplazado literalmente por valor (el resto del texto en la línea).
  - Macro: Es una operación definida mediante `#define`
    - Una macro sin argumentos es tratada como una constante simbólica.
    - Una macro con argumentos, al ser expandida, reemplaza sus argumentos con los argumentos reales encontrados en el programa.
    - Realiza una sustitución de texto, sin chequeo de tipos.



Para encapsular un pedazo de código, se debe hacer en un `do-while` para protegerlo.

- **undef:** elimina la definición de una constante simbólica o macro.
- **if, elif, else, endif**

### #if, #elif, #else y #endif

- Permiten hacer una **compilación condicional** de un conjunto de líneas de código.
- **Sintaxis**

```
#if expresión-constante-1
<sección-1>
#elif <expresión-constante-2>
<sección-2>
.
.
.
#elif <expresión-constante-n>
<sección-n>
<#else>
<sección-final>
#endif
```

- **ifdef y ifndef:** Permiten comprobar si un identificador está o no actualmente definido, es decir, si un `#define` ha sido previamente procesado para el identificador y si sigue definido.

## Headers

### Archivo main.c

```
1 #include <stdio.h>
2 #include "cabecera.h"
3
4 int main()
5 {
6     printf("Ejecutando...\n");
7     VerTexto("Funciona?");
8     printf("Terminado.\n");
9
10    return(0);
11 }
```

### Archivo cabecera.h

```
1 #ifndef CABECERA_H_INCLUDED
2 #define CABECERA_H_INCLUDED
3
4 void VerTexto(char *);
5
6 #endif // CABECERA_H_INCLUDED
```

- Este archivo utiliza «include guards» (guardas include) para evitar múltiples definiciones de la función.

Este archivo no tiene implementaciones, solo prototipos.

### Archivo FuncionVerTexto.c

```
1 #include <stdio.h>
2 #include "cabecera.h"
3
4 void VerTexto(char * cadena)
5 {
6     printf("%s\n", cadena);
7 }
```

- Puede incluirse la cabecera para chequear consistencia y evitar errores.

Compile y verifique que funciona

## Argumentos Main

- `Argc`: cantidad de argumentos recibidos por la función `main`.
- `Argv`: vector que contiene los argumentos en formato string

```

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
    int n1, n2;
    char oper;
    float result;

    if (argc != 4)
        printf("Número incorrecto de parámetros\n");
    else
    {
        n1 = atoi(argv[1]);
        n2 = atoi(argv[2]);
        oper = *argv[3];

        switch (oper){
            case '+': result = n1 + n2; break;
            case '-': result = n1 - n2; break;
            case '*': result = n1 * n2; break;
            default : result = (float)n1 / n2;
        }
        printf ("%d %c %d = %.2f", n1, oper, n2, result);
    }
    return 0;
}

```

2Operandos1Operador.c



atoi convierte a int, mientras que atof convierte a float

## Compilador GCC

- gcc ejemploMain.c: se creara el exe con nombre por defecto.
- gcc -o ejemplo ejemploMain.c || gcc -Wall ejemplo ejemploMain.c (compila con mas warnings)
- gcc -DLINEWIDTH=80 -o Ej2Modif Ej02\_Modif.c (cumple la funcion de define)

# OPTIMIZACIÓN DE CÓDIGO CON GCC

Opción	Descripción
<b>0</b>	Optimizar para la velocidad de compilación - sin optimización del código (por defecto).
<b>1,2,3</b>	Optimizar para aumentar la velocidad de ejecución del código (cuanto mayor sea el número, mayor será la velocidad).
<b>s</b>	Optimizar el tamaño del archivo.
<b>funroll-loops</b>	Optimizar activando el desenrollado de bucles. Es independiente de otras opciones de optimización.

Ejemplo: gcc hola.c -o hola -O1

## Makefile

all: objetivo (hola)  
hola: requisito (hola.o)  
gcc hola.o -o hola  
hola.o: hola.c  
gcc -c hola.c -o hola.o

```
CC = gcc
CFLAGS = -c -Wall
SOURCES = $(wildcard *.c)
OBSJS = $(SOURCES:.c=.o)
EXE = hola.exe

all: $(EXE)

$(EXE): $(OBSJS)
$(CC) $(OBSJS) -o $@

%.o: %.c
$(CC) $(CFLAGS) $< -o $@

clean:
rm -f $(OBSJS) $(EXE)
```

Busca todos los archivos .c en el directorio actual

Genera la lista de archivos objeto

\$@ se refiere al objetivo  
En este caso "hola.o"

\$< corresponde al 1er. requerimiento.  
En este caso "hola.c"