UNIVERSIDAD FASTA FACULTAD DE INGENIERÍA

Estructuras de Datos
Taller de Programación I
Lenguaje C y Memoria dinámica

RESEÑA HISTÓRICA

El lenguaje C fue desarrollado por Dennis Ritchie en 1972 en los laboratorios de Bell mejorando el lenguaje B de Thompson. Pronto, empezaron a surgir muchas implementaciones del lenguaje C, a raíz de su popularidad. Por este motivo, se hizo necesario definir un estándar que está representado hoy por el ANSI C.

ALGUNAS CARACTERÍSTICAS DEL LENGUAJE C

- El lenguaje de programación C está caracterizado por ser de uso general, con una sintaxis sumamente corta y un juego de operadores muy potente.
- C es un lenguaje de "medio-bajo nivel". Esto significa que explota los recursos Hardware así como el manejo de las direcciones y los bits.
- No posee operaciones de entrada-salida, ni métodos de archivos, y tampoco maneja cadenas de caracteres. Estas operaciones se hacen por medio de funciones contenidas en librerías externas al lenguaje, lo que le confiere un alto grado de portabilidad.
- C es un lenguaje semi-estructurado, por la razón de que no se puede declarar funciones dentro de otras. La estructura básica de un programa C se hace alrededor de la función; ésta puede tener sus propias variables locales, acceder a las variables globales del programa, y finalmente devolver uno o más valores o ninguno.
- C es un lenguaje que permite una compilación separada. Se puede partir un archivo en varios archivos más pequeños y que cada uno sea compilado por separado. Luego se enlazan entre sí junto con las rutinas de biblioteca para formar el ejecutable completo. Su ventaja reside en que cambios de código en uno de los archivos no necesita una nueva compilación del programa entero.
- Un programa en C durante su ejecución crea y usa 4 regiones de memoria lógicas diferentes: la primera región contiene el código del programa, la segunda contiene las variables globales, la tercera, es la pila (Stack) que almacena temporalmente las direcciones de las funciones, el estado del procesador, etc., y la cuarta llamada es el Heap (montículo), que es la región libre, que sirven para reservación dinámica de memoria.

La descripción del lenguaje se realiza siguiendo las normas del ANSI C, por lo tanto, todo lo expresado será utilizable con cualquier compilador que se adopte.

PROGRAMAS EN C

Los programas se construyen con:

- ✓ Comentarios.
- ✓ Órdenes para el preprocesador
- ✓ Definiciones de funciones
- ✓ Expresiones formadas con constantes, variables, funciones y operadores.
- ✓ Sentencias

☞ **Ejemplo**

```
#include <stdio.h>              /* inclusión de un archivo */
main(){
   printf("Bienvenido");        /* una sentencia */
   return 0;                    /*termina el programa */
}
```

COMENTARIOS

Para el compilador, los comentarios son inexistentes, por lo que no generan líneas de código, permitiendo abundar en ellos tanto como se desee. En C se toma como comentario todo carácter interno a los símbolos: /* */ y pueden ocupar uno o más renglones

```
/* este es un comentario */
/* Este es otro comentario más
largo que el anterior */
```

FUNCIÓN MAIN()

main() es la función principal, indicando donde empieza el programa su cuerpo es un conjunto de sentencias delimitadas por llaves. Cada sentencia termina con el punto y coma ";".
Es la función invocada desde el sistema operativo cuando comienza la ejecución del programa. La sentencia "return 0" termina el programa y devuelve un valor al Sistema.

```
void main () {
……
}

int main () {
……
return 0;
}
```

ENCABEZADO - ORDENES PARA EL PREPROCESADOR

Son órdenes que el preprocesador interpreta antes de que el código fuente sea compilado, comienzan con # y se encuentran antes del *main()*.
El preprocesador produce un programa escrito en C que es lo que se compila después.
La directiva **#include** es una orden para leer un archivo de texto especificado en el nombre que sigue a la misma **<stdio.h>**. El preprocesador reemplaza la directiva por el contenido de dicho archivo, que contiene las declaraciones de las funciones invocadas por el programa.
La directiva **#include** no es una sentencia de programa, por lo que no es necesario terminarla con el punto y coma '**;**'.
El compilador tiene varias formas de buscar un archivo.

```
#include <stdio.h>            /* Carpeta de instalación */
#include "stdio.h"            /* Carpeta activa en el momento */
#include "g:\milibreria.h"    /* Carpeta indicada */
```

La directiva **#define** se utiliza para declarar constantes que pueden ser identificadores y también expresiones. Lo que *#define* permite en definitiva, es operar con macros (expansión de código).
#define *nombre texto_reemplazo* luego cada vez que se hace mención a *nombre* se reemplaza por el *texto_reemplazo*

☞ **Ejemplo de utilización de define**

```
#define PI 3.1416             /* Definición de constante simbólica */
#define X PI+2                /* Definición de X en base a PI */
#define prod(a,b)(a*b)        /* Macros con parámetros, que si luego se
            escribe x=prod(2,(4+3)) se reemplazará por x=(2*(4+3)); */
```

IDENTIFICADORES

En C, los nombres de las variables, funciones y otros objetos definidos por el usuario se conocen como **identificadores**. La longitud de un identificador puede variar de uno a varios caracteres, siendo el primero una letra o un guión bajo y los siguientes una combinación de letras, números y guiones bajos.
Ningún identificador debe coincidir con las palabras reservadas del lenguaje, ni con el nombre de alguna función de librería.
En C, las minúsculas y las mayúsculas se tratan como distintas (es *case sensitive*)

```
cont, prueba10, balance_total  /* Son válidos */
1cont, hola!, balance..total   /* Son inválidos */
```

TIPOS DE DATOS

Existen cinco tipos de datos básicos en C; **char** (carácter), **int** (entero), **float** (real coma flotante), **double** (real de doble precisión) y **void** (sin valor).
El tipo **void** se usa para declarar funciones que no devuelven ningún valor o para declarar algún puntero genérico.

Modificadores de tipos
Los modificadores de tipos se usan para alterar el tipo base: **short**, **long**, **unsigned, signed**. Los modificadores se pueden aplicar a los tipos base int y char; **long** también se puede aplicar a double.
Por ejemplo, el tipo *int* permite valores en el rango *-32768 a 32767* mientras que *unsigned int* permite valores en el rango *0 a 65535*.

VARIABLES

Todas las variables deben declararse antes de ser usadas; esto es, indicar el tipo de datos al que pertenecen. Esto determina:

- Su representación en memoria
- El conjunto de valores posibles
- Las operaciones permitidas y la forma de realizarlas

☞ **Ejemplo de declaración de variables**

```
int i,j,k, v[20], m[10][10];   /* 3 variables enteras, un vector y una
                                  matriz de enteros  */
float beneficio, perdida;      // 2 variables reales
unsigned int a;                //entero sin signo
```

Existen tres sitios donde se pueden declarar variables: dentro de las funciones "**variables locales"**, en la definición de parámetros de funciones "**parámetros formales"** y fuera de todas las funciones "**variables globales**".

Dónde se declaran las variables

- **Variables locales**

Llamadas también variables automáticas y se declaran dentro de las funciones y son referenciadas sólo por sentencias que estén dentro del bloque donde están declaradas.
Recordar que un bloque está encerrado entre dos llaves. Las variables locales existen sólo cuando se ejecuta el bloque y se destruye al salir de él.

- **Parámetros formales**

Los parámetros formales son los argumentos de una función. Su declaración se hace entre los paréntesis de la función. Su comportamiento es igual que las variables locales de cualquier función; o sea desaparecen al terminar su ejecución.

- **Variables globales**

Las variables globales se conocen y se pueden usar en cualquier parte del programa. Además mantienen su valor a lo largo de la ejecución del programa.
Las variables globales se crean justo después de declaración, y por lo general, se hace al principio del programa y fuera de todas las funciones.
Se puede declarar una variable local a una función con el mismo nombre de una variable global existente, pero el compilador las trata distintamente; Dentro de esa función, prevalece la variable local.

Inicialización de variables
Todas las variables se pueden inicializar en el momento de su declaración.

☞ **Ejemplo de inicialización en la declaración**

```
float cont=0, max=-1;
char c='a';
int x, y = 6;                  // sólo inicializa y
```

EXPRESIONES

Una expresión se forma combinando constantes, variables, operadores e invocaciones a funciones.

☞ **Ejemplo**

```
s = s + i
n == 0
++i
```

Una expresión representa un valor que es el resultado de realizar las operaciones indicadas siguiendo las reglas de evaluación establecidas en el lenguaje.

OPERADORES

Operadores Aritméticos

Los operadores aritméticos comprenden las operaciones básicas: suma (+), resta (-), multiplicación (*), división (/), autoincremento (++), autodecremento (--) y módulo (%).

El operador módulo: % se utiliza para calcular el **resto** del cociente entre dos enteros, y no puede ser aplicado a variables del tipo float ni double.

Se puede observar que no existen operadores de potencia, radicales, logaritmos, etc. En C todas estas operaciones (y muchas otras) se realizan por medio de llamadas a funciones de librería (ver *math.h*).

Los operadores ++/-- suelen emplearse con variables de tipo **int,** aunque se pueden usar sin problemas con cualquier otro tipo de variable.

La colocación de los operadores como **prefijo** ó **sufijo** son equivalente y tienen el mismo efecto, excepto cuando haya otros operadores en la expresión; la posición de los operadores (**++,--**) indica el momento de la operación de incremento o decremento con respecto a las otras operaciones.

☞ **Ejemplo de autoincremento/autodecremento**

```
n = n + 1; es equivalente a n++; o ++n;
m = m - 1; es equivalente a m--; o --m;
m = 4; n = ++m;                /* Se incrementa m y se asigna su valor a
                                  n; (n = m = 5) */
m = 4; n = m++;                /* Se asigna m a n y luego se incrementa,
                                  (n = 4 y m = 5) */
En ambos casos el valor de m es 5, pero el valor de n es distinto.
```

Operadores relacionales

Todas las operaciones relacionales resultan en sólo dos posibles resultados: **falso** o **verdadero**. En C, no existe el tipo booleano, por lo que falso queda representado por cualquier expresión cuyo valor es nulo (cero) y verdadero por cualquier valor distinto de cero. No hay que confundir el operador relacional *igual que* (==) con el de operador de *asignación* (=).

Símbolo	Significado	Símbolo	Significado	Símbolo	Significado
==	igual	<	menor	>	mayor
!=	distinto	<=	menor o igual	>=	mayor o igual

Operadores lógicos

Símbolo	Significado	Símbolo	Significado	Símbolo	Significado
&&	AND	||	OR	!	NOT

Asignación

☞ Sintaxis: identificador = expresión;

El *identificador* representa generalmente una variable y *expresión* puede ser una constante, una variable o una expresión más compleja. El *identificador* se conoce como **lvalue** y la *expresión* por **rvalue**. Una asignación es simplemente la copia del resultado de una expresión **rvalue** sobre otra **lvalue**, por lo tanto, **lvalue** debe poseer una posición de memoria.

```
a =17;
17 = a;                        // Incorrecta! pues 17 no es lvalue
```

Asignación múltiple

☞ Sintaxis: identificador1 = identificador2 = identificador3 = expresión;

Se permite la asignación múltiple, y se evalúa de derecha a izquierda.

```
a = b = c = 5;                  // las variables toman el valor 5
```

Asignación condicional

☞ Sintaxis: identificador = (expresion_1) ? expresion _2 : expresion_3 ;

La expresión *expresion_1* que es tipo relacional con un resultado **verdadero** o **falso** se evalúa, y si el resultado es **verdadero**, se asignará al *identificador* el valor de *expresion_2* y si el resultado es **falso**, se le asignará al Identificador valor de *expresion_3*.

☞ **Ejemplos de asignación condicional**

```
menor =(a<b)? a:b ;             /* Se elige el menor de a y b */
valor =(a>0)? -a:a;             /* Asigna el negativo */
return (a<b)? a:b ;             //return asociado al condicional
```

Operadores combinados con asignación

☞ Sintaxis: identificador **op=** expresion;

C posee otros operadores de asignación que se combinan con los operadores aritméticos para abreviar las operaciones. Así una operación como a = a + b, se puede abreviar como: a += b.

No se admite un espacio en blanco entre los símbolos. Esta asignación se extiende a todos los operadores aritméticos.

☞ **Ejemplo**

```
int i = 5, j = 7;
float f = 5.5, g = -3.25;
f -= g;                         /* f = f-g; f = 8.75 */
j *= (i-3);                     /* j = j*(i-3); j=14 */
j %= 2;                         // j = j % 2
```

Conversión de tipo

Conversión Automática

Cuando dos o más tipos se encuentran dentro de una misma operación o expresión matemática, ocurre una conversión automática del tipo de las variables.

☞ **Ejemplo**

```
int h=9;
float i=7,k;
k = i/2;
k = h/2;
k = h/2.0;
```

Conversión por asignación

En una asignación, *lvalue = rvalue;* el cálculo de *rvalue* se hace de acuerdo con las reglas de conversión automática, pero luego se ajusta, de ser posible, al tipo de *lvalue.*

```
int h;
float i=7;
h = i/2;
```

Conversión explícita (cast)

☞ Sintaxis: (*tipo de dato*) *operando*;

El lenguaje permite cambiar el tipo de una expresión imponiendo el tipo del resultado de una operación mediante la conversión explícita; donde *tipo de dato* puede ser (char, int, float, double, long, unsigned, etc.). El *cast* no cambia el tipo de dato de las variables o expresiones, sino cambia el tipo del valor para efectuar la operación.

☞ **Ejemplo**

```
float x, k=2.7;
int j, i=3;
j = k*i;
j = (int)k*i;
x = k*i;
x = (int)k*i;
x = (int)(k*i);
```

Tabla de precedencia de algunos operadores ordenada de mayor a menor

Operador	Asociatividad
() [] -> .	I a D
sizeof (tipo) ! ++ -- * &	D a I
* / %	I a D
+ -	I a D
>> <<	I a D
== !=	I a D
& (and binario)	I a D
^	I a D
&&	I a D
||	I a D
? :	D a I
= += *= otros	D a I

```
a - b / c * d      /* Primero b/c, luego se multiplica por d, y finalmente
                      se resta el producto a a */
m = a*++n;         /* Primero se incrementa n y luego se multiplica por a
                      y luego se asigna a m */
```

Información adicional : ***Operadores en C, Precedencia y Asociatividad***
https://docs.microsoft.com/es-es/cpp/c-language/c-operators?view=msvc-160

SENTENCIAS

Una sentencia se forma con expresiones. Un bloque empieza con " **{** " y termina con " **}** ", pudiendo contener cualquier número de sentencias y declaraciones.
Todas las sentencias, excepto los bloques, terminan con el símbolo " **;**

ENTRADA / SALIDA

El conjunto de las funciones de E/S están definidas en el archivo **stdio.h**, pero en algunos compiladores se puede tener algunas funciones de E/S definidas en **conio.h**.
Cuando se emplea alguna función de E/S, se debe incluir el archivo al comienzo del programa mediante la directiva **#include <stdio.h>** y el compilador insertará el código fuente al comienzo del archivo.

Entrada/Salida con formato

La función printf()

Para visualizar datos por pantalla, se dispone de la función **printf()** que permite formatear la salida. Esta función puede visualizar una combinación de valores numéricos, caracteres y cadenas de caracteres.

☞ Sintaxis: **printf**(*cadena_de_control*, *arg1*, *arg2*, ..., *argn*);

La *cadena_de_control* contiene la información sobre el formato de las salidas de los *argumentos*, los cuales pueden ser constantes, variables, referencias a funciones, o cualquier expresión C. La *cadena_de_control* está compuesta por grupos de caracteres, donde cada grupo encabezado por el símbolo "**%**" representa el formato del dato de salida. En la *cadena de control*, se puede intercalar texto sin formato para comentar las salidas.
Debe haber igual número de comandos de formato como argumentos. Si hay más argumentos que formatos se descartan los que sobran, pero si hay menos, la salida es indefinida.

Tabla de los códigos de algunos formatos posibles	
Formato	**Visualización de datos**
%c	Carácter
%s	Cadena de caracteres
%d	Entero decimal con signo
%u	Entero decimal sin signo
%f	Coma flotante
%e	Coma flotante en notación científica.
%p	Dirección del puntero.

Ejemplo Formato	Descripción
%10d	escribe un entero en un campo de 10 dígitos.
%5.2f	escribe un float en un campo de 5 dígitos donde 2 de ellos son decimales.
%06d	rellenara con ceros, en vez de espacios en blanco, los números menores de 6
%-6d	justifica a la izquierda el número en un campo de 6 dígitos
%5-7s	escribe una cadena de entre 5 y 7 caracteres. Se truncará si tiene más de 7 y se rellenará con espacios en blanco si tiene menos de 5.

Caracteres no representables	
\n	Salto de línea
\t	Tabulación horizontal
\’	Apóstrofo
\”	Comillas
\\	Barra invertida

.

☞ **Ejemplo**

```
char ch = 'A';
int m = 4;
float x = 14.250;
printf("ch=%c, m=%d", ch, m);          /* ch=A, m=4 */
printf("ch=%d, x=%f", ch, x);          /* ch=65, x=14.250 */
printf("x=%7.1f, \n m=%05d", x, m);    /* x=   14.2,
                                          m=00004 */
printf("La suma x+m=%6.3f",x+m);          /* La suma x+m=18.250 */
```

La función scanf()

Se utiliza para leer los datos desde el teclado y la sintaxis es parecida a la función anterior.

☞ Sintaxis: **scanf**(*cadena_de_control*, *arg1*, *arg2*, ..., *argn*);

La *cadena_de_control* contiene la información sobre el formato de las entradas y son idénticos a los formatos vistos de la función **printf().** Pero, los *argumentos* son direcciones de las variables y para ellos se debe utilizar el operador de las direcciones “&”.

Cuando se trata de leer una cadena, el argumento no debe ir precedido por *ampersand*, ya que el propio nombre del arreglo indica la dirección del mismo.

La ausencia del & cuando el mismo sea necesario, no será detectado como error.

☞ **Ejemplo**

```
int n;
float x;
char cadena[20];
.........
scanf("%f %d",&x, &n);
scanf("%s ", cadena);
```

Entrada/Salida sin formato

Para operar con **caracteres** (teclado/pantalla) se tienen las siguientes funciones:

- ✓ **getche()** espera hasta que se pulse una tecla y devuelve su valor; la tecla pulsada aparece en pantalla.
- ✓ **getch()** opera como la anterior, pero sin eco, es decir el carácter pulsado no aparece en la pantalla.
- ✓ **getchar()** lee un carácter del teclado (con eco) y devuelve su valor cuando se pulsa ENTER.
- ✓ **putchar( )** imprime un carácter en la pantalla.

☞ **Ejemplo**

```
char ch;
ch = getchar();
putchar(ch);
putchar('B');
putchar(98);                    // Escribe B
```

Para leer o escribir una **cadena de caracteres** (teclado/pantalla), se tienen las siguientes funciones:

- ✓ **gets( )** lee una cadena de caracteres hasta pulsar ENTER. La cadena se almacena en un arreglo de caracteres y en lugar de nueva línea (\n), se pone el carácter null (\0).
- ✓ **puts ( )** escribe una cadena de caracteres en la pantalla y traduce el \0 por un \n (newline).

☞ **Ejemplo**

```
char linea[80];
........
gets(linea);
puts(linea);
puts("UNO");
```

SENTENCIAS de DECISION

Sentencia if-else

☞ Sintaxis: **if** (*expresión*) *sentencia1* [*else sentencia 2]*;

El operador condicional (? :)

☞ Sintaxis: (*condicion*) ? *expresion1* : *expresion2*;

Sentencia switch

☞ Sintaxis: **switch** (*expresion*) {
case *constante1***:** *sentencia1***;**
break;
case *constante2***:** *sentencia2***;**
break;
.....
*[***default:** *sentencian]***;**
}

La ejecución del **switch** comienza con la evaluación de *expresión* (int o char), y luego se compara sucesivamente con todas las etiquetas**.** Cuando se iguala a una de ellas, se ejecuta la sentencia correspondiente. Si no aparece la palabra **break,** continúa la ejecución del resto de las sentencias, pero si aparece un **break,** se termina la ejecución del **switch**. Esto implica que los **case** no son excluyentes.
Al final del **switch**, puede aparecer una opción **default**, que implica que si no se ha igualado la *expresión* evaluada a ningún **case,** se ejecute lo que sigue. En este caso no es necesario un **break**.
Observación: si se eliminan los **break** al resultar *verdadera* una comparación, se ejecutarán las sentencias de ese **case** particular pero también las que se hallen por debajo.

☞ **Ejemplos de equivalencia entre estructuras**

```
if (a<b)
   sentencia 1;
else
   sentencia 2;
```

⇨

```
(a<b) ? sentencia 1 : sentencia 2;
```

```
if (ch == 'a' || ch == 'A')
   conta++;
else
   if (ch == 'e' || ch == 'E')
     conte++;
```

⇨

```
switch (ch) {
 case 'a':
 case 'A':conta++;
          break;
 case 'e':
 case 'E':conte++;
          break;
}
```

SENTENCIAS de REPETICION

Sentencia while

☞ Sintaxis: **while** (*expresion*)
Sentencia/s;

☞ **Ejemplo** Mostrar los dígitos de 0 a 9

OPCION 1	OPCION 2
`int digito = 0;` `while(digito <= 9){` `  printf("%d \n", digito);` `  ++digito;` `}`	`int digito = 0;` `while(digito <= 9)` `  printf("%d \n", digito++);`
`int digito = 0;` `while(digito++ <= 9){` `  printf("%d \n", digito);` `  ++digito;` `}`	**//Es este código equivalente a los dos anteriores?**

Sentencia do-while

☞ Sintaxis: **do**
Sentencia/s;
while (*expresion*);

☞ **Ejemplo** Mostrar los dígitos de 0 a 9

```
int digito = 0;
do {
  printf("%d \n", digito);
  ++digito;
} while(digito <= 9)
```

Sentencia for

☞ Sintaxis: **for**(*expresion1***;** *expresion2* **;** *expresion3*)
Sentencia/s;

dónde *expresion1* es una asignación de una o más variables que equivale a su inicialización, *expresion2* representa una condición que debe ser satisfecha para que continúe la ejecución del ciclo y *expresion3* es otra asignación, que comúnmente varía alguna de las variables contenida en *expresión2*. Cuando se ejecuta un **for** por única vez se ejecuta *expresión1*, se evalúa *expresion2* que se comprobará antes de cada iteración del bucle, y al final de cada pasada se evalúa *expresion3*, por lo tanto un bucle **for** es equivalente a un bucle **while** como se muestra a continuación:

expresion1;
while(*expresión2*) {
sentencia/s;
expresion3;
}

☞ **Ejemplo** Mostrar los dígitos de 0 a 9

OPCION 1

```
int digito;
for (digito=0;digito<=9;digito++)
  printf("%d \n", digito);
```

OPCION 2

```
int digito;
 for (digito=0;digito<=9;)
  printf("%d \n", digito++);
```

OPCION 3

```
int digito=0;
for (;digito<=9;)
  printf("%d \n", digito++);
```

EJERCICIOS

1) Leer un número y obtener y mostrar su factorial.
2) Leer un conjunto de números positivos y mostrar el factorial de cada uno de ellos.
3) Leer nombre y 3 notas de un conjunto de N alumnos, mostrar el nombre del alumno que obtuvo el mejor promedio.

TIPO PUNTERO

Una variable de tipo puntero almacena una dirección de memoria.
Cuando se declara un puntero se reserva espacio para albergar una dirección de memoria, pero **no para almacenar el dato al que apunta el puntero**.
El espacio de memoria reservado para almacenar una variable de tipo puntero es siempre el mismo (4 bytes) independientemente del tipo de dato al que se apunte.
Los operadores ***** y **&** se utilizan para operar con direcciones de memoria. Son unarios (afectan sólo a un operando).

- ✓ el operador **&** permite obtener la dirección de una variable, y
- ✓ el operador ***** es el complementario de éste, pues al evaluarse devuelve el contenido de una dirección de memoria, por lo tanto el operando tiene que ser un puntero. El operador ***** sirve también para declarar una variable de tipo puntero a un tipo de dato determinado.

☞ **Ejemplo**

```
int m=1, n=2,z;
int *p, *q;                    /* p y q son punteros a enteros */

p = &m;                        /* p recibe la dirección de m */
*p = 4;                        /* sería incorrecto p=4; equivale m=4; */
printf("%d %p", *p, p);        /* contenido y la dirección de m*/

q = &n;                        /* q recibe la dirección de n */
z= *q;                         /* equivale z = n; */
printf("%d %d", *p, *q);

p = q;
printf("%d %d", *p, *q);
```

FUNCIONES

- ✓ Todo programa en C está formado por una o mas funciones (al menos *main()*).
- ✓ No se pueden definir funciones anidadas.
- ✓ Desde una función se puede invocar a otras, inclusive a ella misma (recursividad).
- ✓ **El mecanismo de paso de parámetros es por valor.**

Prototipos

El **prototipo** de una función es una declaración previa a la definición de dicha función. El prototipo sirve exclusivamente para informarle al compilador el tipo y el número de los argumentos, así como el tipo de dato a devolver.

Los prototipos no son obligatorios en C, sin embargo, son aconsejables ya que facilitan la comprobación de errores en la invocación respecto de la definición correspondiente.

La declaración de la cabecera debe anteceder a la propia definición de la función.

☞ Sintaxis: *tipo_dato* nombreFuncion (*lista_tipo_argumentos*);

dónde *tipo_dato* es el tipo de dato devuelto por la función (en caso de obviarse el mismo se toma el tipo **int** por defecto). Para evitar malas interpretaciones es conveniente explicitarlo.

Cuando la función no devuelve ningún valor ("procedimiento"), se indica por medio de la palabra reservada **void** (sin valor).

lista_tipo_argumentos representa los tipos de datos de los argumentos que pueden ir acompañados o no de los nombres de los argumentos. Una función que no necesita argumentos se informa mediante la palabra **void** puesta entre los paréntesis.

☞ **Ejemplo**

```
int maximo(int a, int b);
void saludo();      /*  o void saludo (void);  */
```

Definición de funciones

La **definición** de una función puede ubicarse en cualquier lugar del programa, con sólo dos restricciones: debe hallarse después de dar su prototipo, y no puede estar dentro de la definición de otra función (incluida main()). Es decir, a diferencia de Pascal, en C las definiciones no pueden anidarse.

La definición debe comenzar con un encabezamiento, que debe coincidir totalmente con el prototipo declarado para la misma, y a continuación se escribirán las sentencias que componen el cuerpo de la función que debe ir encerrado por llaves. En la definición de una función, el encabezamiento no termina con un punto y coma.

☞ **Ejemplo**

```
int maximo(int a, int b){
    Sentencias
}
void saludo(){
    Sentencias
}
```

Salida (Sentencia return)

Para salir de una función se utiliza la sentencia **return** que produce la salida inmediata de la función hacia la sentencia que la invocó. La sentencia puede devolver un valor acorde con el tipo usado al declarar la función.

En las funciones de tipo **void**, debe obviarse la sentencia **return**, y el control vuelve cuando se alcanza al final de la función.

Si la sentencia **return** incluye una expresión, el valor de la expresión se convierte (si es necesario) al tipo de la función, y ese valor se devuelve al medio que hizo la llamada.

☞ **Ejemplo**

```
int maximo(int a, int b){
    return (a>b)? a : b;
}
void saludo(){
    printf("Bienvenido");
}
```

Invocación de una función

En el cuerpo de alguna otra función se produce la invocación, cuyos *parámetros actuales* deben coincidir en tipo y cantidad con los *parámetros formales* del prototipo de la función. Si los parámetros actuales fueran expresiones, se evalúan antes de copiarlos en los formales.

☞ **Ejemplo**

```
int main(){
int a,b;
  scanf("%d %d",&a, &b);
  saludo();
  printf("El máximo de los valores leidos es %d", maximo(a,b));
  return 0;
}
```

.

Los parámetros de una función pueden ser valores (**pasaje por valor**) o direcciones (**pasaje por referencia**)

En el pasaje de parámetros por **Valor** las modificaciones en los valores de los parámetros formales de la función no afectan a las variables/constantes/expresiones del medio que hizo la llamada.

Si se quiere conseguir el efecto lateral de afectar a las variables del medio de llamada, hay que inducir el paso por **Referencia**, pasando direcciones de variables (**&**) en vez de sus valores. Entonces, se usan los parámetros formales desreferenciados (*****) dentro de la función.

PASAJE POR VALOR	PASAJE POR REFERENCIA (DIRECCION)
`void doble(int i) {` `  i *= 2;` `}`	`void doble(int *i){` `  *i *=2;` `}`
`void main() {` `  int n=5;` `  printf("%d\n", n);  // 5` `  doble(n);` `  printf("%d\n", n);  // 5` `}`	`void main() {` `  int n=5;` `  printf("%d\n", n); // 5` `  doble(&n);` `  printf("%d\n", n); //10` `}`

Agrupamiento de los prototipos

Es recomendable agrupar todos los prototipos de las funciones de un programa en uno o varios archivos de cabecera (**.h**) que luego se incluirán como encabezado en los programas que hagan referencia a las funciones allí declaradas.

Visibilidad de las funciones

Una función declarada en un archivo tiene un alcance global; es decir, puede ser invocada desde cualquier punto dentro este archivo. Cuando se quiere acceder a ella desde otro archivo, hay que incluir en este último la declaración (archivo de cabecera) de dicha función.

Si se quisiera restringir el uso de la función a un solo módulo, se usa el modificador **static** en su declaración.

CADENAS

Una cadena de caracteres es un arreglo de caracteres terminado en el carácter nulo (\0).
Como ya hemos visto **gets()** y **puts**() pueden utilizarse para leer y escribir cadenas sin formato. O **scanf()** y **printf()** con el formato %s.

Cadenas constantes

Una cadena constante es una secuencia de caracteres encerrada entre comillas dobles.
Para inicializar una cadena constante:

```
char cad[ ] = ”Hola”;                    /* Ocupa 5 bytes */
```

equivale a:

```
char cad[ ] = {‘H’,‘o’,‘l’,‘a’,‘\0’};    /* Ocupa 5 bytes */
```

equivale a:

```
char *cad = ”Hola”;                      /* Ocupa 5 bytes */
```

Ingreso/Asignación de cadenas

Es necesario reservar espacio para una cadena, un modo de hacerlo es en la declaración de la variable.

```
#define SIZE 30
char cadena1[SIZE], *cadena2;
scanf(“%s”, cadena1);                    /* Correcto */
scanf(“%s”, cadena2);                   /* Incorrecto */
strcpy(cadena2, cadena1);                /* Incorrecto */
```

Algunas Librerías

- **stdio.h** : *algunas funciones* son **puts**(), **gets**(), **printf**(), **scanf**().
- **string.h** : *algunas funciones* son
  - ✓ **strlen**(*cad*) : devuelve la longitud de la cadena *cad* sin el \0.
  - ✓ **strcpy**(*cad1*, *cad2*) : copia el contenido de cad2 en cad1. Sería incorrecto cad1 = cad2. Devuelve un char *. Variante: **strncpy**(*cad1*, *cad2, n*).
  - ✓ **strcat**(*cad1*, *cad2*) : concatena el contenido de cad2 en cad1 (sin controlar que haya espacio). Devuelve un char *.
  - ✓ **strcmp**(*cad1*, *cad2*) : compara cad1 con cad2. Sería incorrecto. cad1 != cad2. Retorna -1 (<0), 0, 1 (>0). Variante: **strncmp**(*cad1*, *cad2, n*).

ARREGLOS

Declaración

☞ Sintaxis: *tipo_dato* nombreVector[*componentes*];
tipo_dato nombreMatrizr[*filas*] [*columnas*];

☞ **Ejemplos de declaración de variables y tipos**

```
int vector[3];                  /* posición 0 a la 2 */
int matriz[2][3];               /* la primera posición es la [0][0] */

typedef int Tvector[10];        /* Tvector es un tipo */
typedef int Tmatriz[10][10];    /* Tmatriz es un tipo */

Tvector v;
Tmatriz mat;
```

Inicialización

En la declaración, es posible asignarle un valor a los elementos.

☞ **Ejemplo de declaración con inicialización**

```
int V[3] = {4, 5, 6};
int Mat[2][3] = { {1,2,3}, {4,5,6} };
```

El compilador puede deducir las dimensiones del arreglo:

```
int v[] = {1, 2, 3, 5, 7};
```

Cuando se pasa un vector como parámetro, el paso de parámetros es por dirección (un vector es, en realidad, un puntero en C). Por tanto, se debe tener cuidado con los efectos colaterales que se producen si, dentro de un módulo, se modifica un vector que es parámetro.

☞ **Ejemplo de utilización**

```
#include <stdio.h>
void doble(int [], int );

void main() {
    int i, v[] = {1,2,3,4,5}; //se deduce su dimensión física
    doble(v,5);
    for(i=0; i<5; i++)
       printf("%d \t ",v[i]);
 }
void doble(int v[], int n){
  int i;
  for(i=0; i<n; i++)
    v[i] *= 2;
}
```

Las cadenas, arreglos y matrices no pueden pasarse por valor

REGISTROS

Declaración

☞ Sintaxis: **struct** *nombre_estructura* {
tipo_variable nombre_variable1;
tipo_variable nombre_variable2;
tipo_variable nombre_variable3; };

```
struct carta {
     int nro;
     char palo; };
```

Para definir variables de tipo **struct**:

```
struct carta {
     int nro;
     char palo; } c1, c2;

typedef struct {
         int nro;
         char palo; } Tcarta;
Tcarta c1, c2;
```

Operación con estructuras y campos

```
c1.palo='O';
c1.nro=2;
c2=c1;
```

Punteros a estructuras

```
typedef struct {
    char palo;
    int nro; } Tcarta;

 void GeneraCarta (Tcarta *);
 Tcarta GeneraCarta (int, char);  //sobrecargado
 void muestra (Tcarta);

 void main() {
    Tcarta c;
    GeneraCarta(&c); //o c=GeneraCarta(8,'O');
    muestra(c);
}
void GeneraCarta (Tcarta * c){
  scanf("%c %d ", &c->palo, &(*c).nro);
  //scanf("%c %d ", &(c->palo), &((*c).nro));

}

Tcarta GeneraCarta (int n, char p){
  Tcarta caux;
  caux.palo=p;  caux.nro=n;
  return caux;
}

void muestra (Tcarta c){
  printf("%c  %d", c.palo, c.nro);
 }
```

PUNTEROS y MEMORIA DINAMICA

Ya vimos que un puntero es un tipo de datos que almacena una dirección de memoria.
Existe una dirección especial que se representa por medio de la constante NULL (definida en <**stdlib.h**>) y se emplea cuando se quiere indicar que un puntero no apunta a ninguna dirección.

Correspondencia entre punteros y vectores

Al declarar un vector, se reserva memoria para almacenar elementos del mismo tipo. Y como ya hemos visto, el identificador es un puntero.

☞ **Ejemplo**

```
int v[3], *ptr;
...
ptr = v;                        // ptr = &v[0];
v[0] = 6;                       // *v = 6;  *ptr = 6;
```

Aritmética de punteros

C permite sumar (o restar) a un puntero un valor entero, desplazando la dirección tantos elementos como se indique. Si se declara

*int *ptr*; entonces *ptr + x*, devuelve un puntero a la posición de memoria *sizeof(int)*x* bytes desde *ptr.*

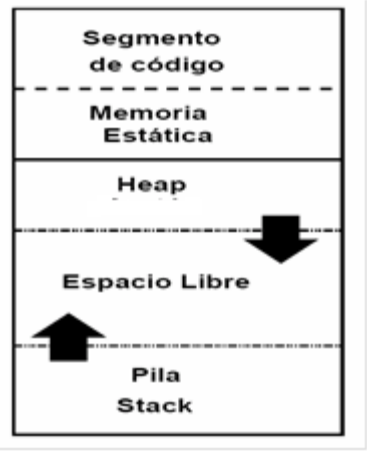
Esto tiene sentido cuando el puntero es la dirección de un arreglo y se permite así el desplazamiento dentro del mismo. C no chequea que el valor del puntero sea válido (apunte a una posición válida dentro del arreglo).

☞ **Ejemplo**

```
int v[10];
int *ptr = v;
          entonces ptr+i apunta a v[i] y *(ptr+i) ≅ v[i]
          ptr++ permite acceder a v[1]
          v++ es inválido, pues es la dirección constante del arreglo
```

Se pueden restar punteros, esto tiene sentido cuando ambos apuntan al mismo arreglo y permite obtener la diferencia entre dos posiciones. No se pueden sumar direcciones de memoria.

GESTION DINAMICA de la MEMORIA



Organización de la memoria

- Segmento de código (código del programa).
- Memoria estática (variables globales y estáticas).
- Pila (stack): Variables (locales), recursividad.
- Heap (montículo): Memoria dinámicas

Reserva y liberación de memoria

Cuando se quiere utilizar el *heap*, primero hay que reservar la memoria que se desea ocupar.
Existen dos posibilidades: función **malloc** (ANSI C) y operador **new** (C++)

Tras utilizar la memoria gestionada dinámicamente, hay que liberar el espacio reservado: función **free** (ANSI C) y operador **delete** (en C++)
Si no se libera la memoria, ese espacio de memoria no podrá volverse a utilizar.

☞ Sintaxis: *void **malloc** (tamaño)*;

La función **malloc** (se encuentra en la librería **stdlib.h**) asigna una región de memoria para un elemento de *tamaño* bytes, y devuelve la dirección del primer byte de esa región.
Los valores almacenados en ese bloque de datos quedan indefinidos.
Se garantiza que la zona de memoria concedida no está ocupada por ninguna otra variable ni otra zona asignada por **malloc**. Si no es posible conceder el bloque (p.ej. no hay memoria suficiente), devuelve un puntero NULL.
La creación del objeto y asignación de memoria se hace en tiempo de ejecución.

*Punteros void**

La función *malloc* devuelve un puntero inespecífico (no apunta a un tipo de datos determinado). En C, estos punteros sin tipo se declaran como *void**. Muchas funciones que devuelven direcciones de memoria utilizan los punteros *void**, éstos pueden convertirse a cualquier otra clase de puntero:

☞ **Ejemplo**

```
char *pch = (char *) malloc (100);  //cadena dinámica o vector ?
int *pint = (int *) malloc(sizeof(int));  //int *pint =(int *)malloc(2);
float *pf;
pf = (float *) malloc(sizeof(float);
```

sizeof

La función *malloc* necesita conocer cuántos bytes se quieren reservar. El tamaño en bytes de un elemento de tipo T se obtiene con la expresión *sizeof* (*T*).

☞ Sintaxis: *void* **free** *(puntero)*;

Se libera el bloque correspondiente al elemento cuya dirección indica *puntero*. Si el puntero tiene valor nulo, no hace nada.

☞ **Ejemplo**

```
int * ptr;
ptr = (int*) malloc (sizeof(int));
free(ptr);
ptr = (int*) malloc (30*sizeof(int));
free (ptr);
```

EJERCICIO

Analizar el siguiente código considerando posibles errores y efectos no deseados

```
typedef struct {
   char palo;
   int nro; }Tcarta;
void main() {
   int * Pt1, * Pt2, X=10;
   Tcarta * PtCarta;
   char * cad;
Pt1 = (int *) malloc(sizeof(int));
scanf("%d",Pt1);
Pt2 = &X;                    //No es manejo de memoria dinámica
printf("\n%d  %d", X, *Pt2);
*Pt2= *Pt1 + 4;
printf("\n%d  %d",*Pt1, *Pt2);
Pt2 = Pt1;
printf("\n%d  %d\n",*Pt1, *Pt2);
free(Pt2); free(Pt1);

PtCarta =(Tcarta *)malloc (sizeof(Tcarta));
scanf("%d %c",&(PtCarta->nro), &((*PtCarta).palo));
printf("\n%d  %c\n",(*PtCarta).nro, PtCarta->palo);

cad = (char *)malloc(30*sizeof(char));
scanf("%s",cad);
printf("\n%c  %c",cad[0], cad[1]);

free(cad);  free(PtCarta);  // cad=NULL?
}
```

ESTRUCTURAS DINÁMICAS

Las estructuras dinámicas se construyen con punteros. Un componente del elemento es un puntero que "apunta" a otro elemento del mismo tipo.

Se consigue así, un conjunto de elementos relacionados cuya cantidad puede aumentar y disminuir en tiempo de ejecución.

Las estructuras dinámicas se manejan con definiciones de datos recursivas.

☞ **Ejemplo**

```
typedef struct nodo {
    char dato;
……..
    struct nodo *sig;} nodo;

typedef nodo * Tlista;

Tlista L;  //vble de tipo puntero a nodo
```

ARCHIVOS

Los archivos, en contraposición con las estructuras de datos vistas hasta ahora (variables simples, vectores, registros, etc.), son estructuras de datos almacenadas en memoria secundaria.

El formato de declaración de un archivo es:

FILE * *nombre_archivo*;

En otros lenguajes la declaración del archivo determina (es determinada por) el tipo de datos que se van a almacenar en él. En C la filosofía es distinta, todos los archivos almacenan bytes y es cuando se realiza la apertura y la escritura cuando se decide cómo y qué se almacena en el mismo; durante la declaración del archivo no se hace ninguna distinción sobre el tipo del mismo.

En la operación de apertura se puede decidir si el archivo va a ser de texto o binario, los primeros sirven para almacenar caracteres, los segundos para almacenar cualquier tipo de dato.
Si deseamos leer un archivo como DATOS.TXT utilizaremos uno de texto, si queremos leer y escribir registros (struct) usaremos uno binario como DATOS.DAT.

Apertura y Cierre
Antes de usar un archivo es necesario realizar una **operación de apertura** del mismo; posteriormente, si se desea almacenar datos en él hay que realizar una **operación de escritura** y si se quiere obtener datos de él es necesario hacer una **operación de lectura**. Cuando ya no se quiera utilizar el archivo se realiza una **operación de cierre** del mismo para liberar parte de la memoria principal que pueda estar ocupando (aunque el archivo en sí está almacenado en memoria secundaria, mientras está abierto ocupa también memoria principal). Las funciones de archivos se encuentran en *stdio.h*
La instrucción para abrir un archivo es ***FILE** * archivo es:*
archivo = ***fopen** (nombre-archivo, modo);*
La función ***fopen*** devuelve un puntero a un archivo que se asigna a una variable de tipo FILE *. Si existe algún **tipo de error** al realizar la operación, por ejemplo, porque se desee abrir para leerlo y éste no exista, devuelve el valor **NULL.**
El *nombre-archivo* será una cadena de caracteres que contenga el nombre (y en su caso la ruta de acceso) del archivo tal y como aparece para el sistema operativo.
El *modo* es una cadena de caracteres que indica el tipo del archivo (texto o binario) y el uso que se va a hacer de él lectura, escritura, añadir datos al final, etc.
Los modos disponibles son:

Modo	Significado
"r"	abre un archivo para lectura. Si el archivo no existe devuelve error
"w"	abre un archivo para escritura. Si el archivo no existe se crea, y si existe se destruye y se crea uno nuevo
"a"	abre un archivo para añadir datos al final del mismo. Si no existe se crea.
+	símbolo utilizado para abrir el archivo para lectura y escritura
b	Indica que el archivo es de tipo binario
t	Indica que el archivo es de tipo texto (por defecto)

Los modos anteriores se combinan para conseguir abrir el archivo en el modo adecuado.
La forma habitual de utilizar la instrucción **fopen** es dentro de una sentencia condicional que permita conocer si se ha producido o no error en la apertura.
Declarando `FILE *arch;` es posible abrir el archivo de algunas de las siguientes formas:

```
arch=fopen ("datos.dat", "rb+"); /* archivo binario ya existente para L/E */
arch=fopen ("datos.txt", "w+");  /* archivo nuevo de texto, si existe se
                                    borra su contenido, para L/E */
arch=fopen ("datos.txt", "a");   /* agregar al final */
if ((arch=fopen("nomarch.dat", "r"))==NULL)
   printf("Error en la apertura. Es posible que el archivo no exista");
```

Cuando se termine el tratamiento del archivo hay que cerrarlo: **fclose**(*nombre_archivo*);

Lectura y Escritura
Para almacenar datos en un archivo es necesario realizar una operación de escritura, de igual forma que para obtener datos hay que efectuar una operación de lectura. En C existen muchas y variadas operaciones para leer y escribir en un archivo; entre ellas:
fread - fwrite, **fgetc - fputc**, **fgets - fputs** → arch binarios
fscanf – fprintf → arch de texto.

Para leer todos los datos de un archivo basta con realizar lecturas sucesivas hasta que **se lee el final del archivo**. La función ***feof** (archivo)* devuelve un valor distinto de 0 cuando se ha consumido la marca EOF.

Acceso Directo a los datos

Cuando se puede acceder a cualquier dato de un archivo sin tener que pasar por anteriores se está realizando un acceso directo a los datos.

La función que permite situarse en un determinado dato del archivo es **fseek** (*archivo*, *posicion*, *origen*); que coloca el puntero del archivo a tantos bytes del origen como indica posición (positivo o negativo) contando a partir del origen señalado.

Valores posibles para *origen*:

SEEK_SET o 0 : principio del archivo.
SEEK_CUR o 1 : posición actual.
SEEK_END o 2 : final del archivo.

En un archivo *f* que almacene números enteros se podría:

```
fseek(f,0,SEEK_SET); // coloca el puntero al principio de f
```

o

```
fseek(f,3*sizeof(int),SEEK_CUR); /* coloca el puntero 3 posiciones más
                                    allá de la posición actual del puntero */
```

Para saber cuál es la posición en la que está el puntero del archivo, C proporciona la función: **ftell** (*archivo*); que devuelve la posición actual en bytes del puntero del archivo con respecto al principio del mismo. Esta función se utiliza sobre archivos binarios. La sentencia `ultimo=ftell(f);` almacena en *ultimo* la cantidad de bytes de *f.*

Cuando un archivo se pasa como parámetro a una función, siempre debe pasar cerrado y el modo de pasarlo es `FILE * arch`

EJERCICIO

Dado un archivo de texto que contiene datos de clientes de un comercio, en cada línea: nombre (cadena de 12), sexo (carácter) y edad.

Obtener y mostrar la edad promedio de los hombres y guardar los datos de los hombres (nombre y edad) en un archivo de registros.

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

typedef struct {
  char nombre[12];
  int edad;} Treg;

void main()   {
   FILE * archt, *archb;
   int cant, suma, edad;
   char sexo, nom[12];
   Treg reg;
```

```
 archt = fopen("Clientes.txt","rt");
 if (archt == NULL)
    printf("archivo clientes no existe!");
  else { //1
    archb = fopen("Hombres.dat","wb");
    cant =suma = 0;
   fscanf(archt,"%s %c %d\n ", nom,&sexo, &edad);
   while(!feof(archt)) {
     if (sexo == 'M') {
        cant++;
        suma+=edad;
        strcpy(reg.nombre, nom);
        reg.edad=edad;
        fwrite(&reg, sizeof(Treg), 1, archb);
     }
    fscanf(archt,"%s %c %d \n", nom,&sexo,&edad);
  }

  if (cant)
       printf("Promedio %i", suma/cant);
 fclose(archt); fclose(archb);

 if ((archb = fopen("Hombres.dat","rb"))==NULL)
      printf("archivo de clientes hombres no existe!");
 else { //2
   fread(&reg, sizeof(Treg), 1, archb);
   while(!feof(archb)) {
       printf("%s %d", reg.nombre, reg.edad);
       fread(&reg, sizeof(Treg), 1, archb);
    } //del while
   fclose(archb);
} //del else 2
}//del else 1
}
```

ALTERNATIVA AL CICLO DE LECTURA del ARCH de TEXTO

```
    while( fscanf(archt,"%s%c %d \n", nom, &sexo,&edad) ==3)
     if (sexo == 'M') {
        cant++;
        suma+=edad;
        strcpy(reg.nombre, nom);
        reg.edad=edad;
        fwrite(&reg, sizeof(Treg), 1, archb);
     }
```

ALTERNATIVA A LA APERTURA del ARCH BINARIO

```
archb = fopen("Hombres.dat","wb+");
      //LUEGO EL ARCH NO DEBE CERRARSE SINO
fseek(archb,0,SEEK_SET);  //o fseek(archb,0,0);
```