



Departamento
de Ingeniería de la
Información y las
Comunicaciones

UMU

TECNOLOGÍA DE LA PROGRAMACIÓN

TÍTULO DE GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023/24

GRUPO 4

TEMA 0. INTRODUCCIÓN A C

Dept. Ingeniería de la Información y las Comunicaciones

Universidad de Murcia

TEMA 0. INTRODUCCIÓN A C

- 0.1 Introducción
- 0.2 Manipulación básica de datos
- 0.3 Sentencias
- 0.4 Funciones
- 0.5 Ámbito y extensión
- 0.6 Arrays y cadenas de caracteres
- 0.7 Apuntadores
- 0.8 Asignación dinámica de memoria
- 0.9 Ficheros
- 0.10 Gestión de errores
- 0.11 El preprocesador de C
- 0.12 Programación modular
- 0.13 Ejercicios resueltos
- 0.14 Ejercicios propuestos

0.1 INTRODUCCIÓN

0.1.1 Historia del lenguaje y características

0.1.2 Fases de desarrollo

0.1.3 Componentes del lenguaje

0.1.4 Estructura de un programa

0.1.5 Comentarios

0.1.6 Bibliotecas estándares

0.1.7 Ejemplo: "Hola Mundo"

0.1.1 HISTORIA DEL LENGUAJE Y CARACTERÍSTICAS (1/2)

C es un lenguaje de programación creado en 1972 por *Dennis M. Ritchie* en los Laboratorios Bell orientado a la implementación del Sistema Operativo *Unix*.

La primera definición estándar del lenguaje la hizo *ANSI* en 1989. Posteriormente, en 1990, fue ratificado como estándar *ISO (ISO/IEC 9899:1990)*.

A mediados de los ochenta se crea una extensión del lenguaje C orientado a objetos, C++, el cual se convierte en estándar *ISO* en 1998. Tras el proceso de estandarización de *ANSI*, la especificación del lenguaje C permaneció relativamente estable durante algún tiempo, mientras que C++ siguió evolucionando. Sin embargo, el estándar continuó bajo revisión hasta la publicación del estándar *ISO 9899:1999*. Este estándar se denomina habitualmente C99. Se adoptó como estándar *ANSI* en marzo de 2000. Posteriormente se han publicado distintos estándares hasta llegar finalmente al *ISO/IEC 9899:2018 (C18)*. Para más detalles sobre el desarrollo del lenguaje C ver <https://www.bell-labs.com/usr/dmr/www/chist.html>.

Actualmente el lenguaje C no está sujeto a más modificaciones, ya que es el lenguaje C++ el que incorporará nuevos cambios. En este tema se describe el lenguaje C definido en el estándar C99 con algunas extensiones de *GNU*.

0.1.1 HISTORIA DEL LENGUAJE Y CARACTERÍSTICAS (2/2)

Características:

- Combina la abstracción de lenguajes de alto nivel con la eficiencia del lenguaje máquina, por lo que está considerado como un lenguaje de “*medio nivel*”.
- *Débilmente tipado*: Conversiones implícitas de tipos.
- No permite *encapsulado*, aunque sí algún mecanismo rudimentario para simularlo.
- Permite programar con múltiples estilos, siendo el estilo estructurado el más empleado, si bien permite ciertas licencias de ruptura.

Ventajas:

- Altamente transportable.
- Muy flexible.
- Genera código eficiente.
- Muy expresivo.

Inconvenientes:

- Poco modular.
- Realiza pocas comprobaciones.
- Da poca disciplina al programador.
- Resulta difícil de interpretar el código escrito por otros programadores.

0.1.2 FASES DE DESARROLLO

Preprocesador

Transforma el código fuente (**ficheros *.c y *.h**):

- Elimina los comentarios.
- Incluye el contenido de otros ficheros fuente.
- Sustituye las macros.
- Incluye código de forma condicional.

Compilador

Analiza léxica, sintáctica y semánticamente el código fuente transformado por el preprocesador y lo convierte en código objeto: **ficheros *.o**

Enlazador

A partir de los códigos objeto y las bibliotecas genera el ejecutable binario: **fichero *.exe** (Windows).

0.1.3 COMPONENTES DEL LENGUAJE

Elementos léxicos

- Identificadores, palabras clave, literales, operadores, separadores.

Tipos de datos

- Tipos básicos: `int`, `float`, `double`, `char`, y modificadores: `unsigned`, `short`, `long`
- Tipos de datos definidos por el programador:
 - Enumerados: `enum`
 - Uniones: `union`
 - Tipos estructurados: `struct`
- Arrays y apuntadores
- Tipos incompletos
- Especificadores de almacenamiento
- Nombrado de tipos: `typedef`

Sentencias

- Expresiones (pueden incluir llamadas a funciones)
- Condicionales: `if`, `switch`
- Iterativas: `while`, `for`, `do...while`
- Ruptura del flujo de ejecución: `break`, `return`, `continue`, `goto`
- Bloques (sentencias compuestas)

Funciones

- Funciones (declaración y definición)

Otros componentes (tratados por el preprocesador)

- Comentarios, inclusión de ficheros, macros, compilación condicional

0.1.4 ESTRUCTURA DE UN PROGRAMA

La estructura típica de un programa en C es la siguiente:

```
// Inclusión de ficheros
// Definición de macros
// Nombrado de tipos
// Declaración de variables globales
// Declaración de funciones
// Definición de funciones
int main()
{
    ...
    return 0;
}
```


0.1.5 COMENTARIOS

Los comentarios se utilizan para explicar alguna parte del código del programa. Se pondrán antes del elemento del programa a comentar. Podrán ser de dos tipos:

Comentarios de una línea (sólo c99):

Se utilizan para establecer un comentario que ocupa una única línea.

```
// Ejemplo de comentario en una línea
```

Comentarios de más de una línea:

```
/* Ejemplo de comentario  
   que tiene más de  
   una línea.  
*/
```

0.1.6 BIBLIOTECAS ESTÁNDARES

El estándar de C contiene un conjunto de bibliotecas que ofrecen recursos los cuales permiten al programador utilizar servicios elementales no contemplados en el lenguaje C.

Las interfaces de estos servicios se definen en sus correspondientes **ficheros de cabecera** (* .h) (sección 0.12).

Ejemplos:

- Entrada y salida de datos: `stdio.h`
- Manejo de cadenas: `string.h`
- Memoria dinámica: `stdlib.h`
- Rutinas matemáticas: `math.h`

0.1.7 EJEMPLO: “HOLA MUNDO”

```
/* Incluye la biblioteca estándar stdio.h para poder utilizar la
función printf
*/
#include <stdio.h>

// Punto de inicio del programa
int main()
{
    /* La función printf escribe en pantalla la cadena de
    caracteres Hola Mundo. La secuencia \n produce que el
    siguiente texto que se imprima lo haga en una nueva línea
    */
    printf("Hola Mundo\n");
    return 0;
}
```

0.2 MANIPULACIÓN BÁSICA DE DATOS

0.2.1 Literales

0.2.2 Tipos básicos

0.2.3 Identificadores

0.2.4 Declaración de variables

0.2.5 Expresiones y operadores

0.2.6 Operadores de asignación

0.2.7 Variables de sólo lectura: `const`

0.2.8 Ejemplo de manipulación básica de datos

0.2.9 Desbordamientos y redondeos

0.2.10 Conversión de tipo

0.2.11 Estructuras: `struct`

0.2.12 Enumeraciones: `enum`

0.2.13 Uniones: `union`

0.2.14 Renombrado de tipos: `typedef`

0.2.15 Tipos incompletos

0.2.1 LITERALES

También denominados constantes, son datos escritos directamente en el código del programa. Pueden ser enteros, reales, caracteres o cadenas de caracteres:

Enteros

- Un literal entero consiste en una secuencia de dígitos con un prefijo opcional para denotar una base. Ejemplos: 34 (base decimal por defecto), 0x3A5F (base hexadecimal), 0451 (base octal).

Reales

- Un literal real consiste en una secuencia de dígitos que representa la parte entera del número, un punto decimal, y una secuencia de dígitos que representa la parte fraccional. Se puede omitir la parte entera o la fraccional, pero no ambas. Pueden estar seguidas por e o E y un exponente entero positivo o negativo. Ejemplos: 25.73, 25., .73, 38e2, 39.5e-3, 5E4

Caracteres

- Un literal carácter consiste en un carácter entre comillas simples. Un literal carácter es de tipo entero. Algunos caracteres requieren secuencias de escape para ser representados. Ejemplos: 'a', '\'', '\n', '\t'

Cadenas de caracteres

- Un literal cadena de caracteres (o string) consiste en una secuencia de cero o más caracteres, dígitos y secuencias de escape entre comillas dobles. Un literal string es de tipo array de caracteres (sección 0.6.2). Ejemplos: "hola", "\"hola\"", "hola\n"

0.2.2 TIPOS BÁSICOS (1/3)

Todos los tipos son numéricos.

También existen modificadores que acompañan a un tipo de datos básico para cambiar su rango.

Tipo	Significado
char	Carácter
int	Entero
float	Real
double	Real doble

Modificador	Significado
short	Corto
long	Largo
unsigned	Sin signo
signed	Con signo

Existe un tipo `void` para indicar la inexistencia de tipo.

Los rangos pueden ser mayores o menores dependiendo de la versión del compilador y de la arquitectura del procesador.

Se realizará una sesión de prácticas dedicada a los tipos, modificadores y rangos.

0.2.2 TIPOS BÁSICOS (2/3)

```
#include <stdio.h>
#include <limits.h>
#include <float.h>
#define printf __mingw_printf
int main(){
    printf("Tipo: bytes [min,max]\n");
    printf("-----\n");
    printf("char: %zu byte [%d,%d]\n",sizeof(char),CHAR_MIN,CHAR_MAX);
    printf("signed char: %zu byte [%d,%d]\n",sizeof(signed char),SCHAR_MIN,SCHAR_MAX);
    printf("unsigned char: %zu byte [%d,%u]\n",sizeof(unsigned char),0,UCHAR_MAX);
    printf("int: %zu bytes [%d,%d]\n",sizeof(int),INT_MIN,INT_MAX);
    printf("short int: %zu bytes [%hd,%hd]\n",sizeof(short int),SHRT_MIN,SHRT_MAX);
    printf("long int: %zu bytes [%ld,%ld]\n",sizeof(long int),LONG_MIN,LONG_MAX);
    printf("long long int: %zu bytes [%lld,%lld]\n",sizeof(long long int),LLONG_MIN,LLONG_MAX);
    printf("unsigned short int: %zu bytes [%d,%hu]\n",sizeof(unsigned short int),0,USHRT_MAX);
    printf("unsigned int: %zu bytes [%d,%u]\n",sizeof(unsigned int),0,UINT_MAX);
    printf("unsigned long int: %zu bytes [%d,%lu]\n",sizeof(unsigned long int),0,ULONG_MAX);
    printf("unsigned long long int: %zu bytes [%d,%llu]\n",sizeof(unsigned long long
        int),0,ULLONG_MAX);
    printf("float: %zu bytes [%.4e,%.4e],0,[%.4e,%.4e]\n",sizeof(float),-FLT_MAX,-FLT_MIN,
        FLT_MIN,FLT_MAX);
    printf("double: %zu bytes [%.4e,%.4e],0,[%.4e,%.4e]\n",sizeof(double),-DBL_MAX,-DBL_MIN,
        DBL_MIN,DBL_MAX);
    printf("long double: %zu bytes [%.4Le,%.4Le],0,[%.4Le,%.4Le]\n",sizeof(long double),-LDBL_MAX,-
        LDBL_MIN,LDBL_MIN,LDBL_MAX);
    // NOTA: SE PUEDE PONER le o Le, pero Le da problemas en algunas plataformas
    printf("-----\n");
}
```

0.2.2 TIPOS BÁSICOS (3/3)

```
Tipo: bytes [min,max]
-----
char: 1 byte [-128,127]
signed char: 1 byte [-128,127]
unsigned char: 1 byte [0,255]
int: 4 bytes [-2147483648,2147483647]
short int: 2 bytes [-32768,32767]
long int: 4 bytes [-2147483648,2147483647]
long long int: 8 bytes [-9223372036854775808,9223372036854775807]
unsigned short int: 2 bytes [0,65535]
unsigned int: 4 bytes [0,4294967295]
unsigned long int: 4 bytes [0,4294967295]
unsigned long long int: 8 bytes [0,18446744073709551615]
float: 4 bytes [-3.4028e+38,-1.1755e-38],0,[1.1755e-38,3.4028e+38]
double: 8 bytes [-1.7977e+308,-2.2251e-308],0,[2.2251e-308,1.7977e+308]
long double: 16 bytes [-1.1897e+4932,-3.3621e-4932],0,[3.3621e-4932,1.1897e+4932]
-----
```


0.2.3 IDENTIFICADORES (1/3)

Un *identificador* es una secuencia de caracteres usado para nombrar una variable, función, un nuevo tipo de datos definido por el programador o una macro del preprocesador.

Los identificadores deben seguir las siguientes **reglas**:

- Un identificador debe empezar por una letra o por el carácter de subrayado seguido de cualquier cantidad de letras, dígitos o subrayados.
- Se distinguen mayúsculas y minúsculas.
- No se pueden utilizar *palabras reservadas* por el compilador como `int`, `char`, `while`, etc.
- Muchos compiladores no permiten letras acentuadas o eñes.

0.2.3 IDENTIFICADORES (2/3)

Palabras reservadas reconocidas por ANSI C89:

```
auto break case char const continue default do double else  
enum extern float for goto if int long register return short  
signed sizeof static struct switch typedef union unsigned void  
volatile while
```

ISO C99 añade las siguientes palabras reservadas:

```
inline _Bool _Complex _Imaginary
```

0.2.3 IDENTIFICADORES (3/3)

Ejemplos de identificadores **válidos**:

```
d, f, c, character, Character, _character, character1, character2
```

Ejemplos de identificadores **no válidos**:

```
Carácter    // El compilador no admite las tildes  
muñeco      // El compilador no admite la ñ  
4c          // No puede empezar por número  
int         // Palabra reservada int
```

0.2.4 DECLARACIÓN DE VARIABLES

Una *variable* está formada por un nombre simbólico (identificador) y un espacio de almacenamiento asociado, el cual contiene un valor.

En C cada variable tiene un tipo. Las variables hay que declararlas como pertenecientes a su tipo antes de utilizarlas.

Sintaxis:

tipo identificador;

Ejemplos de declaraciones válidas:

```
int d;           // Se declara la variable d de tipo int
double f;        // Se declara la variable f de tipo double
char character;  // Se declara la variable character de tipo char
char Character;  // Se declara la variable Character de tipo char
```

0.2.5 EXPRESIONES Y OPERADORES (1/7)

Una *expresión* es una combinación de constantes, variables y funciones, que es interpretada de acuerdo a las normas particulares de *precedencia* y *asociación*. La *evaluación de una expresión* permite realizar las operaciones establecidas por los *operadores* sobre los valores de los *operandos*, obteniendo un valor numérico. Una expresión consiste en, al menos, un operando y cero o más operadores.

Los *operandos* pueden ser:

- literales
- variables
- llamadas a funciones que devuelven valores (sección 4.2)

Los *operadores* pueden ser:

- aritméticos: expresiones aritméticas
- relacionales: expresiones lógicas (relacionales)
- lógicos: expresiones lógicas

0.2.5 EXPRESIONES Y OPERADORES (2/7)

Las *expresiones aritméticas* combinan *operadores aritméticos* junto con operandos para calcular valores numéricos.

Operador Aritmético	Significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (Resto)

0.2.5 EXPRESIONES Y OPERADORES (3/7)

Las *expresiones lógicas* combinan *operadores relacionales* y/o *lógicos* junto con operandos para obtener valores numéricos. Esto es así dado que **C no tiene tipo booleano. C interpreta como falso el valor 0 y como cierto cualquier otro valor distinto de cero.**

El resultado de la evaluación de una expresión lógica es un número entero igual a **1**, si la condición es cierta, y a **0** si la condición es falsa.

0.2.5 EXPRESIONES Y OPERADORES (4/7)

Operador Relacional	Resultado
E1 == E2	1 si E1 es igual que E2; 0 en caso contrario.
E1 != E2	1 si E1 es distinto que E2; 0 en caso contrario.
E1 > E2	1 si E1 es mayor que E2; 0 en caso contrario.
E1 < E2	1 si E1 es menor que E2; 0 en caso contrario.
E1 >= E2	1 si E1 es mayor o igual que E2; 0 en caso contrario.
E1 <= E2	1 si E1 es menor o igual que E2; 0 en caso contrario.

Operador Lógico	Resultado
E1 && E2	1 si E1 y E2 son distintos de cero; 0 en caso contrario (AND).
E1 E2	1 si E1 o E2 son distintos de cero; 0 en caso contrario (OR).
!E	1 si E es cero; 0 en caso contrario (NOT).

0.2.5 EXPRESIONES Y OPERADORES (5/7)

Ejemplos:

```
int a,b,n;      // Declaración de varias variables del mismo tipo
3.14           // Literal
n              // Variable
n+5            // Operador aritmético suma
b+n*3          // Operadores aritméticos suma y producto
(b+n)*3        // Paréntesis para establecer precedencia
a>b            // Operador relacional mayor que
(a>=b)&&(n<0)    // Operador lógico and
```

0.2.5 EXPRESIONES Y OPERADORES (6/7)

Precedencia de los operadores: establece el orden en que son aplicados los operadores para evaluar una expresión.

Asociatividad: establece el orden para operadores con igual precedencia. La asociatividad en C se establece **de izquierda a derecha**.

0.2.5 EXPRESIONES Y OPERADORES (7/7)

Categoría/ Precedencia	Operadores
1. Máxima	(), [], ->, .
2. Unarios	!, ~, +, -, ++, --, &, *, sizeof(), (tipo)
3. Multiplicativos	*, /, %
4. Aditivos	+, -
5. Rotación	<<, >>
6. Relacional	<, <=, >, >=
7. Igualdad y Desigualdad	==, !=
8. Y bit a bit	&
9. Oex bit a bit	^
10. O bit a bit	
11. Y lógico	&&
12. O lógico	
13. Condicional	?:
14. Asignación	=, *=, /=, %=, +=, -=, &=, ^=, =, <<=, >>=
15. Coma	,

0.2.6 OPERADORES DE ASIGNACIÓN (1/4)

Los *operadores de asignación* almacenan valores en las variables.

El operador de asignación estándar = almacena el valor de su operando derecho en la variable especificada en su operando izquierdo.

Sintaxis:

variable = expresión

El operador de asignación puede utilizarse también en el momento de la declaración de una variable para asignarle inicialmente un valor.

Sintaxis:

tipo variable = expresión

0.2.6 OPERADORES DE ASIGNACIÓN (2/4)

Ejemplos:

```
int d;  
d = 5;  
double f1 = 3.56;  
double f2 = 5.5;  
double f3;  
f3 = f1 + f2;
```

Nótese que una asignación es una expresión. Por tanto, la asignación $x=y$, además de asignar a la variable x el valor de la variable y , devuelve el valor asignado, es decir, el valor de y .

Ejemplo:

```
int a = 3;  
int b, c;  
c = 10 - (b=a); // Tras su ejecución, b es 3 y c es 7
```

0.2.6 OPERADORES DE ASIGNACIÓN (3/4)

Operador	Efecto	Devuelve
<code>++x</code> (preincremento)	<code>x = x + 1</code>	<code>x + 1</code>
<code>x++</code> (postincremento)	<code>x = x + 1</code>	<code>x</code>
<code>--x</code> (predecremento)	<code>x = x - 1</code>	<code>x - 1</code>
<code>x--</code> (postdecremento)	<code>x = x - 1</code>	<code>x</code>
<code>x += y</code>	<code>x = x + y</code>	<code>x + y</code>
<code>x -= y</code>	<code>x = x - y</code>	<code>x - y</code>
<code>x *= y</code>	<code>x = x * y</code>	<code>x * y</code>
<code>x /= y</code>	<code>x = x / y</code>	<code>x / y</code>
<code>x %= y</code>	<code>x = x % y</code>	<code>x % y</code>

0.2.6 OPERADORES DE ASIGNACIÓN (4/4)

Ejemplo:

```
int a,b;  
a = 3;  
b = a++; // a = 4, b = 3  
a = 3;  
b = ++a; // a = 4, b = 4  
a = 3;  
b = 4;  
b += a; // b = 7
```

0.2.7 VARIABLES DE SÓLO LECTURA: CONST

Una variable puede declararse de sólo lectura mediante la palabra reservada `const`.

Las variables de sólo lectura no pueden modificarse a lo largo del programa, por lo que debe asignarse su valor al mismo tiempo que se declaran.

Cualquier operación de modificación de una variable de sólo lectura provocará un error de compilación.

Ejemplo:

```
const double pi = 3.14159;
```


0.2.8 EJEMPLO DE MANIPULACIÓN BÁSICA DE DATOS (1/2)

Salida por pantalla con `printf`

Sintaxis: `int printf(const char *formato, ...)`

La función `printf` produce una salida por pantalla de acuerdo a un formato determinado. El formato es una cadena de caracteres que puede hacer uso de:

- Texto ordinario (texto normal).
- Especificadores de conversión asociados a los argumentos.
- Secuencias de escape.

Los puntos suspensivos indican que la función `printf` admite un número variable de argumentos (*función variádica*). Los argumentos deben corresponderse apropiadamente con los especificadores de conversión.

La función devuelve el número de caracteres impresos, o un número negativo si ha ocurrido algún error.

Especificadores de conversión más utilizados:

- `%c` char
- `%d` int
- `%f` float, double
- `%s` char *

Secuencias de escape:

- `\n` Salto de línea
- `\t` Tabulador

0.2.8 EJEMPLO DE MANIPULACIÓN BÁSICA DE DATOS (2/2)

Salida por pantalla con `printf`

```
#include <stdio.h>

int main()
{
    int d = 36;
    double f = 3.61;
    char c = 'c';
    char s[] = "hola";
    printf("%d\n ",d);      // Imprime el decimal 36
    printf("%f\n ",f);      // Imprime el real 3.61
    printf("%.1f\n ",f);    // Imprime el real 3.6
    printf("%c\n ",c);      // Imprime el carácter 'c'
    printf("%s\n ",s);      // Imprime la cadena "hola"
    printf("%10s\n ",s);    // Texto alineado a la derecha
                           // ocupando 10 caracteres
    printf("%-10s\n ",s);   // Texto alineado a la izquierda
    return 0;
}
```

0.2.9 DESBORDAMIENTOS Y REDONDEOS

Desbordamiento

El lenguaje C **no detecta desbordamientos**. Si se produce un resultado fuera del rango, no se interrumpe el programa ni se señala el desbordamiento, sino que simplemente el resultado es erróneo.

Redondeo

Si el resultado de una expresión con operandos enteros es fraccionario **se redondea al entero más cercano a cero**.

Ejemplo:

```
float x = 14/3;      // x = 4.0000
float y = 14.0/3.0;  // y = 4.6667
```

0.2.10 CONVERSIÓN DE TIPO (1/4)

Cuando se evalúa una expresión en C, se pueden producir *conversiones de tipo* a través de dos mecanismos:

Promoción: conversión a un tipo de mayor rango.

Degradación: conversión a un tipo de menor rango.

Estas conversiones pueden ser:

Implícitas: las realiza automáticamente el compilador.

Explícitas: las indica el programador mediante el operador *cast*.

0.2.10 CONVERSIÓN DE TIPO (2/4)

Reglas para la *conversión implícita* de tipos en expresiones aritméticas:

1. Los tipos `char` y `short` se convierten a tipo `int`.
2. Los tipos `unsigned char` y `unsigned short` se convierten a `unsigned`.
3. Los operandos de menor rango se convierten (promueven) al operando de mayor rango y el resultado de la expresión es de ese tipo. El orden es el siguiente:

`int < unsigned int < long < unsigned long < float < double`

Ejemplo: Si tenemos una variable `a` de tipo `int` y una variable `b` de tipo `float`, para evaluar la expresión `a+b`, se promueve la variable `a` al tipo `float` y el resultado de la expresión es del tipo `float`.

0.2.10 CONVERSIÓN DE TIPO (3/4)

Conversión implícita a través del operador de asignación

```
int x = 3;
int y = 3;
float z = 4.6;
y = z; // Conversión (degradación) implícita
      // z se degrada a int
      // y toma el valor 4
z = x; // Conversión (promoción) implícita
      // x se promueve a float
      // z toma el valor 3.0
```

Cuando se realiza una degradación, puede producirse una **pérdida de información**.

0.2.10 CONVERSIÓN DE TIPO (4/4)

Conversión explícita con el operador cast

Se pueden forzar las conversiones de tipo utilizando el operador *cast*:

- El operador *cast* es unario.
- Tiene igual prioridad que el resto de operadores unarios.

Ejemplo:

```
int a = 1;
int b = 2;
float c = a / b;           // c toma el valor 0.0
float d = (float) a / (float) b; // d toma el valor 0.5
```

Nota: El operador *cast* sólo funciona con tipos enteros (incluido char), reales y apuntadores.

0.2.11 ESTRUCTURAS: STRUCT (1/5)

Un tipo `struct` es un tipo de datos definido por el programador compuesto por variables de otros tipos (que pueden ser a su vez de tipo `struct`). Un tipo `struct` permite definir un *producto de tipos* (tipos compuestos por varios elementos o miembros de otros tipos).

Definición de un tipo `struct`

Para definir un tipo `struct` se usa la palabra reservada `struct`, seguida por el nombre del tipo (opcional) y seguida por las declaraciones de los miembros del tipo encerrados entre llaves, y terminado en `;`

Si no se pone nombre al tipo `struct` no se podrá hacer referencia a ese tipo después.

Sintaxis:

```
struct nombre_estructura  
{  
    tipo1 nombre1;  
    ...  
    tipoN nombreN;  
};
```

Ejemplo:

```
struct Punto  
{  
    float x, y;  
};
```


0.2.11 ESTRUCTURAS: STRUCT (2/5)

Declaración de variables de tipo struct

Se pueden declarar variables de tipo struct en el momento de la definición del tipo struct poniendo los nombres de las variables después de la llave cerrada y antes del ;

Ejemplo:

```
struct Punto
{
    float x, y;
} p1, p2; // Declara dos variables p1 y p2 de tipo struct Punto
```

También se pueden declarar variables de tipo struct después de la definición del tipo struct usando la palabra struct y el nombre dado al tipo struct, seguido por uno o más nombres de variables separados por comas.

Ejemplo:

```
struct Punto
{
    float x, y;
};
struct Punto p1, p2;
```

0.2.11 ESTRUCTURAS: STRUCT (3/5)

Inicialización de variables de tipo struct

Los miembros de una variable de tipo struct se pueden inicializar en su declaración.

Ejemplos:

```
struct Punto
{
    float x, y;
} p0 = {0.0, 0.0};           // p0.x = 0.0, p0.y = 0.0
struct Punto p1 = {0.1, 0.2}; // p1.x = 0.1, p1.y = 0.2
struct Punto p2 = {.y=5.0, .x=1.0}; // p2.x = 1.0, p2.y = 5.0
struct Punto p3 = {y:3.2, x:0.0}; // p3.x = 0.0, p3.y = 3.2
struct Punto p4 = {2.0};       // p4.x = 2.0, p4.y = 0.0
```

0.2.11 ESTRUCTURAS: STRUCT (4/5)

Acceso a los miembros de las variables de tipo struct

Para acceder a los miembros de una variable de tipo struct se utiliza la **sintaxis**:

nombre_variable . nombre_campo

Ejemplo:

```
struct Punto
{
    float x, y;
};
struct Punto p;
p.x = 0.0;
p.y = 0.0;
printf("Punto (%f,%f)\n",p.x,p.y);
```

0.2.11 ESTRUCTURAS: STRUCT (5/5)

Asignación de variables de tipo struct

Una variable de tipo struct puede asignarse a otra variable de tipo struct de su mismo tipo mediante el operador de asignación. En este caso, la asignación realiza una **copia miembro a miembro** de la variable de tipo struct origen a la destino. Si un miembro es una estructura de datos de tipo array (ver sección 0.6) cuyo tamaño ha sido establecido en la declaración mediante un literal, también copia el contenido de los elementos del array.

Ejemplo:

```
struct Punto p1 = {0.0,0.0};  
struct Punto p2 = p1;
```

Una variable de tipo struct no puede asignarse a otra variable de tipo struct de distinto nombre aunque sus miembros coincidan. Tampoco es posible utilizar el operador *cast*.

0.2.12 ENUMERACIONES: ENUM (1/4)

Una *enumeración* es un tipo de datos definido por el programador para almacenar valores enteros constantes (por defecto `signed int`) y referirse a ellos a través de nombres. Por defecto, los valores son 0, 1, 2, ..., aunque pueden definirse valores distintos en su inicialización.

Definición de enumeraciones

Para definir una enumeración se utiliza la palabra reservada `enum`, seguida por el nombre de la enumeración (opcional) y seguido por una lista de nombres separados por comas y encerrados entre llaves y terminado en `;`

Si no se pone nombre a la enumeración no se podrá hacer referencia a ella después.

Sintaxis:

```
enum nombre_enumeración  
{  
    nombre1,nombre2,nombreN  
};
```

Ejemplo:

```
enum DiaSemana  
{  
    lunes, martes, miercoles, jueves, viernes, sabado, domingo  
};
```

0.2.12 ENUMERACIONES: ENUM (2/4)

Declaración de variables enumeración

Se pueden declarar variables de tipo enumeración en el momento de la definición de la enumeración poniendo los nombres de las variables después de la llave cerrada y antes del ;

Ejemplo:

```
enum DiaSemana
{
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
}dia1, dia2; // Declara dos variables dia1 y dia2 de tipo enum DiaSemana
```

También se pueden declarar variables de tipo enumerado después de la definición de la enumeración usando la palabra enum y el nombre dado a la enumeración, seguido por uno o más nombres de variables separados por comas.

Ejemplo:

```
enum DiaSemana
{
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
};
enum DiaSemana dia1, dia2;
```

0.2.12 ENUMERACIONES: ENUM (3/4)

Asignación de variables enumeración

Se puede asignar a una variable de tipo enumeración cualquier valor que se pudiera asignar a una variable de tipo `int`, incluyendo valores de otras enumeraciones.

Además, cualquier variable a la que se pueda asignar un valor entero, puede ser asignada un valor de una enumeración.

Lo que no se puede hacer es cambiar los valores de una enumeración una vez han sido definidos, ya que son constantes.

Ejemplo:

```
enum DiaSemana
{
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
};
enum DiaSemana dia1,dia2;
int a = 2;
dia1 = a;
dia2 = jueves;
a = dia2;
lunes = 1; // ERROR
```

0.2.12 ENUMERACIONES: ENUM (4/4)

Ejemplo: Definición del tipo BOOLEAN

```
typedef enum
{
    falso, verdad
}BOOLEAN;
int main()
{
    // Lee caracteres desde teclado hasta que el carácter sea 'F' o 'f'
    BOOLEAN fin = falso;
    while (!fin)
    {
        char c=getchar();
        if ((c=='F')||(c=='f'))
            fin = verdad;
    }
    return 0;
}
```


0.2.13 UNIONES: UNION (1/3)

Una *unión* es un tipo de datos definido por el programador para almacenar varias variables en el mismo espacio de memoria. Aunque se puede acceder a todas las variables en cualquier momento, sólo se debería acceder a una de ellas al mismo tiempo, ya que asignar el valor a una de ellas sobrescribe el valor de las otras.

Definición de uniones

Para definir una estructura se usa la palabra reservada `union`, seguida por el nombre de la unión (opcional) y seguida por las declaraciones de los miembros de la unión encerrados entre llaves, y terminado en `;`. Si no se pone nombre a la unión no se podrá hacer referencia a la unión después.

Sintaxis:

```
union nombre_unión  
{  
    tipo1 nombre1;  
    ..  
    tipoN nombreN;  
};
```

Ejemplo:

```
union Dato  
{  
    char op;  
    double valor;  
};
```

0.2.13 UNIONES: UNION (2/3)

Declaración de variables unión

Se pueden declarar variables de tipo unión en el momento de la definición de la unión poniendo los nombres de las variables después de la llave cerrada y antes del ;

Ejemplo:

```
union Dato
{
    char op;
    double valor;
} d1, d2; // Declara dos variables d1 y d2 de tipo union
```

También se pueden declarar variables de tipo unión después de la definición de la unión usando la palabra union y el nombre dado a la unión, seguido por uno o más nombres de variables separados por comas.

Ejemplo:

```
union Dato
{
    char op;
    double valor;
};
union Dato d1, d2;
```

0.2.13 UNIONES: UNION (3/3)

Inicialización de variables unión

Los miembros de una variable unión se pueden inicializar en su declaración.

Ejemplos:

```
union Dato
{
    char op;
    double valor;
} d0 = {'c'};           // d0.op = 'c'
union Dato d1 = {.op= 'c'}; // d1.op = 'c'
union Dato d2 = {.valor=5.0}; // d2.valor = 5.0
union Dato d3 = {op: 'c'}; // d3.op = 'c'
union Dato d4 = {valor:3.2}; // d4.valor = 3.2
```

0.2.14 RENOMBRADO DE TIPOS: TYPEDEF (1/2)

Se puede crear un nuevo nombre para cualquier tipo de datos mediante la sentencia `typedef`.

Una vez creado el nuevo nombre, puede utilizarse para la declaración de variables pertenecientes a ese tipo.

Sintaxis:

`typedef nombre_tipo_antiguo nombre_tipo_nuevo;`

Ejemplos:

```
typedef int Entero;  
int a;  
Entero n;
```

0.2.14 RENOMBRADO DE TIPOS: TYPEDEF (2/2)

Ejemplos para renombrado de tipo `struct` (formas válidas también para enumeraciones y uniones)

Primero la definición de tipo `struct` y después el nombre del tipo `struct` :

```
struct Punto { float x, y; };  
typedef struct Punto Punto;
```

Tipo `struct` y nombre del tipo :

```
typedef struct Punto { float x, y; } Punto;
```

Tipo `struct` anónimo y nombre del tipo :

```
typedef struct { float x, y; } Punto;
```

Posteriormente, se pueden declarar variables con el nuevo nombre del tipo y con el antiguo, excepto en el último caso que la estructura es anónima:

```
struct Punto punto1; // Esto no puede hacerse en el tercer caso  
Punto punto2;
```

0.2.15 TIPOS INCOMPLETOS

Se pueden definir tipos `struct`, enumeraciones y uniones sin listar sus miembros o valores (en el caso de las enumeraciones) resultando en lo que se denomina un *tipo incompleto*.

En algún momento posterior del módulo se deberá completar el tipo. También podrán declararse variables apuntador a un tipo incompleto en cualquier otra parte del programa.

Esta técnica es comúnmente utilizada para *ocultar información* en *tipos de datos abstractos* (Tema 1) y para implementar *estructuras de datos enlazadas* (Tema 2).

Ejemplo:

```
typedef struct Punto Punto;
typedef enum Color Color;
typedef union Dato Dato;
...
struct Punto { float x, y; };
enum Color { rojo, verde, azul };
union Dato { char op; double valor; };
```

0.3 SENTENCIAS

0.3.1 Sentencias simples y compuestas

0.3.2 Sentencia expresión

0.3.3 Sentencia if

0.3.4 Sentencia while

0.3.5 Sentencia for

0.3.6 Sentencia do while

0.3.7 Sentencias break, continue, return y goto

0.3.8 Sentencia vacía

0.3.9 Sentencia switch

0.3.1 SENTENCIAS SIMPLES Y COMPUESTAS

El lenguaje C distingue entre sentencias simples y sentencias compuestas o bloques.

Sentencia simple: typedef, expresión seguida de ;, if, while, for, do, etiqueta, goto, break, continue, return, ;, switch

Sentencia compuesta o bloque: Varias sentencias entre llaves.

Ejemplo:

```
// Sentencia simple
    a = b * 8;
// Sentencia compuesta
{
    a = 9;
    b = 3;
    c = a * 9 - b;
}
```


0.3.2 SENTENCIA EXPRESIÓN

Cualquier expresión seguida de un `;` es una sentencia simple.

Ejemplo:

```
a+b-5;
```

En este ejemplo, la expresión es evaluada, pero esto resulta inútil porque no se está almacenando su valor en ningún sitio.

Las sentencias de expresión son útiles cuando tienen algún tipo de efecto como almacenar un valor o llamar a una función.

Ejemplo:

```
c = a+b-5;  
a++;  
printf("%d", a+b);
```

0.3.3 SENTENCIA IF

Sentencia `if`: Ejecuta condicionalmente una parte u otra de un programa basado en la evaluación de una expresión dada.

Sintaxis: *if (condición) sentencia_si_cierto*
else sentencia_si_falso

donde *condición* es una expresión de cualquier clase. Si el resultado de la expresión es distinto de cero, la condición se considera cierta y se ejecuta el código de *sentencia_si_cierto*, en caso contrario, se ejecuta el código *sentencia_si_falso*.

La cláusula *else* es opcional.

Se puede utilizar una serie de sentencias *if* para comprobar múltiples condiciones.

Ejemplo (determinar el estado del agua):

```
enum estado{ solido, liquido, gaseoso } estado_agua;  
int temperatura;  
scanf("%d",&temperatura);  
if (temperatura<0) estado_agua=solido;  
else if (temperatura<100) estado_agua=liquido;  
else estado_agua=gaseoso;
```

0.3.4 SENTENCIA WHILE

La sentencia `while` es una sentencia de bucle con una condición de salida al comienzo del bucle.

Sintaxis:

`while (condición) sentencia`

La sentencia `while` primero evalúa la condición. Si la condición es cierta, ejecuta la sentencia y vuelve a evaluar la condición. Si la condición es falsa, la sentencia `while` termina.

Ejemplo (imprimir los números del 0 al 9):

```
int x=0;
while(x<10)
{
    printf("%d ",x);
    x++;
}
```

0.3.5 SENTENCIA FOR (1/2)

La sentencia `for` es una sentencia de bucle cuya estructura permite, fácilmente, la inicialización de variables, evaluación de la condición de salida y la actualización de variables.

Resulta especialmente útil para realizar bucles controlados por un contador.

Sintaxis:

for (expresión_inicial ; condición ; expresión_actualización)
sentencia

- *expresión_inicial* se ejecuta antes de empezar el bucle.
- *condición* se evalúa al comienzo de cada iteración.
- *sentencia* se ejecuta mientras *condición* sea cierta.
- *expresión_actualización* se ejecuta después de *sentencia* en cada iteración del bucle.

Ejemplo (imprimir los números del 0 al 9):

```
for(int x=0; x<10; x++)  
    printf("%d ",x);
```

0.3.5 SENTENCIA FOR (2/2)

Las tres expresiones de la sentencia **for** **son opcionales**:

- Si se omite la expresión inicial, no se ejecuta nada antes del bucle.
- Si se omite condición, la condición de continuación será siempre cierta.
- Si se omite la expresión de actualización, no se hace nada después de cada iteración.

La sentencia no puede omitirse, pero puede corresponderse con la sentencia vacía ; (sección 0.3.8).

Ejemplo (imprimir los números del 0 al 9):

```
int x = 0;
for( ;x<10; )
{
    printf( "%d ", x );
    x++;
}
```

0.3.6 SENTENCIA DO WHILE

Es una sentencia de bucle con una condición de salida al final del bucle, por lo que itera al menos una vez.

Sintaxis:

```
do  
    sentencia  
while (condición)
```

Ejemplo (adivinar un número entero):

```
int miNumero = 7; // Establece un número secreto para el usuario  
int numero;  
do  
{  
    printf("Adivina un numero entero entre 1 y 10: ");  
    scanf("%d",&numero);  
    if (miNumero>numero) printf("Mi numero es mayor.\n");  
    if (miNumero<numero) printf("Mi numero es menor.\n");  
}  
while(miNumero!=numero);
```

0.3.7 SENTENCIAS BREAK, CONTINUE, RETURN Y GOTO

Sentencia `break`

La sentencia `break` termina las sentencias `while`, `do`, `for` o `switch`.

La sentencia `continue`

La sentencia `continue` termina la iteración actual de un bucle y comienza la siguiente iteración.

Sentencia `return`

La sentencia `return` termina la ejecución de una función y retorna el control del programa a la función que realizó la llamada (sección 0.4.1).

Sentencia `goto`

La sentencia `goto` salta incondicionalmente a otro punto del programa indicado mediante una etiqueta (`label`). Esta sentencia es generalmente rechazada en el paradigma de programación estructurada, por lo que no será utilizada en este curso.

Resulta conveniente y buena práctica de programación no terminar los bucles con sentencias del tipo `break`, `continue`, `return` o `goto`, las cuales permiten modificar el flujo de control de ejecución, pero violan las normas de la programación estructurada. En cualquier caso, **siempre es posible terminar un bucle de forma adecuada sin la utilización de estas sentencias.**

0.3.8 SENTENCIA VACÍA

La sentencia vacía es un punto y coma sólo:

```
;
```

Esta sentencia no tiene ningún efecto ni consume tiempo de ejecución. Suele usarse como única sentencia del cuerpo de un bucle o como una expresión.

Ejemplo (leer un entero positivo desde teclado y comprobar si es un número primo):

```
int x;
scanf("%d",&x);
int i=2;
for( ; (i<x) && (x%i!=0) ; i++)
    ; // Sentencia vacía
if (i==x)
    printf("El numero %d es primo",x);
else
    printf("El numero %d no es primo",x);
```


0.3.9 SENTENCIA SWITCH (1/4)

La sentencia `switch` evalúa una expresión y compara el resultado con otras, ejecutando un conjunto de sentencias basándose en el resultado de las comparaciones.

Sintaxis:

```
switch ( expresión )  
{  
  
    case expresión_comparación_1 : sentencia1  
    case expresión_comparación_2 : sentencia2  
  
    ...  
  
    case expresión_comparación_N : sentenciaN  
    default: sentenciaD  
  
}
```

0.3.9 SENTENCIA SWITCH (2/4)

La sentencia `switch` compara la expresión con cada una de las expresiones de comparación, hasta que encuentra una que es igual; entonces ejecuta las sentencias siguientes.

Todas las expresiones deben ser de un tipo entero. Las expresiones de comparación deben ser de tipo entero constante, es decir, literales enteros o expresiones construidas a partir de literales enteros.

También es posible especificar un rango de valores enteros consecutivos en lugar de las expresiones de comparación con la sintaxis:

case expresión_comparación_inferior ... expresión_comparación_superior

La sentencia `break` se utiliza frecuentemente para terminar el procesamiento de un caso particular dentro de una construcción `switch`. Si no se usa `break`, continuará ejecutándose el código asociado al siguiente `case`.

Las sentencias de la cláusula `default`, que es opcional, se ejecutan si la expresión no es igual a ninguna de las expresiones previamente comparadas .

0.3.9 SENTENCIA SWITCH (3/4)

Ejemplo (establecer la hora de alarma del despertador):

```
enum DiaSemana{
    lunes, martes, miercoles, jueves, viernes, sabado, domingo
} dia;
scanf("%d",&dia);
int hora_despertar;
switch(dia)
{
    case lunes ... viernes: hora_despertar = 7; break;
    case sabado: hora_despertar = 9; break;
    case domingo: hora_despertar = 10;
}
```

0.3.9 SENTENCIA SWITCH (3/4)

Ejemplo (determinar el estado del agua):

```
enum estado{
    solido, liquido, gaseoso
} estado_agua;
int temperatura;
scanf("%d",&temperatura);
switch(temperatura)
{
    case -273 ... 0: estado_agua=solido; break;
    case 1 ... 100: estado_agua=liquido; break;
    default: estado_agua=gaseoso;
}
```

0.4 FUNCIONES

0.4.1 Definición de funciones

0.4.2 Llamadas a función

0.4.3 Declaración adelantada

0.4.4 Funciones sin parámetros formales

0.4.5 Procedimientos

0.4.6 Paso de parámetros

0.4.7 Paso y devolución de estructuras

0.4.1 DEFINICIÓN DE FUNCIONES (1/2)

En el lenguaje C, la *declaración* de una *función* consta de los siguientes componentes:

- Tipo de datos devuelto.
- Nombre de la función.
- Lista de *parámetros formales*.

Sintaxis:

tipo_devuelto nombre_función (< lista de parámetros formales >)

donde *< lista de parámetros formales >* puede contener cero, uno o más parámetros formales separados por comas. Cada parámetro formal se especifica mediante su tipo y su nombre.

0.4.1 DEFINICIÓN DE FUNCIONES (2/2)

El *cuerpo* de la función es una sentencia compuesta a continuación de la declaración que debe contener, en todos los caminos del flujo de ejecución de la función, la sentencia `return`, la cual devuelve el resultado de la función. Además de devolver el resultado, la sentencia `return` provoca la terminación de la función, devolviéndose el control de ejecución del programa a la siguiente sentencia después de la llamada a la función.

Sintaxis:

```
{  
...  
return expresión;  
}
```

Ejemplo:

```
double area_triangulo(double base, double altura)  
{  
    return base*altura/2;  
}
```

Nótese que `main` es una función cuyo nombre está reservado por el compilador con la particularidad de que es el punto de inicio del programa.

0.4.2 LLAMADAS A FUNCIÓN (1/2)

Una *llamada a una función* es una expresión. Para llamar a una función se escribe su nombre y entre paréntesis una lista de expresiones separadas por comas.

Sintaxis:

nombre_función (<lista de expresiones>)

La llamada a la función puede formar parte de otras expresiones que requieren su valor, o también la llamada puede realizarse sin que se use el valor devuelto. Por ejemplo, en ejemplos anteriores no se utiliza el valor devuelto por las funciones `printf` y `scanf`.

Cada expresión en la lista se evalúa y su resultado se pasa a la función. Denominamos *parámetro actual* o *argumento* al valor devuelto por la evaluación de cada expresión. Cada parámetro actual debe tener el mismo tipo que el del parámetro formal correspondiente o un tipo compatible.

0.4.2 LLAMADAS A FUNCIÓN (2/2)

Ejemplos (calcular el área de un triángulo):

```
double base = 10.0;
double altura = 5.6;
double area = area_trianguulo(base, altura);
printf("El area es: %f\n", area);
```

Ejemplo (leer un entero con formato correcto):

```
int numero, resultado;
do
{
    printf("Introduce un numero entero: ");
    resultado = scanf("%d",&numero);
    if (resultado!=1) printf("Formato incorrecto.\n");
}
while(resultado!=1);
```

0.4.3 DECLARACIÓN ADELANTADA

Las funciones pueden declararse de forma adelantada indicando únicamente el *prototipo de la función*.

De esta forma se podría utilizar la función en una zona de código anterior a la definición de la función. Esta técnica es típicamente usada en C para *programación modular* (sección 0.12).

Ejemplo:

```
double area_triangulo(double base, double altura); // prototipo
double volumen_prisma_triangular(double b,double h,double h_prisma)
{
    return area_triangulo(b,h)*h_prisma;
}
double area_triangulo(double base, double altura)
{
    return base*altura;
}
```

0.4.4 FUNCIONES SIN PARÁMETROS FORMALES

Si una función se declara sin parámetros formales, el lenguaje no impide que en su llamada puedan utilizarse argumentos. Para forzar a que una función se defina y se invoque sin argumentos, la función debe declararse con `void` en la lista de parámetros formales.

Sintaxis:

tipo_devuelto nombre_función (void)

Ejemplo:

```
double valor_pi(void)
{
    return 3.1416;
}
double area_circulo(double radio)
{
    return valor_pi()*radio*radio;
}
```

0.4.5 PROCEDIMIENTOS

El lenguaje C sólo permite definir funciones. Un *procedimiento* puede definirse como una función que no devuelve ningún valor. A estas funciones se les denomina *funciones void*. En una función void no es obligatoria la sentencia return, aunque puede utilizarse sin argumentos para terminar la función. La llamada a una función void no devuelve nada y por tanto no puede formar parte de expresiones.

Sintaxis:

void nombre_función(<lista de parámetros formales>)

Ejemplo:

```
void ImprimeRango(int inferior, int superior)
{
    for(int i=inferior;i<=superior;i++) printf("%d ",i);
}
int main()
{
    int a = 4;
    int b = 13;
    ImprimeRango(a,b);
    return 0;
}
```

0.4.6 PASO DE PARÁMETROS

Los parámetros en C se pasan siempre por *valor*. En un paso por valor, se transfiere una *copia* del valor del parámetro actual al parámetro formal. Por tanto una función no puede alterar el valor de una variables usada como parámetros actual.

Ejemplo:

```
void incrementa(int n)
{
    n++;
}
int main()
{
    int numero = 6;
    incrementa(numero);
    printf("Número = %d\n", numero); // Imprime Número = 6
    return 0;
}
```

Para conseguir que el valor de una variable usada como parámetro actual en la llamada a una función pueda modificarse desde el código de la función hay que recurrir a los *apuntadores*, que se verán en la sección 0.7.

0.4.7 PASO Y DEVOLUCIÓN DE ESTRUCTURAS

Paso de parámetros **struct** a funciones:

Cuando una variable **struct** se pasa como parámetro a una función, el contenido de todos sus miembros se copia, es decir, se realiza una **asignación** del parámetro actual al parámetro formal.

Devolución de **struct**:

Si una función devuelve una variable **struct**, también se realiza una **asignación** entre el valor devuelto por la función y la variable utilizada en la llamada a la función.

0.5 ÁMBITO Y EXTENSIÓN

0.5.1 Definición de ámbito y extensión

0.5.2 Ámbito de variables, tipos y funciones

0.5.3 Extensión de las variables

0.5.4 Variables locales static

0.5.5 Ejemplo

0.5.1 DEFINICIÓN DE ÁMBITO Y EXTENSIÓN

Ámbito:

El *ámbito* de un identificador de variable, tipo o función es el **contexto del programa en el que el identificador puede ser utilizado.**

Extensión:

La *extensión* (o duración) de una variable es el **periodo de tiempo** desde que la variable **empieza a existir**, ocupando espacio de memoria, hasta que **deja de existir**, liberándose el espacio de memoria que ocupaba.

0.5.2 ÁMBITO DE VARIABLES, TIPOS Y FUNCIONES (1/3)

Variables globales

- Declaradas fuera de cualquier bloque del *módulo* (*fichero .c*).
- Su ámbito es desde el punto en el que se declaran hasta el final del módulo.
- Una variable global puede ampliar su ámbito al resto de módulos si en éstos se declara la variable como `extern` (sección 0.12)

Variables locales

- Declaradas en el interior de un bloque.
- Su ámbito es desde el punto en el que se declaran hasta el final del bloque en el que se han declarado.
- Una variable declarada dentro de un bloque *oculta* a cualquier otra variable con el mismo nombre declarada global o en un bloque de nivel superior.
- Los parámetros de una función actúan como variables locales a la función en cuanto a su ámbito.

Variables dinámicas

- Accesibles mediante apuntadores, se construyen y destruyen en tiempo de ejecución.
- Su ámbito es el correspondiente a la variable apuntador que la apunta.

0.5.2 ÁMBITO DE VARIABLES, TIPOS Y FUNCIONES (2/3)

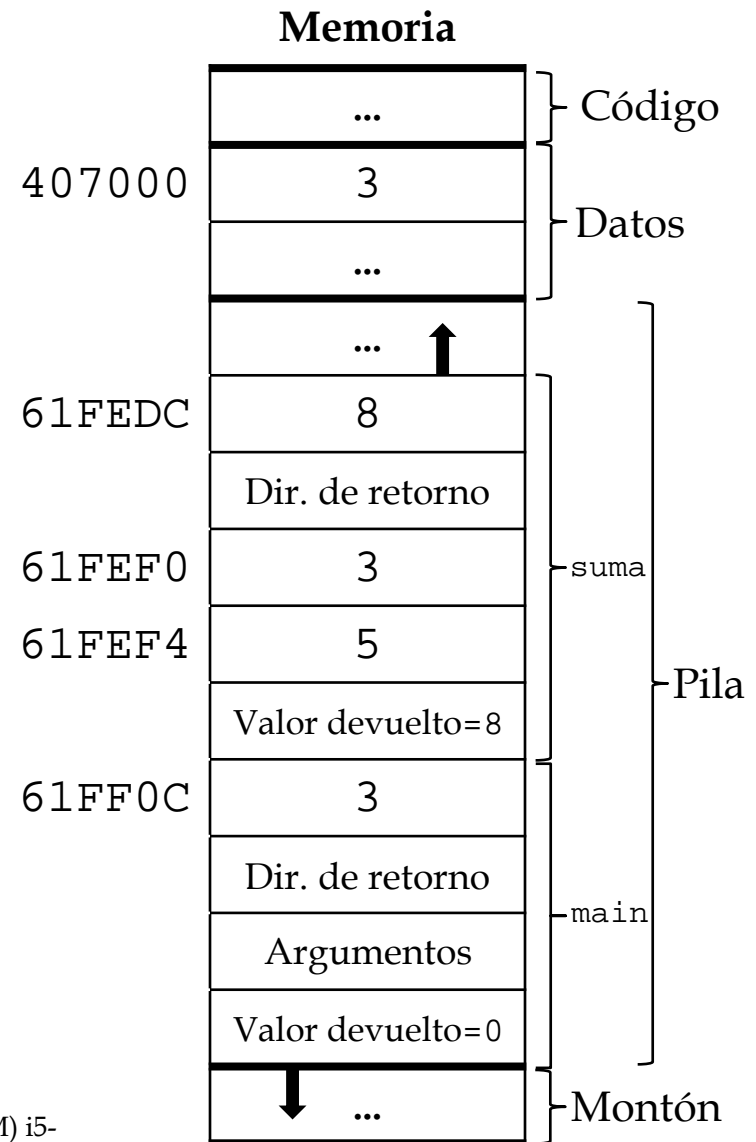
Tabla de símbolos

Identificador	Tipo	Dirección
dato	int	407000
i	int	61FF0C
a	int	61FEF0
b	int	61FEF4
res	int	61FEDC

```
int dato = 3;
int suma( int a, int b ) {
    int res = a + b;
    return res;
}
int main( )
{
    int i = dato;
    printf("Suma = %d\n", suma(i,5));
    return 0;
}
```

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque como se hace en realidad).

Tecnología de la Programación
Tema 0. Introducción a C



0.5.2 ÁMBITO DE VARIABLES, TIPOS Y FUNCIONES (3/3)

Ámbito de tipos de datos

El ámbito de los tipos de datos es **similar al de las variables**, es decir, los tipos de datos con ámbito global se declaran fuera de todo bloque y los declarados dentro de un bloque tienen ámbito local al bloque.

Un tipo de datos local **oculta** a cualquier otro tipo de datos con el mismo nombre definido global o en un bloque superior.

Ámbito de las funciones

El ámbito de las funciones en el estándar C99 es siempre global ya que las funciones se definen siempre fuera de todo bloque. Sin embargo, en algunas extensiones, por ejemplo *GNU*, si se admiten funciones anidadas y por tanto con ámbito local.

0.5.3 EXTENSIÓN DE LAS VARIABLES

Variables globales

- Existen permanentemente mientras el programa se ejecuta.
- La asignación de memoria de las variables globales se realiza en tiempo de compilación. Por esta razón se denominan también *variables estáticas*. Su espacio de almacenamiento es el *segmento de datos (data segment)*.
- Por defecto, las variables globales se inicializan a 0.

Variables locales

- Existen mientras se está ejecutando el código de la subrutina en las que se han declarado. Por esta razón se denominan también variables automáticas. Su espacio de almacenamiento es el *segmento de pila (stack segment)*. No obstante, siempre que sea posible pueden ser alojadas en los registros de la CPU por razones de optimización de recursos.
- Una variable local a un bloque se crea cuando se declara, y se destruye o libera cuando el bloque en el que ha sido declarada termina.
- Los parámetros formales de una función se crean cuando se llama a la función, y se destruyen o liberan cuando la función retorna.
- Una variable local no inicializada contiene valores indefinidos.

Variables dinámicas

- Existen desde que se construyen hasta que se destruyen con instrucciones explícitas del lenguaje. Su espacio de almacenamiento es el *segmento montón (heap segment)*.

0.5.4 VARIABLES LOCALES STATIC

La palabra reservada `static` permite definir variables locales con extensión permanente. Se alojan en el segmento de datos.

La inicialización de una variable `static` se realiza una sola vez al comienzo del programa, si la inicialización se ha realizado en el momento de la declaración.

Ejemplo:

```
int contador(void)
{
    static int cuenta = 0;
    return cuenta++;
}
```

La variable `cuenta` es local a la función `contador`, pero no desaparece al retornar la función, por lo que cada vez que se invoca a la función `contador`, la variable `cuenta` se incrementa.

0.5.5 EJEMPLO

```
typedef double Real; // Tipo global
Real t; // Variable global
Real a; // Variable global
Real mitad(Real t) // El parámetro t oculta a la variable global t
{
    Real a; // La variable local a oculta a la variable global a
    a = t / 2;
    return a;
}
int main()
{
    for(int i=0;i<10;i++) // La variable i es local al bucle for
    {
        t = (Real)i;
        a = mitad(t);
        printf("Numero = %f, Mitad = %f\n",t,a);
    }
    return 0;
}
```

0.6 ARRAYS Y CADENAS DE CARACTERES

0.6.1 Arrays

0.6.2 Cadenas de caracteres

0.6.1 ARRAYS (1/8)

Un *array* es una estructura de datos que permite almacenar uno o más elementos del mismo tipo en memoria contiguamente.

Los elementos de un array son accesibles a través de *índices*. En C los elementos de un array están indexados comenzando por el índice 0.

Declaración de arrays

Para declarar un array se especifica el tipo de datos de los elementos, su nombre y el número de elementos que puede almacenar.

Sintaxis:

tipo nombre [N];

Declara *nombre* como un array de *N* elementos del tipo *tipo*.

N es una expresión que debe devolver un resultado entero mayor o igual que cero. Si el array es global, expresión debe ser constante, ya que su valor se requiere en tiempo de compilación.

Ejemplo:

```
int v[100];  
int tam = 60;  
int v2[tam]; // Válido solo si la declaración de v2 es local
```

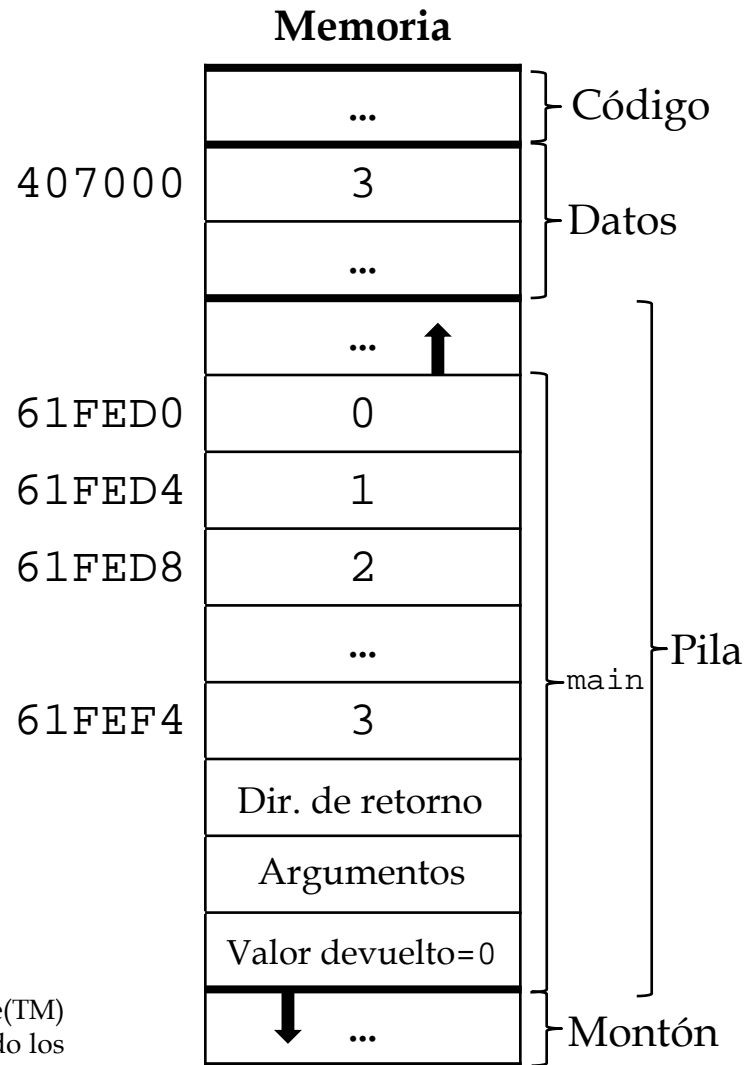

0.6.1 ARRAYS (2/8)

Tabla de símbolos

Identificador	Tipo	Dirección
n	int	407000
v	int[n]	61FED0
i	int	61FEF4

```
int n = 3;
int main( ) {
    int v[n];
    int i;
    for (i=0; i<n; i++)
        v[i] = i;
    return 0;
}
```

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque como se hace en realidad).



0.6.1 ARRAYS (3/8)

Declaración junto con inicialización

Sintaxis:

tipo nombre [N] = { valor1, valor2, ... , valorN };

Se puede omitir el tamaño *N* del array ya que su tamaño viene implícito por el número de valores inicializados, pero si se define debe ser una expresión constante.

También se pueden inicializar los elementos de un array en cualquier orden especificando el índice que se inicializa o el rango de índices. En este caso, los elementos que no se inicializan, son inicializados por defecto a cero y el tamaño del array, si no se indica, viene dado por el valor del mayor índice.

Ejemplos:

```
int v1[3] = {3, 4, 5};
float v2[2] = {3.5, 6.8};
double v3[] = {3.6, 9.8, 0.5, 9.2}; // Vector v3 de tamaño 4
int v4[6] = { [2] 5, [4] 9 }; // Mismo efecto que v4[]={0,0,5,0,9,0}
int v5[] = { [3] = 7, [1] = 2 }; // Mismo efecto que v4[]={0,2,0,7}
int v6[] = {[0 ... 9] = 1, [10 ... 98] = 2, 3}; // v[99] = 3
```

0.6.1 ARRAYS (4/8)

Acceso a los elementos de un array

Para acceder al elemento i -ésimo de un array se utiliza la expresión:

nombre [i]

donde [] es *el operador de índice de array*.

En C, un array de tamaño N se indexa desde el índice 0 hasta el índice $N-1$.

Ejemplo (leer por teclado dos vectores y sumarlos):

```
int n = 100;
int v1[n], v2[n], v3[n];
for(int i=0; i<n; i++)
    scanf("%d %d",&v1[i],&v2[i]);
for(int i=0; i<n; i++)
    v3[i] = v1[i] + v2[i];
```

0.6.1 ARRAYS (5/8)

Declaración de arrays de *múltiples dimensiones*

Se pueden declarar arrays con mas de una dimensión.

Sintaxis:

tipo nombre [N_1] [N_2] ... [N_M]

donde N_1, N_2, \dots, N_M son el número de elementos por cada una de las M dimensiones.

Ejemplos:

```
double matriz1[5][10];  
int tam1 = 3;  
int tam2 = 7;  
int matriz2[tam1][tam2][2];
```

Declaración junto con inicialización de arrays de múltiples dimensiones

Se pueden inicializar los elementos de un array de múltiples dimensiones al mismo tiempo que se declara. En este caso, el tamaño de la primera dimensión puede omitirse.

Ejemplos:

```
int matriz1[2][5] = { {1,2,3,4,5}, {6,7,8,9,10} }; // matriz1 de 2 x 5  
int matriz2[][3] = { {1,2,3}, {4,5,6} };           // matriz2 de 2 x 3
```

0.6.1 ARRAYS (6/8)

Tabla de símbolos

Identificador	Tipo	Dirección
n	int	61FEE0
m	int	61FEDC
x	int	61FED8
v	int[n][m]	61FE88
i	int	61FED4
j	int	61FED0

```
int main( ) {  
    int n = 3; int m = 2;  
    int x = 0;  
    int v[n][m];  
    int i,j;  
    for (i=0; i<n; i++)  
        for (j=0; j<m; j++)  
            v[i][j] = ++x;  
    return 0;  
}
```

$v = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque como se hace en realidad).

Tecnología de la Programación
Tema 0. Introducción a C

Memoria		main	Pila
61FE88	1		
61FE8C	2		
61FE90	3		
61FE94	4		
61FE98	5		
61FE9C	6		
	...		
61FED0	2		
61FED4	3		
61FED8	6		
61FEDC	2		
61FEE0	3		
	Dir. de retorno		
	Argumentos		
	Valor devuelto=0		

0.6.1 ARRAYS (7/8)

Acceso a arrays de múltiples dimensiones

En un array de N dimensiones, para acceder al elemento (i_1, i_2, \dots, i_M) se utiliza la expresión:

nombre [i_1] [i_2] ... [i_M]

Ejemplo (leer por teclado dos matrices y sumarlas):

```
int n = 10;
int m = 10;
int matriz1[n][m], matriz2[n][m] ,matriz3[n][m];
for(int i=0; i<n; i++)
    for(int j=0; j<m; j++)
        scanf("%d %d",&matriz1[i][j],&matriz2[i][j]);
for(int i=0; i<n; i++)
    for(int j=0; j<m; j++)
        matriz3[i][j] = matriz1[i][j] + matriz2[i][j];
```

0.6.1 ARRAYS (8/8)

Consideraciones sobre arrays:

- El lenguaje C no detecta índices fuera de rango.
- Si *a* y *b* son arrays, la expresión *a=b* es ilegal.
- Si *a* es un array de dos dimensiones, la expresión *a[i]* es el array correspondiente a la fila *i*-ésima del array *a*.
- Si *a* es un array de dos dimensiones, la expresión *a[i, j]* equivale a *a[j]*, ya que el operador *,* devuelve el valor de la expresión más a la derecha.
- Existen equivalencias entre los tipos arrays y tipos apuntador, que se verán más adelante en la sección 0.7.

0.6.2 CADENAS DE CARACTERES (1/3)

Una cadena de caracteres (*string*) es una secuencia de cero o más caracteres, dígitos o secuencias de escape.

Declaración de cadenas

En C las cadenas de caracteres se declaran como un array de caracteres:

Sintaxis:

char nombre [tamaño];

En C, toda cadena termina con un *carácter especial nulo* expresado como 0, '\0', o NULL. Por tanto, una cadena declarada como un array de tamaño *N* puede almacenar como máximo *N-1* caracteres, ya que el último elemento siempre contendrá el carácter especial nulo.

Ejemplo:

```
char miCadena[10]; // miCadena puede contener como máximo 9 caracteres
```


0.6.2 CADENAS DE CARACTERES (2/3)

Declaración e inicialización de cadenas

Existen dos formas de inicializar una cadena:

- Mediante un literal de tipo cadena entre comillas:

```
char saludo[5] = "hola";
```

- Como una inicialización de array de char:

```
char saludo[5] = {'h','o','l','a',0};
```

En estos dos casos es frecuente omitir el tamaño de la cadena:

```
char saludo[] = "hola";  
char saludo[] = {'h','o','l','a',0};
```

Si se quieren incluir las dobles comillas como un carácter de la cadena, debe utilizarse la secuencia de escape :

```
char saludo[] = "\"hola\"";  
char saludo[] = {'\\','h','o','l','a','\\',0};
```

0.6.2 CADENAS DE CARACTERES (3/3)

Asignación de cadenas

La asignación de valores a cadenas de caracteres mediante el operador de asignación se deberá hacer carácter a carácter.

Ejemplo:

```
char saludo1[5];
saludo1[0] = 'h';
saludo1[1] = 'o';
saludo1[2] = 'l';
saludo1[3] = 'a';
saludo1[4] = 0;
char saludo2[5];
for(int i=0; i<5; i++) saludo2[i] = saludo1[i];
```

No está permitido utilizar el operador de asignación con cadenas de caracteres, salvo en la declaración con inicialización. Los ejemplos siguientes **son incorrectos**:

```
saludo1 = "hola";
saludo2 = saludo1;
```

Otras operaciones con cadenas de caracteres:

Las operaciones fundamentales para manejar cadenas de caracteres, tales como copia, comparación, lectura, escritura, etc, se ofrecen en bibliotecas estándar de C como `<string.h>` y `<stdio.h>` y requieren el uso de apuntadores, por lo que se verán en la sección 0.7.

0.7 APUNTAADORES

- 0.7.1 Definición de apuntador
- 0.7.2 Declaración de apuntadores
- 0.7.3 Asignación de direcciones
- 0.7.4 Operador de desreferencia
- 0.7.5 El apuntador nulo
- 0.7.6 Paso de apuntadores como parámetros
- 0.7.7 Algunas consideraciones sobre apuntadores
- 0.7.8 Aritmética y comparación de apuntadores
- 0.7.9 Apuntadores y arrays
- 0.7.10 Apuntadores y cadenas de caracteres
- 0.7.11 Apuntadores y estructuras

0.7.1 DEFINICIÓN DE APUNTADOR

Un apuntador almacena la *dirección de memoria* de una variable o constante previamente almacenada. La variable apuntada puede ser de un tipo tanto primitivo como definido por el programador.

Algunas de las **utilidades** de los apuntadores en C son:

- Paso de parámetros por *referencia* en funciones.
- Implementación de *estructuras enlazadas*.
- Arrays y cadenas de caracteres.
- Gestión eficiente de la memoria.

0.7.2 DECLARACIÓN DE APUNTADES

Para declarar un apuntador se especifica un nombre para la variable apuntador y un tipo de dato que indica de qué tipo de variables el apuntador almacenará direcciones de memoria. También hay que incluir el *simbolo* ***** antes del nombre de la variable apuntador.

Sintaxis:

*tipo * nombre_variable_apuntador;*

Ejemplo:

```
double * p;           // p es un apuntador a un tipo double
char * a, * b;        // a y b son apuntadores a tipo char
int * c, d;           // c es un apuntador a int y d es un int
```

0.7.3 ASIGNACIÓN DE DIRECCIONES

El *operador de referencia* `&` devuelve la dirección de una variable. Por tanto la operación `&` puede utilizarse para extraer direcciones de variables y asignarlas a variables apuntador con el operador de asignación:

nombre_variable_apuntador = & nombre_variable;

Podemos asignar también a una variable apuntador el valor de otra variable apuntador.

Ejemplo:

```
double d;      // Declara una variable d de tipo double
double * p1;   // Declara un apuntador p1 a double
double * p2;   // Declara un apuntador p2 a double
p1 = &d;       // Asigna a p1 la dirección de la variable d
p2 = p1;       // Asigna a p2 el valor de p1
```

En este caso las dos variables apuntador `p1` y `p2` apuntan a la misma variable `d`.

Otro mecanismo de asignar direcciones de memoria a la variable apuntador es mediante el uso de *asignación dinámica de memoria* (sección 0.8).

0.7.4 OPERADOR DE DESREFERENCIA

El *operador de desreferencia* `*` (o de *indirección*) accede al contenido de la variable apuntada por un apuntador. Nótese que este operador coincide con el utilizado para declarar variables de tipo apuntador.

Sintaxis:

`* nombre_variable_apuntador;`

Puede estar en la parte izquierda de una sentencia de asignación, para asignar un valor a la variable apuntada, o formando parte de una expresión, para devolver el valor de la variable apuntada.

Ejemplo:

```
double d;           // Declara una variable d de tipo double
double * p1;        // Declara un apuntador p1 a double
double * p2;        // Declara un apuntador p2 a double
p1 = &d;            // Asigna a p1 la dirección de la variable d
p2 = p1;            // Asigna a p2 el valor de p1
*p1 = 3.6;          // Asigna 3.6 a la variable d. Mismo efecto que d = 3.6
double e = *p2;     // Asigna a e el valor de la variable d.
                    // Mismo efecto que e = d
```

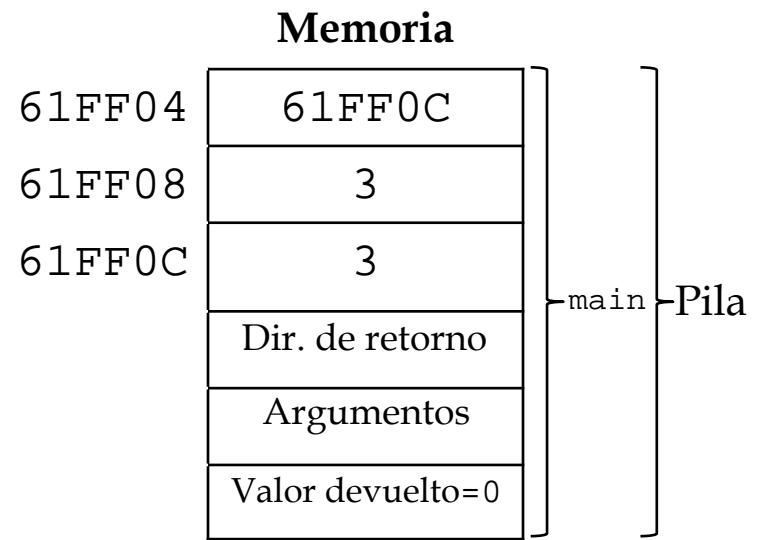
0.7.4 OPERADOR DE DESREFERENCIA (2/2)

Tabla de símbolos

Identificador	Tipo	Dirección
n	int	61FF0C
m	int	61FF08
p	int *	61FF04

```
int main( ) {  
    int n = 3;  
    int m;  
    int * p;  
    p = &n;  
    m = *p;  
    return 0;  
}
```

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque como se hace en realidad).



0.7.5 EL APUNTADOR NULO

El apuntador nulo es un caso especial de apuntador, su valor es 0.

En la biblioteca `<stdio.h>` se define la macro `NULL` con valor 0 para representar al apuntador nulo.

Suele utilizarse para *inicializar apuntadores* y como *marca de fin* en estructuras enlazadas (Tema 2).

Ejemplo:

```
double * p1;  
p1 = NULL;           // Mismo efecto que p1 = 0  
double * p2 = NULL;  // Mismo efecto que double * p2 = 0
```

0.7.6 PASO DE APUNTADES COMO PARÁMETROS (1/4)

Aunque el paso de parámetros en C es siempre por valor, se puede simular un *paso por referencia* mediante el uso de apuntadores.

Para ello:

1. El parámetro formal en la función se declara como un apuntador al tipo del parámetro que se va a pasar por referencia.
2. En la llamada a la función, se pasa la dirección del parámetro actual con el operador de referencia & seguido del nombre del parámetro actual.
3. En el código de la función se accede al contenido del parámetro actual mediante el operador de desreferencia * seguido del nombre del parámetro formal.

0.7.6 PASO DE APUNTADES COMO PARÁMETROS (2/4)

Ejemplo (intercambio del contenido de dos variables):

```
void intercambia(int * n, int * m)
{
    int aux = *n;
    *n = *m;
    *m = aux;
}

int main()
{
    int n = 1;
    int m = 2;
    intercambia(&n,&m);          // se pasa la dirección de n y m
    printf("n=%d m=%d\n",n,m);  // Imprime n=2 m=1
    return 0;
}
```

0.7.6 PASO DE APUNTADES COMO PARÁMETROS (3/4)

Ejemplo: función `scanf`

Sintaxis: `int scanf (const char * formato, ...);`

La función `scanf` de la biblioteca `<stdio.h>` recibe datos del stream de entrada estándar `stdin` (correspondiente al teclado) bajo el control de la cadena apuntada por ***formato*** que especifica las secuencias de entrada permitidas y cómo han de ser convertidas para su asignación. Los puntos suspensivos indican que la función `scanf` admite un número variable de argumentos (*función variádica*). Los argumentos deben corresponderse apropiadamente con los especificadores de conversión.

Los especificadores de conversión son los mismos que en la función `printf`:

- `%c` `char`
- `%d` `int`
- `%f` `float`
- `%lf` `double`
- `%s` `char *`

En la llamada, se debe incluir el operador `&` en el paso de argumentos, ya que estos se van a modificar. En el caso de lectura de cadenas de caracteres no se pone el operador `&` puesto que una cadena de caracteres es ya un apuntador.

Devuelve el número de datos de entrada asignados, o el valor de la macro `EOF` (igual a -1) si ocurre un error de entrada antes de realizar ninguna conversión.

0.7.6 PASO DE APUNTADES COMO PARÁMETROS (4/4)

Ejemplo: función `scanf`

```
int a;  
float f;  
double d;  
char c;  
char cadena[10];  
/* Lee desde el teclado un entero, un real, un real de  
   doble precisión, un carácter y una cadena de  
   caracteres. */  
scanf("%d %f %lf %c %s", &a, &f, &d, &c, cadena);
```

0.7.7 ALGUNAS CONSIDERACIONES SOBRE APUNTADES

- Es necesario que los apuntes estén correctamente inicializados antes de acceder a la zona de memoria a la que apuntan.

```
int * p;  
*p = 3; // ERROR: Se accede a una zona desconocida  
        // de memoria
```

- Es posible que un apuntes a un tipo determinado apunte a una variable de cualquier otro tipo. El compilador da un aviso, pero no un error. En general no deben realizarse este tipo de acciones.
- Los apuntes que apuntan a variables automáticas (locales) no deben utilizarse cuando el bloque de la variable automática ha terminado, ya que la variable automática ha sido eliminada de la pila.

0.7.8 ARITMÉTICA Y COMPARACIÓN DE APUNTADES

El lenguaje C permite sumar o restar cantidades enteras a un apuntador, así como la comparación entre apuntadores.

Se pueden utilizar los siguientes operadores aritméticos:

- Operadores binarios: +, -
- Operadores unarios: --, ++ (pre y post)
- Operadores de asignación: +=, -=

Además, los apuntadores pueden ser comparados usando los operadores relacionales ==, <=, >=, <, >, !=

Si *p* es un apuntador, la expresión *p+i* devuelve una dirección (apuntador) que es la dirección de *p* sumándole *i* veces el tamaño (en bytes) del tipo al que apunta *p*.

0.7.9 APUNTADORES Y ARRAYS

0.7.9.1 Apuntadores y arrays de una dimensión

0.7.9.2 Apuntadores y arrays de múltiples dimensiones

0.7.9.3 Paso de arrays como parámetros a funciones

0.7.9.1 APUNTADORES Y ARRAYS DE UNA DIMENSIÓN

Si v es un array de una dimensión, la expresión $v[i]$ es equivalente a la expresión $*(v+i)$. Esto es así porque **un array se representa mediante una dirección constante cuyo valor es la dirección del primer elemento**. El nombre de un array puede utilizarse por tanto como un apuntador al que se le puede aplicar aritmética de apuntadores, excepto que no puede alterarse su valor inicial constante.

El nombre de un array de elementos de un tipo concreto puede usarse también para asignar su dirección de comienzo a una variable apuntador a ese tipo. La siguiente asignación es por tanto compatible:

```
int v[10];           // Array de 10 enteros
int * p = v;         // Mismo efecto que int * p = &v[0];
```

Ejemplo (inicializar a cero los elementos de un array):

```
int v[10];           // Array de 10 enteros
for(int i=0; i<10; i++)
    *(v+i) = 0;       // Mismo efecto que v[i] = 0;
```

0.7.9.2 APUNTADES Y ARRAYS DE MÚLTIPLES DIMENSIONES

Los arrays de múltiples dimensiones reciben un tratamiento especial.

Supongamos, sin pérdida de generalidad, arrays de dos dimensiones. Si `v` es un array de dos dimensiones, entonces la expresión `v[i][j]` es equivalente a la expresión `*(*(v+i)+j)`.

Dado que los elementos de un array de múltiples dimensiones se almacenan de forma contigua en memoria, un array de dos dimensiones de enteros, de tipo `int[N][M]`, **no es compatible** con los tipos `int*[M]` ni `int**`, ya que estos últimos no asumen la contigüidad de los datos, por lo que **un array de múltiples dimensiones de tipo `int[N][M]`, no puede ser asignado a variables de tipos `int*[M]` ni `int**`.**

Ejemplo (inicializar a cero los elementos de un array de dos dimensiones):

```
int v[10][5];           // Array (10 x 5)
for(int i=0;i<10;i++)
    for(int j=0;j<5;j++)
        *(*(v+i)+j) = 0; // Mismo efecto que v[i][j] = 0;
```

NOTA: Aunque la expresión `v[i]` es equivalente a la expresión `*(v+i)`, y la expresión `v[i][j]` es equivalente a `*(*(v+i)+j)`, es usual utilizar siempre el operador de índice `[]` para acceder a los elementos de un array.

0.7.9.3 PASO DE ARRAYS COMO PARÁMETROS A FUNCIONES (1/2)

Cuando un array se pasa como parámetro a una función, el parámetro formal puede definirse con alguna de las siguientes **sintaxis**:

1. *tipo * identificador*
2. *tipo identificador []*
3. *tipo identificador [literal]* (el literal es ignorado)

En el paso de un array como parámetro a una función, se pasa por valor la dirección del primer elemento del array. Por tanto, una modificación del contenido del array a través del parámetro formal tiene efecto se mantiene cuando finaliza la función, ya que se simula **un paso por referencia**.

En un array de M dimensiones se usarán tantos operadores `[]` como dimensiones, pero **se deben definir los tamaños de, al menos, las $M-1$ últimas dimensiones** (el tamaño de la primera dimensión es opcional y si se define es ignorado).

No es posible conocer el número de elementos del array en el ámbito de la función, por lo que en los casos que se requiera **se debe pasar también como parámetro el tamaño del array**.

0.7.9.2 PASO DE ARRAYS COMO PARÁMETROS A FUNCIONES (2/2)

Ejemplo (función que inicializa a cero los elementos de un vector):

```
void InicializarVector(int * v, int n)// Mismo efecto con int v[]
{
    for(int i=0;i<n;i++)
        v[i] = 0; // Mismo efecto que *(v+i) = 0;
}
int main(){ int v[5]; InicializarVector(v,5); return 0;}
```

Ejemplo (función que inicializa a cero los elementos de un array de dos dimensiones):

```
void InicializarMatriz(int v[][5], int n)
{
    for(int i=0;i<n;i++)
        for(int j=0;j<5;j++)
            v[i][j] = 0; // Mismo efecto que (*(v+i)+j) = 0;
}
int main(){ int v[10][5]; InicializarMatriz(v,10); return 0;}
```

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (1/7)

De igual forma, una cadena de caracteres es compatible con el tipo `char *`, dado que una cadena de caracteres es un array de `char`. La siguiente asignación es por tanto correcta:

```
char s[10];    // Array de 10 caracteres
char * p = s;  // Mismo efecto que char * p = &s[0];
```

Por otra parte, si `s` es una cadena de caracteres, la expresión `s[i]` es equivalente a `*(s+i)`, es decir, devuelve el carácter `i`-ésimo de la cadena de caracteres.

Ejemplo (imprimir una cadena de caracteres carácter a carácter):

```
char s[] = "hola";           // Cadena de caracteres
int i = 0;
while(*(s+i)!=0)             // Mismo efecto que while (s[i]!=0)
{
    printf("%c",*(s+i)); // Mismo efecto que printf("%c",s[i]);
    i++;
}
```

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (2/7)

Biblioteca <string.h>

La biblioteca <string.h> ofrece una serie de recursos de utilidad para trabajar con cadenas de caracteres. A continuación se muestran algunas de las funciones más utilizadas:

Copia (strcpy):

*char * strcpy(char * s1, const char * s2);*

Copia la cadena apuntada por s2, incluyendo el carácter nulo, a la cadena apuntada por s1. Devuelve el valor de s1. Si la cadena apuntada por s1 no tiene suficiente espacio para almacenar la cadena apuntada por s2, entonces el comportamiento no está definido.

Ejemplo:

```
char cadena1[10];  
strcpy(cadena1, "hola");  
char cadena2[20];  
strcpy(cadena2, cadena1);
```

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (3/7)

Biblioteca `<string.h>`

Concatenación (`strcat`):

*`char * strcat(char * s1, const char * s2);`*

Añade una copia de la cadena apuntada por `s2` incluyendo el carácter nulo, al final de la cadena apuntada por `s1`. El carácter inicial de la cadena apuntada por `s2` sobrescribe el carácter nulo al final de la cadena apuntada por `s1`. Devuelve el valor de `s1`. Si la cadena apuntada por `s1` no tiene suficiente espacio para almacenar la cadena apuntada por `s2`, entonces el comportamiento no está definido.

Ejemplo:

```
char cadena1[10] = "hola";  
char cadena2[10] = " mundo";  
char saludo[20];  
strcpy(saludo, cadena1);  
strcat(saludo, cadena2);
```

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (4/7)

Biblioteca `<string.h>`

Longitud (`strlen`):

*`unsigned int strlen(const char * s);`*

Devuelve el número de caracteres de la cadena apuntada por `s`, sin incluir el carácter nulo.

Ejemplo:

```
char cadena[10] = "hola";  
int n = strlen(cadena); // Asigna a n el valor 4
```


0.7.10 APUNTADORES Y CADENAS DE CARACTERES (5/7)

Biblioteca `<string.h>`

Comparación (`strcmp`):

```
int strcmp(const char * s1, const char * s2);
```

Compara la cadena apuntada por `s1` con la cadena apuntada por `s2`. Devuelve un número mayor, igual o menor que cero, según sea la cadena apuntada por `s1` mayor, igual o menor que la cadena apuntada por `s2`, en orden lexicográfico.

Ejemplo (comprobar si una cadena de caracteres existe en un array):

```
int busca(char texto[][20], int n, char * s) {
    int i=0;
    while((i<n)&&strcmp(texto[i],s)) i++;
    return (i<n);
}

int main(){
    char texto[][20]={"hola","mundo"};
    char s[20];
    scanf("%s",s);
    if (busca(texto,2,s)) printf("La cadena se encuentra en el texto.\n");
    else printf("La cadena no se encuentra en el texto.\n");
    return 0;
}
```

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (6/7)

Biblioteca <stdio.h>

La biblioteca <stdio.h> ofrece una serie de recursos de utilidad para manipular datos de entrada y salida. A continuación se muestran algunas de las funciones más utilizadas con cadenas de caracteres.

Visualización de cadenas de caracteres (puts):

```
int puts(const char * s);
```

Escribe la cadena apuntada por *s* en el stream de salida estándar *stdout* (correspondiente a la pantalla) y añade un carácter de nueva línea a la salida. El carácter nulo final no se escribe.

Devuelve EOF si ocurre un error de escritura; en caso contrario devuelve un valor positivo.

Ejemplo:

```
char cadena[20] = "hola";  
puts(cadena);
```

Otras alternativas para visualizar cadenas de caracteres son las funciones *fputs*, *printf* y *fprintf*.

0.7.10 APUNTADORES Y CADENAS DE CARACTERES (7/7)

Biblioteca `<stdio.h>`

Lectura de cadenas de caracteres (**gets**):

*`char * gets(char * s);`*

Lee una cadena de caracteres desde el stream de entrada estándar `stdin` (correspondiente al teclado) en la cadena apuntada por `s` hasta que encuentra un carácter de nueva línea. El carácter de nueva línea se descarta y se escribe en la cadena `s` un carácter nulo después del último carácter. Si la cadena apuntada por `s` no tiene suficiente espacio para almacenar la cadena leída, entonces el comportamiento no está definido.

Devuelve `s` si la operación se ha realizado con éxito. Si ocurre un error de lectura, el contenido de la cadena apuntada por `s` queda indeterminado y se devuelve el apuntador nulo.

Ejemplo:

```
char s[20];
gets(s);
printf("La cadena leida es :%s\n",s);
```

Otras alternativas para leer cadenas de caracteres son las funciones `fgets`, `scanf` y `fscanf`.

0.7.11 APUNTADORES Y ESTRUCTURAS (1/2)

Un apuntador puede apuntar a un tipo struct.

Ejemplo:

```
typedef struct
{
    float x, y;
} Punto;
Punto p;
Punto * q = &p;
(*q).x = 3.0; // Mismo efecto que p.x = 3.0;
(*q).y = 4.0; // Mismo efecto que p.y = 4.0;
```

0.7.11 APUNTAADORES Y ESTRUCTURAS (2/2)

El operador de miembro -> para struct referenciados con apuntadores

Para acceder a los miembros de una variable de tipo struct mediante un apuntador, el lenguaje C dispone del operador -> con la siguiente sintaxis:

ptr -> miembro

que es equivalente a:

*(*ptr).miembro*

En el ejemplo anterior, las siguientes instrucciones:

```
( *q ) . x = 3 . 0 ;  
( *q ) . y = 4 . 0 ;
```

son equivalentes a:

```
q->x = 3 . 0 ;  
q->y = 4 . 0 ;
```

0.8 ASIGNACIÓN DINÁMICA DE MEMORIA

0.8.1 Definición de variable dinámica

0.8.2 Ámbito y extensión de las variables dinámicas

0.8.3 Reserva de memoria: malloc, calloc, realloc

0.8.4 Liberación de memoria: free

0.8.5 Consideraciones sobre asignación dinámica de memoria

0.8.6 Asignación dinámica de arrays y cadenas

0.8.1 DEFINICIÓN DE VARIABLE DINÁMICA

En esta sección se introducirán los mecanismos de *asignación dinámica de memoria*. Esta técnica de programación trata con *variables dinámicas*, las cuales permiten una gestión de los recursos de memoria más eficiente adaptada a las necesidades de la aplicación.

Una variable es *dinámica* cuando su espacio de almacenamiento se asigna y se libera en **tiempo de ejecución** mediante **instrucciones explícitas** del lenguaje de programación.

No hay que confundir variables dinámicas con variables automáticas, ya que las variables automáticas (locales), aunque también se crean y se destruyen en tiempo de ejecución, este proceso se controla de forma automática por el compilador cuando comienza o termina un bloque.

Las variables dinámicas se almacenan en un segmento de memoria distinto al de las variables estáticas y automáticas, denominado *segmento heap* o *segmento montón*.

0.8.2 ÁMBITO Y EXTENSIÓN DE LAS VARIABLES DINÁMICAS

Las variables dinámicas **no tienen nombre**. Se accede a ellas a través de variables de tipo apuntador, las cuales sí poseen un identificador de variable.

El *ámbito de una variable dinámica* es el ámbito de la variable apuntador que apunta a ella.

La *extensión de una variable dinámica* es desde que la variable se crea en tiempo de ejecución hasta que se destruye, ambas operaciones mediante instrucciones del lenguaje.

0.8.3 RESERVA DE MEMORIA: MALLOC, CALLOC, REALLOC (1/4)

Existen varias funciones de la biblioteca estándar `<stdlib.h>` que permiten **reservar y liberar memoria dinámicamente**.

La función **malloc** reserva un bloque de memoria de un tamaño dado como parámetro. Devuelve un apuntador al inicio de la zona de memoria reservada.

Sintaxis:

```
void * malloc( unsigned int numero_bytes );
```

El tamaño se especifica en bytes y se garantiza que la zona de memoria reservada no está ocupada por ninguna otra variable dinámica. Si **malloc** no puede reservar un bloque porque no hay memoria suficiente, devuelve el apuntador nulo (NULL).

0.8.3 RESERVA DE MEMORIA: MALLOC, CALLOC, REALLOC (2/4)

La función **malloc** devuelve un apuntador a un tipo no especificado `void`. Un apuntador a `void` puede convertirse a un apuntador a cualquier otro tipo mediante el operador *cast*.

Ejemplo:

```
int * p = (int *)malloc(4*100);
```

Esta instrucción reserva una variable dinámica de 400 bytes para albergar a 100 enteros (cada entero ocupa 4 bytes).

El casting explícito no es necesario, ya que se hace un casting implícito al evaluar la expresión. La siguiente instrucción sería por tanto equivalente a la anterior:

```
int * p = malloc(400);
```

En adelante no se realizará casting explícito en estos casos.

0.8.3 RESERVA DE MEMORIA: MALLOC, CALLOC, REALLOC (3/4)

El operador **sizeof** devuelve el tamaño (en bytes) de un tipo o una expresión.

Sintaxis:

sizeof (tipo)

sizeof expresión

Si el operando es una expresión devuelve el tipo que debería devolver la expresión si fuese evaluada.

Este operador se puede utilizar en conjunción con la función `malloc` para determinar el tamaño de memoria que se quieren reservar.

Ejemplo (reserva 100 enteros de tipo `int`):

```
int * p = malloc(sizeof(int)*100);
```

0.8.3 RESERVA DE MEMORIA: MALLOC, CALLOC, REALLOC (4/4)

La función **calloc** es similar a **malloc**, pero inicializa con el valor 0 todos los bits reservados.

Sintaxis: *void * calloc (unsigned int numero_elem, unsigned int tam_elem);*

La función **realloc** cambia el tamaño de una variable dinámica a un nuevo tamaño.

Sintaxis: *void * realloc(void * ptr, unsigned int numero_bytes);*

Cambia el tamaño del bloque de memoria apuntado por `ptr` al tamaño especificado por `numero_bytes`. El contenido del bloque de memoria se conserva hasta el menor de los tamaños nuevo y viejo. Si el tamaño nuevo es mayor que el tamaño viejo, la memoria añadida no será inicializada. Si `ptr` es `NULL`, entonces la llamada `realloc(ptr, numero_bytes)` es equivalente a `malloc(numero_bytes)`. Si `ptr` no es `NULL`, y `numero_bytes` es cero, entonces la llamada `realloc(ptr, numero_bytes)` es equivalente a `free(ptr)` (sección 0.8.4). A menos que `ptr` sea `NULL`, su valor debe haber sido devuelto por una llamada previa a `malloc`, `calloc` o `realloc`. Si se ha requerido mover el bloque de memoria apuntado por `ptr`, entonces se libera todo el bloque antiguo.

0.8.4 LIBERACIÓN DE MEMORIA: FREE

La función **free** libera el bloque de memoria apuntado por un apuntador previamente devuelto por una llamada a `malloc`, `calloc` o `realloc`. En otro caso el comportamiento de la llamada está indefinido.

Sintaxis:

*`void free (void * apuntador);`*

Ejemplo:

```
int * p = malloc(sizeof(int)*100);  
... // Código que utiliza la variable dinámica apuntada por p  
free(p);
```

0.8.5 CONSIDERACIONES (1/2)

Es importante no dejar nunca una zona de memoria reservada con `malloc`, `calloc` o `realloc` sin ningún apuntador que la apunte, ya que esta zona no podría ser accesible ni liberada.

El programador es responsable de liberar la memoria cuando sea necesario utilizando `free`.

Si una zona liberada por `free` estaba apuntada por otros apuntadores, estos otros apuntadores pasarán ahora a apuntar a una zona no reservada.

0.8.5 CONSIDERACIONES (2/2)

Tabla de símbolos

Identificador	Tipo	Dirección
n	int	61FF0C
p	int *	61FF08
q	int *	61FF04

```
#include <stdlib.h>
int main( ) {
    int n = 3;
    int * p = malloc(sizeof(int));
    int * q = malloc(sizeof(int));
    *p = n;
    *q = *p+1;
    free(p);
    free(q);
    return 0;
}
```

Nota: Las direcciones son reales obtenidas con un Procesador Intel(R) Core(TM) i5-4460 CPU@ 3.20GHz 8,00 GB, GNU GCC compiler. No se han visualizado los valores de los registros modificados por las funciones ni las direcciones al frame anterior. Las direcciones en la tabla de símbolos son absolutas (no relativas al comienzo del bloque como se hace en realidad).

Tecnología de la Programación
Tema 0. Introducción a C

Memoria

61FF04	9967128	main	Pila
61FF08	9967096		
61FF0C	3		
	Dir. de retorno		
	Argumentos		
	Valor devuelto=0		Montón
	...		
9967096	3		
	...		
9967128	4		
	...		

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (1/10)

Como se vio en secciones anteriores, un apuntador a un tipo es compatible con un array de una dimensión de elementos de ese tipo. Por tanto, una vez reservada una variable dinámica de n elementos de un tipo mediante `malloc`, el apuntador devuelto puede tratarse como un array de n elementos de ese tipo.

Ejemplo (Reservar de forma dinámica e inicializar a cero n enteros contiguos en memoria heap):

```
int n = 10;    // Número de elementos
int * p = malloc(sizeof(int)*n);
for(int i=0; i<n; i++)
    p[i] = 0;
...
free(p);
```

Lo mismo ocurre con las cadenas de caracteres. **Ejemplo:**

```
char * cadena = malloc(sizeof(char)*20);
strcpy(cadena, "hola");
printf("cadena = %s", cadena);
...
free(cadena);
```


0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (2/10)

Para reservar memoria de forma dinámica que permita representar un array de dos dimensiones $n \times m$ (n filas por m columnas), por ejemplo, de elementos de tipo `int`, se puede declarar una variable de tipo `int **` (**apuntador de apuntadores**). Es decir, un apuntador a un bloque de memoria que representa las n filas, donde cada elemento es un apuntador a un bloque de memoria que contiene los m elementos de esa fila.

Se deberá, por tanto, hacer una primera llamada a `malloc` para reservar el bloque de memoria que represente las n filas, y posteriormente realizar n llamadas a `malloc` para reservar los m elementos de cada una de las n filas.

Para liberar el array, primero deberán realizarse n llamadas a `free` para liberar los m elementos de cada una de las n filas, y posteriormente una última llamada a `free` que libere el bloque de memoria que representaba las n filas.

Ejemplo (Reservar de forma dinámica un array de $n \times m$ enteros):

```
int ** a = malloc(sizeof(int*)*n);
for(int i=0; i<n; i++)
    a[i] = malloc(sizeof(int)*m);
...
for(int i=0; i<n; i++)
    free(a[i]);
free(a);
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (3/10)

Paso de parámetros de arrays reservados dinámicamente

Para arrays de una dimensión creados dinámicamente, el parámetro formal de una función se define de igual forma que el paso de los arrays creados en el segmento de datos o pila (sección 0.7.9.3).

Para arrays de dos dimensiones creados dinámicamente, el parámetro formal debe definirse como un doble apuntador o como un array de apuntadores.

Ejemplo (prototipo de una función que imprime una matriz creada dinámicamente de dos dimensiones de enteros):

```
void ImprimeMatriz(int ** a, int n, int m);
```

o también:

```
void ImprimeMatriz(int * a[], int n, int m);
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (4/10)

Funciones que devuelven apuntadores

Una función no puede devolver un tipo array. En su lugar, la función puede devolver un apuntador al bloque de memoria que contiene el array. Su prototipo debe ser el siguiente:

*tipo * nombreFuncion (<lista de parametros>)*

En estos casos, el array debe construirse en el interior de la función y finalmente devolverse. Esta construcción ha de realizarse mediante asignación dinámica de memoria con las funciones `malloc`, `calloc` o `realloc`.

Para el caso de arrays de dos dimensiones, se devolverá un apuntador a un bloque de memoria que representa las filas, donde cada elemento es un apuntador a un bloque de memoria que contiene los elementos de esa fila. Su prototipo es el siguiente:

*tipo ** nombreFuncion (<lista de parametros>)*

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (5/10)

Ejemplo (función que devuelve un array de n enteros):

```
int * CreaVector(int n)
{
    int * v = malloc(sizeof(int)*n); // La memoria del array
                                     // se reserva en el heap
    return v; // Retorna una dirección del heap
}
```

El siguiente ejemplo es **incorrecto**, ya que la función devuelve una dirección de la pila, lo que puede producir un fallo puesto que al retornar la función, la memoria del array en la pila se libera.

```
int * CreaVector(int n)
{
    int v[n]; // La memoria del array se reserva en la pila
    return v; // Retorna una dirección de la pila
}
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (6/10)

Ejemplo (función que imprime un array de n enteros):

```
void ImprimeVector(int * v, int n)
// Mismo efecto que void ImprimeVector(int v[], int n)
{
    for(int i=0; i<n; i++)
        printf("%d ",v[i]);
}
```

Ejemplo (función que crea, imprime y libera un vector de 3 enteros):

```
int main()
{
    int n = 3;
    int * v = CreaVector(n);
    ImprimeVector(v,n);
    free(v);
    return 0;
}
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (7/10)

Ejemplo (función que crea dinámicamente una matriz de $n \times m$ enteros):

```
int ** CreaMatriz(int n, int m)
{
    int ** a = malloc(sizeof(int*)*n);
    for(int i=0; i<n; i++)
        a[i] = malloc(sizeof(int)*m);
    return a;
}
```

Ejemplo (función que imprime una matriz de $n \times m$ enteros creada dinámicamente):

```
void ImprimeMatriz(int ** a, int n, int m)
// Mismo efecto que void ImprimeMatriz(int * a[], int n, int m)
{
    for(int i=0; i<n; i++)
    {
        for(int j=0; j<m; j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
}
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (8/10)

Ejemplo (función que libera una matriz de $n \times m$ enteros creada dinámicamente):

```
void LiberaMatriz(int ** a, int n)
// Mismo efecto que void LiberaMatriz(int * a[], int n)
{
    for(int i=0; i<n; i++)
        free(a[i]);
    free(a);
}
```

Ejemplo (función que crea, imprime y libera una matriz de 2×3 enteros):

```
int main()
{
    int n = 2;
    int m = 3;
    int ** a = CreaMatriz(n,m);
    ImprimeMatriz(a,n,m);
    LiberaMatriz(a,n);
    return 0;
}
```

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (9/10)

Ejemplo (función `main`):

Todo programa C requiere definir una función llamada `main`. Esta función se invoca desde el sistema operativo cuando comienza la ejecución del programa. La orden de ejecución del programa puede contener argumentos que se le pasarán a la función `main`, y ésta puede devolver un valor `int` que informe de cómo ha terminado la ejecución.

La función `main` puede entonces tener una serie de parámetros formales opcionales. También es opcional devolver un entero o no devolver nada.

Prototipo de la función `main`:

```
int main ( int argc, char * argv[] );
```

o también:

```
int main ( int argc, char ** argv );
```

donde:

- **argc** es el número de argumentos (el nombre de la aplicación se considera como un argumento más).
- **argv** es un array de cadenas de caracteres de tamaño **argc** que contiene los argumentos (el nombre de la aplicación se ubica en **argv[0]**).

La función `main` devuelve un entero con el resultado de la ejecución.

0.8.6 ASIGNACIÓN DINÁMICA DE ARRAYS Y CADENAS (10/10)

Ejemplo (función main):

```
int main(int argc, char * argv[])
{
    printf("%d\n",argc);
    for(int i=0; i<argc; i++)
        printf("%s ",argv[i]);
    return 0;
}
```

Si en la línea de comandos ejecutamos:

```
C:\Ejemplo>nombre_programa esto es una prueba
```

se imprimiría por pantalla lo siguiente:

```
5
nombre_programa esto es una prueba
```

0.9 FICHEROS

0.9.1 El tipo FILE

0.9.2 Streams estándares: stdin, stdout, stderr

0.9.3 Apertura de ficheros: fopen

0.9.4 Cierre de ficheros: fclose

0.9.5 Lectura de ficheros: fgets, fscanf

0.9.6 Escritura a ficheros: fputs, fprintf

0.9.7 Final de un fichero: feof

0.9.8 Acceso aleatorio: rewind, fseek

0.9.9 Ejemplos

0.9.1 EL TIPO FILE

La biblioteca `<stdio.h>` contiene funciones, macros y tipos para manipular datos de entrada y salida, es decir, lectura y escritura de datos. La lectura y escritura de datos puede realizarse a ficheros.

En C un fichero se declara mediante un *descriptor de fichero* consistente en un **apuntador a un tipo FILE** que ofrece `<stdio.h>`

Sintaxis:

*FILE * descriptor_fichero;*

FILE es el tipo de datos que usa C para representar un *stream* (canal de comunicación). El tipo FILE almacena toda la información necesaria para controlar un stream, incluyendo su *indicador de posición de fichero*, un *apuntador a su buffer* asociado, un *indicador de errores* que registra si se ha producido un error de lectura y/o escritura, y un *indicador de final de fichero* que registra si se ha llegado al final del fichero.

Una vez declarado, el **proceso básico** sobre un fichero consiste en:

1. Apertura del fichero.
2. Lectura de datos del fichero y/o escritura de datos en el fichero.
3. Cierre del fichero.

0.9.2 STREAMS ESTÁNDARES: STDIN, STDOUT, STDERR

Un *stream* es un concepto abstracto de alto nivel para representar un canal de comunicación a un fichero, dispositivo o proceso. Los streams se encargan de convertir cualquier tipo de dato a texto legible por el usuario, y viceversa, además de hacer manipulaciones binarias de los datos y cambiar la apariencia y el formato en que se muestra la salida.

Un stream puede ser, por ejemplo, un **fichero**, el **teclado** (sólo lectura) o la **pantalla** (sólo escritura). En C se establecen los siguientes *streams estándares*:

- **stdin**: stream de entrada estándar (normalmente teclado).
- **stdout**: stream de salida estándar (normalmente pantalla).
- **stderr**: stream de error estándar (normalmente pantalla), usado para mensajes de error y aspectos de diagnóstico del programa.

0.9.3 APERTURA DE FICHEROS: FOPEN

Para abrir un fichero se utiliza la función `fopen` con la siguiente **sintaxis**:

FILE * fopen(const char * nombre, const char * modo);

Abre un fichero cuyo nombre es la cadena apuntada por `nombre`, adjudicándole un stream, y `modo` apunta a una cadena de caracteres que define el modo de apertura:

- **r**: sólo lectura; el fichero debe existir.
- **w**: escritura; se crea un fichero nuevo o se sobrescribe si ya existe.
- **a**: añadir; se abre para escritura, el puntero para escritura se sitúa al final del fichero; si el fichero no existe, se crea.
- **r+**: lectura y escritura; el fichero debe existir.
- **w+**: lectura y escritura; se crea un fichero nuevo o se sobrescribe si ya existe.
- **a+**: añadir, lectura y escritura; se abre para escritura, el puntero para lectura y escritura se sitúa al final del fichero; si el fichero no existe, se crea.
- **t**: fichero de texto; si no se especifica la cadena "t" ni la "b", se asume por defecto que es "t".
- **b**: fichero binario.

La función `fopen` devuelve un apuntador al stream adjudicado, y `NULL` si no se ha podido abrir el fichero (por ejemplo, porque no se ha encontrado).

0.9.4 CIERRE DE FICHEROS: FCLOSE

Para cerrar un fichero previamente abierto se utiliza la función `fclose` con la siguiente **sintaxis**:

```
int fclose ( FILE * f );
```

Despeja el stream apuntado por `f`, cierra el fichero asociado y lo desasocia del stream. Devuelve 0 si el fichero fue cerrado con éxito, y EOF en caso contrario.

Ejemplo:

```
FILE * f;  
char nombre[] = "Datos.txt";  
f = fopen(nombre,"w"); // Abre f para escritura  
...      // Instrucciones de escritura a f  
fclose(f);      // Cierra f  
f = fopen(nombre,"r"); // Abre f para lectura  
...      // Instrucciones de lectura de f  
fclose(f);      // Cierra f
```

0.9.5 LECTURA DE FICHEROS: FGETS, FSCANF

La lectura desde un fichero puede hacerse con distintas funciones. Por ejemplo, la función `fgets` lee una línea de un fichero con la siguiente **sintaxis**:

```
char * fgets ( char * cadena , int n , FILE * f );
```

donde `cadena` es una cadena de caracteres en la cual se guardará la línea leída, `n` es el número máximo de caracteres, incluyendo el carácter nulo, por lo que se leerán un máximo de `n-1` caracteres, y `f` es el descriptor de fichero desde el que se lee. Un carácter nulo es escrito inmediatamente después del último carácter leído en la cadena de caracteres. En indicador de posición de fichero apunta a la posición del fichero donde se va a leer el siguiente dato.

La función `fgets` devuelve `cadena` si es realizada con éxito. Si un final de fichero (EOF) es encontrado y ningún carácter ha sido leído en la cadena de caracteres, entonces el contenido de `cadena` permanece invariable y la función devuelve `NULL`. Si ocurre un error de lectura durante el proceso, el contenido `cadena` queda indeterminado y la función devuelve `NULL`.

Otra función que permite realizar la lectura desde un fichero reconociendo el formato de los datos es `fscanf`, que se utiliza de igual forma que la función `scanf` añadiendo el descriptor de fichero:

```
int fscanf ( FILE * f , const char * formato , ... );
```

Se puede utilizar la función `fscanf` utilizando el stream de entrada estándar. **Ejemplo:**

```
fscanf(stdin,"%s",c);    // Equivale a scanf("%s",c);
```

0.9.6 ESCRITURA A FICHEROS: FPUTS, FPRINTF

La escritura a un fichero también puede hacerse con distintas funciones. Por ejemplo, la función `fputs` escribe una línea en un fichero con la siguiente **sintaxis**:

```
int fputs ( char * cadena , FILE * f );
```

donde `cadena` apunta a una cadena de caracteres que contiene la línea a escribir en el fichero, y `f` es el descriptor de fichero en el que se va a escribir. El carácter nulo no es escrito. Las líneas se van escribiendo en el fichero una detrás de otra, desde el principio del fichero (`w`) o desde el final (`a`).

La función **`fputs`** retorna EOF si ocurre un error de escritura, si no, entonces retorna un valor no negativo.

Para realizar la escritura a un fichero con formato de los datos se puede utilizar `fprintf` de igual forma que la función `printf` añadiendo el descriptor de fichero:

```
int fprintf ( FILE * f, const char * formato, ... );
```

Se puede utilizar la función `fprintf` utilizando el stream de salida estándar. **Ejemplo:**

```
fprintf(stdout, "Hola"); // Equivale a printf("Hola");
```


0.9.7 FINAL DE UN FICHERO: FEOF

La función `fEOF`, con la siguiente **sintaxis**:

```
int fEOF ( FILE * f );
```

Comprueba el indicador de final de fichero para el descriptor de fichero `f`. Devuelve un valor distinto de 0 (cierto) si se ha llegado al final del fichero, y 0 (falso) en caso contrario.

Se utiliza para saber cuándo se ha llegado al final del fichero para dejar de leer.

0.9.8 ACCESO ALEATORIO: REWIND, FSEEK

La función `rewind`, con la siguiente **sintaxis**:

```
void rewind ( FILE * f );
```

mueve el indicador de posición de fichero asociado a **f** al comienzo del fichero.

La función `fseek`, con la siguiente **sintaxis**:

```
int fseek ( FILE * f, long int desplazamiento, int origen );
```

mueve el indicador de posición de fichero asociado a **f** a una posición con un desplazamiento respecto a **origen**, que puede ser:

SEEK_SET: desde el principio del fichero.

SEEK_CUR: desde la posición del puntero.

SEEK_END: desde el final del fichero.

La función `fseek` devuelve 0 si la función se ha realizado con éxito, y un valor distinto de 0 en caso contrario.

Ejemplo:

```
fseek(f,-5,SEEK_END); // Posición 5 bytes antes el final
fgets(cadena,5,f);    // Lee los primeros 4 char
fseek(f,5,SEEK_SET);  // Posición 5 bytes después del inicio
fgets(cadena,5,f);    // Lee los primeros 4 char
```

0.9.9 EJEMPLOS (1/3)

El siguiente programa asigna, a los elementos de un array de enteros, los valores leídos desde un fichero de texto y posteriormente escribe los valores de los elementos del array en otro fichero de texto.

```
void LeeArray(char * nombreFichero, int v[], int n)
{
    FILE * f = fopen(nombreFichero,"r");
    for(int i=0; i<n; i++)
        fscanf(f,"%d",&v[i]);
    fclose(f);
}
void EscribeArray(char * nombreFichero, int v[], int n)
{
    FILE * f = fopen(nombreFichero,"w");
    for(int i=0; i<n; i++)
        fprintf(f,"%d ",v[i]);
    fclose(f);
}
int main()
{
    int n = 10;
    int v[n];
    LeeArray("f1.txt",v,n);
    EscribeArray("f2.txt",v,n);
    return 0;
}
```

0.9.9 EJEMPLOS (2/3)

El siguiente programa copia un fichero de texto, línea a línea a otro fichero de texto. La función `CopiaTexto` recibe como parámetros los descriptores de fichero, por lo que puede ser utilizada con los streams estándares.

```
void CopiaTexto(FILE * f1, FILE * f2)
{
    int n = 100;
    char s[n];
    while(!feof(f1))
    {
        fgets(s,n,f1);
        fputs(s,f2);
    }
}

int main()
{
    FILE * f1 = fopen("f1.txt","r");
    FILE * f2 = fopen("f2.txt","w");
    CopiaTexto(f1,f2);
    fclose(f1);
    fclose(f2);
    return 0;
}
```

0.9.9 EJEMPLOS (3/3)

El siguiente programa copia un fichero de texto, carácter a carácter a otro fichero de texto.

```
void CopiaTexto(FILE * f1, FILE * f2)
{
    char c;
    while(!feof(f1))
    {
        fscanf(f1,"%c",&c);
        if (!feof(f1)) fprintf(f2,"%c",c);
    }
}

int main()
{
    FILE * f1 = fopen("f1.txt","r");
    FILE * f2 = fopen("f2.txt","w");
    CopiaTexto(f1,f2);
    fclose(f1);
    fclose(f2);
    return 0;
}
```

0.10 GESTIÓN DE ERRORES (1/2)

Muchas de las funciones de las bibliotecas de C devuelven un valor especial para indicar si ha habido un fallo o no. Típicamente este valor especial es -1, un apuntador NULL, o una constante definida para este propósito como por ejemplo EOF.

Sin embargo, este valor informa sólo de que ha ocurrido un error. Para conocer que clase de error se ha producido se puede consultar el código de error almacenado en la variable `errno` que contiene el valor del **último error producido** y está declarada en el fichero de cabecera `<errno.h>` como una variable de tipo `extern int`. La biblioteca `<errno.h>` también define macros para los códigos de error producidos durante la ejecución de un programa.

Los mensajes de error particulares asociados a los valores de la variable `errno` se pueden obtener usando la función `strerror` de la biblioteca `<string.h>` con la siguiente sintaxis:

```
char * strerror(int numeroError);
```

Otra opción es imprimirlo directamente en el stream estándar de error `stderr` mediante la función `perror` de la biblioteca `<stdio.h>` con la siguiente sintaxis:

```
void perror(const char * mensajePrevio);
```

La cadena de caracteres `mensajePrevio` es un mensaje definido por el programador que se imprime antes del propio mensaje de error. Si esta cadena es la cadena vacía o el valor NULL, entonces no se imprime ningún mensaje previo.

0.10 GESTIÓN DE ERRORES (2/2)

Ejemplo:

```
#include <errno.h>
#include <string.h>
#include <stdio.h>
int LeeEnteroError(char * nombreFichero, int * n)
{
    FILE * f = fopen(nombreFichero,"r");
    if(f == NULL) return -1;
    fscanf(f,"%d",n);
    fclose(f);
    return 0;
}
int main()
{
    int m;
    char * nombreFichero = "ff1.txt";
    if (LeeEnteroError(nombreFichero,&m)==0) printf("m = %d\n",m);
    else
    {
        fprintf(stderr,"No se puede abrir el archivo %s\n",nombreFichero);
        fprintf(stderr,"Error numero %i\n",errno);
        fprintf(stderr,"Descripcion: %s \n", strerror(errno));
        // Mismo efecto que perror("Descripcion");
    }
    return 0;
}
```

0.11 EL PREPROCESADOR DE C

0.11.1 Definición de preprocesador

0.11.2 Inclusión de ficheros

0.11.3 Definición de macros

0.11.4 Compilación condicional

0.11.1 DEFINICIÓN DE PREPROCESADOR

Un *preprocesador* es un programa separado que es invocado por el compilador antes de que comience la fase de compilación.

El *preprocesador de C* es un procesador de macros que el compilador de C utiliza de forma automática para transformar el programa. Permite eliminar comentarios y reconocer una serie de directivas para:

1. Incluir ficheros.
2. Definir macros.
3. Incluir código de forma condicional (compilación condicional).

El preprocesador tiene su **propio lenguaje** independiente del lenguaje C y todas sus órdenes comienzan con el **carácter #**.

0.11.2 INCLUSIÓN DE FICHEROS

Sintaxis:

#include <fichero>

#include "fichero"

Se inserta el código del fichero dentro del código fuente actual. Si el nombre del fichero se escribe entre < >, el fichero se busca en una lista estándar de directorios del sistema; si el nombre se escribe entre " ", entonces se busca primero en el directorio actual, después en los directorios indicados por el usuario y finalmente en los directorios propios del sistema.

Ejemplo:

```
#include <stdio.h>
int main()
{
    printf ("Hola, mundo\n");
    return 0;
}
```

0.11.3 DEFINICIÓN DE MACROS

Sintaxis:

```
#define nombre_macro codigo_sustituto  
#define nombre_macro ( par1, par2, ... ) codigo_sustituto  
#undef nombre_macro
```

La directiva `#define` sustituye bien un identificador, bien una función, por un fragmento de código.

Ejemplo:

```
#define MAX 100  
#define MULTIPLICA(x,y) (x)*(y)  
...  
#undef MAX  
#undef MULTIPLICA
```

0.11.4 COMPILACIÓN CONDICIONAL (1/3)

La compilación condicional permite insertar para su compilación o no determinadas partes de código según se cumplan ciertas condiciones.

El preprocesador de C permite realizar compilación condicional utilizando las *directivas condicionales*:

#if

#ifdef

#ifndef

0.11.4 COMPILACIÓN CONDICIONAL (2/3)

Sintaxis:

```
#ifdef nombre_macro  
    código-si-definida  
#else  
    código-si-no-definida  
#endif
```

Inserta para su compilación *codigo-si-definida*, si está definida la macro *nombre_macro*, y en otro caso inserta *codigo-si-no-definida*.

Ejemplo:

```
int a,b,c;  
#ifdef MULTIPLICA  
    c = MULTIPLICA(a,b);  
#else  
    c = a*b;  
#endif
```

0.11.4 COMPILACIÓN CONDICIONAL (3/3)

Sintaxis:

```
#ifndef nombre_macro  
    código-si-no-definida  
#else  
    código-si-definida  
#endif
```

Inserta `codigo-si-no-definida`, si no está definida la macro `nombre_macro`, y en otro caso inserta `codigo-si-definida`.

Ejemplo:

```
int a,b,c;  
#ifndef MULTIPLICA  
    c = a*b;  
#else  
    c = MULTIPLICA(a,b);  
#endif
```

0.12 PROGRAMACIÓN MODULAR

0.12.1 Definición de programación modular

0.12.2 Ficheros cabecera

0.12.3 Variables globales static y extern

0.12.1 DEFINICIÓN DE PROGRAMACIÓN MODULAR

La *programación modular* es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más **legible y manejable**, así como más **fiable, mantenible, reusable, modificable, evaluable y depurable**. Un módulo debe tener una tarea bien definida y usualmente requiere de otros para poder operar. La comunicación entre módulos se realiza mediante una *interfaz de comunicación* que también debe estar bien definida.

El lenguaje C ofrece algunas construcciones para desarrollar el modelo de programación modular, como son las *funciones* y los *ficheros de compilación separada*. Una aplicación puede estar compuesta por *módulos*, cada uno de los cuales ofrece un *conjunto de recursos* (funciones, tipos de datos, variables y macros) los cuales podrán ser utilizados por otros módulos. Este conjunto de recursos público constituye la interfaz de comunicación del módulo. Un módulo en C es un *fichero fuente con extensión .c* el cual puede ser **compilado por separado**, creando un *fichero .o*.

Una aplicación compuesta de varios módulos requiere **compilar todos los módulos** y posteriormente **enlazarlos** generando el *fichero ejecutable* (.exe en windows).

También es posible crear una *librería (estática o dinámica)* con una colección de módulos. Las librerías estáticas se copian en el programa cuando se compila. Las librerías dinámicas no se copian si no que se van buscando dinámicamente cuando el programa lo requiere.

0.12.2 FICHEROS CABECERA (1/2)

Por cada fichero de módulo .c podemos asociar un *fichero cabecera .h*, que contendrá la declaración de todas las funciones, tipos, variables y macros que se hacen públicas al resto de módulos (**interface**). Cada módulo se compila de forma independiente al resto de módulos.

Los módulos que vayan a utilizar recursos de un módulo .c podrán hacerlo incluyendo el fichero .h asociado:

```
#include "Fichero.h"
```

Para evitar la definición múltiple de recursos de un mismo fichero de cabecera, se pueden utilizar las directivas condicionales.

0.12.2 FICHEROS CABECERA (2/2)

Ejemplo (Módulo Suma que define el tipo Entero y la función suma):

Fichero "Suma.h"

```
#ifndef __SUMA_H
#define __SUMA_H
typedef int Entero;
Entero suma(Entero a, Entero b);
#endif
```

Fichero "Suma.c "

```
#include "Suma.h"
Entero suma(Entero a, Entero b)
{
    return a+b;
}
```

Ejemplo (Módulo Main que contiene la función main y utiliza recursos del módulo Suma):

Fichero "Main.c"

```
#include "Suma.h"
int main()
{
    Entero a = suma(3,5); return 0;
}
```

0.12.3 VARIABLES GLOBALES EXTERN Y STATIC

Relacionado con la programación modular, el lenguaje C ofrece dos *tipos de almacenamiento de variables globales* mediante el uso de las palabras reservadas `extern` y `static`.

- **extern:** Permite que una variable sea global a varias unidades de compilación. Para ello la variable se define en el espacio global de uno de los ficheros y después se declaran como `extern` al comienzo de los demás módulos (usualmente dentro de los ficheros de cabecera).
- **static:** El significado varía según se aplique a una variable local o global. En ambos casos, la variable es estática (extensión permanente) y su inicialización se hace una sola vez, al comienzo de la ejecución del programa.
 - Cuando la variable es local, su efecto es que dicha variable sigue teniendo ámbito local, pero tiene extensión permanente.
 - Cuando la variable es global, su efecto es que el ámbito se limita al fichero donde está definida.

0.13 EJERCICIOS RESUELTOS (1/14)

Objetivos:

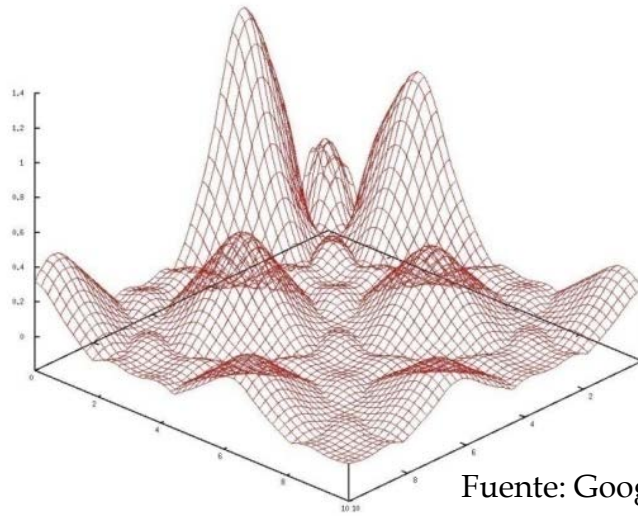
- Definición de funciones.
- Paso de parámetros por valor y por referencia en funciones.
- Definición y manipulación de arrays de una y dos dimensiones.
- Definición y manipulación de cadenas de caracteres.
- Funciones que devuelven arrays y cadenas de caracteres.
- Asignación dinámica de memoria en arrays y cadenas de caracteres.
- Funciones que devuelven estructuras de datos (`struct`).

0.13 EJERCICIOS RESUELTOS (2/14)

1) Implementar en C la función *bump de Keane* definida como:

$$\text{bump}(\mathbf{x}) = - \left| \left\{ \sum_{i=1}^m \cos^4(x_i) - 2 \prod_{i=1}^m \cos^2(x_i) \right\} / \left(\sum_{i=1}^m i x_i^2 \right)^{0.5} \right|$$

donde \mathbf{x} es un vector de tamaño m de números reales.



Fuente: Google Imágenes

0.13 EJERCICIOS RESUELTOS (2/14)

```
#include <math.h>
double bump(double x[], int m)
{
    double sumc4 = 0.0;
    double prodc2 = 1.0;
    double sum2 = 0.0;
    for (int i=1; i<=m; i++)
    {
        double cosx = cos(x[i-1]);
        sumc4 += pow(cosx,4);
        prodc2 *= pow(cosx,2);
        sum2 += i * pow(x[i-1],2);
    }
    return -fabs((sumc4 - 2.0 * prodc2)/pow(sum2,0.5));
}
```

0.13 EJERCICIOS RESUELTOS (2/14)

- 2) Implementar en C la función `PrecioRebajado` donde `v1` es un array que representa el precio en euros de `n` productos y `v2` es un array que representa el descuento (en porcentaje) aplicado a cada producto. La función debe devolver un nuevo array con el precio final rebajado de los `n` productos.

```
double * PrecioRebajado(double * v1, double * v2, int n)
{
    double * v3 = malloc(n*sizeof(double));
    for(int i=0; i<n; i++)
        v3[i] = v1[i]-v1[i]*v2[i]/100;
    return v3;
}
```

0.13 EJERCICIOS RESUELTOS (3/14)

- 3) Implementar en C la función `arrayPotencias`, la cual, dado un array `c` de `n` costos, devuelve un nuevo array de `n` reales en donde el elemento i -ésimo del array tiene el valor $c_i x^i$, $i=0, \dots, n-1$. Por ejemplo: $n=3$, $c=\{1, 2, 3\}$ y $x=2$ devuelve el array $\{1, 4, 12\}$.

```
float * arrayPotencias(int n, float c[n], float x)
{
    float * a = malloc(n*sizeof(float));
    for(int i=0; i<n; i++)
        a[i] = c[i]*pow(x,i);
    return a;
}
```


0.13 EJERCICIOS RESUELTOS (4/14)

- 4) Implementar en C la función `copia`, la cual devuelve una nueva cadena de caracteres que es copia de la cadena `s`.

```
char * copia(char * s) {  
    int n = 0;  
    while(s[n]!=0)  
        n++;  
    char * c = malloc((n+1)*sizeof(char));  
    for(int i=0; i<n+1; i++)  
        c[i] = s[i];  
    return c;  
}
```

0.13 EJERCICIOS RESUELTOS (5/14)

- 5) Implementar en C la función `cuenta`, la cual devuelve el número de caracteres iguales a `c` en la cadena `s`.

```
int cuenta(char * s, char c)
{
    int n = 0;
    for(int i=0; s[i]!=0; i++)
        if (s[i] == c)
            n++;
    return n;
}
```

0.13 EJERCICIOS RESUELTOS (6/14)

- 6) Implementar en C la función identidad, la cual asigna la matriz identidad a la matriz a de dimensión $n \times n$.

```
void identidad(int n, int a[n][n])
{
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            if (i==j)
                a[i][j] = 1;
            else
                a[i][j] = 0;
}

void identidad(int n, int a[n][n]) //versión reducida
{
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            a[i][j] = i==j;
}
```

0.13 EJERCICIOS RESUELTOS (7/14)

- 7) Implementar en C la función `subarray`, la cual, dado un array `v` de `n` números enteros, devuelve el subarray del array `v` comprendido entre los índices `i` y `j`. Si `i < 0` ó `i > j` ó `j ≥ n` devuelve `NULL`.

```
int * subarray(int * v, int n, int i, int j)
{
    if ((i < 0) || (i > j) || (j ≥ n))
        return NULL;
    int * aux = malloc(sizeof(int) * (j - i + 1));
    for(int k = i; k ≤ j; k++)
        aux[k - i] = v[k];
    return aux;
}
```

0.13 EJERCICIOS RESUELTOS (8/14)

- 8) Implementar en C la función `sucesion_suma` la cual devuelve un array de n elementos enteros en donde cada elemento a_i tiene el valor $a_i = \sum_{j=0}^i j$, $i = 0, \dots, n-1$, $n > 0$ (ver figura 1).

0	1	2	3	4	5	6	7	8
0	1	3	6	10	15	21	28	36

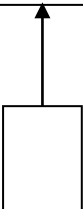


Figura 1: Ejemplo de array para $n=9$.

0.13 EJERCICIOS RESUELTOS (9/14)

```
int * sucesion_suma(int n){  
    int * nueva = malloc(sizeof(int)*n);  
    nueva[0] = 0;  
    for(int i = 1 ; i<n ; i++)  
        nueva[i] = nueva[i-1]+i;  
    return nueva;  
}
```

0.13 EJERCICIOS RESUELTOS (10/14)

9) Dada la siguiente definición en C:

```
typedef struct  
{  
    double minimo;  
    double maximo;  
    double media;  
}stats;
```

Implementar en C la función `estadisticos` que devuelve el mínimo, máximo y media de los elementos de un array `a` de `n` elementos, $n > 0$.

0.13 EJERCICIOS RESUELTOS (11/14)

```
stats * estadisticos(double * a, int n)
{
    stats * s = malloc(sizeof(stats));
    s->minimo = a[0];
    s->maximo = a[0];
    s->media = a[0];
    for(int i=1; i<n; i++)
    {
        if (a[i]<s->minimo) s->minimo = a[i];
        if (a[i]>s->maximo) s->maximo = a[i];
        s->media += a[i];
    }
    s->media /= n;
    return s;
}
```


0.13 EJERCICIOS RESUELTOS (12/14)

10) Teniendo en cuenta que la función de `Fibonacci` se define de la siguiente forma:

`Fibonacci(0)=0`

`Fibonacci(1)=1`

`Fibonacci(n)=Fibonacci(n-1)+Fibonacci(n-2), n≥2`

Implementar en C la siguiente función (ver figura 2).

```
// Devuelve un array con los n+1 primeros valores de la función
// de Fibonacci (desde Fibonacci(0) hasta Fibonacci(n)).
// Pre: n≥0
int * FibonacciArrayCrea(int n) {...}
```

0.13 EJERCICIOS RESUELTOS (13/14)

0	1	2	3	4	5	6	7	8
0	1	1	2	3	5	8	13	21




Figura 2. Ejemplo de array devuelto por la función `FibonacciArrayCrea(8)`.

0.13 EJERCICIOS RESUELTOS (14/14)

```
int * FibonacciArrayCrea(int n) {  
    int * a = malloc((n+1)*sizeof(int));  
    a[0]=0;  
    if (n>0)  
        a[1]=1;  
    for(int i=2; i<=n; i++)  
        a[i] = a[i-1]+a[i-2];  
    return a;  
}
```

0.14 EJERCICIOS PROPUESTOS

Objetivos: Aplicaciones para manipulación de ficheros de texto.

- 1) Realizar un programa que lea datos enteros desde teclado, los asigne a una matriz de $n \times m$ datos (n y m se leen desde teclado), imprima los datos de la matriz en pantalla y en un fichero de texto, libere la matriz y cierre el fichero. Si algún dato leído no es un número entero, el programa repetirá la lectura de ese dato, hasta que completen los $n \times m$ datos.
- 2) Realizar un programa que lea datos enteros de un fichero de texto (el del ejercicio propuesto anterior), construya una matriz de $n \times m$ datos con los datos del fichero (n y m se leen desde teclado), imprima los datos de la matriz en pantalla, libere la matriz y cierre el fichero. Si no existen $n \times m$ datos en el fichero, el fichero no existe, o algún dato no es entero, deberá producirse una situación de error e imprimirse en pantalla, abortando el programa.