Programación II

Repaso de Programación en C

Iván Cantador
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contenidos

- Fundamentos de C
- Estructuras y uniones
- Definición de tipos de datos
- Arrays
- Punteros
- Memoria dinámica
- Manejo de ficheros
- Funciones de interés

Contenidos

- **Fundamentos de C**
- Estructuras y uniones
- Definición de tipos de datos
- Arrays
- Punteros
- Memoria dinámica
- Manejo de ficheros
- Funciones de interés

Características de C

- C es un lenguaje de programación estructurado
- Está considerado de **alto nivel**
- Posee, sin embargo, características de medio-bajo nivel
  - Variables débilmente tipadas
  - Punteros
  - Sentencia `goto`
  - Código ensamblador

Tipos de datos primitivos

Tipo de dato	Nombre	Bytes**	Descripción	Ejemplos

Caracter	**char**	1	Representa un carácter ASCII, aunque se puede usar como un número entre -128 y +127	`char c1;`
				`c1 = 'a';`
				`printf("%c", c1);`

Entero corto	**short**	2	Entero de 16 bits con signo	`short s1 = 0;` `printf("%d", s1);`

Entero	**int**	2 o 4	Entero de 16 o 32 bits con signo	`int i1;`
				`i1 = 288;  /*Decimal*/`
				`i1 = 032;  /*Octal*/`
				`i1 = 0xf0; /*Hexadecimal*/`
				`printf("%d", i1);`

Entero largo	**long**	4 u 8	Entero de 32 o 64 bits con signo	`long l1;` `printf("%ld", l1);`

Real	**float**	4	Número real en coma flotante	`float f1;`
				`f1 = 3.142;`
				`f1 = 3e-12;`
				`printf("%f", f1);`
				`printf("%5f", f1);`

Real doble	**double**	8	Número real en coma flotante, con doble precisión	`double d1;`
				`d1 = 4.3e-3;`
				`printf("%f", d1);`
				`printf("%4.3f",d1);`

* Los tipos de datos primitivos puede ir precedidos por `unsigned`, para forzar a que sean sin signo, e.g. `unsigned int`

** Los tamaños pueden variar dependiendo del microprocesador y sistema operativo – ver **sizeof**

Conversiones entre tipos de datos

- Conversión de tipos de datos
  - **Implícita**: la expresión se convierte al tipo de mayor rango

    double ← float ← long ← int ← short ← char

    ```
    f = 0.9 / 2;
    ```
  - **Explícita**: la expresión se convierte mediante un "*casting*"

    (tipo) expresion

    ```
    i = (int) (0.9 / 2);
    ```


Variables locales vs. variables globales


```
#include <stdio.h>

int var_glo = 3;

void mi_funcion(int var_arg) {
     int var_loc;

     var_loc = 1;

     var_arg = var_loc + var_arg + var_glo;
}

int main() {
     int var_main = 2;

     mi_funcion(var_main);

     printf("Valor = %d\n", var_main);

     return 0;
}
```

Operadores (I)

Tipos de operador	Operadores	Descripción	Ejemplos

Aritméticos	Suma: `+`
	Resta: `-`
	Multiplicación: `*`
	División: `/`
	Módulo: `%`
		Operadores aritméticos elementales
		Más operaciones en las librerías matemáticas
			`char c = 'A'-'a';`
			`int i = c%4;`
			`double d = (2/3)*3;`

Relacionales	Menor: `<`
	Menos o igual: `<=`
	Mayor: `>`
	Mayor o igual: `>=`
	Igual: `==`
	Distinto: `!=`
		Operadores de comparación entre enteros, caracteres y reales	`int cond1 = (32>=4);`
			`int cond2 = (cond1<6);`
			`cond1 == cond2;`

Booleanos	NOT lógico: `!` AND lógico: `&&` OR lógico: `||`	Operaciones lógicas, aplicadas sobre enteros, siendo 0 = *false* y otro valor *true*	`int a, b, c;` `(!(a && (b || c)))`

Operadores de bits	NOT nivel bit: `~`
	AND nivel bit: `&`
	OR nivel bit: `|`
	XOR nivel bit: `^`
	Despl. izq.: `<<`
	Despl. der.: `>>`
		Operaciones lógicas aplicadas a nivel de bit
		En los desplazamientos: x << y, significa desplazar y bits el valor x
			`int a = 0xff, b = 0xf0;`
			`(a & b);`
			`(a | b);`
			`a << 2;`

Operadores (II)

Tipos de operador	Operadores	Descripción	Ejemplos

Asignación	Suma con asig.: `+=`
	Resta con asig.: `-=`
	Multiplic. con asig.: `*=`
	División con asig.: `/=`
		Operaciones de asignación a una variable
		Las de forma `x+=y` son equivalentes a `x=x+y`
			`int a, b, c;`
			`a = b = c = 0;`
			`a += 1;`
			`b += c += a;`

Incrementales	Preincremento: `++c`
	Postincremento: `c++`
	Predecremento: `--c`
	Postdecremento: `c--`
		Postincremento: devuelve el valor actual de `c` y luego hace `c=c+1`
		Preincremento: hace `c=c+1` y devuelve el nuevo valor de `c`
			`char c, d= 0;`
			`c = d++;`
			`d = c++;`

Concatenación de expresiones	*expr1* , *expr2*	Sirve para unir varias expresiones en una misma línea de código	`int i;` `i = (i=2, i++, i*= 2);`

Condicional	*exp1* ? *exp2* : *exp3*	Expresión condicional
		Si *exp1* es *true* el resultado es *exp2*, si no *exp3*
			`int a, b, max, min;`
			`max = (a>b ? a : b);`
			`min = (a<b ? a : b);`

Precedencia de operadores

- **Precedencia de operadores** (de mayor a menor)
  1. Aritméticos
  2. Relacionales
  3. Booleanos
  4. Asignaciones

```
// Expresiones equivalentes
a = ! b * c > d;
a = !((b * c) > d);
```

- **Precedencia de operadores aritméticos** (de mayor a menor)
  1. ( ) paréntesis
  2. * / multiplicación, división
  3. + – suma, resta

```
a = 1 + 2 * 3;      // El resultado es 7
a = (1 + 2) * 3;    // El resultado es 9
```


Sentencias de control de flujo (II)

- Condicionales simples
  - Sentencias de selección **if**, **if-else**
    - La condición siempre va entre paréntesis
    - Saltos: **goto**

```
if (condicion)
   sentencia;
```

```
if (condicion) {
   bloque_sentencias
}
```

```
if (condicion) {
   sentencia o bloque
}
else {
   sentencia o bloque
}
```

```
if (condicion) {
   sentencia o bloque
}
else if (condicion) {
   sentencia o bloque
}
else {
   sentencia o bloque
}
```


Sentencias de control de flujo (II)

- Condicionales múltiples
  - Sentencia de selección **switch**
    - Según el valor de una expresión (entre paréntesis y de tipo `char` o `int`) salta a una sentencia identificada por un valor constante: **case**
    - Puede o no existir un caso por defecto: **default**
    - Una vez en el caso seleccionado, la ejecución es secuencial, siempre que no se encuentre un punto de ruptura: **break**

```
switch (expresion) {
  case cte1:
    sentencias
  case cte2:
    sentencias
  ...
  default:
    sentencias
}
```

```
switch (expresion) {
  case cte1:
    sentencias
    break;
  case cte2:
    sentencias
    break;
  ...
  default:
    sentencias
}
```

Sentencias de control de flujo (III)

- Repeticiones / bucles
  - Bucles **while**, **do-while**
    - Repetición mientras una condición sea cierta

```
while (condicion)
  sentencia;
```

```
while (condicion) {
  sentencia o bloque
}
```

```
do {
  sentencia o bloque
} while (condicion);
```

  - Bucles **for**
    - Iteración con inicialización, condición de finalización e "incremento"

```
for(expr1; condicion; expr2)
  sentencia;
```

```
for(expr1; condicion; expr2) {
  sentencia o bloque
}
```

```
expr1;
while(condicion) {
  sentencia o bloque
  expr2;
};
```


Rupturas de ejecución en un bucle (I)

- Finalización de la ejecución de un bucle: **break**
- Finalización de la iteración actual de un bucle: **continue**

Rupturas de ejecución en un bucle (II)


```c
#include <stdio.h>

int main() {
     int i, j, r=0;

     for( i=0; i<10; i++ ) {
          for( j=0; j<10; j++ ) {
               if( i % 2 == 0 ) {
                 break;
               }
               r++;
          }
     }

     return r;
}
```


Rupturas de ejecución en un bucle (III)


```c
#include <stdio.h>

int main() {
     int lista[10];
     int i=0, j=0;

     while( i<10 ) {
          lista[i] = i++;
     }

     for( i=0; i<10; i++ ) {
          if( lista[i] == 8 ) {
               continue;
          }
          else if( lista[i] % 2 == 0 ) {
               j += i;
          }
     }

     return j;
}
```

Funciones

- En C no hay distinción entre **funciones** y procedimientos (que no devuelven un resultado)
- Formato de una función

```
<tipo_retorno> <nombre_funcion>(<argumentos_entrada>) {
    <codigo>
}
```

  - Si no se devuelve un resultado, el tipo de retorno es *void*
  - Si se devuelve un resultado, toda finalización de ejecución de la función se indicará con una sentencia **`return`** `<valor>;`
  - Si no se reciben datos, no se pone nada como argumentos de entrada

Funciones: ejercicio

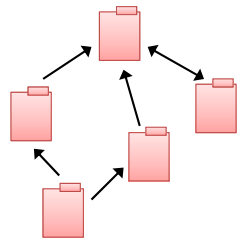
- Ejercicio: implementar una función sumar, invocándola desde *main*, que reciba como entrada un *array* de enteros y devuelva la suma de los elementos de ese *array*

Funciones: ejercicio


```c
#include <stdio.h>

int sumar(int lista[], int longitud) {
    int i, suma;

    suma = 0;
    for( i=0; i<longitud; i++ ) {
       suma += lista[i];
    }
}

int main() {
    int numeros[] = {1, 2, 3, 5, 7};

    printf("La suma es %d.\n", sumar(numeros, 5));

    return 0;
}
```

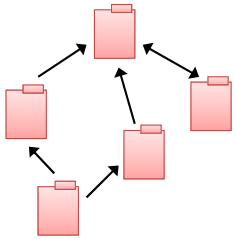

Programación modular

- La **programación modular** consiste en descomponer la complejidad de una aplicación informática en distintos componentes o *módulos*, donde cada módulo contiene un conjunto de funcionalidades relacionadas
  - Ejemplo. Un **programa de gestión de entradas de cine** podría tener módulos principales como:



Programación modular

- La **programación modular** consiste en descomponer la complejidad de una aplicación informática en distintos componentes o *módulos*, donde cada módulo contiene un conjunto de funcionalidades relacionadas
  - Ejemplo. Un **programa de gestión de entradas de cine** podría tener módulos principales como:
    - Interfaz gráfica de usuario
    - Gestión de películas
    - Gestión de salas
    - Gestión de transacciones de pago
    - Impresión de entradas



Programación modular en C (I)

- En C cada módulo de un programa está compuesto (en general y al menos) por 2 ficheros:
  - **Fichero de cabecera** (*header*)
    - De extensión **.h**
    - Contiene la definición de tipos de datos y macros, y la declaración de funciones del módulo
  - **Fichero fuente** (*source*)
    - De extensión **.c**
    - Contiene la definición (implementación) de las funciones declaradas en el fichero de cabecera

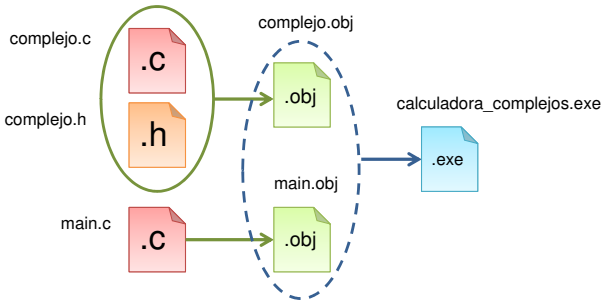
Programación modular en C (II)

```c
/* complejo.h */

#ifndef COMPLEJO_H
#define COMPLEJO_H

/* Macros */

#define PI 3.1416

/* Definición de tipos de datos estructurados */

typedef struct _Complejo {
     double re;
     double im;
} Complejo;

/* Declaracion de funciones */

double moduloComplejo(Complejo *a);

Complejo *sumaComplejos(Complejo *a, Complejo *b);

#endif
```

```c
/* complejo.c */

#include <stdio.h>
#include <math.h>
#include "complejo.h"

/* Definición de funciones */

double moduloComplejo(Complejo *c) {
     double modulo;

     if( !c ) return -1;

     modulo = sqrt(c->re * c->re + c->im * c->im);

     return modulo;
}

Complejo *sumaComplejos(Complejo *a, Complejo *b) {
     Complejo *s = NULL;

     if( !a || !b ) return NULL;

     s = (Complejo *) malloc(sizeof(Complejo));
     if( !s ) return NULL;

     s->re = a->re + b->re;
     s->im = a->im + b->im;

     return s;
}
```


Compilación y enlazado

- La generación de un programa escrito en C se lleva a cabo en 2 pasos:
  - **Compilación**
    - ficheros de código fuente (.c) → ficheros objeto (.obj)
  - **Enlazado** o "**lincado**" (*link*)
    - ficheros objeto (.obj / .o) → fichero ejecutable (.exe)



Preprocesado C

- El **preprocesador** es una componente del compilador que procesa los ficheros antes de la propia compilación, admitiendo ciertas "directivas" que empiezan por #
  - #include
  - #define
  - #ifdef, #indef, #else, #endif

#include: Inclusión de librerías de funciones

- Directiva **#include**
  - Incluye el contenido de un fichero en el punto del programa donde se encuentra el comando
  - El fichero a incluir será en general de cabecera de una "librería" de funciones
  - Sintaxis

```
#include <nombre_fichero>      /* Librerías de sistema */
#include "nombre_fichero"      /* Librerías de usuario */
```

  - Librerías de sistema habituales
    - **stdio.h** (E/S, I/O) estándar
    - **stdlib.h** Funciones y tipos de datos generales
    - **math.h** Funciones matemáticas
    - **string.h** Funciones para manejo de cadenas de caracteres
    - **time.h** Funciones y tipos para fechas, horas, tiempo del sistema
    - **mem.h** Funciones mara manejo de memoria

#define: Definición de macros

- Directiva **#define**
  - Define una constante (***macro***) con o sin parámetros, con el fin de que se sus apariciones se sustituyan textualmente por su valor
  - Ejemplos de macros sin parámetros

```
#define PI 3.1416
#define ERR 0
#define OK 1
#define MENSAJE "Mensaje predefinido"
```

  - Ejemplos de macros con parámetros

```
#define CUADRADRO(N)  (N)*(N)
#define MAX(A,B)  ((A)>(B)?(A):(B))
...
int i=1, j, k;
j = CUADRADO(i+1);
k = MAX(i*PI, j);
...
```


#ifdef, #ifndef, #else, #endif

- Directivas **#ifdef**, **#ifndef**, **#else**, **#endif**
  - Permiten añadir o quitar bloques de sentencias en tiempo de compilación según cierta condición (del tipo "si macro definido" y "si macro no definido")
  - Ejemplo

```
#define DEBUG
...
#ifdef DEBUG
printf("La ejecución pasa por aquí\n");
#endif
...
#ifndef ERR
#define ERR 0
#endif
#ifndef OK
#define OK 1
#endif
```

Contenidos

- Fundamentos de C
- **Estructuras y uniones**
- Definición de tipos de datos
- Arrays
- Punteros
- Memoria dinámica
- Manejo de ficheros
- Funciones de interés

Estructuras

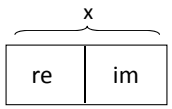
- Una estructura (**struct**) es un tipo de dato Complejo heterogéneo, i.e. contiene varios datos atómicos que pueden ser de diferentes tipos

```
struct [etiqueta] {
    tipo1 dato1 [, dato2 ...];
    . . .
    [tipoN-1 datoN-1] [, datoN ...];]
} [variable(s)];
```


Estructuras

- Ejemplo de estructura: número complejo

```
/* x, y, z: tres variables de estructura _Complejo */
struct _Complejo {
    float re;
    float im;
} x, y;

struct _Complejo z;
```



Observación: lo de utilizar una etiqueta para nombrar la estructura que empiece por guión bajo _ no os obligatorio; es un convenio que muchos siguen (al igual que utilizar i, j, … como variables en bucles) → más adelante se verá el porqué

Estructuras

- Referencia a los atributos (miembros, campos) de una estructura

```
float parteReal(struct _Complejo c) {
    return c.re;
}

float parteImaginaria(struct _Complejo c) {
    return c.im;
}
```

- Asignación de estructuras

```
struct _Complejo x, y;

x.re = 1;
x.im = 2;

y = x;    /* Copia los valores en los atributos de x a y */
```

Observación: la asignación es correcta con *struct* de atributos primitivos; la asignación directa entre *arrays* (y por tanto entre cadenas de caracteres) no está definida en C

- La E/S por teclado/pantalla se realiza atributo a atributo

```
struct _Complejo x;

scanf("%f %f",&(x.re),&(x.im)); /* & operador de dirección*/

printf("%f %f\n", x.re, x.im);
```


- Las funciones pueden devolver estructuras

```
struct _Complejo conjugado(struct _Complejo c) {
    c.im = -c.im;
    return c;
}
```


- El manejo de tablas de estructuras se hace de la forma habitual

```
struct _Complejo tabla[20];

/* Asignación de valores al primer elemento de tabla */
tabla[0].re = 1;
tabla[0].im = 2;
```

- Una estructura puede tener atributos que a su vez son estructuras

```
struct _Punto2D {
    float x;
    float y;
};

struct _Triangulo {
    struct _Punto2D vertices[3];
};
```

Observación: el orden de definición de las estructuras es fundamental (e.g., para definir un "triángulo" hay que haber definido antes un "punto")

- Inicialización de los valores de una estructura (en la declaración de la variable)

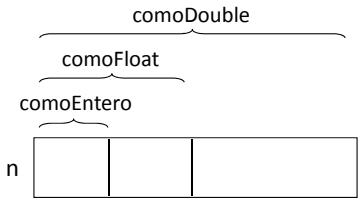
```
struct _Cliente {
    unsigned long dni;
    char nombre[256];
    int diaNacimiento;
    int mesNacimiento;
    int anioNacimiento;
};

struct _Cliente c = {12345678, "Jose", 31, 12, 1989};
```


- **Punteros a estructuras**
  - El acceso a los atributos de una estructura se puede realizar mediante "**->**"

```
struct _Cliente c;
struct _Cliente *pc = NULL;

c.anioNacimiento = 1989;

pc = &c;
(*pc).anioNacimiento = 1990;      /* Forma de acceso 1 */
pc->anioNacimiento = 1991;        /* Forma de acceso 2 */
```


- Una unión (**union**) es un registro donde todos los atributos ocupan (comparten) la misma posición de memoria
  - Sus atributos son excluyentes
  - De uso poco frecuente

```
union _Numero {
    int comoEntero;
    float comoFloat;
    double comoDouble;
} n;

n.comoEntero = 4;
printf("%f", n.comoDouble);
```



- Otro ejemplo

```
union _Identificador {
    unsigned long dni;
    char nombre[256];
};

union _Identificador id;

...
id.dni = 12345678;
strcpy(id.nombre, "Juan Pérez");
...
```


- Fundamentos de C
- Estructuras y uniones
- **Definición de tipos de datos**
- Arrays
- Punteros
- Memoria dinámica
- Manejo de ficheros
- Funciones de interés

- Definición de nuevos tipos de datos: **typedef**

```
typedef <tipo_existente> <tipo_nuevo>;
```

  - Ejemplos:

```
typedef unsigned char byte;
typedef int entero;

byte b;
entero e;

b = 0x56;
e = 5;
```


- Definición de nuevos tipos de datos: **typedef**

```
typedef <tipo_existente> <tipo_nuevo>;
```

  - Ejemplo:

```
typedef struct _Complejo {
    float re;
    float im;
} Complejo;

Complejo x;   /* Alternativa: struct _Complejo x; */

x.re = 3;
x.im = -5;
```

Definición de tipos de datos

- Definición de nuevos tipos de datos: **typedef**

```
typedef <tipo_existente> <tipo_nuevo>;
```

  - Ejemplo:

```
typedef struct _Empleado {
    unsigned long dni;
    char nombre[256];
    _Empleado *supervisor;
} Empleado;
```

¡Manera de referenciar una estructura desde ella misma!

Definición de tipos de datos

- Tamaño (en Bytes) de un tipo de datos: operador *sizeof*

```
typedef struct _Empleado {
    unsigned long dni;
    char nombre[256];
    _Empleado *supervisor;
} Empleado;

/* sizeof(Empleado) = sizeof(unsigned long) +
                      256*sizeof(char) +
                      sizeof(_Empleado *) */
sizeof(Empleado);
```


Definición de tipos de datos

- **Enumeraciones**: *enum*
  - Asigna nombres simbólicos a constantes enteras

```
enum [etiqueta] {cte1 [= valor1] [, cte2 = [valor2] ...]};
```

```
enum boolean {false, true};    /* false = 0, true = 1 */
enum boolean b = true;         /* b = 1 */

enum estaciones {primavera=1,verano=2,otono=3,invierno=4};
enum estaciones e = verano;    /* e = 2 */
```


Definición de tipos de datos

- **Enumeraciones**: *enum*
  - Ejemplos con typedef

```
typedef enum {false, true} boolean;
boolean b = true;

typedef enum {ERR, OK} status;
status st = ERR;

typedef enum {ap=5, not=7, sob=9} Calificacion;
Calificacion c = not;
```

Contenidos

- Fundamentos de C
- Estructuras y uniones
- Definición de tipos de datos
- **Arrays**
- Punteros
- Memoria dinámica
- Manejo de ficheros
- Funciones de interés

Arrays

- Un **array** (~*tabla*) almacena un número fijo de datos en posiciones de memoria consecutivas
  - Declaración

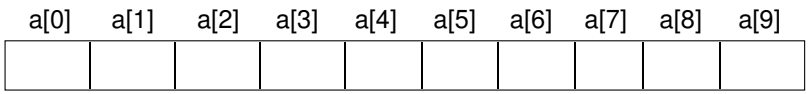
```
<tipo_dato> <nombre>[tamaño];

int a[3];
float b[4], c[5];
```

  - Inicialización (en la declaración):

```
int a[3] = {1, 2, 3};
float b[] = {1, 2, 3, 4}; /* Se puede omitir el tamaño */
float b[5] = {1, 2, 3};   /* Se inicializan los 3 primeros
                             valores */
```


Arrays

- El primer elemento de un array **a** de *N* elementos es **a[0]**, el segundo es **a[1]**, ..., el último es **a[*N*-1]**
  - Ejemplo: array de 10 elementos



Arrays

- La asignación entre arrays **NO** está definida en C
  - Solución:
    - Recorrer los elementos de los arrays mediante un bucle

```
int a[] = {1, 2, 3};
int b[3];
int i;

for( i=0; i<3; i++ ) {
    b[i] = a[i];
}
```

- **Cadenas de caracteres**
  - En C una cadena de caracteres es un array de char, donde el último carácter es el "carácter de fin de cadena": '\0'

```
char s1[] = "Hola";
char s2[] = {'H', 'o', 'l', 'a', '\0'};
```


- **Arrays multidimensionales** (~*matrices*)

```
int m1[2][3];
int m2[][3] = {{1, 2, 3}, {4, 5, 6}};

int m3[10][10][5];
```


- Una array multidimensional de dimensión *N*x*M* de tipo T es equivalente a un **puntero** a tipo T con *N*M* elementos

```
int m[2][3] = {{1, 2, 3}, {4, 5, 6}};
int *pm = NULL;
int i, j;

for( i=0; i<2; i++ ) {
     for( j=0; j<3; j++ ) {
          printf("m(i,j)=%d\n", m[i][j]);
     }
}

pm = m;
for( i=0; i<2*3; i++) {
     printf("m(i,j)=%d", i, j, pm[i]);
}
```

	m[0][0]	m[0][1]	m[0][2]	m[1][0]	m[1][1]	m[1][2]
pm	1	2	3	4	5	6

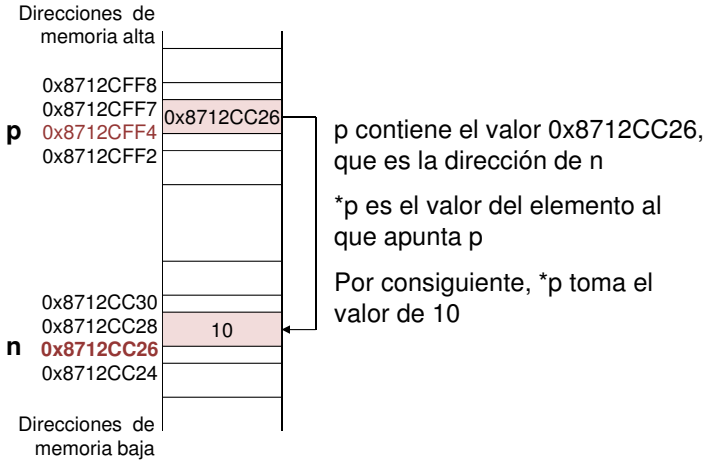
Contenidos

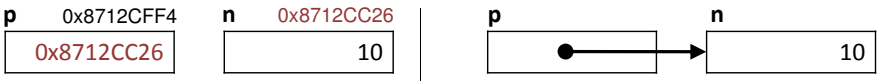
Punteros

- Un **puntero** es una variable donde se almacena la dirección de memoria de otra variable de cierto tipo

```
int n = 5;    /* Asigna el valor 5 a la posición de memoria
                 identificada como n y que tiene espacio para
                 guardar un número entero */
int *p = NULL;
p = &n;                    /* Guarda en p la dirección de n */
                           /* "p apunta a n" */

printf("%x\n", &n);        /* Imprime la dirección de n */
printf("%x\n", p);         /* Imprime la dirección de n */

*p = 10;      /* Asigna el valor 10 a la posición de memoria
                 apuntada por p (es decir, a la de n) */
```

Punteros





Punteros

- **Declaración**

```
<tipo_dato> *<nombre_variable>;

int *pi = NULL;    /* Puntero a entero */
long *pl = NULL;   /* Puntero a long */
float *pf = NULL;  /* Puntero a float */
double *pd = NULL ;/* Puntero a double */
char *pc = NULL;   /* Puntero a char (cadena de caracteres) */

void *pv = NULL;   /* Puntero a void */

int **ppi = NULL; /* Puntero a puntero a entero (matriz) */

Complejo *ps = NULL; /* Puntero a tipo definido (estructura) */
```


Punteros

- **Asignación**

```
p = &<nombre_variable>;

int n;
int *pi = NULL;

pi = &n;   /* p apunta a n */

*pi = 10;  /* Equivalente a n = 10 */

/* CUIDADO CON LOS "PUNTEROS LOCOS" */
char c = 'A';
char *pc;

*pc = c;   /* ¡DESASTRE! (¿a qué posición apuntaba pc?) */
           /* Moraleja: siempre inicializar punteros a NULL */
```

2 usos diferentes de *
(aparte del de operador aritmético de multiplicación)

- **Punteros nulos**: aquellos que apuntan a NULL (que es un valor, 0, definido en stdio.h)

```
p = NULL;
```

- Comprobación de puntero nulo

```
if( p ) ...       /* Equivale a if( p != NULL ) ... */
if( !p ) ...      /* Equivale a if( p == NULL ) ... */
```


- **Verificación de tipos**
  - En C, las variables puntero han de direccionar realmente a variables del mismo tipo de dato ligado a los punteros en sus declaraciones

```
int i;
long l;
long *p = NULL;       /* Puntero a long */

p = &l;               /* Correcto */
p = &i;               /* ¡Incorrecto! */
```


- **Punteros a tipo no definido**
  - En C, los punteros a **void** pueden apuntar a variables de cualquier tipo → C es un lenguaje débilmente tipado

```
int i;
long l;
void *p = NULL;       /* Puntero a void */

p = &l;               /* Correcto */
p = &i;               /* Correcto */
```


- **Compatibilidad en la asignación entre punteros**
  - Se pueden asignar punteros del mismo tipo

```
int *p1;
int *p2;

p1 = p2;
```

  - Se pueden asignar punteros a void a cualquier otro

```
int *p1;
void *p2;

p1 = p2;
p2 = p1;
```

  - Se pueden asignar punteros de distintos tipos con *casting* explícito

```
float *p1;
int *p2;

p1 = (float *) p2; /* No se pierde información */
p2 = (int *) p1;   /* Cuidado: se puede perder información */
```

- **Aritmética de punteros**
  - En C, a un puntero P de tipo T se le puede sumar/restar un número entero N, dando como resultado la dirección de memoria que se corresponde con direccion(P) + N * sizeof(T)

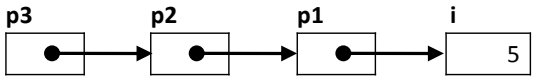
```
char c[10];
int i[10];
float f[10];

c + 2;    /* Dirección de c + 2 Byte, si sizeof(char)=1 Byte */
i + 2;    /* Dirección de i + 4 Bytes, si sizeof(int)=2 Bytes */
f + 2;    /* Dirección de f + 8 Bytes, si sizeof(float)=4 Bytes */
```

  - Cuidado: en C no se comprueban límites de arrays al compilar

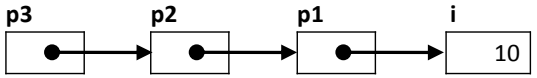
```
float f[10];
...f + 500...; /* 500>>10: compila, pero provoca desbordamiento */
```


- **Punteros a puntero**

```
int i = 5;
int *p1 = &i;
int **p2 = &p1;
int ***p3 = &p2;
```




```
*p1 = 10;
**p2 = 10;
***p3 = 10;
```



- **Punteros y arrays**
  - El nombre de un array es el nombre simbólico de la dirección del primer byte del array => "apunta" a una dirección de memoria fija (no es un puntero que pueda variar)
  - Un array de tipo T es "equivalente" a un puntero a tipo T

```
int e[10], *pe = NULL;

*e = 5;              /* equivale a e[0] = 5 */
*(e+4) = 5;          /* equivale a e[4] = 5 */
pe = e + 2;          /* equivale a pe = &e[2] */
```

  - Se puede asignar un array a un puntero, pero no al revés

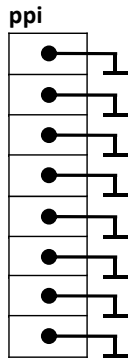
```
pe = e;              /* e = pe da un error de compilación */
```

  - Con un puntero a tipo T se pueden usar los corchetes

```
pe[2] = 5;
pe[4] = *(pe + 1);   /* equivale a pe[4] = pe[1] */
```


- **Array de punteros**

```
int *ppi[8];    /* Array de 8 punteros a entero */
int i;

for( i=0; i<8; i++ ) {
    p[i] = NULL;
}
```



- A distinguir…

```
int *ptr          /* puntero a int */
int ptr[]         /* array de int */
int *ptr[]        /* array de punteros a int */
int (*ptr)[]      /* puntero a array de int */
int *(*ptr)[]     /* puntero a array de punteros a int */
...
```

- **Punteros a estructura**
  - El acceso a los atributos de una estructura a través de un puntero a ella se realiza normalmente con '->', aunque se puede hacer mediante '.'

```
Complejo c;
Complejo *pc = NULL;

pc = &c;

/* Correctas */
printf("%f + %fi\n", pc->re, pc->im);
printf("%f + %fi\n", (*pc).re, (*pc).im);

/* Incorrectas */
printf("%f + %fi\n", *pc.re, *pc.im);
printf("%f + %fi\n", *(pc.re), *(pc.im));
```


- **Punteros a estructura**
  - Se suelen usar al invocar una función que recibe como argumentos de entrada estructuras => mejora en la eficiencia

```
/* Preferible */
void imprimirComplejo(Complejo *c) {
    printf("%f %fi", c->re, c->im);
}

/* No preferible */
void imprimirComplejo(Complejo c) {
    printf("%f %fi", c.re, c.im);
}
```


- **Paso de argumentos a una función**
  - **Por valor** (mecanismo de invocación a una función por defecto en C)
    - La función recibe como argumento **copia**(s) de la(s) variable(s) de entrada
    - Las copias de variable se guardan de forma temporal (durante la ejecución de la función) en la PILA DEL PROGRAMA
  - **Por referencia**
    - La función recibe como argumento **puntero**(s) a la(s) variables de entrada

Punteros

- **Paso de argumentos a una función**

Por valor

```
void swap(int a, int b) {
    int t;

    t = a;
    a = b;
    b = t;
}

/* MAL */
```

Por referencia

```
void swap(int *a, int *b) {
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

/* BIEN */
/* Recomendable cuando los
   argumentos de entrada son
   estructuras */
```


Contenidos

- Fundamentos de C
- Estructuras y uniones
- Definición de tipos de datos
- Arrays
- Punteros
- **Memoria dinámica**
- Manejo de ficheros
- Funciones de interés

Memoria dinámica

- En C se puede reservar memoria "dinámicamente", es decir en tiempo de ejecución
- El manejo de la memoria dinámica se realiza mediante punteros

```
char *linea = NULL;
int **matriz = NULL;
```

- Toda memoria reservada de forma dinámica ha de liberarse antes de la finalización de la ejecución
- Las funciones para manejo de memoria se encuentran definidas en la librería **<stdlib.h>**
  - **malloc, calloc, realloc**
  - **free**

Memoria dinámica

- Reserva de memoria: **malloc**

```
void *malloc(size_t numBytes)
```

- Liberación de memoria: **free**

```
void free(void *puntero)
```

Memoria dinámica


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

int main(int argc, char *argv[]) {
     char *nombre = NULL;

     nombre = (char *) malloc(256 * sizeof(char));
     if( !nombre ) {
          fprintf(stderr, "Error en main: reserva de memoria.\n");
          return 1;
     }

     prinft("Introduzca su nombre: ");
     gets(nombre);
     printf("Hola %s!\n", nombre);

     if( nombre ) free(nombre);

     return 0;
}
```


Memoria dinámica

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

int main(int argc, char *argv[]) {
     int **matriz = NULL, filas=2, columnas=3, i, j;

     /* Reserva de memoria para una matriz */
     matriz = (int **) malloc(filas * sizeof(int *));
     if( !matriz ) {
          return 1;
     }
     for( i=0; i<filas; i++ ) {
          matriz[i] = (int *) malloc(columnas * sizeof(int));
          if( !matriz[i] ) {
               for( j=0; j<i; j++ )
                    free(matriz[j]);
               free(matriz);
               return 1;
          }
     }
     /* Liberacion de memoria de una matriz */
     for( i=0; i<filas; i++ )
          free(matriz[i]);
     free(matriz);

     return 0;
}
```


Memoria dinámica


```c
#include <stdio.h>
#include <stdlib.h>
#include "complejo.h"

int main(int argc, char *argv[]) {
     Complejo *c = NULL;

     /* Reserva de memoria para un tipo de dato definido */
     c = (Complejo *) malloc(sizeof(complejo));
     if( !c ) {
        fprintf(stderr, "Error en main: reserva de memoria.\n");
        return 1;
     }

     prinft("Introduzca parte real: ");
     scanf("%f", &c->re);
     scanf("%f", &c->im);
     printf("Numero Complejo creado: %f + %fi\n", c->re, c->im);

     if( c ) free(c);

     return 0;
}
```


Memoria dinámica

- Reserva de memoria inicializada a 0: **calloc**

```
void *calloc(size_t numElementos, size_t numBytesElemento)
```

- Reasignación de memoria: **realloc**

```
void *realloc(void *puntero, size_t numBytes)
```

Contenidos

- Fundamentos de C
- Estructuras y uniones
- Definición de tipos de datos
- Arrays
- Punteros
- Memoria dinámica
- **Manejo de ficheros**
- Funciones de interés

Manejo de ficheros

- El manejo de ficheros se realizar mediante (punteros a) variables de tipo **FILE**, definido en la librería **stdio.h**

```
#include <stdio.h>
FILE *f = NULL;
```

- Tipos de fichero
  - De texto
  - Binarios
- Operaciones con ficheros
  - Apertura
  - Lectura/escritura
  - Cierre

Manejo de ficheros

- Apertura de un fichero: **fopen**

```
FILE *fopen(<pathFichero>, <modoApertura>);
```

```
FILE *f = NULL;
f = fopen("c:\prog2\datos.txt", "r");
if( f == NULL ) { /* error */ }
```

- Modos de apertura

	Fichero de texto	Fichero binario
Abrir para lectura	`"r"`	`"rb"`
Abrir/crear para escritura	`"w"`	`"wb"`
Abrir para añadir al final	`"a"`	`"ab"`
Abrir para lectura/escritura	`"r+"`	`"r+b"`
Abrir/crear para lectura/escritura	`"w+"`	`"w+b"`
Abrir/crear para añadir al final	`"a+"`	`"a+b"`

Manejo de ficheros

- Cierre de un fichero: **fclose**

```
fclose(<fichero>);
```

```
FILE *f = NULL;
f = fopen("c:\prog2\datos.txt", "r");
if( f != NULL ) {
    fclose(f);
}
```

Manejo de ficheros

- Lectura/escritura de un fichero de texto: **fscanf**, **fprinft**

```
int n, numLeidos;
FILE *f = NULL;

f = fopen("c:\prog2\datos.txt", "r+");
if( f == NULL ) {
    printf("Error en la apertura del fichero de texto.\n");
    return ERR;
}

numLeidos = fscanf(f, "%d\n", &n);
fprintf(f, "%d\n", n);

fclose(f);
```


Manejo de ficheros

- Lectura/escritura de un fichero de texto: **fgetc**, **fputc**

```
char c;
FILE *f = NULL;

f = fopen("c:\prog2\datos.txt", "r");
if( f == NULL ) {
    printf("Error en la apertura del fichero de texto.\n");
    return ERR;
}

while( (c = fgetc(f)) != EOF ) {   /* EOF = End Of File */
    printf("Caracter leido: %c\n", c);
}

fclose(f);
```


Manejo de ficheros

- Lectura/escritura de un fichero de texto: **fgets**, **fputs**

```
char linea[256];
FILE *f = NULL;

f = fopen("c:\prog2\datos.txt", "r");
if( f == NULL ) {
    printf("Error en la apertura del fichero de texto.\n");
    return ERR;
}

while( fgets(linea, 256, f) != NULL ) {
    printf("Linea leida: %s\n", linea);
}

fclose(f);
```


Manejo de ficheros

- Lectura/escritura de un fichero de texto: **fgets**, **fputs**

```
char linea[256];
FILE *f = NULL;

f = fopen("c:\prog2\datos.txt", "r");
if( f == NULL ) {
    printf("Error en la apertura del fichero de texto.\n");
    return ERR;
}

while( !feof(f) ) {
    fgets(linea, 256, f);
    printf("Linea leida: %s\n", linea);
}

fclose(f);
```

Manejo de ficheros

- Lectura/escritura de un fichero binario: **fread**, **fwrite**

```
char buffer[256];
FILE *f = NULL;

f = fopen("c:\prog2\datos.dat", "rb");
if( f == NULL ) {
    printf("Error en la apertura del fichero binario.\n");
    return ERR;
}

while( !feof(f) ) {
    fread((void *)buffer, sizeof(char), 256, f);
}

fclose(f);
```


Manejo de ficheros

- Posicionamiento en un fichero: **fseek**

```
fseek(<fichero>, <offset>, <origen>)

fseek(f, 0L, SEEK_SET);    /* Al comienzo del fichero */
fseek(f, 0L, SEEK_CUR);    /* En la posicion actual */
fseek(f, 0L, SEEK_END);    /* Al final del fichero */

fseek(f, 256L, SEEK_SET);  /* En el Byte 256 */
```

- Funciones relacionadas
  - **ftell**: devuelve la posición actual de lectura/escritura
  - **rewind**: "rebobinado"; equivale a `fseek(f, 0L, SEEK_SET)`

Contenidos

- Fundamentos de C
- Estructuras y uniones
- Definición de tipos de datos
- Arrays
- Punteros
- Memoria dinámica
- Manejo de ficheros
- **Funciones de interés**

Funciones de interés

- Función **main**
  - Salida
    - Tipo: *void*, *int*
    - Valor devuelto al sistema operativo por el programa al acabar su ejecución
  - Entrada
    - **int argc**: número de argumentos del programa **+ 1**
    - **char *argv[]**: array con el nombre (argv[0]) y argumentos (argv[1], argv[2], …) del programa

Funciones de interés

- Función **main**

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

    printf("Nombre del programa: %s\n", argv[0]);

    printf("Numero de argumentos: %d\n", argc-1);

    printf("Argumentos:\n");
    for( i=1; i<argc; i++ ) {
        printf("\t%s\n", argv[i]);
    }
}
```


Funciones de interés

- **Cadenas de caracteres** (string.h, stdlib.h)

Función	Sintaxis	Descripción
strlen	`unsigned int strlen(char *s)`	Devuelve la longitud de la cadena (sin contar el '\0')
strcpy	`char *strcpy(char *des, char *ori)`	Copia la cadena ori en la cadena des y devuelve des
strcat	`char *strcat(char *des, char *ori)`	Concatena a la cadena des la cadena ori y devuelve des
strcmp	`int strcmp(char *s1, char *s2)`	Compara dos cadenas s1 y s2. El resultado es 0 si s1 = s2 (mismos caracteres), <0 si s1<s2, >0 si s1>s2
atoi	`int atoi(char *s)`	Convierte una cadena a un entero. Funciones relacionadas: atof, *itoa*, *ftoa*, …

Funciones de interés

- **Memoria** (mem.h, string.h)

Función	Sintaxis	Descripción
memcpy	`void *memcpy(void *des, void *ori, size_t num)`	Copia numBytes desde la posición apuntada por ori a la posición apuntada por des
memcmp	`void *memcmp(void *s1, void *s2, size_t num)`	Compara numBytes de s1 y s2. El resultado es como el de strcmp
memset	`void *memset(void *s, int c, size_t num)`	Escribe en numBytes de s el valor de c (int convertido a unsigned char)

Funciones de interés

- **Caracteres** (ctype.h)

Función	Sintaxis	Descripción
isalpha	`int isalpha(int c)`	Devuelve "true" (un número distinto de 0) si c (convertida a unsigned char) es una letra
isupper	`int isupper(int c)`	Devuelve "true" si c es una letra mayúscula
islower	`int islower(int c)`	Devuelve "true" si c es una letra minúscula
isspace	`int isspace(int c)`	Devuelve "true" si c es un espacio en blanco, un tabulador '\t' o un salto de línea '\n'
toupper	`int toupper(int c)`	Devuelve c en mayúscula
tolower	`int tolower(int c)`	Devuelve c en minúscula

Funciones de interés

- **Otras** (stdlib.h, time.h)

Función	Sintaxis	Descripción
system	`int system(char *comando)`	Ejecuta un comando del sistema operativo, devolviendo 0 si no hay error
exit	`void exit(int status)`	Cierra todos los ficheros y termina la ejecución del programa, devolviendo status
rand	`int rand(void)`	Devuelve un número aleatorio uniformemente distribuido en el rango [0, RAND_MAX] Número aleatorio en [M, N]: rand () % (N–M+1) + M "Semilla" aleatoria: srand(time(NULL))
clock	`clock_t clock (void)`	Devuelve el número de pulsos de reloj desde que se inició el proceso. Para convertir pulsos de reloj a segundos se usa la constante CLK_PER_SEC o la maco CLK_TCK
time	`time t time(time t*)`	Devuelve la fecha y hora actuales, guardándolas también en el argumento t si éste no es NULL

Funciones de interés

- **strtok** (string.h)
  - "Tokeniza" (trocea) una cadena de caracteres atendiendo a un conjunto de caracteres *delimitadores*

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

void main() {
     char *cadena = "primero segundo, tercero\tcuarto";
     char *delimitadores = " ,\t";
     char *token = NULL;

     printf("Tokens de %s:\n", cadena);

     token = strtok(cadena, delimitadores);
     while( token ) {
          printf("%s\n", token);
          token = strtok(NULL, delimitadores);
     }
}
```