ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Introducción a C

IWEB 2013-2014

Santiago Pavón

ver: 2013.09.14

Historia de C

- Creado por Dennis Ritchie
  - Laboratorios AT&T Bell - 1972
- Estándar en 1989 - ANSI C
- Último estándar: C99

Características de C

- Pocas palabras reservadas.
- Tipos:
  - números enteros (int, long).
  - números reales (float, double).
  - carácter (char).
  - tipos compuestos (struct, union).
  - punteros (memoria y arrays).
- Tipado débil:
  - Se producen conversiones implícitas de tipos.
  - Casting: forzar conversión de tipo.
- Uso de librerías externas.
- Preprocesador.
- No hay excepciones, recolección de basura, comprobación de rangos, ...
- No es un lenguaje orientado a objetos.

Ciclo de Vida

- Editar el programa con extensión **.c**

```
$ vi HolaMundo.c
```

- Compilar: **gcc**

```
$ gcc -g HolaMundo.c -o HolaMundo
```

- Depurar: **gdb**

```
$ gdb HolaMundo
```

- Mejor usar IDEs: Eclipse, Xcode, Microsoft Visual C++, ...

HolaMundo.c

```
/*
 * Ejemplo: HolaMundo.c
 */

#include <stdio.h>

int main(void) {

      char msg[] = "Hola Mundo";

      puts(msg);

      return 0;
}
```

Comentarios

- Varias líneas: `/* ... */`
- Hasta fin de línea: `//` ....

Preprocesador

- Incluir ficheros de cabeceras con la macro **`#include`**.

  `#include <stdio.h>`

  - Declaración de constantes, tipos, funciones.
  - **`stdio.h`** = funciones de entrada/salida estándares.

- Usar:
  - **`< >`** para buscar en directorios del sistema.
  - **`" "`** para buscar primero en directorio de trabajo.
  - Opción del compilador **-I*directorio*** para indicar donde buscar los ficheros.

- Definir macros:
  - `#define PI 3.1416`
  - `#define Company "UPM"`
  - `#define SUMA(a,b) ((a)+(b))`
  - Realiza sustituciones en línea en el código.

- Condicionales:
  - `#if #ifdef #ifndef #else #elif #endif`
  - Tratadas por el preprocesador: antes de compilar.
  - gcc: la opción **`-Dnombre=valor`** crea definiciones para el preprocesador.
- Eliminar definición: `#undef`
- Generar errores o advertencias al compilar: `#error` `#warning`

Variables

- Declararlas antes de usarlas.

```
int edad;
```

- Asignar un valor:

```
edad = 33;
```

- Inicializar al declarar:

```
int edad = 33;
```

- Declarar e inicializar varias variables:

```
int dias, meses, edad = 33, pelos = 0;
```

- Nombres de variables:
  - Empiezan por letra, y contienen letras, dígitos y _
  - Mayúsculas y minúsculas son distintas.

- El **ámbito** de una variable:
  - Es el **bloque** o la función en la que se declara.
  - Si la variable se declara fuera de una función, su ámbito es **global**.
    - Persisten durante toda la vida del programa.
    - Puede accederse a ellas y ser modificadas desde cualquier función.

- Variables **`static`**:
  - **Una variable global** declarada como **`static`** es **privada** del fichero en el que se declara.
    - No puede hacerse **`extern`**.
  - **Dentro de una función**, las variables **`static`**:
    - Siguen siendo locales a la función.
    - Se inicializan solo al inicializar el programa.
    - **No se reinicializan** con cada llamada a la función.

Expresiones Aritméticas

- Formadas por variables, literales, operadores.

```
dias = meses * 30;
```

- Operadores:

```
+ - * / % =  +=  -=  *=  /=  %=
```

  - Los operadores tienen distintas prioridades.
    - Usar paréntesis para cambiar el orden de evaluación.

Funciones

- Las funciones deben declararse antes de usarlas.
- La declaración de una función se llama prototipo de la función:

```
int factorial(int n);

float media(float, float);

void imprime(char* msg);
```

- `void` indica que no se devuelve valor o que no hay argumentos.

- Los ficheros de cabeceras contienen declaraciones/prototipos de funciones.

main()

- El programa empieza su ejecución en la función **main**.

```
int main(void);

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

  - **argc** = número de parámetros en la línea de comandos.
  - **argv** = array de strings con los parámetros de la línea de comandos.
    - Incluye el nombre del programa

- **main** devuelve 0 si su ejecución fue exitosa, o distinto de 0 para indicar algún error.

Definición de Funciones

```
tipo nombreFuncion(tipo arg2, tipo arg, ...) {
   declaracion de variables;
   sentencias;
}
```

- La definición debe coincidir con su prototipo (si existe).
- Un bloque entre llaves con la declaración de variables locales y las sentencias.
- Declarar las variables antes de usarlas.
- Solo devuelve un valor del tipo indicado.
  - Usar **`void`** si no se devuelve un valor.
- Si no hay argumentos, poner **`void`** entre los paréntesis.

String

- Literales entre comillas.

```
char msg[] = "Hola Mundo";
char name[] = "Ale" "jandro";
char saludo[] = {'h','o','l','a',0};
```

- Se implementan como un array de caracteres (**`char`**) terminado en null ('**`\0`**').
- Pueden contener caracteres especiales escapados con **`\`**

`\\` **`\"`** **`\b`** **`\t`** **`\n`** **`\r`**
`\ooo` (*octal*)
`\xhh` (*hexadecimal*)

- **`<string.h>`** proporciona funciones para manipular strings.

Entrada/Salida por Consola

- Streams de entrada, salida y error:
  - **stdin**, **stdout**, **stderr**.
- Páginas de manual: Ejecutar:

```
$ man stdio
$ man getchar
```

- Imprimir por pantalla:

```
puts(string);
putchar(caracter);
```

- Leer del teclado:

```
c = getchar();
str = gets(str);
```

Tipos Numéricos

- Tipos básicos:

  `char` - un byte
  `int` - un entero del tamaño natural de la máquina.
  `float` - real punto flotante de precisión simple.
  `double` - real punto flotante de doble precisión.

- Calificadores:

  `short`
  `long`
  `signed`
  `unsigned`

- **`<limits.h>`** y **`<float.h>`** contienen constantes sobre el tamaño de cada tipo.

`short int x;`	Entero corto
`short x;`	
`unsigned short int x;`	Entero corto sin signo
`unsigned short x;`	
`int x;`	Entero
`unsigned int x;`	Entero sin signo
`long int x;`	Entero largo
`long x;`	
`unsigned long x;`	Entero largo sin signo
`unsigned long int x;`	
`float x;`	Real
`double x;`	Real doble precisión
`long double x;`	Real largo de presición extendida
`char x;`	Carácter
`signed char x;`	
`unsigned char x;`	Carácter sin signo

- Los tipos **`unsigned`** contienen el doble de valores.
  - No se reserva un bit para el signo.
- El tamaño de estos tipos (su rango de valores) depende de la máquina y del compilador.
  - Solo se garantiza que:
    - `sizeof(char)  <  sizeof(short)`
    - `sizeof(short) <= sizeof(int)`
    - `sizeof(int)   <= sizeof(long)`
    - `sizeof(short) <= sizeof(float)`
    - `sizeof(float) <  sizeof(double)`
- La disposición de los bytes depende de la arquitectura de la máquina:
  - **Big endian**: los bits más significativos ocupan las direcciones de memoria menores.
  - **Little endian**: Los bits menos significativos ocupan las direcciones de memoria menores.

- Literales:
  - char: `'A'  '\x41'  '\0101'`
  - entero: `33`
  - entero long: `33l  33L`
  - entero sin signo: `33u  33U`
  - unsigned long: `33UL`
  - double: `3.3`
  - float: `3.3f  3.3F`
  - double long : `3.3l  3.3L`
  - octal: `033`
  - hexadecimal: `0x33`

Enumerados

- `enum BOOL {NO, YES};`
  - **`NO`** es 0
  - **`YES`** es 1

- `enum DiasSemana {lunes, martes, miercoles};`
  - **`lunes`** es 0
  - **`martes`** es 1
  - **`miercoles`** es 2

- `enum Color {rojo=3, verde, amarillo, azul=7};`
  - **`rojo`** es 3
  - **`verde`** es 4
  - **`amarillo`** es 5
  - **`azul`** es 7

Operadores

- Numéricos: `+` `-` `*` `/` `%`
- Relacionales: `>` `>=` `<` `<=` `==` `!=`
  - En C el valor falso es cero, y distinto de cero es verdadero.
  - Cuidado al comparar valores reales con el operador **==**. La precisión es finita.
  - No confundir la asignación **=** y la comparación **==**.
- Lógicos: `&&` `||` `!`
  - Se evalúa en cortocircuito.
- Pre/Post Incremento/Decremento: `x++` `x--` `++x` `--x`
- Bits: `&` `|` `^` `~` `<<` `>>`
- Asignación: `=` `+=` `-=` `*=` `/=` `%=`
- Condicional (`?:`): *expresión_lógica* `?` *valor_para_verdad* `:` *valor_para_falso*

Conversión de Tipo

- El tipado es débil.
- Conversiones implícitas de tipos:
  - En una expresión los valores pueden promocionar a tipos de mayor precisión.
    - `double x = 7.5/2;` -----> 2 promociona a 2.0
  - El tipo **`char`** promociona automáticamente a **`int`**.

- Conversiones explícitas de tipos - Casting:
  - `int x = (int)7.5;`

Sentencias

- Terminadas en `;`

```
puts("hola");
int x = 2*y;
```

- Bloque: es una sentencia creada con varias sentencias entre llaves.

```
{ int aux =x;
  x = y;
  y = aux;
}
```

  - No termina en `;`
  - El bloque puede esta vacío: `{ }`
  - Pueden declararse variables dentro de un bloque.
  - Los bloques pueden anidarse.

Control de Flujo

- Sentencias condicionales: **`if`** y **`switch`**
- Bucles: **`while`**, **`for`** y **`do-while`**
- Alteración del flujo: **`break`** y **`continue`**
- Saltos: **`goto`**

- En C no existe el tipo booleano.
  - En las condiciones debe usarse una expresión que evalúe a 0 para falso, y a distinto de 0 para verdadero.

```
if (x>0)
   puts("Positivo");
```

Solo se imprime si **x** es positivo

```
if (x>0)
   puts("Positivo");
else
   puts("Negativo o cero");
```

El **else** es opcional

```
if (x)
   if (x>0)
      puts("Positivo");
   else
      puts("Negativo");
else
   puts("Cero");
```

Si **x** no es cero

if anidado

```
if (x>0)
   puts("Positivo");
else
   if (x<0)
      puts("Negativo");
   else
      puts("Cero");
```

El **else** afecta al **if** más cercano

```
if (x>0)
   puts("Positivo");
else if (x<0)
   puts("Negativo");
else
   puts("Cero");
```

Idéntico al ejemplo anterior, pero escrito con otro estilo

```
if (x != 0)
    if (x>0)
        puts("Positivo");
else
    puts("Cero");
```

Seguro que nos hemos equivocado.

El **else** afecta al **if** más cercano.

Si **x** vale **-5** se imprime **Cero**.
Si **x** es **0** no se imprime nada.

```
if (x != 0) {
    if (x>0)
        puts("Positivo");
} else
    puts("Cero");
```

Lo soluciono con un bloque.

El **else** afecta al **if** más cercano

La sentencia switch compara el valor de la variable con todos los puntos de entrada en orden. Se entra por el caso con el valor coincidente, y se ejecutan las sentencias hasta encontrar un break.

Un entero o char

```
switch (dia) {
    case 1:
        msg = "lunes";
        break;
    case 2:
        msg = martes;
        break;
    case 6:
    case 7:
        msg = "festivo";
        break;
    default:
        msg = "otro";
        break;
}
```

Punto de entrada para **dia** igual a **1**

break para terminar ejecución del switch

Entradas para varios casos

Entrada para los demás casos

- Mientras se cumpla la condición se ejecuta la sentencia:

```
while (condicion) sentencia
```

- Ejemplo:

```
int x = 0;
while (x < 10) {
    puts("hola");
    x++;
}
```

- Las iteraciones se controlan con tres expresiones
  - **inicialización**.
  - condición de finalización
  - actualización.

```
for (inicialización ; condición ; actualización)
        sentencia
```

  - Las expresiones son opcionales.
    - Si no se proporciona la condición, se considera verdadero.
  - Pueden usarse sentencias compuestas separadas por comas

- Ejemplos:

```
int x,i,j;
for (x=0 ; x<10 ; x++) puts("hola");
for (i=1, j=10 ; i!=j ; i++, j--) puts("hola");
```

- Se ejecuta la sentencia mientras se cumpla la condición:

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

  - Primero se ejecuta la sentencia y luego se comprueba la condición.
  - La sentencia seguro que se ejecuta por lo menos una vez.

- Ejemplo:

```
int x = 0;
do {
   puts("hola");
   x++;
} while (x < 10);
```

- **`break`**
  - Se usa para terminar el bucle más cercano o salir de un `switch`.
- **`continue`**
  - Se usa para pasar a la siguiente iteración del bucle más cercano.
- Ejemplo:

```
int x, y=10;
printf("%i es divisible por ",y);
for (x=1 ; ; x++) {
   if (y%x != 0) continue;
   printf("%i",x);
   if (x>=y) break;
   printf(", ");
}
```

--> 10 es divisible por 1, 2, 5, 10

- **goto** realiza un salto a otro punto del código identificado con una etiqueta.

```
for (;;) {
   for (;;) {
      if (condicion)
         goto fuera;
   }
}
fuera:
. . .
```

Módulos

- Un programa C no tiene que ser un único fichero.
- Separar funcionalidades en módulos independientes.
  - La interface del módulo se especifica en un fichero de cabecera con la extensión **`.h`**
  - La implementación del módulo se realiza en un fichero con extensión **`.c`**
- El fichero de cabecera informa de:
  - las funciones del módulo, incluyendo el prototipo de las funciones.
  - las variables globales del módulo, redeclarándolas con la palabra reservada **`extern`**.
    - **`extern`** indica al compilador que la variable ha sido definida en otro sitio.
- Para usar un módulo debe incluirse el fichero de cabecera.
  - Usar directivas del preprocesador en el fichero .h para evitar múltiples inclusiones.
- Pasar al compilador todos los ficheros .c para que se enlacen juntos.
  - Hay varias formar de compilar y enlazar los ficheros.

```
/*
 *  Polinomio: a*x^2 + b*x + c
 */

#ifndef __POLINOMIO_H__
#define __POLINOMIO_H__

// Coeficientes (variables globales)
extern float a, b, c;

// Calcula las raices
float raiz1();
float raiz2();

#endif
```

polinomio.h

```c
#include <math.h>
#include "polinomio.h"

float a, b, c;

float raiz1() {

    return (-b + sqrt(b*b - 4*a*c)) / (2*a);
}

float raiz2() {

    return (-b - sqrt(b*b - 4*a*c)) / (2*a);
}
```

polinomio.c

```c
#include <stdio.h>
#include "polinomio.h"

int main() {

      a = 1;
      b = -1;
      c = -2;

      float r1 = raiz1();
      float r2 = raiz2();

      printf("Raiz 1 = %f\n",r1);
      printf("Raiz 2 = %f\n",r2);

      return 0;
}
```

raices.c

```
$ gcc raices.c polinomio.c -o raices
$ ./raices
Raiz 1 = 2.000000
Raiz 2 = -1.000000
$
```

Salida Estándar: Formato

- **<stdio.h>** proporciona una función para generar una salida formateada.

```
int printf(char format[], arg1, arg2, ...);
```

  - El primer argumento es el formato.
    - Contiene strings literales y especificaciones de formato.
      - Empiezan por **%**.
  - Los demás argumentos son los valores a imprimir y que se asocian a cada especificación **%**.
  - Devuelve el número de caracteres impresos.

- Formato: **%[flags][width][.precision][length]<type>**

- Ejemplos:

```
printf("Hola Mundo\n");                          // Hola Mundo
printf("%s tiene %d años\n",nombre,edad); // Pepe tiene 7 años
printf("%.2f",1.23456);                           // 1.23
```

Más sobre Entrada/Salida

- Entrada formateada:

  `scanf`

- Entrada y salida formateada sobre strings:

  `sprintf`, **`scanf`**

- Ficheros:

  `fopen`, **`fclose`**, **`getc`**, **`fgets`**, **`putc`**, **`fputs`**, ...

Punteros y Variables

- Los punteros son direcciones de memoria.
- Las variables residen en una dirección de memoria.
- Operador **&**:
  - La dirección de memoria donde reside una variable se obtiene con el operador **&**.
- Tipo puntero:
  - Dado un tipo `t`, el tipo `t*` representa los valores dirección a variable o valor de tipo `t`.
- Operador *:
  - Dereferencia un puntero para acceder a la variable o valor apuntado.
  - Un puntero dereferenciado se comporta como una variable normal.
- **null** es el puntero a nada. Es un 0.

- Ejemplo:

```
int x = 1;
int * px = &x;
int y = *px + 2;
```

  - **&x** es la dirección o puntero donde **x** reside en la memoria.
  - **px** es una variable de tipo puntero a entero, es decir, una variable de tipo **int***.
    - En el ejemplo, **px** almacena la dirección de la variable **x**.
  - ***px** es el valor almacenado en la dirección apuntada por **px**.
    - En el ejemplo ***px** es un 1 dado que **px** contiene la dirección de **x**. Tras la asignación **y** valdrá 3.

Función: Devolver Varios Valores

- Usar punteros en los parámetros formales de la función

```
void raices(float *pr1, float *pr2) {

      *pr1 = (-b + sqrt(b*b - 4*a*c)) / (2*a);
      *pr2 = (-b - sqrt(b*b - 4*a*c)) / (2*a);
}
```

- Invocar la función pasando las direcciones de las variables donde se dejarán los resultados.

```
float r1, r2;

raices(&r1, &r2);

printf("Raiz 1 = %f\n",r1);
printf("Raiz 2 = %f\n",r2);
```

Raíz 1 = 2.000000
Raíz 2 = -1.000000

Arrays

- Un array es un puntero a una zona de memoria donde se almacenan de forma continua los elementos del array.

```
int a[5];
a[2] = 25;
int b[] = {2,4,6,8,10};
```

  - `a` y `b` son arrays de 5 enteros.
    - el tamaño del array `a` se indica en la definición
    - el tamaño del array `b` se obtiene del literal.
  - `a` y `b` son punteros al primer elemento del array.
  - `a[3]` es el cuarto entero guardado en el array `a`.
  - `*(a+3)` es el cuarto entero guardado en el array `a`.
    - aritmética de punteros: se suma al puntero `a` tres veces el tamaño del dato apuntado.

sizeof()

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

```
int x;
int a[10];

sizeof(int) ->  4
sizeof(x)   ->  4
sizeof(a)   -> 40
```

struct

- Una estructura **`struct`** define un nuevo tipo de datos como una agrupación de varios valores,
  - cada miembro de la estructura tiene un nombre y un tipo.

```
struct persona {
    char nombre[100];
    int edad;
};

struct persona p1 = {"Luis",33};
printf("Nombre = %s",p1.nombre);
struct persona p2, *pp;
p2 = p1;
pp = &p1;
pp->edad = 40;
```

- En los literales de inicialización se especifica el valor de todos los miembros de la estructura.
- Para acceder a los miembros de una estructura se usa el operador "**.**" o "**->**":
  - "**.**" para valores struct.
  - "**->**" para valores puntero a struct.
- Al asignar una estructura, pasarla como parámetro o devolverla en una función, se copian todos los valores.
  - Si un miembro es un puntero se copia la dirección, no el valor apuntado.
  - Para evitar realizar muchas copias o para compartir la misma estructura, se usan punteros a las estructuras.
- Un miembro de una estructura puede ser otra estructura.
- El nombre de una estructura es opcional

```
struct {int x, int y} punto;
```

- Un array de estructuras se declara e inicializa así:

```
struct persona p[3] = {{"Luis",19}, {"Ana",14}, {"Ivan",77}};
struct persona p[3] = {"Luis",19,"Ana",14,"Ivan",77};
```

union

- Una **union** es una variable que puede alojar un valor de entre varios posibles valores.

```
union clave {
    char telefono[10];
    long hash;
    int id;
} c;
```

  - La variable **c** es de tipo **union clave**.
  - El valor almacenado en **c** puede ser uno de entre las siguientes posibilidades: un string con un número de **teléfono**, un **hash** de tipo long, o un **id***entificador* de tipo entero.

```
c.telefono = "555 555555";
c.hash = 1234567891;
c.id = 666;
```

typedef

- Facilidad para crear nombres de tipos.
  - No crea tipos nuevos.
  - Crea nombres o alias para referirse a un tipo ya existente.

```
typedef int Edad;

typedef char * String;

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

Memoria

- **Stack**:
  - Zona de memoria donde se crean las variables que se declaran.
  - La memoria de estas variables se crea y destruye automáticamente cuando se crea o destruye un ámbito.
- **Heap**:
  - Zona de memoria donde se obtiene la memoria dinámica.
  - Esta memoria se obtiene y se libera con **`malloc`**, **`calloc`**, **`free`**, ...

malloc - calloc - free

- `void* malloc(size_t n)`
  - Obtiene un bloque de memoria del tamaño indicado.
    - Devuelve un puntero a la zona de memoria reservada o NULL en caso de fallo.
  - La zona de memoria reservada no se inicializa con ningún valor.

```
int* a = (int*)malloc(sizeof(int)*20);
```

- `void* calloc(size_t n, size_t size)`
  - Obtiene memoria para un array de n elementos de tamaño size.
    - Devuelve un puntero a la zona de memoria reservada o NULL en caso de fallo.
  - La zona de memoria reservada se ha inicializado con 0s.

```
int* a = (int*)calloc(20,sizeof(int));
```

- Otros: **realloc**, **valloc**, ...
- `void free(void*)`
  - Libera la memoria reservada con **malloc**, **calloc**, **realloc**, ...

La Gestión de la Memoria

- La memoria pedida debe liberarse cuando ya no se necesite.
  - Para evitar que el programa se quede sin memoria. (*leaks*)
- Un bloque de memoria solo puede liberarse una vez.
- No puede usarse la memoria liberada.
- Errores:
  - Liberar dos veces la misma zona de memoria.
  - Usar (acceder/modificar) una zona de memoria ya liberada.
    - Puede haber sido o será reasignada a otra misión.

      Se machacarán datos, se colgará el programa o el ordenador, funcionamientos erróneos de forma aleatoria, etc...
- Recordad que no hay recolección automática de memoria.
- El programador debe implementar sus propias estrategias para gestionar la memoria.
  - Es una labor difícil.

- Peligro: un puntero es inválido cuando el ámbito de la variable a la que apunta se destruye.

```
char * saludo() {
    char msg[] = "hola";
    return msg;
}
```

  - La dirección devuelta por la función apunta a una variable que ya no existe.
  - Esa dirección es basura.

Ejemplos

Hola Mundo

```
#include <stdio.h>

int hola(void) {
    printf("Hola Mundo\n");
    return 0;
}

int main(int argc,const char* argv[]) {
   hola();
   return 0;
}
```

Una función escribe Hola Mundo

Máximo Común Divisor

```
int mcd(int a, int b) {
      while (a*b != 0)
         if (a>b)
           a %= b;
         else
           b %= a;
      return a+b;
}
int main(int argc,const char* argv[]) {
   int v1 = 12;
   int v2 = 15;
   printf("mcd(%i,%i)=%i\n",v1,v2,mcd(v1,v2));
```

La función mcd devuelve el máximo común divisor de dos números.

Calcular la mitad

```
void mitad(float v, float* res) {
     *res = v / 2;
     v = 6.6;
}

int main(int argc,const char* argv[]) {
     float f1 = 2.5;
     float f2;
     mitad(f1, &f2);
     printf("Mitad de %.2f = %.2f\n",f1,f2);

// Mitad de 2.50 = 1.25
```

El resultado se devuelve en el segundo parámetro

Tirar Dados

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

#define TABLA_DADOS_SIZE 15

int main(int argc,const char* argv[]) {
      int dado[TABLA_DADOS_SIZE];

      srand(clock()); // Inicializacion
      for (int i=0 ; i<TABLA_DADOS_SIZE ; i++) {
          dado[i] = rand() % 6 + 1;
      }
      int total = 0;
      for (int i=0 ; i<TABLA_DADOS_SIZE ; i++) {
          total += dado[i];
      }
      printf("Total = %d\n",total);
```

Guardo 15 números aleatorios en un array e imprimo la suma

Tamaños

```
void tamanos(void) {
  printf("char -> %lu bytes.\n",sizeof(char)); // 1
  printf("short int -> %lu bytes.\n",sizeof(short int)); // 2
  printf("int -> %lu bytes.\n",sizeof(int)); // 4
  printf("long int -> %lu bytes.\n",sizeof(long int)); // 8
  printf("float -> %lu bytes.\n",sizeof(float)); // 4
  printf("double -> %lu bytes.\n",sizeof(double)); // 8
  printf("long double -> %lu bytes.\n",sizeof(long double)); // 16
  printf("unsigned int -> %lu bytes.\n",sizeof(unsigned int)); // 4
}
```

Distancia entre dos puntos

```
#include <math.h>

typedef struct {
   double x;
   double y;
} Punto;

double distancia(Punto p1, Punto p2) {
   return sqrt(pow(p1.x-p2.x,2) + pow(p1.y-p2.y,2));
}

int main (int argc, const char * argv[]) {
   Punto p1 = {0.5, 1.5};
   Punto p2 = {4.5, 4.5};
   printf("Distancia = %f\n", distancia(p1,p2));
   …
```

Calcular la distancia entre dos puntos, donde los puntos son structs.

Gestión de Memoria

```
Punto *p3 = malloc(sizeof(Punto));
if ( NULL == p3 ) {
    printf("No queda memoria.");
    exit(1);
}
p3->x = -1.5;
p3->y = -3.5;
printf("Dist = %f\n", distancia(p2,*p3));

free(p3);
```

Pedir memoria para un Punto.

Distancia entre p2 y *p3.

Liberar la memoria.

Días de la Semana

```
enum Dias {
     Lunes,
     Martes,
     Sabado = 5,
     Domingo
};

enum Dias hoy = Martes;
printf("Hoy es %d\n", hoy);    // Hoy es 1
```

Tipo enumerado

Acumulador

```
int acumula(int v) {
    static int total = 0;
    total += v;
    return total;
}

acumula(1);
acumula(2);
acumula(3);
int t = acumula(4);
printf("Total acumulado = %d\n", t); //10
```

Variable estática para acumular la suma de los valores pasados en todas las llamadas a la función.