Sistemas Informáticos Industriales

Apuntes de

Punteros y memoria dinámica en C

Licencia

Grado en Electrónica y Automática
Departamento de Informática de Sistemas y Computadores
Escuela Técnica Superior de Ingeniería del Diseño

punteros_C_r9.odt

Contenido

1. Punteros y memoria dinámica en C....3
1.1 Introducción....3
1.2 Objetivos....3
1.3 Esquema....3
1.4 Qué es un puntero....4
1.5 Operaciones aritméticas con punteros....6
1.6 Vectores y punteros....8
1.7 Punteros en funciones. Paso por referencia....10
1.8 Memoria dinámica....13
1.9 Punteros a estructuras....15
1.10 Resumen....17
1.11 Bibliografía comentada....17
1.12 Actividades resueltas....17
1.12.1 Medias aritméticas....17
1.12.2 Números complejos....19

1 PUNTEROS EN C

1.1 INTRODUCCIÓN

El uso de punteros en el lenguaje C/C++ será indispensable para el desarrollo de los objetivos de la asignatura, y los deberemos considerar como una mera herramienta indispensable para la comprensión y desarrollo de posteriores contenidos.

Es posible desarrollar programas básicos en C sin emplear punteros, pero los punteros y C están tan íntimamente ligados que, llegados a un punto, es imposible ir más allá sin su conocimiento.

Esta unidad está pensada para trabajarla de forma lineal sin necesidad de acudir a otras fuentes ni materiales adicionales.

En puntos concretos se intercalan actividades que permitirán practicar los conocimientos adquiridos.

En caso de dificultad en la resolución de actividades sí se recomienda acceder a fuentes externas, por ejemplo, a la bibliografía recomendada.

1.2 OBJETIVOS

Saber qué es un puntero.

Saber apuntar elementos y objetos mediante punteros.

Saber realizar operaciones de movimiento de punteros (operaciones aritméticas).

Saber acceder y crear estructuras de tamaño fijo (estructuras y vectores).

Saber acceder y crear estructuras de tamaño variable creadas dinámicamente.

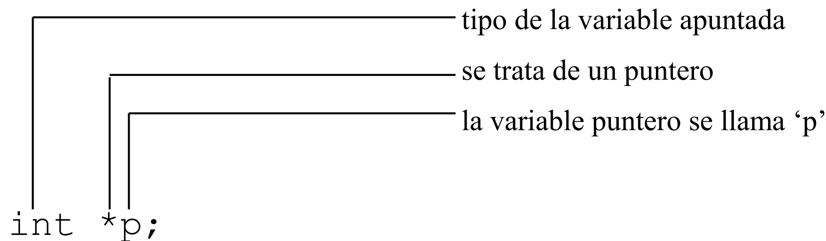
1.3 ESQUEMA

El esquema de la unidad coincide con el desarrollo ordenado de los objetivos y que se plasma en la correspondiente tabla de contenidos.

1.4 QUÉ ES UN PUNTERO

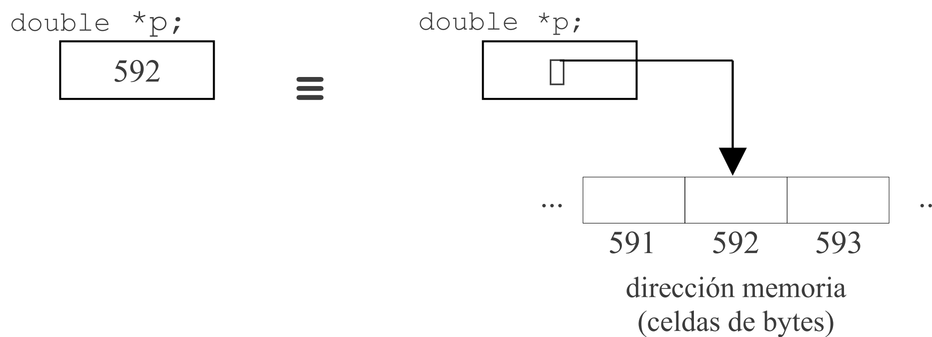
Un puntero es un sinónimo de “dirección de memoria principal”, y una variable es aquella que se usa para contener esas direcciones de memoria.

Como el resto de las variables, las variables puntero es necesario declararlas, por ejemplo:



El ***tipo*** del puntero indica que la variable puntero se va a emplear para contener la dirección en memoria de un determinado tipo de dato.

En jerga informática, “la dirección de memoria que contiene un puntero” y donde “apunta” son sinónimos.



Una variable puntero puede contener el valor especial **`NULL`**, que significa que no contiene una dirección de memoria válida.

Dos operadores fundamentales para trabajar con punteros son `*` y `&`:

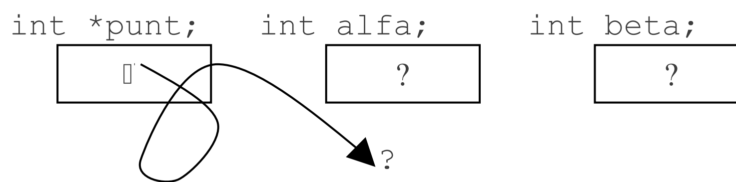
- `*p` significa "contenido de la dirección apuntada por el puntero `p`"
- `&x` significa "dirección de memoria del objeto `x` (por ejemplo, una variable)"

El siguiente ejemplo permite aclarar el significado de dichos operadores. Sea el siguiente programa:

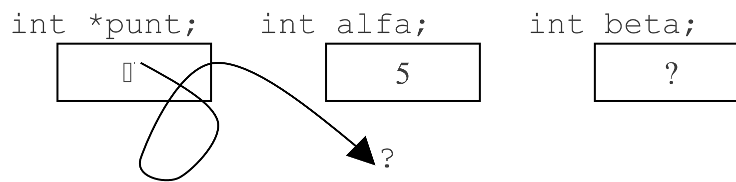
```
void main (void)
{
   int *punt;
   int alfa, beta;

   alfa = 5;          (1)
   punt = &alfa;      (2)
   beta = *punt;      (3)
   *punt = 7;         (4)
}
```

Al entrar en la función `main()` se crean en memoria las variables locales. Estas variables contendrán "basura".

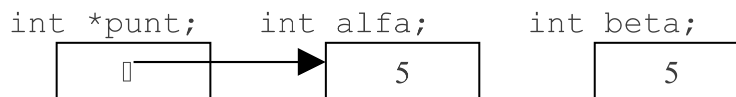


Al ejecutarse la línea (1) las variables contendrán:

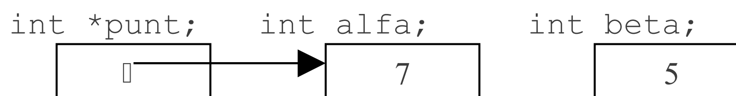


Al ejecutarse la línea (2), a la variable `punt` se le asigna la dirección de memoria de la variable `alfa`, quedando las variables de la siguiente forma:

Al ejecutarse la línea (3) a la variable `beta` se le asigna el entero al que apunta en ese momento el puntero `punt`. En este caso `punt` apunta a la posición de memoria donde se almacena el contenido de `alfa`, por lo que las variables quedan de la siguiente forma:



Por último, la línea (4) hace que se modifique el contenido apuntado por `punt`., con lo que `alfa` pasa a valer 7 y quedan las variables de la siguiente forma:



Actividad

Haz un programa en el que se definan 4 variables de tipo `double`. Define 4 punteros y haz que apunten a cada una de las 4 variables. Usando estos punteros, obtén la suma `a1+a2+a3` y deposítala en `resultado`.

```
double a1, a2, a3, resultado;
```

Actividad

Haz lo mismo, pero ahora sólo puedes declarar 2 punteros y tienes que conseguir hacer toda la operación utilizando únicamente punteros, sin hacer referencia explícita ni a `a1`, `a2`, `a3` ni a `resultado`, únicamente puedes utilizar sus direcciones de memoria.

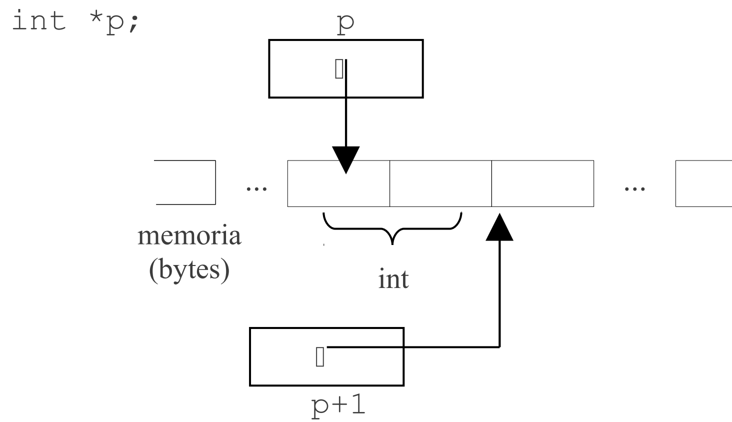
Toda la operación no se podrá hacer de golpe, necesitarás realizar varios pasos e ir acumulando valores intermedios.

1.5 OPERACIONES ARITMÉTICAS CON PUNTEROS

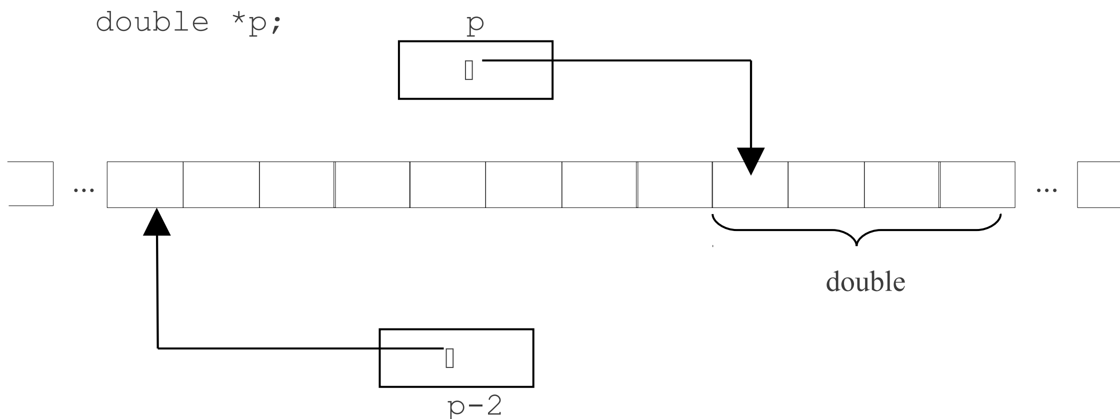
Con punteros podemos hacer operaciones aritméticas enteras. (+, -, ++, --, ...) que equivaldrán a hacer que el puntero apunte más hacia delante o más hacia atrás en la memoria (direcciones de memoria mayores o menores respectivamente).

Para las operaciones aritméticas es importantísimo tener en cuenta que la unidad de incremento/decremento de un puntero es el tamaño en bytes del tipo de variable apuntada.

Para ilustrar esta idea supongamos, por ejemplo, que un `int` ocupa dos bytes en memoria. Si sumamos 1 al puntero esto significa "apuntar" al siguiente `int` en memoria, es decir, en realidad el puntero se ha incrementado en dos unidades.



Si, por ejemplo, se trata de un `double`, y suponiendo que se almacena en 4 bytes, restar 2 al puntero equivaldrá al siguiente escenario:



Ejemplo:

```
int *p;
int alfa;
p = ??;    //a cualquier sitio
alfa = *(p+5);
```

Ejemplo:

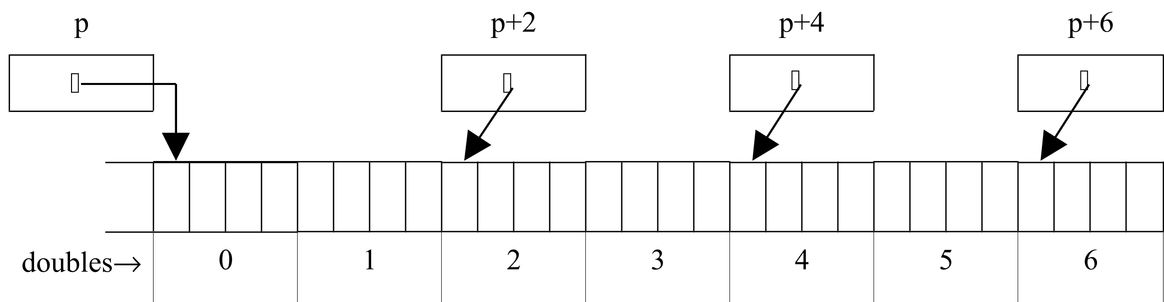
Se desea recorrer la memoria a partir de una posición X y localizar el número entero (`int`) –3525.

```
int *p;
p = ??;    // a cualquier sitio X
while (*p != -3525) {
    p++;
}
```

Actividad:

Acumula en una variable de tipo `double` la suma de 100 `double` que están separados 2 `doubles` a partir de un puntero que apunta a una dirección X.

Por ejemplo, suponiendo que un `double` ocupa 4 bytes, se pretende acceder a los elementos mostrados en la siguiente figura.



ATENCIÓN

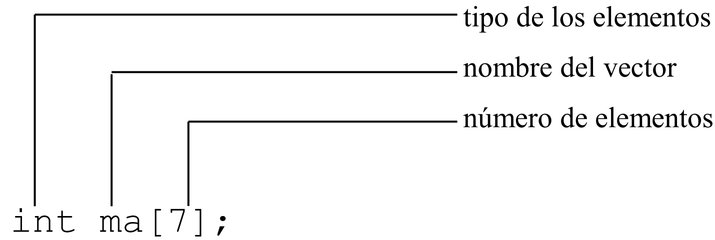
El tamaño de un tipo de dato depende del tipo de procesador, el sistema operativo, el compilador, etc. Para no tener problemas, se puede usar el operador `sizeof(tipo dato)` para saber cuántos bytes ocupa el tipo de dato. Por ejemplo, la siguiente línea en el compilador Keil para un microcontrolador 8051 dará 2, en Borland C++ Builder para Microsoft Windows dará 4, y para el compilador gcc para Linux sobre PC dará 4.

```
printf("El tamaño del tipo int es %d\n", sizeof(int));
```

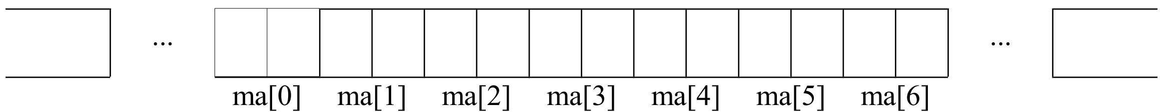
1.6 Vectores y punteros

Un vector (matriz, array) es un conjunto de elementos del mismo tipo dispuestos secuencialmente en memoria.

Por ejemplo, dada la siguiente declaración de un vector de C,

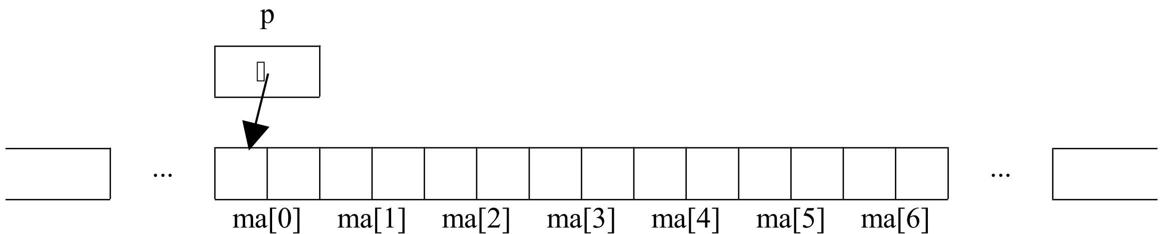


si cada entero es representado internamente con 2 bytes, en la memoria tendremos la siguiente distribución de bytes:



Podemos usar un puntero para acceder a los elementos. De hecho, es muy normal hacerlo así:

```
int ma[7];
int *p;
p = &ma[0];
```



Si queremos leer el elemento `ma[2]` del vector, se podrá hacer de 2 formas:

```
x = ma[2];
```

ó

```
p = &ma[0];
x = *(p+2);
```

Podemos, por ejemplo, rellenar el vector anterior con el valor 77 en todas las posiciones, de dos formas:

```
for (i = 0; i<7; i++) {
    ma[i] = 77;
}
```

ó

```
p = &ma[0];
for (i=0; i<7; i++)
    {*p=77; p++;}              // otra forma *(p++) = 77;
```

La diferencia es que la segunda versión es más rápida y la única posible en ciertos casos que se verán más adelante.

Es tan normal usar punteros con vectores que podemos hacer referencia a la dirección del primer elemento de un vector de dos formas:

```
int *p;
int ma[7];
```

`p = &ma[0];` ó **`p = ma;`**

Actividad

Usando punteros calcula el sumatorio de un vector de 100 `double` y depositalo en una variable. La declaración de variables es:

```
double valores[100];
double sumatorio;
```

1.7 PUNTEROS EN FUNCIONES. PASO POR REFERENCIA

El lenguaje C solo soporta el denominado *paso de parámetros por valor*, lo que significa que los argumentos de las funciones son copiados a una zona de memoria local a la función, sin disponerse de acceso a los datos originales.

Muchos lenguajes de programación soportan el denominado *paso de parámetros por referencia*, en el que se permite trabajar con los elementos originales. En C esto se consigue pasando "por valor" un puntero que contiene la dirección del objeto referenciado.

El siguiente ejemplo permite aclarar el significado. Sea el siguiente programa:

```
void duplica (double x)                   (1)
{
   x = 2.0 * x;                           (2)
}
void main (void)
{
   double dato;

   dato = 5.0;                           (3)
   duplica (dato);                       (4)
   printf ("Valor vale %lf\n", dato);   (5)
}
```

- Al crearse la variable local `dato` (3), en ésta se almacena el dato 5.0

- Al llamar a la función con el parámetro `dato` (4), su contenido es copiado a la variable local `x` (1), es decir, se está haciendo **un paso de parámetros por valor**, estando el valor copiado en una variable local (y, por tanto, que solo existe dentro de la función `duplica()`).
- Al ejecutarse (2), la variable local `x` pasa a tener el valor 10.0. Así, se duplica el valor de la variable `x`, pero no se toca el valor de `dato`.
- Al salir de la función, la variable `x` se destruye automáticamente, perdiéndose su contenido.
- Finalmente, al mostrar el contenido de la variable `dato` por pantalla (5) aparecerá un 5.0

En el siguiente caso,

```
void duplica (double *x)                   (1)
{
   *x = 2.0 * (*x);                        (2)
}
void main (void)
{
   double dato;

   dato = 5.0;                            (3)
   duplica (&dato);                       (4)
   printf ("Valor vale %lf\n", dato);     (5)
}
```

- Al crearse la variable local `dato` (3) en esta se almacena el dato 5.0
- Al llamar a la función con el parámetro &`dato` (4), se está pasando la dirección en memoria de la variable `dato`, así, a la variable puntero `x` (1) se le copia la dirección en memoria de `dato`. Como se está pasando un parámetro por valor en un puntero se dice que se está haciendo **un paso por referencia** de una variable.
- Al ejecutarse (2), los punteros están actuando sobre los contenidos en memoria a los que apuntan, que en este caso coinciden con la dirección en memoria de `dato`. Así, lo que está ocurriendo realmente es que se está duplicando la variable `dato`.
- Al salir de la función, la variable puntero `x` se destruye automáticamente, perdiéndose su contenido, pero su acción sobre `dato` no se pierde.
- Finalmente, al mostrar el contenido de la variable `dato` por pantalla (5) aparecerá en este caso un 10.0

<u>Ejemplo:</u>

El siguiente fragmento permite a la función `scanf()` copiar un dato en una variable, pues se le pasa por referencia

```
int a;
...
scanf("%d",&a);
```

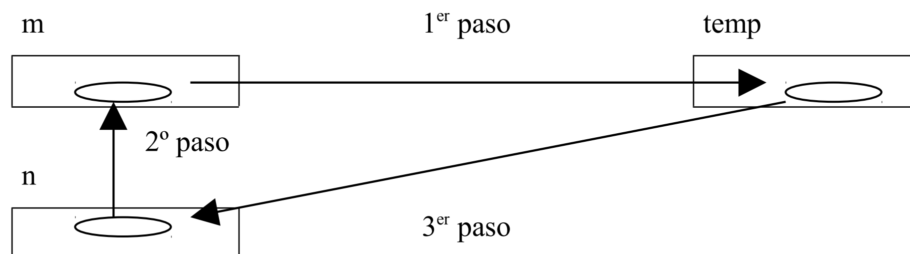
Actividad

Una tarea habitual de los programas que ordenan información es tener una función que intercambie dos datos. Realiza la función con prototipo `void intercambia (double *x, double *y)`; de manera que al llamarla con la dirección de dos variables, intercambie su contenido.

Ejemplo de llamada:

```
double a, b;
...
intercambia (&a, &b);
```

Pistas: para intercambiar dos variables `m`, `n` se hace:



Actividad

Desarrollar una función que permita calcular la media aritmética de los valores contenidos en vectores usando el siguiente prototipo, en el que se le pasa la dirección en memoria del primer valor y la cantidad de valores a tratar:

```
double media (double *p, int elementos);
```

Recuerda:

$$media(datos) = \frac{\sum_i valor_i}{n}$$

Como ejemplo de uso, si suponemos un vector que contiene 5000 doubles correspondientes a tomas de temperatura y otro con 10000 doubles correspondientes a tomas de presión se podría hacer:

```
double temperatura[5000];
double presion[10000];

...
printf("La temperatura media es %lf y la varianza %lf\n",
        media(temperatura, 5000), varianza(temperatura,
5000));
printf("La presión media es %lf y la varianza %lf\n",
        media(presion, 10000), varianza(presion, 10000));
```

La ventaja de haber pasado una referencia al vector, en lugar del vector en sí, es que hemos ahorrado memoria y ganado en velocidad al no hacer una copia de los valores que contiene el vector (en C no se puede hacer esto, no tiene sentido). Además, se ha logrado acceder a los datos originales.

Actividad

Desarrollar la función `varianza()` que se muestra en el ejemplo anterior aprovechando la función `media()` desarrollada. El prototipo de la función es:

```
double varianza (double *p, int elementos);
```

Recuerda:

$$\mathrm{var}(datos) = \frac{\sum_i (valor_i - me}{n -}$$

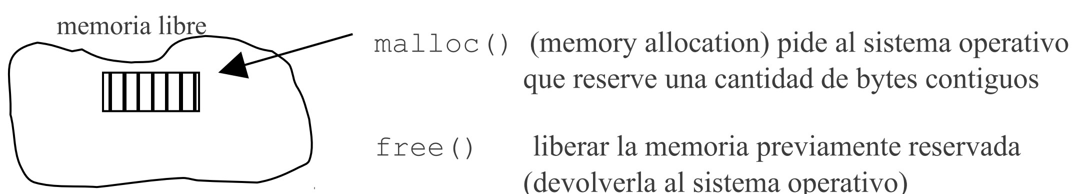
1.8 MEMORIA DINÁMICA

Hasta ahora, las estructuras de datos de C que se conocen permiten crear almacenamientos de información de tamaño fijo y estructura predeterminada.

Con la "memoria dinámica" se pueden crear almacenamientos temporales de tamaño y estructura variable, pudiéndose crear abstracciones de datos como pilas, colas, listas enlazadas, árboles, tablas hash, etc.

Para ilustrar la necesidad de memoria dinámica, imagínese cómo se pueden almacenar en memoria un número arbitrario de números sin saber, en principio, cuántos van a ser, o mantener un texto en memoria o imágenes de los que no se sabe ni la cantidad ni el tamaño. Esto se puede resolver utilizando las técnicas que se van a describir.

La siguiente figura ilustra la idea; se puede considerar a la memoria principal del computador como una enorme extensión libre gestionada por el sistema operativo. Los programas en ejecución podrán "pedir" al sistema operativo que les reserve "parcelas", parcelas que deberán devolver cuando no las necesiten para que el espacio liberado pueda ser usado por otros.



Para usar estas funciones hay que incluir uno de los siguientes archivos de cabecera: `stdlib.h` o `alloc.h`. Sus prototipos son:

cantidad de bytes a reservar

```
void *malloc (size_t tamaño);
```

puntero al principio de la memoria reservada. Si no consigue reservarla devolverá NULL

```
void free (void *bloque);
```

puntero al principio de la memoria previamente reservada

Ejemplo:

Reservar una zona de memoria para almacenar N doubles. Almacenar en ellos los valores del 1 a N.

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

void main (void)
{
   int i, n;
   double *p, *pd;

   printf("Dame N: ");
   scanf("%d", &n);
   p = (double*) malloc (n*sizeof(double));
   if (p == NULL) {
    printf("No hay bastante memoria libre\n");
    exit(1);
   }
   pd = p;
   for (i=0; i<n; i++) {
      *pd=i;
      pd++;
   }

   ...
   free(p);
```

typecast, convertir (void *) a (double *)

cantidad de bytes necesaria para almacenar n doubles

```
}
```

Actividad.

Haz un programa que solicite qué cantidad de números se le van a introducir, reserve memoria (doubles), los pida y los almacene. Usando las funciones `media()` y `varianza()` antes realizadas que devuelva estos datos y libere la memoria.

Para pedir un double por teclado utilizar `scanf("%lf",&variable);`

Para pedir un int por teclado utilizar `scanf("%d",&variable);`

1.9PUNTEROS A ESTRUCTURAS

Es muy habitual en C que la creación de estructuras complejas de datos (listas, colas, árboles, etc.) consista en declarar estructuras C (`struct`) y crear dichas estructuras utilizando memoria dinámica.

Para tener una referencia de dónde están esas estructuras creadas en memoria, se hará uso de punteros, que apuntarán al comienzo de dicha estructura en memoria. Para acceder a los miembros de la estructura se usará el operador `->` con esta sintaxis:

```
nombre_puntero -> nombre_miembro
```

El siguiente ejemplo permite entender la utilización del operador:

```
struct signo{      // no es una variable, es una declaracion
   int dia;        // de tipo de estructura
   int mes;
}

void main (void)
{
   struct signo dato;
   struct signo *puntero *puntero2;

   // acceso directo a la estructura
   dato.dia = 24;
   dato.mes = 3;

   // con punteros
   puntero = &dato;
   puntero -> dia = 24
   puntero -> mes = 3;

   puntero2 = (struct signo *) malloc (2*sizeof(struct sig-
no));
   puntero2 -> dia = 24
```

```
      puntero2 -> mes = 3;
      puntero2 ++;   //apuntar al siguiente elemento
      puntero2 -> dia = 10
      puntero2 -> mes = 5;

   }
```

Actividad

Crea una función capaz de sumar un vector de números complejos. Para representar los números complejos se suele emplear un tipo estructurado cuya definición es

```
struct complejo {
   double real;
   double imaginario;
}
```

El prototipo de la función a desarrollar que debe ser capaz de sumar los complejos es,

```
void suma_complejos(struct complejo *datos, int num_datos,
                    struct complejo *resultado);
```

Como ejemplo de uso, el siguiente programa emplea la función para sumar un vector de N complejos que previamente se ha creado en memoria dinámica.

```
void main(void) {
   struct complejo res;
   struct complejo *p, vector[20];
   ...
   suma_complejos(vector, 20, &res);
   ...
   p = (struct complejo *)malloc(N*sizeof(struct complejo));
   ...
   suma_complejos(p,N,&res);
   ...
}
```

1.10 RESUMEN

Llegados a este punto, se debe saber qué es un punteros y cómo utilizarlo para acceder a los distintos elementos “apuntables” del lenguaje (variable, otros punteros, vectores, estructuras, funciones, etc.).

Prácticamente todo se puede apuntar en C y, para complicar las cosas, muchas veces es necesario mover los punteros para conseguir acceder a determinadas zonas. Esto se logra mediante operaciones aritméticas sencillas.

Por desgracia para los programadores, los punteros y C están tan ligados que hay operaciones que es imposible realizar sin su intervención. A cambio, el lenguaje C es tan cercano a la máquina, que obtendremos rendimientos equivalentes a desarrollar la aplicación en código máquina, pero a varios órdenes de magnitud de sencillez.

Por ejemplo, la creación de estructuras de tamaño variable que se adapten dinámicamente a las necesidades de la aplicación necesitarán de los mecanismos de memoria dinámica y punteros. En C++ existen mecanismos que simplifican el aprovechamiento de estas técnicas, pero, de igual manera, los punteros volverán a aparecer.

1.11 BIBLIOGRAFÍA COMENTADA

- Programación estructurada en C. Antonakos, Mansfield. Prentice-Hall.

  Un libro pretende enseñar el lenguaje C de una manera muy clara y didáctica.

1.12 ACTIVIDADES RESUELTAS

1.12.1 MEDIAS ARITMÉTICAS

```
/* uso de punteros en vetores y en memoria dinamica
*************************/
/* DSII curso 2009/2010 */

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

/* calcular la media aritmetica de los elemtos apuntados
********************/
double media ( double *p, int elementos){

    int i;
    double suma;

    suma=0.0;
    for(i=0;i<elementos;i++) {
        suma = suma + (*p);
        p++;
    }
    return (suma / elementos);
```

```
}

/* calcular la varianza de los elemtos apuntados
****************************/
double varianza (double *p, int elementos){

    int i;
    double med, resta, var;

    resta = 0.0;

    med = media (p, elementos);

    for (i=0 ;i<elementos; i++){
        resta = resta + ( ( (*p)-med ) * ( (*p)-med ) );
        p++;
    }
    var = resta / (elementos-1);
    return (var);
}

/* programa principal
*****************************************************/
int main (void) {

    double temperatura[500000];
    double presion[10000];

    int i, num_temperaturas;

    double *p_datos, *pd;


    // una prueba con vectores

    for (i=0;i< 400000;i++) {
        temperatura[i] = 3.5;
    }

    printf("La temperatura media es %lf\n", media(temperatura, 400000));
    printf("La varianza es %lf\n", varianza(temperatura, 400000));

    printf("La temperatura media es %lf\n", media(&temperatura[1000],
20));  //1
    printf("La varianza es %lf\n", varianza(&temperatura[1000], 20));


    // ahora probamos con memoria dinamica

    printf("¿Cuantas temperaturas quieres? ");
    scanf("%d", &num_temperaturas);
    printf("Pidiendo memoria ...\n");
    p_datos = (double*) malloc (num_temperaturas*sizeof(double));

    if (p_datos == NULL)
    {
        printf("No hay bastante memoria libre\n");
        exit(1);
    }

    // rellenar con datos
    printf("Rellenando memoria ...\n");
    pd = p_datos;
```

```
    for (i=0;i<num_temperaturas;i++) {
        //*pd = i/10.0;
        *pd = (double)i/10;
        pd++;
    }

    printf("Calculando ...\n");
    printf("media=%lf varianza=%lf", media(p_datos, num_temperaturas),
varianza(p_datos,num_temperaturas));

    printf("Devolviendo la memoria ...\n");
    free(p_datos);
    printf("Devuelta!");

    return(0);
}
```

1.12.2 NÚMEROS COMPLEJOS

```
/* ejemplo de uso de estructuras y punteros aplicado a los números complejos
*/
#include <stdio.h>
#include <stdlib.h>

struct complejo{
    double real;
    double imaginario;
};

void suma_complejos (struct complejo *datos, int num_datos, struct complejo
*resultado){
    int i;

    resultado->real=0.0;
    resultado->imaginario=0.0;
    for ( i=0 ; i < num_datos ; i++){
        resultado->real = resultado->real + datos->real;
        resultado->imaginario = resultado->imaginario + datos->imaginario;
        datos++;
    }

}

int main(int argc, char *argv[])
{
    struct complejo res;
    struct complejo res2;
    struct complejo *p,*p2, vector [20];
    int i, j;

    /* probamos primero con un vector de complejos */

    for (i=0;i<20;i++){
        vector[i].real=1.0;
        vector[i].imaginario=2.0;
    }

    suma_complejos(vector,20,&res);

    printf("La suma de los 20 valores complejos es: %lf + %lfi\n", res.real,
```

```
res.imaginario);

    /* ahora jugamos con memoria dinámica */
    p = (struct complejo *)malloc(50*sizeof(struct complejo));

    p2=p; // creo este vector para guardar el comienzo del p
    for (j=0;j<50;j++){
        p2->real=1.0;
        p2->imaginario=2.0;
        p2++;
    }

    suma_complejos(p,50,&res2);

    printf("La suma de los 50 valores es: %lf + %lfi\n", res2.real,
res2.imaginario);

    getchar();
    return (0);
}
```