Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

Capítulo 5. Los punteros en C

Tabla de contenidos

Los punteros son uno de los aspectos más potentes de la programación en C, pero también uno de los más complejos de dominar. Los punteros permiten manipular la memoria del ordenador de forma eficiente. Dos conceptos son fundamentales para comprender el funcionamiento de los punteros:

1. El tamaño de todas variables y su posición en memoria.
2. Todo dato está almacenado a partir de una dirección de memoria. Esta dirección puede ser obtenida y manipulada también como un dato más.

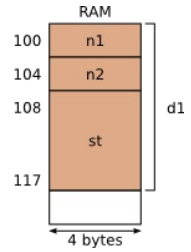
Los punteros son también una de las fuentes de errores más frecuente. Dado que se manipula la memoria directamente y el compilador apenas realiza comprobaciones de tipos, el diseño de programas con punteros requiere una meticulosidad muy elevada que debe ir acompañada de una dosis idéntica de paciencia. Programar eficientemente usando punteros se adquiere tras escribir muchas líneas de código pero requiere una práctica sostenida.

Te recomendamos que leas este documento con tu entorno de desarrollo preparado para probar pequeños fragmentos de código y programas. Debes compilarlos y ejecutarlos para comprobar que se comportan como esperas.

5.1. Todo dato tiene una dirección en memoria

Todos los datos se almacenan a partir de una dirección de memoria y utilizando tantos bytes como sea necesario. Por ejemplo, considera la siguiente definición de datos y su correspondiente representación en memoria a partir de la dirección 100.

```
struct contact
{
    int n1;
    float n2;
    char st[10];
} d1;
```

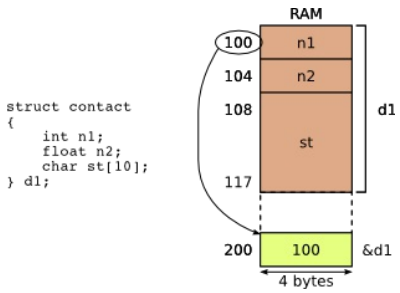


Asumiendo que `int` y `float` ocupan 4 bytes cada uno, y `char` ocupa 1 byte, si `d1` está almacenada a partir de la posición 100, entonces su campo `n1` tiene esa misma dirección, el campo `n2` está en la posición 104 y el campo `st` está almacenado a partir de la posición 108.

5.2. La indirección

Supongamos que las direcciones de memoria se representan internamente con 32 bits (4 bytes).

Entonces, podemos almacenar la dirección de d1 (el número 100) en otro lugar en memoria que ocupará 4 bytes por ser una dirección de memoria. La siguiente figura muestra esta situación con la dirección de memoria almacenada en la posición 200.



En C se obtiene la dirección de memoria de cualquier variable mediante el operando & seguido del nombre de la variable. En la figura anterior, la expresión &d1 devuelve el valor 100.

El valor que devuelve el operador & depende de la posición de memoria de su operando, y por tanto, no está bajo el control del programador. A la dirección de memoria almacenada como dato en la posición 200 se le denomina "puntero" pues su valor "apunta" a donde se encuentra la variable d1. Otra forma de enunciarlo: en la posición 200 hay un puntero a d1.

Si en la posición 200 está almacenado un puntero a la variable d1 se pueden acceder a los datos de esta estructura mediante una "indirección". Se toma el dato almacenado en la posición 200 y su valor (el número 100) se interpreta ahora como una dirección. Se accede a esa dirección y de ahí se accede a los campos de d1. Acabamos de acceder a d1 de forma indirecta, o a través de una "indirección".

La indirección se puede aplicar múltiples veces en un mismo acceso en lo que se conoce como "indirección múltiple". Siguiendo con el ejemplo anterior, podemos almacenar ahora la dirección del puntero (esto es, el valor 200) en otra posición de memoria, por ejemplo, la posición 300. En esta posición 300 tenemos la dirección de la dirección de d1. O de forma análoga, en la posición 300 hay un puntero a un puntero a d1. Igualmente, se puede acceder a los datos de d1 pero esta vez mediante una doble indirección. Se pueden construir indirecciones múltiples con un número arbitrario de niveles.

5.3. El tipo de datos "puntero a"

El operador & seguido del nombre de una variable devuelve su dirección de memoria. El tipo de datos del resultado es "puntero a" seguido del tipo de la variable utilizada. La regla para obtener la sintaxis y el significado de estos tipos de datos es:

> Para todo tipo de datos T existe un tipo de datos que se denomina "Puntero a T" definido como "T *".

En la siguiente tabla se muestran las consecuencias de aplicar esta regla a los tipos de datos básicos en C, así como ejemplos de declaración de variables.

Tipo T	Tamaño (bytes) [a]	Puntero a T	Tamaño (bytes)	Ejemplo de uso
int	4	int *	4	int *a, *b, *c;
unsigned int	4	unsigned int *	4	unsigned int *d, *e, *f;
short int	2	short int *	4	short int *g, *h, *i;
unsigned short int	2	unsigned short int *	4	unsigned short int *j, *k, *l;
long int	4	long int *	4	long int *m, *n, *o;
unsigned long int	4	unsigned long int *	4	unsigned long int *p, *q, *r;
char	1	char *	4	char *s, *t;
unsigned char	1	unsigned char *	4	unsigned char *u, *v;
float	4	float *	4	float *w, *x;
double	8	double *	4	double *y, *z;
long double	8	long double *	4	long double *a1, *a2;

[a] Valores para el Nokia N810 pero pueden cambiar dependiendo de la plataforma

Además de los punteros a los tipos de datos creados, C permite declarar un puntero genérico de tipo void *. Una variable de este tipo almacena un puntero a cualquier dato. Se recomienda utilizar este tipo de datos sólo cuando sea estrictamente necesario.

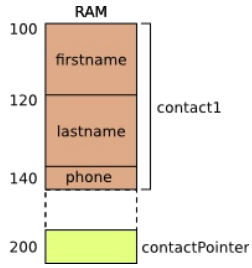
tipo de datos solo cuando sea estrictamente necesario.

El tamaño de los punteros es siempre el mismo independientemente del dato al que apuntan porque todos ellos almacenan una dirección de memoria. Para las estructuras de datos, la regla se aplica de forma idéntica. El siguiente ejemplo muestra la definición de una estructura y la declaración de una variable de tipo estructura y otra de tipo puntero a esa estructura:

```
struct contact
{
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact contact1;

struct contact *contactPointer;
```



La variable `contact1` es de tipo estructura y ocupa 44 bytes (20 + 20 + 4), mientras que `contactPointer` es de tipo puntero y ocupa 4 bytes.

5.3.1. Preguntas de autoevaluación

1. Se define un array de dos enteros de 4 bytes del tipo `int array[2]` y está almacenado a partir de la posición 100 de memoria. Se ejecuta el código:

   ```
   array[0] = 20;
   array[1] = 30;
   ```

   ¿Qué valor tiene el entero almacenado en la posición de memoria 104?

   - 2
   - 20
   - 30
   - 104

   ¿Y en qué dirección está almacenado el primer elemento del array?

   - 100
   - 101
   - 102
   - 103

2. En un programa se definen estas dos variables:

   ```
   int i = 10;
   int *i_ptr;
   ```

   Si la variable `i` está almacenada en la posición de memoria 100, ¿qué valor contiene la variable `i_ptr`?

   - La valor 100 que es la dirección de la variable `i`
   - El valor 104 porque es la posición siguiente a la que ocupa `i`.
   - Ningún valor porque no ha sido inicializada.
   - Su propia dirección de memoria.

3. Si una variable tiene que almacenar la dirección de la dirección de un carácter, ¿qué tipo debe tener su declaración?

   - `char *`
   - `** char`
   - `char **`
   - `string **`

5.4. Asignación de una dirección a un puntero

Dada una variable `var` de tipo `t` y una variable `var_ptr` de tipo puntero a t (`t *`) es posible asignar

```
var_ptr = &var
```

El siguiente ejemplo muestra la declaración y asignación de valores a punteros (fichero pointer_example_1.c):

```
1  #include <stdio.h>
2  int main()
3  {
4      int num1, num2;
5      int *ptr1, *ptr2;
6
7      ptr1 = &num1;
8      ptr2 = &num2;
9
10     num1 = 10;
11     num2 = 20;
12
13     ptr1 = ptr2;
14     ptr2 = NULL;
15
16     return 0;
17 }
```

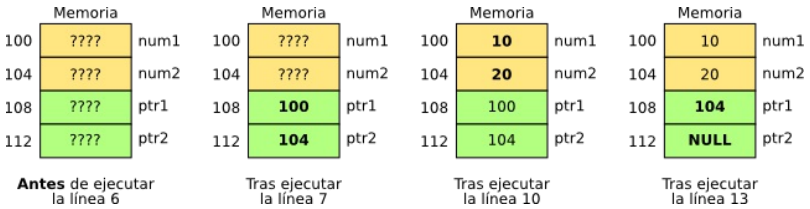
Las líneas 4 y 5 definen dos enteros y dos punteros a enteros respectivamente. Los punteros se comportan como el resto de variables, no tienen valor inicial al comienzo del programa. Las líneas 7 y 8 asignan las direcciones de las dos variables. La dirección de una variable existe desde el principio de un programa, y por tanto esta asignación es correcta a pesar de que todavía no hemos almacenado nada en las variables `num1` y `num2`.

Esta es una fuente común de anomalías. La dirección de una variable se puede obtener en cualquier punto del programa. Un error frecuente consiste en obtener la dirección de una variable cuando esta no tiene valor alguno asignado. El valor del puntero es correcto, pero el valor al que apunta no. En el ejemplo anterior, antes de ejecutar la línea 10, el programa tiene dos punteros inicializados, pero los valores a los que apuntan no lo están.

La línea 13 es una asignación de un puntero a otro. El valor de `ptr2` es la dirección de `num2` tal y como se ha asignado anteriormente. Como resultado, `ptr1` contiene también ahora la dirección de `num2`.

La línea 14 asigna al puntero `ptr2` la constante `NULL` que está definida en el fichero `stdio.h` incluido en la primera línea del programa. Esta constante representa el "puntero vacío". Su valor numérico es cero, y cuando un puntero tiene este valor no está apuntando a nada.

La siguiente figura muestra la evolución de las variables en diferentes puntos de ejecución. Las direcciones de memoria en las que están almacenadas las variables son arbitrarias.



Sugerencia

Copia y pega el contenido del programa anterior en un fichero de texto en tu entorno de desarrollo. Compila para comprobar su corrección sintáctica. Realiza cambios en las declaraciones y asignaciones y re-compila para comprobar su corrección.

5.4.1. Preguntas de autoevaluación

1. Se define las variables `a` de tipo entero, `b` de tipo puntero a entero, y `c` de tipo puntero a puntero a entero. ¿Cómo se consigue que `c` tenga la dirección de la dirección del entero `a`?
   - `c = b; b = a;`
   - `c = b; b = &a;`

- `c = &b; b = a;`
- `c = &b; b = &a;`

2. Dadas dos variables de cualquier tipo en un programa en C, considera la siguiente expressión:

```
&a == &b;
```

   - La expresión es incorrecta.
   - La expresión es correcta y es siempre falsa.
   - La expresión es correcta y es siempre cierta.
   - La expresión es correcta y es cierta o false dependiendo de los valores de `a` y `b`.

5.5. La indirección a través de punteros

Los punteros tienen dos cometidos. El primero es almacenar una dirección de memoria (ver sección 5.4). El segundo es utilizar la dirección de memoria almacenada para acceder al dato al que apunta mediante una indirección (ver sección 5.2 es una indirección).

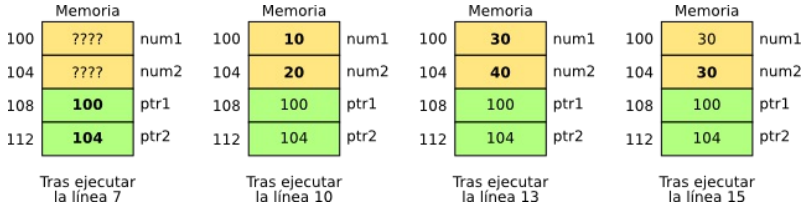
En C la indirección se denota con el operador * seguido del nombre de una variable de tipo puntero. Su significado es "accede al contenido al que apunta el puntero". Desafortunadamente, este operador coincide con el utilizado para denotar los tipos de datos punteros y para declarar este tipo de variables. Es muy importante tener una distinción muy clara entre estos dos usos del operador. Al traducir un programa, el compilador no comprueba ni que el puntero contiene una dirección de memoria correcta, ni que en esa dirección de memoria hay un dato correcto. Esta característica hace que el diseño de programas con punteros sea complejo.

El siguiente programa muestra el uso del operador de indirección (fichero pointer_example_2.c):

```
1  #include <stdio.h>
2  int main(int argc, char** argv)
3  {
4      int num1, num2;
5      int *ptr1, *ptr2;
6
7      ptr1 = &num1;
8      ptr2 = &num2;
9
10     num1 = 10;
11     num2 = 20;
12
13     *ptr1 = 30;
14     *ptr2 = 40;
15
16     *ptr2 = *ptr1;
17
18     return 0;
19 }
```

La línea 13 asigna el valor 30 a la dirección de memoria almacenada en `ptr1` mediante una indirección. Como el puntero tiene la dirección de `num1` (asignada en la línea 7) esta linea es análoga a asignar el valor 30 directamente a `num1`. Análogamente, la línea 14 asigna el valor 40 a la dirección a la que apunta `ptr2`. Esta asignación es equivalente a asignar el valor 40 a `num2`. La línea 16 contiene dos indirecciones. La expresión a la derecha del signo igual obtiene el dato en la posición indicada por la dirección almacenada en `ptr1` (la dirección de `num1`) y este dato se almacena en la posición indicada por el puntero `ptr2` (la dirección de `num2`). Al terminar el programa, la variable `num2` contiene el valor 30 a pesar de que no ha sido asignado directamente.

La siguiente figura muestra la evolución de estas cuatro variables a lo largo del programa. De nuevo, las direcciones de memoria en las que están almacenadas son arbitrarias.



La coincidencia entre el operador de indirección y el de definición de tipo puntero "*" puede dificultar

La coincidencia entre el operador de indirección y el de definición de tipo puntero "*" puede dificultar la comprensión del código. Por ejemplo, la siguiente línea

```
int *r = *p;
```

contiene dos operandos "*". El primero se define el tipo "puntero a entero" y va seguido de la declaración de la variable "r". El segundo es una indirección que se aplica a la dirección almacenada en el puntero "p".

5.5.1. Acceso indirecto a campos de una estructura

Supongamos que se ha definido una estructura con nombre "struct s". Aplicando la regla de definición de tipo puntero, un puntero a esta estructura tiene el tipo "struct s *". En el siguiente fragmento de código se define y declara una estructura, y luego se declara un puntero que se inicializa a su dirección.

```
1  struct s
2  {
3      int x;
4      char c;
5  }
6
7  struct s element;
8  struct s *pointer;
9
10 pointer = &element;
```

Cuando se realiza una indirección con pointer es sólo para acceder a los campos de la estructura. Esta operación requiere dos operadores, la indirección (operador "*" y el acceso a los campos de una estructura (operador "."). La sintaxis es:

```
(*pointer).x = 10;
(*pointer).c = 'l';
```

Estas dos operaciones en C se agrupan en el operador "->" que se sitúa entre el puntero y el nombre del campo de la estructura. La notación equivalente, por tanto, es:

```
pointer->x = 10;
pointer->c = 'l';
```

El operador "->" se escribe siempre a continuación de un puntero que apunta a una estructura, y precede al nombre de uno de los campos de esa estructura. En el siguiente programa se muestra como se pueden copiar los datos de una estructura a otra utilizando este operador.

```
1  /* Definición de la estructura */
2  struct coordenadas
3  {
4      float x;
5      float y;
6      float z;
7  };
8
9  int main()
10 {
11     /* Declaración de dos estructuras */
12     struct coordenadas location1, location2;
13     /* Declaración de dos punteros */
14     struct coordenadas *ptr1, *ptr2;
15
16     /* Asignación de direcciones a los punteros */
17     ptr1 = &location1;
18     ptr2 = &location2;
19
20     /* Asignación de valores a la primera estructura */
21     ptr1->x = 3.5;
22     ptr1->y = 5.5;
23     ptr1->z = 10.5;
24
25     /* Copia de valores a la segunda estructura */
26     ptr2->x = ptr1->x;
27     ptr2->y = ptr1->y;
```

```
27      ptr2->y = ptr1->y;
28      ptr2->z = ptr1->z;
29
30      return 0;
31  }
```

Sugerencia

Copia y pega el contenido del programa anterior en un fichero de texto en tu entorno de desarrollo. Compila y ejecuta. Define una segunda estructura e incluye un puntero a ella como campo de la estructura "coordenada". Escribe código para manipular esta nueva estructura.

5.5.2. Preguntas de autoevaluación

1. El puntero a contiene la dirección de memoria del puntero b que contiene la dirección de memoria del entero c. ¿Cual de las siguientes expresiones le asigna el valor 30 al entero c?
   - *a = 30
   - **a = 30
   - ***a = 30
   - No se puede hacer esa asignación

2. Considera el siguiente fragmento de código:

```
struct data
{
    int *ptr;
    int num;
} a;

a.ptr = &(a.num);
```

   ¿Cuál de las siguientes expresiones asigna el valor 10 al campo "num" de la estructura "a"?
   - *(a.ptr) = 10
   - a.ptr = 10
   - a.*ptr = 10
   - a.ptr = *10
   - La asignación no se puede hacer utilizando sólo el campo "ptr".

5.6. Punteros a punteros

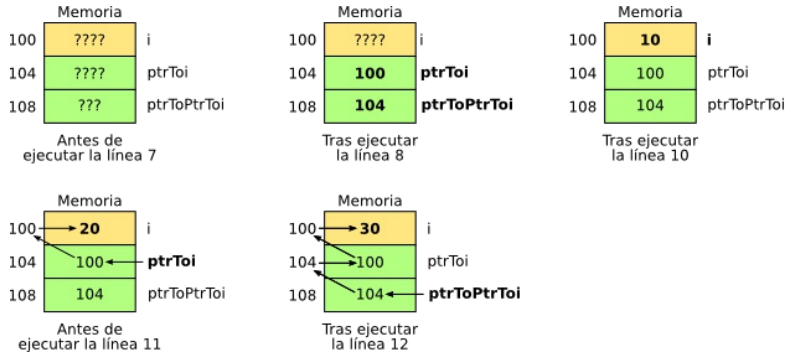
Los punteros también pueden representar el concepto de indirección múltiple (ver sección 5.2). La regla de derivación por la que de un tipo T se deriva el tipo "puntero a T" también se puede aplicar a este nuevo puntero para obtener el tipo "puntero a puntero a T" definido como "T **".

Estos punteros múltiples tienen el mismo tamaño que un puntero simple, la diferencia sólo está en el número de indirecciones para obtener el dato. El siguiente programa es un ejemplo en el que se manipulan punteros con varios niveles de indirección:

```
1   #include <stdio.h>
2   int main()
3   {
4       int i;
5       int *ptrToi;            /* Puntero a entero */
6       int **ptrToPtrToi;      /* Puntero a puntero a entero */
7
8       ptrToPtrToi = &ptrToi; /* Puntero contiene dirección de puntero */
9       ptrToi = &i;            /* Puntero contiene dirección de entero */
10
11      i = 10;                 /* Asignación directa */
12      *ptrToi = 20;           /* Asignación indirecta */
13      **ptrToPtrToi = 30;     /* Asignación con doble indirección */
14
15      return 0;
```

```
16 }
```

La siguiente figura muestra la evolución del valor de todas las variables durante la ejecución del código. Nótese que aunque no hay ningún valor asignado a la variable `i`, la dirección de cualquier variable sí puede ser obtenida y almacenada sin problemas.



Sugerencia

Copia y pega el contenido del programa anterior en un fichero de texto en tu entorno de desarrollo. Utiliza la instrucción "`printf("%d\n", x);`" para imprimir el entero al que se refiere "`x`". Modifica las asignaciones e imprime el valor de las variables para mostrar el funcionamiento de la indirección.

5.7. Uso de la indirección

El uso de punteros en C sólo se justifica si ofrecen una clara ventaja frente a la manipulación de datos directamente. A continuación se describen dos situaciones en las que el uso de punteros redunda en código más eficiente, es decir, que ejecuta a una mayor velocidad, o que ocupa un menor espacio en memoria.

5.7.1. Parámetros por referencia

Hay situaciones en que nos interesará crear funciones que modifiquen el valor de la variable que se le pasa como parámetro y que esta modificación retorne a la función llamadora. En ese caso, decimos que los parámetros se pasan por referencia. Cuando se pasa una variable por referencia, el compilador no pasa la copia de un valor del argumento (como hemos visto en la sección de funciones); sinó que pasa una refrencia que indica a la función dónde se encuentra la variable en memoria. Por tanto, al pasar la dirección de memoria, cuando se hacen cambios sobre esa variable en la función se estará variando el valor de la variable y no el de su copia. Un ejemplo muy claro de función que pasa valores por referencia es el de la función "swap" o "intercambiador". La función "swap" recibe dos parámetros e intercambia el valor del uno por el otro. Observa el código que iplementa esta función.

```
1  #include <stdio.h>
2
3       /* Definición de función "swap". Fíjate que las variables se reciben como puntero a esas variables. */
4      void sswap ( int *x, int *y )
5       {
6           /*Declaramos una variable temporal*/
7           int tmp;
8           tmp = *x;
9           *x = *y;
10          *y = tmp;
11      }
12
13      int main()
14      {
15            int a, b;
16            a = 1;
17            b = 2;
18
19            /*Llamamos a la función "swap" pasándole la dirección a las variables a y b.*/
20            swap( &a, &b );
21            /*Imprime los valores de a y b intercambiados*/
22            printf(" a = %d b = %d\n", a, b );
23
24      }
25
```

Si copias el código y lo ejecutas verás que el programa imrprime los valores de a y b intercambiados. Esto ocurre porque la función, en vez de recibir los parámetros por valor, los recibe por referencia, es decir, recibe su dirección de memoria. Cuando, dentro de la función, se alteran los valores de a y b, se accede directamente al contenido de la dirección de memoria. Por esa razón, fuera del ámbito de la función, a y b mantienen los cambios que han sufrido en la función.

Sugerencia

Modifica el programa anteror sin usar indirecciones. Verás que, cuando salgas de la función los valores de a y b serán los mismo que al inicio, sin haberse intercambiado.

5.7.2. Estructuras como parámetros

Cuando se llama a una función en C, los valores de los parámetros se copian desde el ámbito llamador (lugar en el que está la llamada a la función) al ámbito llamado (el cuerpo de la función). El siguiente programa muestra la ejecución de una función que recibe como parámetro dos estructuras.

```
1  #include <stdio.h>
2  #include <math.h>
3  /* Definición de la estructura */
4  struct coordenadas
5  {
6      float x;
7      float y;
8      float z;
9  };
10 /* Definición de función que calcula la distancia entre dos puntos */
11 float distancia(struct coordenadas a, struct coordenadas b)
12 {
13     return sqrtf(pow(a.x - b.x, 2.0) +
14                  pow(a.y - b.y, 2.0) +
15                  pow(a.z - b.z, 2.0));
16 }
17
18 int main()
19 {
20     /* Declaración e inicialización de dos variables */
21     struct coordenadas punto_a = { 3.5e-120, 2.5, 1.5 };
22     struct coordenadas punto_b = { 5.3e-120, 3.1, 6.3 };
23     float d; /* Almacenar el resultado */
24
25     /* Llamada a la función con las dos estructuras */
26     d = distancia(punto_a, punto_b);
27     /* Imprimir el resultado */
28     printf("%f\n", d);
29
30     return 0;
31 }
```

En la línea 26 se realiza una copia de las estructuras `punto_a` y `punto_b` en el ámbito de la función `main` a las estructuras `a` y `b` en el ámbito de la función `distancia`. Esta copia tiene dos efectos: el programa invierte tanto tiempo como datos haya en la estructura, y durante la ejecución de la función se mantienen en memoria dos copias completas de las estructuras. El uso de punteros ofrece una alternativa más eficiente. La función puede reescribirse para que reciba dos ***punteros a las estructuras***. La definición de la función pasa a ser:

```
float distancia(struct coordenadas *a_ptr, struct coordenadas *b_ptr)
{
    return sqrtf(pow(a_ptr->x - b_ptr->x, 2.0) +
                 pow(a_ptr->y - b_ptr->y, 2.0) +
                 pow(a_ptr->z - b_ptr->z, 2.0));
}
```

La llamada a la función en la línea 26 pasa ahora a ser

```
d = distancia(&punto_a, &punto_b);
```

Con esta nueva versión, el programa ejecuta exactamente los mismos cálculos, obtiene el mismo resultado, pero su ejecución es más rápida, y utiliza menos memoria.

Sugerencia

El programa del ejemplo anterior utiliza las funciones de biblioteca "`pow`" y "`sqrtf`". En una ventana con el intérprete de comandos utiliza el comando **man** seguido de cada uno de estos nombres para averiguar qué hacen. Para utilizar estas funciones se debe incluir el fichero `math.h` mediante la directiva en la línea 2. Además, la biblioteca de funciones matemáticas ha de ser incluida explícitamente al compilar mediante la opción **-lm** en el comando

```
gcc -Wall -o programa fichero.c -lm
```

en el que debes reemplazar `fichero.c` con el nombre de tu fichero.

Modifica el programa del ejemplo para incrementar la precisión en los cálculos. Compara los dos resultados obtenidos.

5.7.3. Enlaces entre estructuras de datos

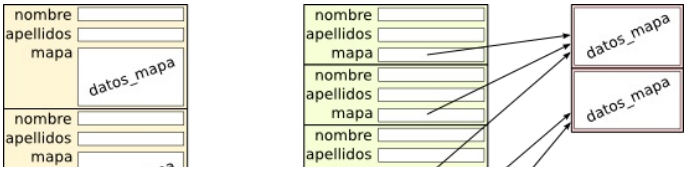
Otro escenario en el que los punteros son de gran utilidad es para "enlazar" estructuras de datos. Supongamos que en la agenda de tu teléfono móvil puedes guardar para cada contacto un mapa. Una posible estructura de datos almacenaría los datos del contacto y tendría un campo para almacenar el mapa. A continuación se muestra una posible definición de las estructuras de datos:

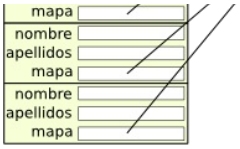
```
1  #define SIZE 64
2  struct datos_mapa
3  {
4  /* Datos referentes al mapa */
5  ...
6  };
7
8  struct datos_contacto
9  {
10     char nombre[SIZE];
11     char apellidos[SIZE];
12     ...
13     /* Datos del mapa para este contacto */
14     struct datos_mapa mapa;
15 };
```

Si la agenda almacena 100 contactos, cada uno de ellos tiene el espacio para almacenar los datos del mapa. Pero imaginemos que este mapa ocupa mucho espacio en memoria debido a los gráficos, y que varios usuarios tienen el mismo mapa, por lo que largas porciones de memoria son réplicas. Lo ideal es mantener la posibilidad de tener un mapa por contacto, pero a la vez evitar tener mapas duplicados. Una posible solución es guardar por separado los usuarios y los mapas. En la estructura del usuario se "enlaza" el mapa pertinente. De esta forma, si varios usuarios comparten el mismo mapa, todos estarán enlazados a una única copia. Este enlace se puede implementar con el uso de punteros tal y como se muestra en las siguientes definiciones:

```
1  #define SIZE 64
2  struct datos_mapa
3  {
4  /* Datos referentes al mapa */
5  ...
6  };
7
8  struct datos_contacto
9  {
10     char nombre[SIZE];
11     char apellidos[SIZE];
12     ...
13     /* Enlace a los datos del mapa */
14     struct datos_mapa *mapa;
15 };
```

En la siguiente figura se muestra la diferencia en tamaño de memoria utilizado en las dos soluciones para el caso en que cinco contactos comparten dos mapas.





Con la nueva solución, el acceso a los datos del mapa se debe hacer con el operador de indirección "->" porque el campo de los datos del mapa es un puntero.

5.7.4. Preguntas de autoevaluación

1. Considera el siguiente fragmento de código:

```
struct data
{
    struct data *s;
} a, b, c;
```

¿Qué tres líneas de código son necesarias para crear una estructura de datos circular (`a.s` apunta a `b`, `b.s` apunta a `c`, y `c.s` apunta a `a`)?

   - `a.s = b; b.s = c; c.s = a;`
   - `&a.s = b; &b.s = c; &c.s = a;`
   - `a.s = *b; b.s = *c; c.s = *a;`
   - `a.s = &b; b.s = &c; c.s = &a;`

2. Considera el siguiente fragmento de código:

```
struct data
{
    int i;
    int j;
} a;
struct data *b = &a;
```

¿Cómo se puede calcular la suma de los campos `i` y `j` de la variable `a` utilizando sólo la variable `b`?

   - `b->i + b->j`
   - `*b->i + *b->j`
   - `b.i + b.j`
   - No se puede calcular.

5.8. Bibliografía de apoyo

- ***Punteros, inicialización e indirección.***:
  - (Teoría) ""The GNU tutorial"", páginas 47-54.
  - (Teoría y ejemplos) "Programación en C: Metodología, algoritmos y estructura de datos", páginas 406-415.
  - (Teoría y ejemplos) "Practical C Programming", páginas 158-166.
- ***Problemas resueltos***: "Problemas resueltos de Programación en Lenguaje C", problemas 4.1-4.5.

5.9. Autoevaluación automática

1. El tamaño de un puntero depende del tipo de datos al que apunta

- Verdadero
- Falso

2. Una variable puntero puede tener delante

- El operador &
- El operador *
- El operador & y el *

3. Una variable entera puede tener delante

- El operador &
- El operador *
- El operador & y el *

4. Sea "p" un puntero a una estructura que ha sido reservada con memoria dinámica. Para acceder a un campo de dicha estructura

- siempre algo de la forma "p->"
- siempre algo de la forma "p."
- algo de la forma "p->" si el campo al que se accede es un puntero
- algo de la forma "p->" si el campo al que se accede no es un puntero

5. Indique cual de los siguientes no es un uso habitual de los punteros

- pasar parámetros por referencia
- establecer enlaces entre estructuras de datos
- restringir el acceso de lectura a los ficheros

5.10. Ejercicios sobre punteros

Los siguientes problemas requieren que conozcas los operandos de manipulación de punteros, cómo se sitúan los datos en memoria y el concepto de indirección. Te recomendamos además, que escribas las definiciones de tipos, declaraciones y fragmentos de código en un fichero de texto plano en tu entorno de desarrollo y que compruebes los resultados mediante su compilación y ejecución. Para los ejercicios se suponen los siguientes tamaños de los tipos de datos básicos:

Tipo	Tamaño (bytes)
`char`, `unsigned char`	1
`short int`, `unsigned short int`	2
`int`, `unsigned int`, `long int`, `unsigned long int`	4
`float`	4
`double`, `long double`	8
Puntero de cualquier tipo	4

Suponte que se definen las siguientes estructuras de datos para guardar la información sobre las celdas con las que tiene posibilidad de conexión un teléfono móvil:

```
1  #define SIZE 100
2  /* Información sobre la celda */
3  struct informacion_celda
4  {
5      char nombre[SIZE];                /* Nombre de la celda */
6      unsigned int identificador;      /* Número identificador */
7      float calidad_senal;              /* Calidad de la señal (entre 0 y 100) */
8      struct informacion_operador *ptr_operador; /* Puntero a una segunda estructura */
9  };
10
11 /* Información sobre el operador */
12 struct informacion_operador
```

```
13 {
14     char nombre[SIZE];           /* Cadena de texto con el nombre */
15     unsigned int prioridad;     /* Prioridad de conexión */
16     unsigned int ultima_comprob; /* Fecha de la última comprobación */
17 };
```

Responde a las siguientes preguntas:

1. ¿Qué tamaño en bytes ocupa una variable de tipo `struct informacion_celda` en memoria?
2. Las siguientes dos líneas declaran dos variables. ¿Cuál de ellas ocupa más espacio en memoria?

```
struct informacion_celda a;
struct informacion_celda *b;
```

3. ¿Qué tamaño tienen las siguientes variables?

```
struct informacion_celda *ptr1, *ptr2;
struct informacion_operador *i1, *i2;
```

4. Si una variable de tipo `struct informacion_celda` está almacenada en la posición de memoria 100, ¿qué dirección tienen cada uno de sus campos?
5. Si una variable de tipo `struct informacion_celda *` está almacenada en la posición de memoria 100, ¿qué dirección tiene cada uno de sus campos?
6. ¿Qué cambios debes hacer en las definiciones de la parte izquierda para que sean equivalentes a las descripciones de la parte derecha?

```
struct informacion_celda c;
struct informacion_celda **c_ptr;
```

```
// variable de tipo estructura informacion_celda
   // puntero a estructura informacion_celda;
```

7. ¿Se pueden hacer las siguientes asignaciones? ¿Qué declara exactamente la linea 3?

```
1 struct informacion_celda c;
2 struct informacion_celda *c_ptr = &c;
3 struct informacion_celda d;
4 struct informacion_celda *d_ptr = c_ptr;
```

8. Considera la siguiente declaración y asignación:

```
1 struct informacion_celda c;
2 struct informacion_celda *c_ptr;
3
4 c_ptr = *c;
```

   ¿Es correcta? Y si lo es, ¿Qué contiene la variable `c_ptr` (no se pregunta por lo que apunta, sino su contenido)?

9. Si se declara una variable como "`struct informacion_celda c;`", ¿qué tipo de datos es el que devuelve la expresión "`&c.ptr_operador`"?
10. Dado el siguiente código:

```
 1 struct pack3
 2 {
 3     int a;
 4 };
 5 struct pack2
 6 {
 7     int b;
 8     struct pack3 *next;
 9 };
10 struct pack1
11 {
12     int c;
13     struct pack2 *next;
14 };
15
16 struct pack1 data1, *data_ptr;
17 struct pack2 data2;
```

```
18 struct pack3 data3;
19
20 data1.c = 30;
21 data2.b = 20;
22 data3.a = 10;
23 dataPtr = &data1;
24 data1.next = &data2;
25 data2.next = &data3;
```

Decide si las siguientes expresiones son correctas y en caso de que lo sean escribe a que datos se acceden.

Expresión	Correcta	Valor
data1.c		
data_ptr->c		
data_ptr.c		
data1.next->b		
data_ptr->next->b		
data_ptr.next.b		
data_ptr->next.b		
(*(data_ptr->next)).b		
data1.next->next->a		
data_ptr->next->next.a		
data_ptr->next->next->a		
data_ptr->next->a		
data_ptr->next->next->b		

11. Supongamos que se escriben las siguientes declaraciones y asignaciones en un programa:

```
1 info_celda c;
2 info_celda_ptr c_ptr = &c;
3 info_operador op;
4 info_operador_ptr op_ptr = &op;
```

La estructura "c" contiene el campo "ptr_operador" precisamente para almacenar la información relativa al operador. ¿Qué expresión hay que usar en el código para guardar la información del operador "op" como parte de la estructura "c"? Teniendo en cuenta los valores que se asignan en las declaraciones, escribe cuatro versiones equivalentes de esta expresión (utiliza "c", "c_ptr", "op" y "op_ptr").

12. Supón ahora que la aplicación en la que se usan estas estructuras necesita almacenar la información para un máximo de 10 celdas. ¿Qué estructura de datos definirías?

13. Escribe un bucle con la variable declarada en el ejercicio anterior que asigne al campo ptr_operador el valor vacío.

14. La información sobre las celdas que se almacena en la estructura del ejercicio anterior la debe utilizar la aplicación para ***recordar*** cuál de ellas es la más próxima. Esta información puede cambiar a lo largo del tiempo. ¿Qué tipo de datos sugieres para almacenar esta información? Ofrece dos alternativas.

15. Se dispone de una estructura de tipo "info_celda c" que a su vez, en el campo "ptr_operador" tiene un puntero a una estructura "info_operador cr". ¿Qué tamaño tiene la estructura "c"? ¿Qué tamaño total ocupa la información incluyendo la información sobre el operador?

16. Escribe el cuerpo de la siguiente función:

```
void fill_in(info_celda_ptr dato, unsigned int id, float sq, info_operador_ptr op_ptr)
```

que asigna el valor de los parámetros "id", "sq" y "op_ptr" a los campos "identificador", "calidad_senal" y "ptr_operador" respectivamente de la estructura apuntada por el parámetro "dato".

¿Cómo explicas que esta función asigne valores a unos campos y no devuelva resultado?

17. Considera las dos versiones del siguiente programa:

Versión 1	Versión 2
#include <stdio.h> struct package	#include <stdio.h> struct package

```
{
int q;
};

void set_value(struct package data, int value)
{
data.q = value;
}

int main()
{
struct package p;
p.q = 10;
set_value(p, 20);
printf("Value = %d\n", p.q);
return 0;
}
```

```
{
int q;
};

void set_value(struct package *d_ptr, int value)
{
d_ptr->q = value;
}

int main()
{
struct package p;
p.q = 10;
set_value(&p, 20);
printf("Value = %d\n", p.q);
return 0;
}
```

La versión 1 del programa imprime el valor 10 por pantalla, y la versión 2 imprime el valor 20. Explica por qué.

Sugerencia

Puedes copiar y pegar el código en un fichero en tu entorno de desarrollo y verificar que los dos programas se comportan tal y como se dice.

18. ¿Qué cantidad de memoria ocupan estas dos estructuras? ¿Cuál es su diferencia?

```
info_celda t[SIZE];
cell_info *t[SIZE];
```

19. Una aplicación de gestión de fotografías en tu móvil tiene definido el catálogo de fotos de la siguiente forma:

```
#define SIZE_NAME

struct picture_info
{
    char name[SIZE_NAME];
    int date_time;
} pictures[SIZE];
```

¿Qué tamaño tiene esta estructura de datos? La aplicación necesita crear una segunda tabla del mismo número de elementos, pero en lugar de tener los datos de las fotos quiere tener los punteros a los datos de las fotos. En otras palabras, es una tabla con idéntico número de elementos que la anterior, pero sus elementos no son estructuras sino punteros a las correspondientes estructuras de la tabla `pictures`. Escribe la declaración y el código para rellenar esa tabla.

20. Se dispone de la siguiente definición de datos:

```
#define SIZE 4
struct coordinates
{
    int latitude;
    int longitude;
} places[SIZE];

places[0].latitude  = 200;
places[0].longitude = 300;
places[1].latitude  = 400;
places[1].longitude = 100;
places[2].latitude  = 100;
places[2].longitude = 400;
places[3].latitude  = 300;
places[3].longitude = 200;
```

La tabla almacena cuatro puntos obtenidos del GPS de tu móvil, cada uno de ellos con su latitud y longitud que son números enteros. Hay una aplicación que ha obtenido estos puntos en este orden, pero necesita acceder a los datos de tres formas distintas. La primera es en el orden en que han sido obtenidos, y por tanto, tal y como está la tabla. El segundo es ordenados

que han sido obtenidos, y por tanto, tal y como está la tabla. El segundo es ordenados crecientemente por latitud y el tercero ordenados crecientemente por longitud. El acceso a estos datos en base a estos tres órdenes es continuo. Una primera solución podría ser reordenar los elementos de la tabla cada vez que se cambia de orden (por ejemplo, llega una petición de datos ordenados por latitud, pero la anterior se ordenaron por longitud, y entonces se reordena la tabla). Pero es extremadamente ineficiente. ¿Por qué? ¿Crees que utilizando punteros puedes ofrecer una alternativa más eficiente que evite reordenar los datos continuamente?