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 2. Tipos de datos en C

Tabla de contenidos

2.1. Tipos de datos básicos

- 2.1.1. Enteros
- 2.1.2. Letras y cadenas
- 2.1.3. Números reales
- 2.1.4. Tablas
- 2.1.5. Tamaño de los tipos de datos básicos

2.2. Tipos de datos estructurados
2.3. Uniones
2.4. Enumeraciones
2.5. Bibliografía de apoyo
2.6. Ejercicios

Las estructuras de datos del lenguaje C son más simples que las que ofrece Java porque no existe el concepto de "clase" ni de "objeto". C ofrece tipos de datos básicos y dos construcciones para crear datos más complejos. El control de acceso a datos que ofrece Java (métodos y campos privados, públicos y protegidos) no existe en C. Las variable son globales, locales a un fichero, o locales a un bloque de código.

2.1. Tipos de datos básicos

C ofrece tres tipos de datos básicos:

- Números enteros definidos con la palabra clave `int`
- Letras o caracteres definidos con la palabra clave `char`
- Números reales o en coma flotante definidos con las palabras claves `float` o `double`

2.1.1. Enteros

Se definen con "`int`" y admiten de forma opcional dos prefijos modificadores:

- "`short`" y "`long`": Modifica el tamaño en bits del entero. Existen por tanto tres tipos de enteros: "`int`", "`short int`" (que se puede abreviar como "`short`"), y "`long int`" (que se puede abreviar como "`long`").

  El lenguaje C no define tamaños fijos para sus tipos de datos básicos. Lo único que garantiza es que un `short int` tiene un tamaño ***menor o igual*** que un `int` y este a su vez un tamaño menor o igual a un `long int`. Esta característica del lenguaje ha complicado la creación de programas que sean compatibles entre varias plataformas.

- "`unsigned`": define un número natural (mayor o igual a cero).

Sugerencia

En tu entorno de desarrollo crea un fichero de texto con la siguiente estructura (puedes simplemente copiar y pegar el texto del siguiente cuadro):

```
int main()
{

}
```

Inserta en la función `main` varias definiciones de enteros para probar todas las combinaciones posibles (hasta diez). Para comprobar que la sintaxis utilizada es correcta abre una ventana con el intérprete de comandos y, en la carpeta donde se encuentra el fichero creado, ejecuta el comando **gcc -Wall -o programa fichero.c** reemplazando **fichero.c** por el nombre del fichero que has creado. Si el comando no imprime mensaje alguno por pantalla, tu programa es correcto. Verás que el compilador genera un fichero con extensión "`.o`", puedes borrarlo.

Preguntas de autoevaluación

Responde a las siguientes preguntas (comprueba tus respuestas también compilando el programa):

1. El siguiente programa compila sin errores:

```
void main()
{
  int i;
  long int j;
  long k;
  short int l;
  short m;
}
```

   - Verdadero
   - Falso

2. El siguiente programa produce un error al compilar porque se está asignando una variable entera a una sin signo (natural).

```
void main()
{
  unsigned i;
  int j = 0;

  i = j;
}
```

   - Verdadero
   - Falso

3. La declaración `unsigned short x` es idéntica a `short unsigned x`.

   - Verdadero
   - Falso

   Escribe un breve programa y comprueba la respuesta con el compilador.

2.1.2. Letras y cadenas

Las variables de tipo letra se declaran como "`char`". Para referirse a una letra se rodea de comillas simples: `'M'`. Como las letras se representan internamente como números, el lenguaje C permite realizar operaciones aritméticas como `'M' + 25`.

Las cadenas de texto o *strings* son simplemente tablas de "`char`". Las funciones de biblioteca para manipular estas cadenas asumen que el último byte tiene valor cero. Las cadenas de texto se escriben en el programa rodeadas de dobles comillas y contienen el valor cero al final. A continuación se muestran dos definiciones:

```
#define SIZE 6
char a = 'A';
char b[SIZE] = "hello";
```

¿Por qué la segunda definición es una tabla de seis elementos si la palabra tiene sólo cinco letras?

Sugerencia

Reutiliza el programa de la sección anterior y añade definiciones de letras y cadenas. Para estas últimas prueba a poner diferentes tamaños de tabla (demasiado pequeños y demasiado grandes para la cadena). Escribe también expresiones aritméticas sobre las letras. Recuerda que si el compilador no emite mensaje alguno, el programa es correcto.

Preguntas de autoevaluación

1. Considera la siguiente declaración:

```
#define SIZE 6
char m[SIZE] = 'strag';
```

Es incorrecta porque la cadena debe ir rodeada de dobles comillas.

- Verdadero
- Falso

Es incorrecta porque el tamaño debe ser 5 (tiene cinco letras).

- Verdadero
- Falso

Si te parece que las respuestas correctas están equivocadas, escribe esa declaración en un programa y compílalo.

2. Un programa C que imprime el resultado de la expresión `'M' + 25` es correcto e imprime una "f".

- Verdadero
- Falso

Te recomendamos que escribas ese programa. Para imprimir pon `printf("%c\n", 'M' +`

```
25);.
```

2.1.3. Números reales

Los números reales se definen con "`float`" o "`double`". La diferencia entre ambas es la precisión que ofrece su representación interna. Hay un número infinito de reales, pero se representan con un número finito de bits. A mayor número de bits, mayor número de reales se representan, y por tanto, mayor precisión. Los reales definidos con "`double`" tienen un tamaño doble a los definidos con "`float`". Al igual que en el caso de los enteros, el tamaño de estas representaciones varía de una plataforma a otra.

Algunas plataformas ofrecen números reales con tamaño mayor al "`double`" que se definen como "`long double`". Los tamaños típicos para los tipos "`float`", "`double`" y "`long double`" son 4, 8 y 12 bytes respectivamente. A continuación se muestran varias definiciones de números reales.

```
float a = 3.5;
double b = -5.4e-12;
long double c = 3.54e320;
```

Sugerencia

Añade al programa de los apartados anteriores definiciones de números reales. Prueba a definir números muy grandes o pequeños para ver la capacidad de representación de los tres tipos. Compila para ver si las definiciones son correctas.

2.1.4. Tablas

Las tablas en C son prácticamente idénticas a las de Java, con el tamaño entre corchetes a continuación del nombre. Al igual que en Java, los índices de la tabla comienzan por cero. A continuación se muestran algunos ejemplos:

```
#define SIZE_TABLE 100
#define  SIZE_SHORT 5
#define SIZE_LONG 3
#define SIZE_NAME 10

int table[SIZE_TABLE];
short st[SIZE_SHORT] = { 1, 2, 3, 4, 5 };
long lt[SIZE_LONG] = { 20, 30, 40};
char name[SIZE_NAME];
```

Los elementos de la tabla se acceden con el nombre de la tabla seguido del índice entre corchetes.

Una de las diferencias entre C y Java es que el acceso a una tabla en C no se verifica. Cuando se ejecuta un programa en Java si se accede a una tabla con un índice incorrecto, se genera una excepción de tipo "`ArrayIndexOutOfBounds`". Estas comprobaciones no se hacen ***nunca*** en C (a no ser que se escriban explícitamente en el programa). Si se accede a una tabla con un índice incorrecto se manipulan datos en una zona de memoria incorrecta y el programa continua su ejecución.

Tras este acceso incorrecto pueden suceder dos cosas. La primera es que la memoria a la que ha accedido por error esté fuera de los límites del programa. En este caso la ejecución termina de manera abrupta y en el intérprete de comandos se muestra el mensaje "**segmentation fault**". La otra posibilidad es que se acceda a otro lugar dentro de los datos del programa. Esta situación seguramente producirá un error cuyos síntomas sean difíciles de relacionar con el acceso incorrecto.

Tablas de múltiples dimensiones

C permite la definición de tablas de múltiples dimensiones escribiendo los diferentes tamaños rodeados de corchetes y concatenados. El acceso se realiza concatenando tantos índices como sea preciso rodeados de corchetes. Al igual que en el caso de las tablas unidimensionales, no se realiza ningún tipo de comprobación de los índices cuando se accede a un elemento. A continuación se muestra la definición de tablas de más de una dimensión.

```
#define MATRIX_A 100
#define MATRIX_B 30
#define COMMON_SIZE 10

int matrix[MATRIX_A][MATRIX_B];
long squarematrix[COMMON_SIZE][COMMON_SIZE];
char soup[COMMON_SIZE][COMMON_SIZE];
```

Sugerencia

Añade al programa de los apartados anteriores definiciones y manipulación de tablas de los diferentes tipos de datos básicos. Comprueba que son correctos sintácticamente mediante el compilador.

2.1.5. Tamaño de los tipos de datos básicos

En C, el tamaño de los tipos de datos básicos puede variar de una plataforma a otra. Esta característica está detrás de buena parte de las críticas que recibe este lenguaje, pues de ella se derivan problemas de compatibilidad (una aplicación se comporta de forma diferente cuando se ejecuta en plataformas diferentes).

A modo de ejemplo, en la siguiente tabla se incluyen los tamaños de los tipos de datos para las plataformas Linux/Intel i686.

Tabla 2.1. Tamaños de los tipos de datos en C en la plataforma Linux/Intel i686

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`	8
`long double`	12

2.2. Tipos de datos estructurados

C permite definir estructuras de datos que agrupan campos de otros tipos de datos. La sintaxis se muestra a continuación:

```
struct nombre_de_la_estructura
{
    tipo_1 nombre_del_campo1;
    tipo_2 nombre_del_campo2;
    ...
    tipo_N nombre_del_campoN;
};
```

```
};
```

La construcción anterior sólo ***define*** un nuevo tipo de datos, no se declara variable alguna. Es decir, la construcción anterior tiene la misma entidad que el tipo "`int`" o "`float`". El nombre del nuevo tipo estructurado definido es "`struct nombre_de_la_estructura`". Por ejemplo:

```
1  #define FIRST_SIZE 100
2  #define LAST_SIZE 200
3  #define CONTACTS_NUM 100
4
5  /* Definición de la estructura */
6  struct contact_information
7  {
8      char firstname[FIRST_SIZE];
9      char lastname[LAST_SIZE];
10     unsigned int homephone;
11     unsigned int mobilephone;
12 };
13
14 /* Declaración de variables con esta estructura */
15 struct contact_information person1, person2, contacts[CONTACTS_NUM];
```

Las líneas 6 a 12 definen un nuevo tipo de datos estructurado que contiene cuatro campos, los dos primeros son tablas de letras y los dos últimos son enteros. A pesar de que estos campos tienen nombres y tamaños, hasta el momento no se ha declarado ninguna variable. Es en la línea 15 en la que se sí se declaran tres variables de este nuevo tipo estructurado. La última de ellas es una tabla de 100 de estas estructuras. Asegúrate de que tienes clara la diferencia entre la "definición" de un tipo de datos y la "declaración" de variables de ese tipo. La siguiente figura muestra estos conceptos para una estructura y un tipo básico.

	Tipo de datos básico	Tipo de datos estructurado
Definición	`int`	`struct contact_information { char firstname[100]; char lastname[200]; int homephone; int mobilephone; };`
Declaración	`int a, b[100];`	`struct contact_information a, b[100];`

Sugerencia

Copia y pega en un fichero de texto en tu entorno de trabajo el código del ejemplo anterior. Compila para comprobar que es correcto. Realiza cambios en la estructura: número de campos, tipos de datos, declaración de nuevas variables, etc. Comprueba que todos ellos mantienen el texto correcto utilizando el compilador.

La definición de una estructura y la declaración de variables se pueden combinar en la misma construcción, pero preferimos que lo hagas por separado, por legibilidad:

```
struct contact_information
{
    char firstname[FIRST_SIZE];
    char lastname[LAST_SIZE];
    unsigned int homephone;
    unsigned int mobilephone;
} person1, person2 contacts[CONTACTS_NUM];
```

El acceso a los campos de una variable estructurada se denota por el nombre de la variable seguido

de un punto y del nombre del campo tal y como se muestra en el siguiente ejemplo..

```
#define FIRST_SIZE 100
#define LAST_SIZE 200
#define CONTACTS_NUM 100

struct contact_information
{
    char firstname[FIRST_SIZE];
    char lastname[LAST_SIZE];
    unsigned int homephone;
    unsigned int mobilephone;
};

int main(int argc, char *argv[])
{
    struct contact_information person1;

    person1.firstname[0] = 'A';
    person1.firstname[1] = 0;
    person1.lastname[0] = 'B';
    person1.lastname[1] = 0;
    person1.homephone = 975556768;
    person1.mobilephone = 666555444;
}
```

Los tipos estructurados pueden anidarse. El único requisito es que la definición de un tipo preceda a su uso. Por ejemplo:

```
1  #define SIZE 100
2  struct point_data
3  {
4      int coord_x;
5      int coord_y;
6  };
7
8  struct polygon {
9      char description[SIZE];
10     struct point_data points[SIZE;
11 };
12
13 struct polygon p;
```

¿Cuántas variables se han declarado en el código anterior? ¿Cuántos tipos de datos se han definido? Imagínate que eres el compilador y que debes reservar memoria para almacenar los datos que se acaban de declarar. ¿Cuáles de estas líneas se traducirían en reserva de memoria? ¿De qué tamaño?

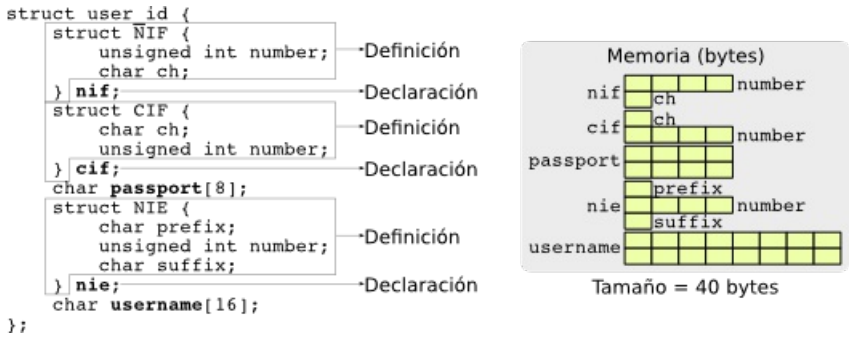
2.3. Uniones

En una estructura, cada campo tiene espacio en memoria para almacenar su valor. Pero hay situaciones especiales en la que las estructuras de datos pueden desperdiciar memoria. Supongamos que una aplicación puede identificar a los usuarios mediante uno de los siguientes posibles cinco datos:

- NIF: ocho dígitos seguidos de una letra.

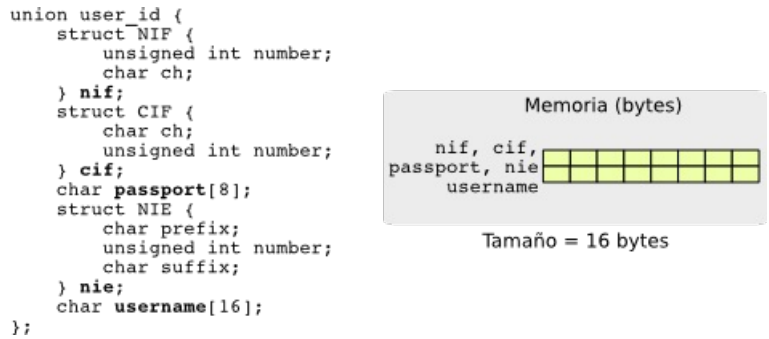
- CIF: letra seguida de 8 dígitos.
- Pasaporte: ocho letras y/o números.
- NIE: letra seguida de 7 dígitos y una segunda letra.
- Nombre de usuario: cadena de texto de hasta 16 letras.

Una posible estructura de datos para almacenar esta inforamación se muestra en la parte izquierda de la siguiente figura:



Si asumimos que los enteros ocupan 4 bytes y las letras 1 byte, la estructura necesita 40 bytes para ser almacenada en memoria tal y como se muestra en la parte derecha de la figura anterior. Pero de todos los campos sólo uno contiene información, el resto están vacíos. Esto quiere decir que la estructura de datos sólo utilizará entre un 12.5% y un 40% del espacio que ocupa. Más de la mitad de la memoria se desperdicia.

C ofrece una estructura en la que ***todos los campos comparten el mismo espacio de memoria*** y que está pensada precisamente para aquellos casos en las que sólo uno de esos campos se utiliza en cada momento. Esta estructura se define reemplazando la palabra "`struct`" por "`union`". La siguiente figura muestra la definición de los datos de usuario utilizando `union`.



Cuando se define una "`union`" se reserva sólo el espacio del mayor de los campos. Los datos se almacenan comenzando por la misma posición de memoria y con la estructura del campo seleccionado. Esta construcción no guarda en ningún lugar cuál de los campos se está utilizando. Si esa información es necesaria, el programador debe almacenarla en una estructura de datos auxiliar. El acceso a los campos de una "`union`" se realiza igual que en una estructura.

2.4. Enumeraciones

Este tipo de datos se utiliza cuando se necesitan variables para almacenar un conjunto reducido de valores no cubiertos por ninguno de los tipos básicos. Supongamos una aplicación quiere almacenar el tipo de conexión de la que dispone el dispositivo y que puede tener los valores GPRS, Wifi, Bluetooth o ninguna. C permite definir un tipo de datos que sólo puede tener estos valores mediante la palabra clave "`enum`" de la siguiente forma:

```
enum type_of_connection { GPRS, Wifi, Bluetooth, Ninguna};
```

A partir de esta definición existe un nuevo tipo de datos `enum type_of_connection` que puede ser utilizado como cualquier otro:

```
enum type_of_connection connection_type;
connection_type = GPRS;
connection_type = Bluetooth;
```

2.5. Bibliografía de apoyo

- ***Tipos numéricos (teoría y ejemplos)***: "The GNU tutorial", páginas 19-22.
- ***Arrays (teoría y ejemplos)***: "The GNU tutorial", páginas 91-100.
- ***Strings, funciones para su manejo (teoría y ejemplos)***:
  - "Practical C Programming":
    - En línea: sección 5.2.
    - Libro: páginas 37-39.
  - "The GNU tutorial", páginas 105-111.
- ***Structs y Unions (teoría y ejemplos)***:
  - "Programación en C: Metodología, algoritmos y estructura de datos", páginas 380-401.
  - "The GNU tutorial", páginas 205-114.
- ***Problemas resueltos***: "Problemas resueltos de Programación en lenguaje C", problemas 2.10, 2.12, 2.13, 2.15.

2.6. Ejercicios

Para resolver los siguientes ejercicios te recomendamos que crees un fichero de texto en tu entorno de trabajo y que escribas en él las soluciones. Compila y (si procede) ejecuta el programa para verificar que realiza las operaciones esperadas. Puedes utilizar el siguiente esqueleto:

```
#include <stdio.h>
int main()
{

    /* Escribe aquí tu código */

    return 0;
}
```

1. Escribe la definición de una estructura que almacene una tabla de 10 enteros, una cadena de 20 letras, un número real de precisión simple y uno de precisión doble. A continuación, en una línea separada, declara dos variables de este tipo. Compila el programa para verificar que la sintaxis es la correcta.
2. Un programador ha escrito las siguientes definiciones:

```
#define SIZE 100

struct map_annotation
{
```

```
    char note[SIZE];
    struct coordinates point;
};

struct coordinates
{
   float longitude;
   float latitude;
};

struct map_annotation annotations[SIZE];
```

¿Qué problema tienen? (siempre puedes cortar y pegarlas en tu programa y utilizar el compilador para comprobar)

3. Incluye en tu programa la variable `cadena` para almacenar una cadena de texto de 10 letras definida como `char cadena[SIZE];`, donde `SIZE` es 11. Se quiere procesar esa cadena de forma que cada letra pase a valer una letra que está cuatro posiciones mas avanzadas en el abecedario. Escribe el código para ejecutar esta operación (no te preocupes por las letras "w", "x", "y" y "z". Al final del programa incluye la siguiente línea para imprimir la cadena por pantalla:

```
printf("%s\n", cadena);
```

4. Descarga el programa enum.c en tu entorno de desarrollo Linux. Abre una ventana con un intérprete de comandos, compílalo con el comando **gcc -Wall -o enum enum.c** y ejecútalo con el comando **./enum**.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    enum status { ON = 0, OFF = 1 };
    enum status a, b;

    a = 0;
    a++;
    a = 20;
    b = a++;

    /* Imprime a y luego b en la misma linea */
    /* Print a and then b in the same line */
    printf("%d %d\n", a, b);
    return 0;
}
```

¿Qué valores imprime por pantalla? A la vista del resultado, ¿qué conclusión se deriva del funcionamiento de los tipos de datos enumerados en C?

5. Define una estructura de datos para almacenar el resultado de un sorteo de la Lotería Primitiva. Es decir, un grupo de seis números, un número complementario y un número de reintegro, todos ellos diferentes y entre los valores del 1 al 49. Calcula el tamaño que ocupa tu estructura de datos con los tamaños mostrados en la sección 2.1.5 (ver las Normas del juego).

6. Declara una tabla de 100 enteros y rellena su contenido con un valor cualquiera. Declara una

segunda tabla también de 100 enteros. Escribe el código que almacena en cada elemento de la segunda tabla la suma de elementos hasta esa misma posición de la primera tabla.

7. Se necesita diseñar una estructura de datos para el buzón de entrada de un teléfono móvil. Los mensajes en ese buzón son de dos tipos, o SMS, o MMS. Los mensajes SMS tiene un remitente (un número de teléfono), una fecha/hora (que es un número entero), un texto de no más de 140 letras y un tipo de mensaje que puede ser SMC o normal. Los mensajes MMS tienen los mismos campos excepto que en lugar del tipo de mensaje tienen una cadena de texto de hasta 200 letras con la ruta a la imagen que contiene. Escribir las definiciones de las estructuras de datos necesarias para tener una tabla de 100 mensajes como buzón de entrada.

8. Descarga el siguiente programa equal.c en tu entorno de trabajo Linux.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    struct equal { int x; };
    struct equal e1, e2;
    e1.x = 1;
    e2 = e1; /* SPECIAL LINE */
    e1.x = 2;
    printf("%d\n", e2.x);
    return 0;
}
```

Abre un terminal con un intérprete de comando, compila el programa con el comando **gcc -Wall -o equal equal.c** y ejecútalo con el comando **./equal**. ¿Puedes adivinar qué resultado se va a imprimir por pantalla? A la vista del resultado explica qué ocurre exactamente en la línea `e2 = e1` y en general con el operando = aplicado a estructuras en C.

Si tienes curiosidad, el programa Equal.java contiene la versión de este código en Java, y el resultado que aparece en pantalla no es el mismo que el de C. ¿Por qué crees que pasa esto?

```
public class Equal {
    int x;
    public Equal() {
 x = 1;
    }
    public static void main(String args[]) {
 Equal e1 = new Equal();
 Equal e2 = e1; /* SPECIAL LINE! */
 e1.x = 2;
 System.out.println(e2.x);
    }
}
```