Arrays de longitud variable en C

Guillermo Hernández

2023-12-08

Versión en línea: https://www.dih5.es/vla.html.

1 Uso básico de los arrays de longitud variable

¿Es correcto el siguiente código C, en el que se emplea **una variable para definir el tamaño de un array**?

```c
#include <stdio.h>

int main(){
    int a,i;
    scanf("%d", &a);
    int v[a];
    for (i=0;i<a;i++)
        v[i]=i;

    for (i=0;i<a;i++)
        printf("%d ", v[i]);

    puts("OK");
    return 0;
}
```

La respuesta correcta es que **depende**. Si lo compilamos con las opciones usuales y probamos con un valor pequeño para la entrada, el funcionamiento es correcto; pero, introduciendo un valor grande (pongamos 10000000) nos encontraremos un *segmentation fault*.

El problema reside en que el tamaño del vector `v` no se puede determinar en tiempo de compilación, sino que se hace en tiempo de ejecución. Esta es una característica que está solo presente en algunas versiones del lenguaje, a partir de C99, y que se llama arrays de tamaño variable o VLA (*Variable-Length Array*). El problema que puede ocurrir es el que hemos visto, que no se puede controlar la situación de si se ha podido obtener realmente la cantidad de memoria necesaria.

En C hay mecanismos de memoria dinámica que permitan considerar esa situación (los estudiamos en Programación II), pero este problema también lo podemos resolver utilizando memoria estática (y así no nos salimos del temario de Programación I). Un ejemplo típico, como los que vemos en la asignatura, es el siguiente:

```c
#include <stdio.h>
#define MAX 10000

int main(){
    int a,i;
    int v[MAX];

    scanf("%d", &a);
    while(a <0 || a > MAX){
        printf("Error: el valor debe estar entre 0 y %d. Reintente.", MAX);
        scanf("%d", &a);
    }

    // [...]
    return 0;
}
```

Es verdad que si modificamos `MAX` también podemos dar lugar a un código que no se pueda ejecutar, pero la macro, además de servir naturalmente como documentación, nos ofrece un mayor control sobre la situación. Por ejemplo, podemos modificar su valor en tiempo de compilación si la definimos solo condicionalmente:

```c
#ifndef MAX
#define MAX 10000
#endif
```

Esto nos permite definir su valor al llamar al compilador: `gcc -DMAX=100 a.c`.

Por otro lado, si estamos trabajando con valores pequeños, es verdad que podemos usar los arrays de tamaño variable de C99, pero debemos controlar que tengan valores «razonables», como hemos visto.

La pregunta final… ¿es una buena idea usar arrays de tamaño variable? Esta característica se volvió «de soporte opcional» para los compiladores en C11; además, C++ no la ha adoptado (aunque los compiladores pueden proporcionarlo como extensión). Esto nos hace pensar que no todos los programadores los valores positivamente; sin embargo, dado que es una opción que nos proporciona alguna versión del lenguaje, podemos hacer uso de ella si:

- No estamos obligados a usar C89/C90/ANSI (el estándar más clásico).

- Controlamos manualmente que el array tenga un tamaño válido y «razonable», con la dificultad de que este concepto puede ser complicado de acotar con exactitud.

Para la asignatura de Programación I, lo mejor es que sigais el patrón de tamaño en memoria + tamaño real, porque en los enunciados puede haber condiciones que imposibiliten el uso de los arrays de tamaño variable; también se parecerá más al código que estáis acostumbrados a ver y será más difícil cometer errores.

Sin embargo; si queréis profundizar más en los VLA y entender por qué existen, podéis seguir leyendo.

2 Uso con funciones

En caso de utilizar VLA en una función, debemos tener en cuenta que para que el tipo esté correctamente definido, los parámetros de los que dependa deben aparecer antes en la signatura. El siguiente ejemplo muestra un VLA en pantalla.

```
void print_matrix(int a, int b, int m[a][b]){
    int i, j;
    for (i = 0; i < a; i++)
        for (j = 0; j < b; j++)
            printf("%d%c", m[i][j], j + 1 == b ? '\n' : ' ');
    printf("\n");
}
```

Sería incorrecta una signatura como `void print_matrix(int m[a][b], int a, int b)`, porque el tipo de `m` no está definido sin conocer `a` y `b`.

3 Uso con memoria dinámica

Podemos tener los mejor de los dos mundos si combinamos la definición de tipos de VLA con las operaciones de reserva dinámica de memoria (Programación II):

- Definimos funciones que usen en su interfaz VLA, de modo que dentro de ellas podemos utilizar el operador corchete con normalidad.
- Utilizamos un puntero a un VLA, esto es, algo como un `int (*m)[a][b]` para reservar memoria, pudiendo comprobar si ha fallado la reserva.

El siguiente código muestra un ejemplo completo

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

// Inicia un array a x b con enteros aleatorios entre 0 y 99
```

```c
void init_random(int a, int b, int m[a][b]){
    int i, j;
    for (i = 0; i < a; i++)
        for (j = 0; j < b; j++)
            m[i][j] = rand() % 100;
}

void print_matrix(int a, int b, int m[a][b]){
    int i, j;
    for (i = 0; i < a; i++)
        for (j = 0; j < b; j++)
            printf("%2d%c", m[i][j], j + 1 == b ? '\n' : ' ');
    printf("\n");
}

int main() {
    int a=-1, b=-1;
    scanf("%d %d", &a, &b);
    if (a < 0 || b < 0){ // No hace falta comprobar un máximo aquí.
    printf("Dimensiones incorrectas\n");
        return 1;
    }

    int (*m)[a][b] = malloc(sizeof(*m)); // Equivale a sizeof(int)*a*b.
    if (m == NULL) {
        printf("Error reservando memoria\n");
        return 2;
    }

    init_random(a, b, *m);
    print_matrix(a, b, *m);

    free(m);

    return 0;

}
```

Este sí es, a mi juicio, **un escenario en que los VLA son ventajosos**, pues nos permite utilizar el operador subíndice (corchete) y la reserva es sencilla (una única llamada a `malloc` y la comprobación de error correspondiente) Puede compararse con otros esquemas, como veremos a continuación.

3.1 Definiendo un tamaño máximo

El compilador necesita conocer el tamaño de cada dimensión excepto de la primera. Si trabajamos con matrices cuyo número de columnas en memoria es `COL`:

```
void f(int a, int b, int m[][COL]);
```

Tiene la desventaja de que depende de COL, obligando a desperdiciar memoria en cada array.

3.2 Puntero a memoria contigua

Podemos utilizar un vector como si fuera una matriz

```
void f(int a, int b, int *m);
```

La reserva sería sencilla, como la de un vector unidimensional, pero nos obliga a usar operaciones aritméticas para encontrar la posición:

- La posición `i`, `j` está en `m[i*b+j]`.
- Al índice `k` le corresponde una posición `k/b`, `k%b`.

En más dimensiones esto se complica.

3.3 Usando arrays de punteros a filas

Una última opción son las representaciones mediante un array de punteros a puntero, creando las filas con memoria dinámica. Esta estructura de datos, que en algunos sitición denominan vector de Iliffe, la usamos frecuentemente en asignaturas de Programación del Grado.

```
void init_random(int a, int b, int **m);
```

El acceso se puede hacer mediante la notación corchete, pero la reserva y liberación se complica:

```
// Reserva
size_t i, j;
int **m = malloc(a * sizeof(int *));
if (m == NULL){
    fprintf(stderr, "Error reservando memoria\n");
    return 1;
}

for (i = 0; i < a; i++){
    m[i] = malloc(b * sizeof(int));
    if (m[i] == NULL) {
        fprintf(stderr, "Error reservando memoria\n");
```

```
        // Liberar la memoria que se consiguió reservar
        for (i--; i >= 0; i--)
            free(matrix[i]);

        free(matrix);

        return 1;
    }
}
// [...]

//Liberación
for (i = 0; i < a; i++)
    free(matrix[i]);

free(matrix);
```

4 Conclusiones

Los VLA pueden ser útiles para escribir código con el que **manejar arrays multidimensionales utilizando la notación corchete**, delegando así en el compilador la aritmética subyacente y facilitando la reserva de memoria dinámica, pero **si se utilizan de forma descuidada** pueden dar lugar a **errores** en el manejo de la **memoria**.