Daniel Enrique Acuña Valdivieso AV18036

Size of

Sizeof es un operador muy utilizado en el lenguaje de programación C . Es un operador unario en tiempo de compilación que se puede usar para calcular el tamaño de su operando. El resultado de sizeof es de tipo integral sin signo, que generalmente se denota por size_t. sizeof se puede aplicar a cualquier tipo de datos, incluidos los tipos primitivos, como los tipos enteros y de coma flotante, los tipos de puntero o los tipos de datos compuestos, como Estructura, unión, etc.

Para reservar memoria se debe saber exactamente el número de bytes que ocupa cualquier estructura de datos. Tal y como se ha comentado con anterioridad, una peculiaridad del lenguaje C es que estos tamaños pueden variar de una plataforma a otra. ¿Cómo sabemos, entonces, cuántos bytes reservar para una tabla de, por ejemplo, 10 enteros? El propio lenguaje ofrece la solución a este problema mediante la función sizeof().

La función recibe como único parámetro o el nombre de una variable, o el nombre de un tipo de datos, y devuelve su tamaño en bytes. De esta forma, sizeof(int) devuelve el número de bytes que se utilizan para almacenar un entero. La función se puede utilizar también con tipos de datos estructurados o uniones tal y como se muestra en el siguiente programa (que te recomendamos que te descargues, compiles y ejecutes):

```
#include <stdio.h>
#define NAME_LENGTH 10
#define TABLE_SIZE 100
#define UNITS_NUMBER 10

struct unit
{  /* Define a struct with an internal union */
  int x;
  float y;
  double z;
  short int a;
  long b;
  union
  { /* Union with no name because it is internal to the struct */
    char name[NAME_LENGTH];
    int id;
    short int sid;
  } identifier;
```

```
};

int main(int argc, char *argv[])
{
  int table[TABLE_SIZE];
  struct unit data[UNITS_NUMBER];

  printf("%d\n", sizeof(struct unit)); /* Print size of structure */
  printf("%d\n", sizeof(table));       /* Print size of table of ints */
  printf("%d\n", sizeof(data));        /* Print size of table of structs */

  return 0;
}
```

Con esta función puedes, por tanto, resolver cualquier duda que tengas sobre el tamaño de una estructura de datos. Basta con escribir un pequeño programa que imprima su tamaño con sizeof().

Malloc

C

Gestión de memoria dinámica

Definido en la cabecera <stdlib.h>
void* malloc(size_t tamaño);

Asigna determinados bytes de tamaño de almacenamiento no inicializado.

Si la asignación tiene éxito, devuelve un puntero al byte más bajo (el primero) en el bloque de memoria asignado que está alineado adecuadamente para cualquier tipo de objeto con alineación fundamental.

Si el tamaño es cero, el comportamiento está definido en la implementación (el puntero nulo puede ser devuelto, o algún puntero no nulo puede ser devuelto que no puede ser usado para acceder al almacenamiento, pero tiene que ser pasado a free).

malloc es seguro para hilos: se comporta como si sólo accediera a las posiciones de memoria visibles a través de su argumento, y no a cualquier almacenamiento estático. Una llamada previa a free o realloc que desasigna una región de memoria se sincroniza con una llamada a un malloc que asigna la misma o parte de la misma región de memoria. Esta sincronización se produce después de cualquier acceso a la memoria por parte de la función de desasignación y antes de cualquier acceso a la memoria por parte de malloc. Hay un solo orden total de todas las funciones de asignación y desasignación que operan en cada	(desde C11)

región particular de la memoria.

Parametros

tamaño- número de bytes a asignar

Valor de retorno

En caso de éxito, devuelve el puntero al principio de la nueva memoria asignada. Para evitar una fuga de memoria, el puntero devuelto debe estar desasignado con free() o realloc().

En caso de fallo, devuelve un puntero nulo.

Ejemplo

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

int main(void)
{
    int *p1 = malloc(4*sizeof(int));  // asigna suficiente para un arreglo de 4 int
    int *p2 = malloc(sizeof(int[4])); // lo mismo, nombrando el tipo directamente
    int *p3 = malloc(4*sizeof *p3);   // lo mismo, sin repetir el nombre del tipo

    if(p1) {
        for(int n=0; n<4; ++n) // rellena el arreglo
            p1[n] = n*n;
        for(int n=0; n<4; ++n) // lo imprime fuera
            printf("p1[%d] == %d\n", n, p1[n]);
    }

    free(p1);
    free(p2);
    free(p3);
}
```

Salida:

```
p1[0] == 0
p1[1] == 1
p1[2] == 4
p1[3] == 9
```

Free

Cuando una zona de memoria reservada con **`malloc`** ya no se necesita, puede ser *liberada* mediante la función **`free`**.

```
void free (void* ptr);
```

`ptr` es un puntero de cualquier tipo que apunta a un área de memoria reservada previamente con **`malloc`**.
Si **`ptr`** apunta a una zona de memoria indebida, los efectos pueden ser desastrosos, igual que si se libera dos veces la misma zona.