Portillo Merlos Melvin Ernesto PM18008

`sizeof()`

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:

```
#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

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).

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);
}
```

free

Desasigna el espacio asignado previamente por malloc(), calloc(), aligned_alloc, o realloc().

Si `ptr` es un puntero nulo, la función no hace nada.

El comportamiento es indefinido si el valor de `ptr` no es igual a un valor devuelto antes por {lc| malloc()}}, calloc(), realloc(), o aligned_alloc().

El comportamiento es indefinido si el área de memoria a la que se refiere `ptr` ya ha sido desasignada, es decir, `free()` o realloc() ya ha sido llamada con `ptr` como argumento y ninguna llamada a malloc(), calloc() or realloc() resultó en un puntero igual a `ptr` después.

El comportamiento es indefinido si después de que `free()` retorna, se hace un acceso a través del puntero `ptr` (a menos que otra función de asignación resulte en un valor de puntero igual a `ptr`).

Ejemplo:

```c
#include <stdlib.h>

int main(void)
{
    int *p1 = malloc(10*sizeof *p1);
    free(p1); // cada puntero asignado debe ser liberado

    int *p2 = calloc(10, sizeof *p2);
    int *p3 = realloc(p2, 1000*sizeof *p3);
    if(p3) // p3 no nulo significa que p2 fue liberado por realloc
       free(p3);
    else // p3 nulo significa que p2 no fue liberado
       free(p2);
}
```