Guía 04.

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.

El operador
sizeof () de uso se usa de manera diferente según el tipo de operando.

1. Cuando el operando es un tipo de datos.
Cuando sizeof () se usa con los tipos de datos como int, float, char ... etc., simplemente devuelve la cantidad de memoria asignada a esos tipos de datos.

Veamos ejemplo:

filter_none
editar
play_arrow
brillo_4

```
#include <stdio.h>
int main()
{
    printf("%lu\n", sizeof(char));
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(float));
    printf("%lu", sizeof(double));
    return 0;
}
```

Salida:

```
1

4 4

4 4

8
```

Nota : sizeof () puede dar una salida diferente según la máquina, hemos ejecutado nuestro programa en el compilador gcc de 32 bits.

2. Cuando el operando es una expresión. Cuando sizeof () se usa con la expresión, devuelve el tamaño de la expresión. Veamos ejemplo:

filter_none
editar
play_arrow
brillo_4

```
#include <stdio.h>
int main()
{
    int a = 0;
    double d = 10.21;
    printf("%lu", sizeof(a + d));
    return 0;
}
```

Salida:

```
8
```

Como sabemos por el primer caso, el tamaño de int y double es 4 y 8 respectivamente, a es una variable int, mientras que d es una variable doble. el resultado final será doble, por lo tanto, la salida de nuestro programa es de 8 bytes.

Necesidad de Sizeof.

1. Para encontrar el número de elementos en una matriz. Sizeof se puede usar para calcular el número de elementos de la matriz automáticamente. Veamos el ejemplo:

filter_none
editar
play_arrow
brillo_4

```
#include <stdio.h>
int main()
{
    int arr[] = { 1, 2, 3, 4, 7, 98, 0, 12, 35, 99, 14 };
    printf("Number of elements :%lu", sizeof(arr) /
sizeof(arr[0]));
    return 0;
```

```
}
```

Salida:

```
Número de elementos: 11
```

2. Para asignar un bloque de memoria de forma dinámica . sizeof se usa mucho en la asignación de memoria dinámica. Por ejemplo, si queremos asignar memoria para la cual es suficiente para contener 10 enteros y no sabemos el tamaño de (int) en esa máquina en particular. Podemos asignar con la ayuda de sizeof.

filter_none

brillo_4

```
int* ptr = (int*)malloc(10 * sizeof(int));
```

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).

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 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.

Free

Definido en la cabecera <stdlib.h>
void free(void* ptr);

Desasigna el espacio asignado previamente por malloc(), calloc(), aligned_alloc, (desde C11) 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() (desde C11).
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)

free es seguro para los 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 a free que desasigna una región de memoria *se sincroniza con* una llamada a cualquier función de asignación posterior que asigne 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 la función de asignación. Hay un solo orden total de todas las funciones de asignación y desasignación que operan en cada región particular de la memoria.	(desde C11)

Parametros
ptr - puntero a la memoria para desasignar
Valor de retorno
(ninguno)

Observaciones
La función acepta (y no hace nada con) el puntero nulo para reducir la cantidad de casos especiales. Tanto si la asignación tiene éxito como si no, el puntero devuelto por una función de asignación puede pasarse a free().