WIKIPEDIA

Malloc

La **asignación dinámica de memoria en el Lenguaje de programación C**, conocida también como **malloc** (abreviatura del inglés ***memory allocation***), se realiza a través de un grupo de funciones en la biblioteca estándar de C , es decir, `malloc` , `realloc` , `calloc` y `free` . En C++, se incluyen estas funciones por retrocompatibilidad, pero han sido sustituidas en gran parte por los operadores `new` y `new[]`.

Están disponibles muchas implementaciones diferentes del mecanismo de asignación de memoria real, utilizado por `malloc` . Su rendimiento varía tanto en tiempo de ejecución y memoria requerida.

Índice

Fundamento

El lenguaje de programación C gestiona la memoria de forma estática, automática o dinámica. Las variables estáticas se asignan en la memoria principal, por lo general junto con el código ejecutable del programa, y persisten durante toda la vida del programa; las variables automáticas se asignan sobre la pila (stack), comienzan cuando se invocan las funciones y acaban cuando se llama a `return`. Para las variables estáticas y automáticas se requiere que el tamaño de la asignación sea constante en tiempo de compilación (antes de C99 , que permite "arrays" automáticos de longitud variable). Si el tamaño requerido no se conoce hasta el tiempo de ejecución (por ejemplo, si los datos de tamaño arbitrario se están leyendo del usuario o desde un archivo de disco), la utilización de objetos de datos de tamaño fijo es insuficiente.

La vida útil de la memoria asignada es también una preocupación. Ni la memoria estática ni automática es adecuada para todas las situaciones. Los datos automáticos asignados no persisten en varias llamadas de función, mientras que los datos estáticos persisten durante toda la vida del programa, sean o no necesarios. En muchas situaciones, el programador requiere una mayor flexibilidad en la gestión de la vida útil de la memoria asignada.

Estas limitaciones se evitan mediante el uso de la gestión de memoria en la que la memoria es más explícitamente (y más flexiblemente) manejada, típicamente mediante la asignación desde el montón (heap), un área de memoria estructurada para este fin. En C, la función `malloc`, perteneciente a la cabecera `stdlib.h`, se utiliza para asignar un bloque de memoria en el montón. El programa accede a este bloque de memoria a través de un puntero que `malloc` regresa. Cuando ya no se necesita la memoria, se pasa el puntero a la función `free`, la cual libera la memoria de modo que se puede utilizar para otros fines.

Algunas plataformas ofrecen llamadas de biblioteca que permiten en tiempo de ejecución la asignación dinámica de la pila C en lugar de la pila (por ejemplo Unix alloca(), Microsoft Windows de CRTL malloca()). Esta memoria se libera automáticamente cuando la función de llamada termina. La necesidad de este se ve reducida por los cambios en el estándar C99, que añade soporte para arrays de longitud variable de ámbito de bloque que tienen tamaños que determine en tiempo de ejecución.

Resumen de las funciones

Las funciones de asignación de memoria dinámica en C se definen en la cabecera `stdlib.h`. (`cstdlib` en C++)

Función	Descripción
`malloc`	asigna el número especificado de bytes
`realloc`	aumenta o disminuye el tamaño del bloque de memoria especificada. Reasigna si es necesario
`calloc`	asigna el número especificado de bytes y los inicializa a cero(0).
`free`	libera el bloque de memoria especificada de nuevo al sistema

Las diferencias entre malloc() y calloc()

Hay dos diferencias entre estas funciones. En primer lugar, `malloc()` toma un único argumento (la cantidad de memoria para asignar en bytes), mientras calloc() necesita dos argumentos (el número de variables para asignar en la memoria y el tamaño en bytes de una sola variable).

En segundo lugar, `malloc()` no inicializa la memoria asignada, mientras `calloc()` inicializa todos los bytes del bloque de memoria asignada a cero.

Ejemplo de uso

Creación de un vector (array) de 10 enteros con el alcance automático es muy sencillo:

```
int array[10];
```

Sin embargo, el tamaño de la matriz se fija en tiempo de compilación. Si se quiere asignar una gama similar de forma dinámica, se puede utilizar el siguiente código:

```
/* Asignar espacio para una matriz con diez elementos de tipo int. Algunos programadores
incluyen el conversor explícito  "(int *)"

antes del malloc aunque no sea necesario. */
int * array = malloc(10 * sizeof(int));
```

```c
/* Comprueba que la memoria se asignó correctamente, en caso contrario se gestiona el error.
*/
if (NULL == array) {
    /* gestión del error en la asignación… */
}

/* Si llegamos a este punto significa que la memoria ha sido asignada correctamente… */

/* Una vez hayamos finalizado el uso de la memoria debemos liberar la misma para usos
futuros. */

free(array);

/* Nos aseguramos que el puntero ya no se usa asignándolo a NULL (u otra región de memoria
asignada). */
array = NULL;
```

malloc() devuelve un puntero nulo (NULL) para indicar que no hay memoria disponible, o que se ha producido algún otro error que impidió la asignación de la memoria.

Tipo de seguridad

`malloc` devuelve un puntero nulo (void *), lo que indica que es un puntero a una región de tipo de datos desconocido. Se requiere el uso del `casting` en C + + debido al fuerte sistema de tipos, mientras que éste no es el caso de C. La falta de un tipo de puntero específico de retorno de `malloc` es un comportamiento de tipo-inseguro de acuerdo a algunos programadores: la asignación de memoria de `malloc` se basa en el recuento de bytes pero no en el tipo de dato. Esto es distinto a la asignación `new` de C++ que devuelve un puntero basado en el tipo de operando.

Uno puede usar un "cast" (ver conversión de tipos) de este puntero a un tipo específico:

```c
int * ptr;
ptr = malloc (10 * sizeof (* ptr)); /* sin una conversión */
ptr = (int *) malloc (10 * sizeof(int)*ptr)); /* con un cast */
ptr = reinterpret_cast <int *> (malloc (10 * sizeof (* ptr))); /* con un cast, para C++ */
```

Hay ventajas y desventajas para la realización del cast.

Ventajas del cast

- Incluyendo el cast permite un programa o función compilar como C++.
- El cast permite versiones anteriores a 1989 de `malloc` que originalmente devuelven un `char *`.
- `cast` puede ayudar al desarrollador a identificar incongruencias entre las llamadas y las asignaciones de datos, especialmente si el puntero se declara muy lejos de la llamada a `malloc()`.

Desventajas del cast

- Bajo el estándar ANSI C, el `cast` es redundante.
- La adición de `cast` puede enmascarar la no inclusión de la cabecera stdlib.h , en el que el prototipo de `malloc` se encuentra. En ausencia de un prototipo para `malloc`, la norma requiere que el compilador de C asume `malloc` devuelve un `int` . Si no hay un `cast`, se

emite un warning cuando se asigna este entero en el puntero, sin embargo, con el `cast`, esta advertencia no se produce, ocultando un error. En ciertas arquitecturas y modelos de datos (por ejemplo, LP64 en sistemas de 64 bits, en los que long y los punteros son de 64 bits e int es de 32 bits), este error de hecho puede dar lugar a un comportamiento indefinido, ya que el declarado implícitamente `malloc` devuelve un valor de 32 - bits, mientras que la función de realidad definida devuelve un valor de 64 bits. Dependiendo de convenciones de llamada y distribución de la memoria, esto puede dar lugar a aplastamiento pila. Este problema es menos probable que pase inadvertido en los compiladores modernos, ya que producen de manera uniforme las advertencias de que una función no declarada se ha utilizado, por lo que seguirá apareciendo una advertencia. Por ejemplo, el comportamiento por defecto para GCC es mostrar una advertencia que dice "declaración implícita incompatible de la función incorporada" independientemente que el `cast` esté presente o no.
- Si el tipo del puntero se cambia, hay que arreglar todas las líneas de código donde `malloc` fue llamado y había echado (a menos que fue arrojado a un `typedef`).

Errores comunes

El uso inadecuado de asignación de memoria dinámica con frecuencia puede ser una fuente de errores.

La mayoría de los errores comunes son los siguientes:

- **No comprobar errores de asignación.** La asignación de memoria no está garantizada que tenga éxito. Si no se comprueba la asignación de memoria, puede producirse un error del programa o de la totalidad del sistema.
- **Las pérdidas de memoria.** Si no se desasigna memoria usando `free` conduce a la acumulación de la memoria de un solo uso, que ya no es utilizado por el programa. Estos recursos de memoria desperdiciados pueden conducir a errores de asignación cuando se hayan agotado esos recursos.
- **Los errores lógicos.** Todas las asignaciones deben seguir el mismo patrón: la asignación mediante malloc , el uso para almacenar datos, desasignación mediante free . Las fallas que se adhieran a este patrón, como el uso de la memoria después de una llamada a la `free` o antes de una llamada a `malloc` , llamada a `free` dos veces seguidas ("doble liberación"), etc, por lo general conducen a una caída del programa.

Límites de tamaño Allocation

El mayor bloque de memoria posible `malloc` asignable depende del sistema, en particular del tamaño de la memoria física y la implementación del sistema operativo. Teóricamente, el número más grande debe ser el valor máximo que se puede mantener en un tipo `size_t`, que es un entero que depende del tamaño de un área de memoria asignado. El valor máximo es de $2^{\text{CHAR\_BIT} \times \text{sizeof(size\_t)}} - 1$ o la constante `SIZE_MAX` en el estándar C99.

Enlaces externos

Inglés

- Wikilibros alberga un libro o manual sobre **C dynamic memory management**.

- Definition of malloc in IEEE Std 1003.1 standard (http://www.opengroup.org/onlinepubs/9699919799/functions/malloc.html)
- Lea, Doug; *The design of the basis of the glibc allocator* (http://gee.cs.oswego.edu/dl/html/malloc.html)
- Gloger, Wolfram; *The ptmalloc homepage* (http://www.malloc.de/en/index.html)
- Berger, Emery; *The Hoard homepage* (http://www.hoard.org)
- Douglas, Niall; *The nedmalloc homepage* (http://www.nedprod.com/programs/portable/nedmalloc/)
- Evans, Jason; *The jemalloc homepage* (http://www.canonware.com/jemalloc/) Archivado (https://web.archive.org/web/20100418045630/http://www.canonware.com/jemalloc/) el 18 de abril de 2010 en Wayback Machine.
- *Simple Memory Allocation Algorithms* (https://web.archive.org/web/20060409094110/http://www.osdcom.info/content/view/31/39/) on OSDEV Community
- Berger, Emery; *Hoard: A Scalable Memory Allocator for Multithreaded Applications* (http://www.cs.umass.edu/~emery/pubs/berger-asplos2000.pdf)
- Michael, Maged M.; *Scalable Lock-Free Dynamic Memory Allocation* (http://www.research.ibm.com/people/m/michael/pldi-2004.pdf)
- Bartlett, Jonathan; *Inside memory management* - The choices, tradeoffs, and implementations of dynamic allocation (http://www-106.ibm.com/developerworks/linux/library/l-memory/)
- Memory Reduction (GNOME) (https://web.archive.org/web/20050823083743/http://live.gnome.org/MemoryReduction) wiki page with lots of information about fixing malloc
- C99 standard draft, including TC1/TC2/TC3 (http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf)
- Some useful references about C (http://paste.tclers.tk/1596)
- ISO/IEC 9899 – Programming languages – C (http://www.open-std.org/jtc1/sc22/wg14/www/standards)
