# malloc和free

## 前言

在C语言中完成动态内存分配和释放功能的是标准库函数malloc和free。malloc和free并不是关键字，只是C标准库中提供的普通函数。

## malloc

### malloc的定义

函数原型：

void \*malloc(size\_t size);

功能：malloc函数用于在堆内存中动态分配指定大小的连续内存空间。它的作用是在程序运行时根据需要申请一定量的内存，以便存储数据或对象等。

参数：size参数表示要分配的内存空间的大小，以字节为单位。

返回值：如果分配成功，malloc函数返回一个指向所分配内存空间起始地址的指针，该指针的类型为void\*，可以根据需要将其转换为其他数据类型的指针。如果分配失败，函数返回NULL指针，通常是因为系统没有足够的可用内存来满足请求。

以下是VS2012中malloc的源码，我们可见其如何实现的。

void \* \_\_cdecl \_malloc\_base (size\_t size)

{

void \*res = NULL;

// validate size

if (size <= \_HEAP\_MAXREQ) {

for (;;) {

// allocate memory block

res = \_heap\_alloc(size);

// if successful allocation, return pointer to memory

// if new handling turned off altogether, return NULL

if (res != NULL)

{

break;

}

if (\_newmode == 0)

{

errno = ENOMEM;

break;

}

// call installed new handler

if (!\_callnewh(size))

break;

// new handler was successful -- try to allocate again

}

} else {

\_callnewh(size);

errno = ENOMEM;

return NULL;

}

RTCCALLBACK(\_RTC\_Allocate\_hook, (res, size, 0));

if (res == NULL)

{

errno = ENOMEM;

}

return res;

}

它的返回值是，如果分配成功则返回指向被分配内存的指针(此存储区中的初始值不确定)，否则返回空指针NULL。当内存不再使用时，应使用free()函数将内存块释放。

### malloc的使用：

//malloc的使用

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

int main()

{

int\* str = 0;

int\* p = 0;

str = (int\*)malloc(10\*sizeof(int));//开辟十个整型空间

if (NULL == str)

{

printf("%s\n", strerror(errno));//若开辟失败

//使用报错函数strerror（errno） 要引用头文件<string.h>

}

else

{

p = str;

}

free(p);

p = NULL;

return 0;

}

## free

### free的定义

函数原型：

void free(void \*ptr);

功能：free函数用于释放先前通过malloc、calloc或realloc等函数分配的动态内存空间，将其归还给系统，以便其他部分的程序可以再次使用该内存。

参数：ptr是一个指向要释放的内存空间的指针，该指针必须是先前通过动态内存分配函数返回的指针。

返回值：free函数没有返回值。

接下来我们看看free的源代码

void \_\_cdecl \_free\_base (void \* pBlock)

{

int retval = 0;

if (pBlock == NULL)

return;

RTCCALLBACK(\_RTC\_Free\_hook, (pBlock, 0));

retval = HeapFree(\_crtheap, 0, pBlock);

if (retval == 0)

{

errno = \_get\_errno\_from\_oserr(GetLastError());

}

}

## 动态内存分配使用注意事项

### malloc

1. 使用sizeof计算数据类型的长度，有助于提高程序的可移植性。
2. 使用前要检查所请求的内存是否成功分配。
3. 操作内存时不要超出分配边界。
4. C++不允许将 一个void\*类型指针赋给任何其它指针。因此在C++中使用malloc时，必须要做类型转换。
5. 连续调用内存分配函数分配的存储空间的顺序和邻接是不确定的。
6. C内存分配函数分配的对象有分配存储期限。这些对象的生存期并不限定于创建它的范围内，因此在任何一个函数内调用malloc()，函数返回后，已分配的内存依然存在。
7. 因为动态内存分配可能会失败，因此对每次返回的指针都进行检查确保它并非NULL非常重要。

### free

1. 合法指针传入：传递给free的指针必须源自malloc、calloc或realloc函数。向free传入非动态分配内存的指针，如未初始化、已释放或非动态分配函数返回的指针，可能引发未定义行为，像内存泄漏、程序崩溃等，可能致使程序立即或稍后终止。对NULL指针调用free，不会执行任何操作。
2. 完整内存块释放：只能整块释放动态分配的内存，尝试释放其中一部分会导致程序立即或稍后终止。不过，realloc可缩小动态分配内存，相当于释放尾部部分内存。
3. 避免访问已释放内存：切勿访问已被free释放的内存。若将指向动态分配内存的指针复制并分散于程序各处，使用任一指针前，务必确认其指向的内存未被其他指针释放，否则会引发错误或未定义行为。
4. 野指针处理：执行free(p)后，内存归还系统，但指针变量p的地址值不变，此时p成为野指针。为防止误操作，建议释放后立即将p赋值为NULL（即p = NULL），明确p不再指向有效内存区域。
5. 确保传递给free的指针是有效的，并且是通过动态内存分配函数获得的指针。如果传递了一个无效的指针，例如未初始化的指针、已经释放过的指针或者不是由动态内存分配函数返回的指针，可能会导致程序出现未定义行为，如内存泄漏、程序崩溃等。
6. 禁止重复释放：不要对同一指针多次调用free函数，除非重新分配了该指针指向的内存。多次释放同一内存空间会造成严重错误。

### realloc

1. 对于realloc不要将返回结果再赋值给原指针，即ptr=realloc(ptr,new\_size)是不建议使用的，因为如果内存分配失败，ptr会变为NULL，如果之前没有将ptr所在地址赋给其他值的话，会发生无法访问旧内存空间的情况，所以建议使用temp=realloc(ptr,new\_size)。
2. realloc的size小于原来的内存块大小时，会造成数据丢失。size大于原来的内存大小时，可能会重新分配内存，因此原来的指针ptr指向的旧内存会被free掉，因此不可以再使用ptr取原来内存中的数据，应该改用realloc返回的新指针。
3. 传递给realloc的指针必须是先前通过malloc(), calloc(), 或realloc()分配的,或者为NULL

### calloc

调用calloc()时，要确保参数相乘时不会超过类型大小而变为负数。（最好室友unsigned int或者size\_t类型。

## 常见的动态内存错误

### 对空指针的解引用操作

使用malloc函数开辟一个很大的空间，大小为INT\_MAX，此时会有一个空指针，进行判断，如果为空指针就立马结束这个程序。（ps:这里的INT\_MAX的使用要引用头文件limits.h）

例：

//错误写法

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <limits.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

int main()

{

int i = 0;

int\* p = (int\*)malloc(INT\_MAX);

for (i = 0; i < 5; i++)

{

\*(p + i) = i;

}

return 0;

}

//正确写法：

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <limits.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

int main()

{

int i = 0;

int\* p = (int\*)malloc(INT\_MAX);

if (p == NULL)

{

printf("%s\n", strerror(errno));//这里是将错误报出来

return 0;//发现是空指针，提前结束

}

for (i = 0; i < 5; i++)

{

\*(p + i) = i;

}

return 0;

}

### 对动态开辟空间的越界访问

不可以不申请就使用动态内存空间，会报错的。

Tips：没有开辟的空间是不能使用的

### 对非动态开辟内存使用free释放

int main()

{

int p=0;

int\* a=&p;

free(a);//这个样子是错误的

return 0;

}

### 使用free释放一块动态开辟内存的一部分

开辟动态空间的时候，一定要把起始位置用变量存好，否则到时会无法释放内存。

//使用free释放一块动态开辟内存的一部分

//错误写法：

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <stdlib.h>

int main()

{

int i = 0;

int\* p = (int\*)malloc(10 \* sizeof(int));

//错误写法：

for (i = 0; i < 5; i++)

{

\*p = i;

p++;//这里会改变p的原始位置，使得无法指向一开始开辟动态内存空间的位置，最终报错

}

free(p);

p = NULL;

return 0;

}

//正确写法：

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <stdlib.h>

int main()

{

int i = 0;

int\* p = (int\*)malloc(10 \* sizeof(int));

//正确写法：

for (i = 0; i < 5; i++)

{

\*(p + i) = i;

}

free(p);

p=NULL;

return 0;

}

### 对同一块动态内存多次释放

一块空间释放后不可再释放，但释放完后p置为空指针再次释放时不会报错。

### 动态开辟内存忘记释放（内存泄露）

即使在函数中开辟内存空间也要记得释放。因为出了函数在外面想释放也无法释放。

但如果返回首元素的地址，free了也行，就是无论怎么样，一定要释放。

在任何地方开辟的内存空间都最好要释放。

## 找出下面问题：

### T1：

void GetMemory(char\* p)

{

p=(char\*)malloc(100);

}

void Test(void)

{

char\* str=NULL;

GetMemory(str);

strcpy(str,"hello world");

printf(str);

}

int main()

{

Test();

return 0;

}

出现的问题：

在这里str是空指针，而p只是新建的一个形参，运行完函数后无法返回，p不存在了，但是内存空间还未被释放，而这个空间的地址此时是没有人能够知道的。也并不能将str里面的NULL改变，所以在strcpy时会出错，因为str此时为NULL指针，会造成非法访问内存，程序会崩溃。

而且在使用过程中只进行了动态内存的开辟，没有进行动态内存的释放，可能会造成动态内存泄露。

改进方法：

#define \_CRT\_SECURE\_NO\_WARNINGS 1

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char\* GetMemory(char\* p)

{

p = (char\*)malloc(100);

return p;

}

void Test(void)

{

char\* str = NULL;

str = GetMemory(str);

strcpy(str, "hello world");

printf(str);

free(str);

str = NULL;

}

int main()

{

Test();

return 0;

}

函数的栈帧与创建：p尽管销毁，因为会先把p里面的值放入到寄存器中，寄存器里面不会销毁，之后再从寄存器位置传进去str。

### T2：

char\* GetMemory(void)

{

char p[] = "hello world";

return p;

}

void Test(void)

{

char\* str = NULL;

str = GetMemory();

printf(str);

}

int main()

{

Test();

return 0;

}

出现的问题：

返回栈空间地址问题：

这里虽然能把p的地址传回去，但是在函数运行完后在函数里面创建的数据会被销毁，也就是说虽然能通过指针找到原来的内存所指向的地方，但是数据都以被销毁。

int test()

{

int a = 10;

return a;

}

int main()

{

int m = test();

printf("%d\n", m);

return 0;

}

这样是可以的，因为返回的是栈空间的变量而不是栈空间的地址。

总结：

在创造函数如果返回地址而不是返回值，在用的时候可能依然是在函数内的值，但也有很大可能不是，可能不是的原因是有关函数栈帧方面，如果在引用地址前再写上一段例如："printf("23333\n");"，可能会导致覆盖掉原来地址上的数据，所以无法通过传址来输出真正的值，因为会被覆盖掉。

### T3：

void GetMemory(char\*\* p, int num)

{

\*p = (char\*)malloc(num);

}

void Test(void)

{

char\* str = NULL;

GetMemory(&str, 100);

strcpy(str, "hello");

printf(str);

}

int main()

{

Test();

return 0;

}

出现的问题：

除了缺失free没有太大毛病了。这里能够打印出hello。

### T4：

void Test(void)

{

char\* str = (char\*)malloc(100);

strcpy(str, "hello");

free(str);

if (str != NULL)

{

strcpy(str, "world");

printf(str);

}

}

int main()

{

Test();

return 0;

}

出现的问题：

这里的free其实是把动态内存空间还给系统了，但是str的话没有定为空指针，仍然存着当初指向开辟的内存空间的地址，那么就还可以通过str找到当初开辟的内存空间，只是这个时候因为释放（free）str了，所以此时没有访问空间的权限，也就无法将world拷贝到str所指向的空间。

正确改法：

所以，在每次free后面都要记得设置为空指针。

## 总结

malloc和free是 C 语言中进行动态内存管理的重要工具，通过合理地使用它们，可以根据程序的实际需求灵活地分配和释放内存，提高内存的使用效率，实现更强大和灵活的程序功能。但同时，也需要谨慎使用，以避免出现内存管理方面的错误。

以上就是今天要讲的内容，后续会有更多内容。

## 参考资料

版权声明：本文参考了其他资料和CSDN博主的文章，遵循CC 4.0 BY-SA版权协议，现附上原文出处链接及本声明。

1. [https://blog.csdn.net/zxx910509/article/details/62881131](https://blog.csdn.net/zxx910509/article/details/62881131" \t "https://blog.csdn.net/zxx910509/article/details/_blank)
2. [https://blog.csdn.net/Green\_756/article/details/123595906](https://blog.csdn.net/Green_756/article/details/123595906" \t "https://blog.csdn.net/Green_756/article/details/_blank)