



# 第11章 动态数据结构的C语言实现

## ——常见的内存错误及其解决对策



哈尔滨工业大学

苏小红

sxh@hit.edu.cn

# 常见的内存错误及其对策

## ■ 分类

- \* 1) 内存分配未成功，却使用了它
- \* 2) 内存分配成功，但尚未初始化就引用它
- \* 3) 内存分配成功，且已初始化，但操作越界
- \* 4) 释放了内存，却继续使用它（野指针，也称悬空指针）
- \* 5) 没有释放内存，造成内存泄漏
- \* 6) 重复释放同一块内存

## ■ 特点

- \* 编译器不能自动发现这类错误，通常在程序运行时才能捕捉到
- \* 时隐时现，无明显症状

# 常见的内存错误-1

- 内存分配未成功，却使用了它
- 起因
  - \* 没有意识到内存分配会不成功，新手易犯
- 解决对策
  - \* 在使用内存前检查指针是否为空指针（NULL）

```
p = (int *)malloc(n*sizeof(int));  
if (p == NULL)  
{  
    printf("No enough memory!\n");  
    exit(1);  
}
```

```
void Fun(void)  
{  
    p = (int *)malloc(n*sizeof(int));  
    if (p == NULL) return;  
    .....  
}
```

# 常见的内存错误-1

- 内存分配未成功，却使用了它
- 起因
  - \* 没有意识到内存分配会不成功，新手易犯
- 解决对策
  - \* 在使用内存前检查指针是否为空指针（NULL）

```
p = (int *)malloc(n*sizeof(int));  
if (p == NULL)  
{  
    printf("No enough memory!\n");  
    exit(1);  
}
```

```
void Fun(int *p)  
{  
    assert(p != NULL);  
    .....  
}
```

# 常见的内存错误-2

- 内存分配成功，但是尚未初始化就将其作为右值使用
- 起因
  - \* 没有初始化的观念，误以为内存的默认值全为0
- 解决对策
  - \* 即使是赋0值也不可省略，不要嫌麻烦
  - \* 对用`malloc()`动态分配的内存，最好用函数`memset()`进行清零操作
  - \* 对于指针变量，即使后面有对其进行赋初值的语句，也最好是在定义时就将其初始化为`NULL`

# 常见的内存错误-3

- **内存分配成功，并且已经初始化，但操作越界**
  - \* 例如，使用数组时常发生下标“多1”或“少1”的操作
  - \* 再如，`p = (int *)malloc(n*2);`
- **解决对策：**
  - \* 用循环语句遍历n个元素的数组时，注意下标从0开始，到n-1结束
  - \* 计算动态分配内存的字节数的时候，始终使用sizeof运算符
  - \* 使用strcpy()、gets()以及memcpy()等函数时要小心

```
char a[100], b[50];  
memcpy(b, a, sizeof(b));
```

```
char a[100], b[50];  
memcpy(b, a, sizeof(a)); //越界
```

# 常见的内存错误-4

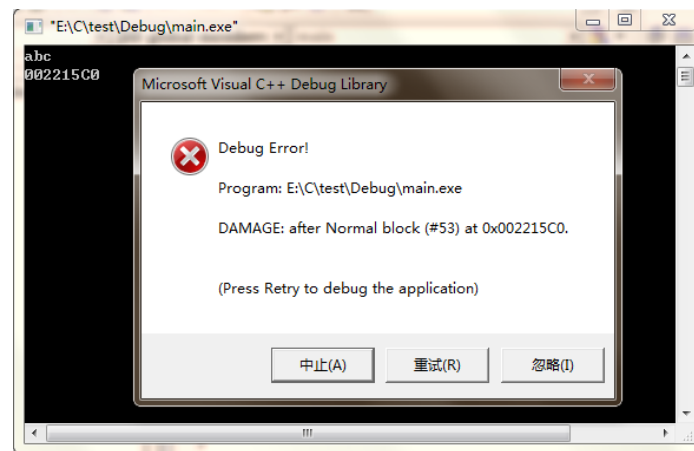
## ■ 释放了内存，却继续使用它

## ■ 起因（1）

- \* 用free释放指针指向的内存以后，没有将指针设置为NULL，导致产生**悬空指针（Dangling Pointer）**也称为**野指针**
  - \* 指向无效内存的指针，不是空指针
  - \* **问题：为什么if语句对野指针不起作用？**

```
...  
free(p);  
...  
if (p != NULL) //不起作用  
{  
    strcpy(p, "def"); //试图修改已释放的内存  
}
```

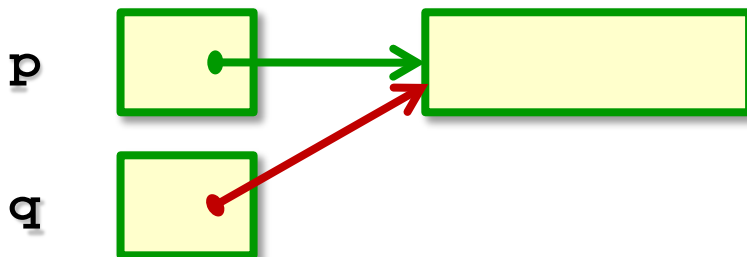
VC6.0



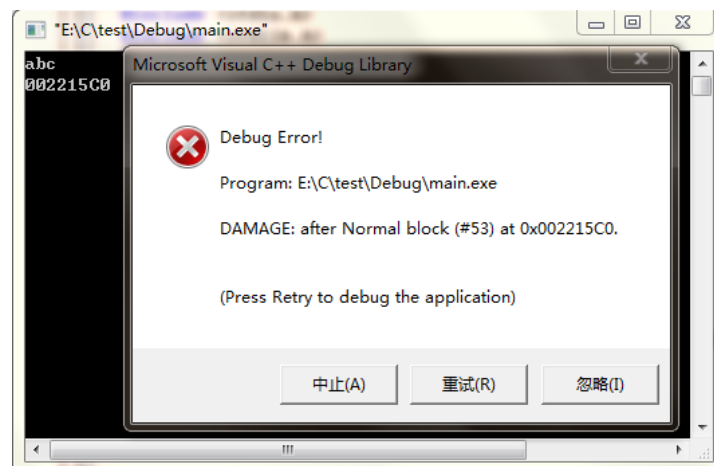
# 常见的内存错误-4

- 一种较为隐蔽的情况：当几个指针指向相同的内存块时

```
char *p, *q;  
p = (char*)malloc(256);  
q = p; //当几个指针指向相同的内存块时  
...  
free(p);  
if (q != NULL) //不起作用  
{  
    strcpy(q, "def"); //试图修改已释放的内存  
}
```



VC6.0





# 常见的内存错误-4

- 释放了内存，却继续使用它
- 解决对策：
  - \* 尽量把free集中在函数的出口处
  - \* 若不能，则指针free后立即将其置为NULL

```
...  
  
p = NULL;  
if (p != NULL) //起作用  
{  
    strcpy(p, "def");  
}
```

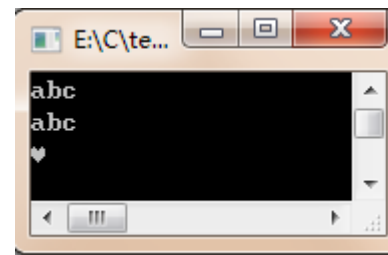
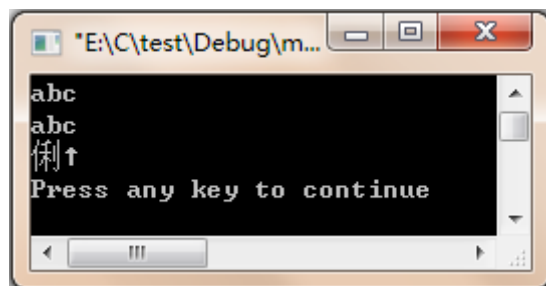
# 常见的内存错误-4

```
#include <stdio.h>
char* GetInput(void);
int main()
{
    char *p = NULL;
    p = GetInput();
    puts(p);
    return 0;
}

char* GetInput(void)
{
    char s[80];
    scanf("%s", s);
    puts(s);
    return s;
}
```

## ■ 起因（2）

- \* 函数的return语句返回了不该返回的内存地址



**解决对策：不要从函数返回局部变量的地址**

warning: function return address of local variable

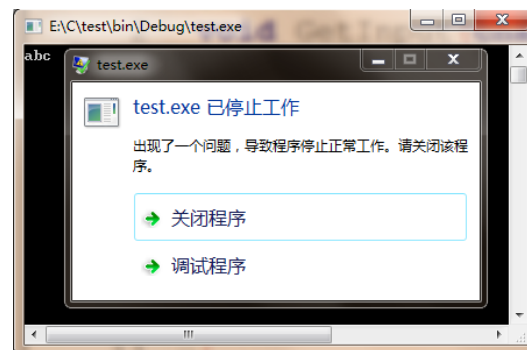
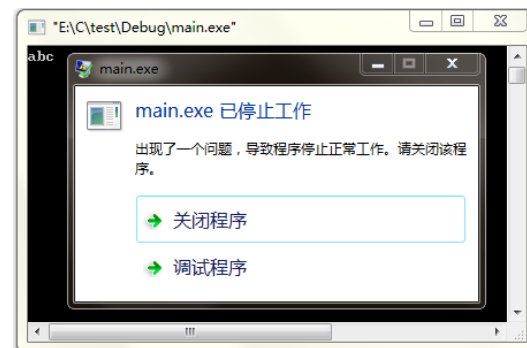
warning C4172: returning address of local variable or temporary

# 常见的内存错误-4

错误原因：向空指针指向的内存写数据

```
#include <stdio.h>
char* GetInput(void);
int main()
{
    char *p = NULL;
    p = GetInput();
    puts(p);
    return 0;
}
char* GetInput(void)
{
    char s[80];
    scanf("%s", s);
    puts(s);
    return s;
}
```

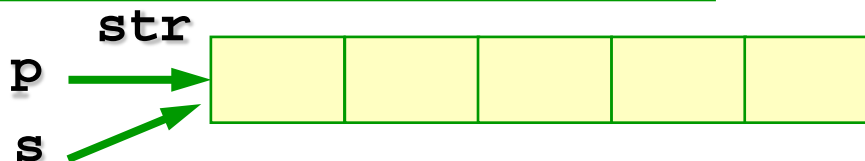
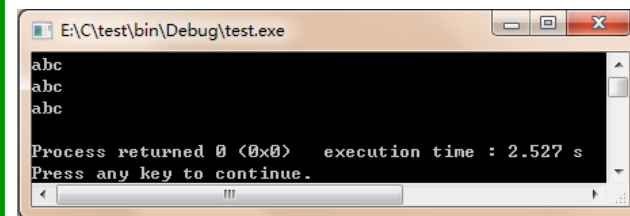
```
#include <stdio.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    puts(p);
    return 0;
}
void GetInput(char *s)
{
    assert(s != NULL);
    scanf("%s", s);
    puts(s);
}
```



```
#include <stdio.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    puts(p);
    return 0;
}
void GetInput(char *s)
{
    scanf("%s", s);
    puts(s);
}
```

```
#include <stdio.h>
void GetInput(char *s);
int main()
{
    char str[80];
    char *p = str;
    GetInput(p);
    puts(p);
    return 0;
}
void GetInput(char *s)
{
    scanf("%s", s);
    puts(s);
}
```

解决对策：指针  
变量使用前一定  
要初始化

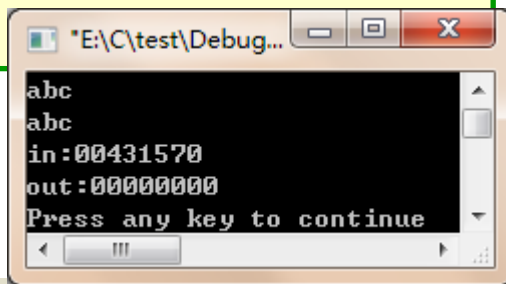


```
#include <stdio.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    puts(p);
    return 0;
}
void GetInput(char *s)
{
    scanf("%s", s);
    puts(s);
}
```

如何从函数返回动态分配的内存的地址？

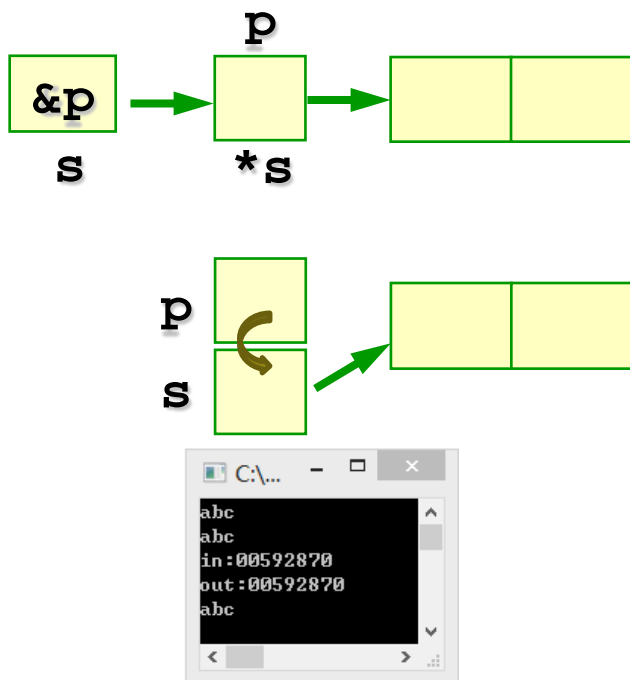
```
#include <stdio.h>
#include <stdlib.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    puts(p);
    return 0;
}
void GetInput(char *s)
{
    s = (char*)malloc(80);
    scanf("%s", s);
    puts(s);
}
```

```
#include <stdio.h>
#include <stdlib.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    printf("out:%p\n", p);
    if (p != NULL)
    {
        puts(p);
    }
    return 0;
}
void GetInput(char *s)
{
    s = (char*)malloc(80);
    scanf("%s", s);
    puts(s);
    printf("in:%p\n", s);
}
```



```
E:\C\test\Debug...
abc
abc
in:00431570
out:00000000
Press any key to continue
```

## 用二级指针做函数参数



## 如何从函数返回动态分配的内存的地址？

```

#include <stdio.h>
#include <stdlib.h>
void GetInput(char **s);
int main()
{
    char *p = NULL;
    GetInput(&p);
    printf("out:%p\n", p);
    if (p != NULL)
    {
        puts(p);
    }
    return 0;
}

void GetInput(char **s)
{
    *s = (char*)malloc(80);
    scanf("%s", *s);
    puts(*s);
    printf("in:%p\n", *s);
}

```

```

#include <stdio.h>
#include <stdlib.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    printf("out:%p\n", p);
    if (p != NULL)
    {
        puts(p);
    }
    return 0;
}

void GetInput(char *s)
{
    s = (char*)malloc(80);
    scanf("%s", s);
    puts(s);
    printf("in:%p\n", s);
}

```

用return返回  
动态分配的内存地址

用return只能返回  
一个动态内存地址

如何从函数返回动态  
分配的内存的地址？

```
#include <stdio.h>
#include <stdlib.h>
char* GetInput(void);
int main()
{
    char *p = NULL;
    p = GetInput();
    if (p != NULL)
    {
        puts(p);
    }
    return 0;
}
char* GetInput(void)
{
    char *s;
    s = (char*)malloc(80);
    scanf("%s", s);
    puts(s);
    return s;
}
```

```
#include <stdio.h>
#include <stdlib.h>
void GetInput(char *s);
int main()
{
    char *p = NULL;
    GetInput(p);
    if (p != NULL)
    {
        puts(p);
    }
    return 0;
}
void GetInput(char *s)
{
    s = (char*)malloc(80);
    scanf("%s", s);
    puts(s);
}
```

# 常见的内存错误-5

- 忘记了释放内存，造成内存泄漏（Memory Leak）
- 特征
  - \* 每被调用一次，就丢失一块内存（制造垃圾）
  - \* 需要运行相当一段时间后才能发现
- 释放内存意味着内存可以被系统再回收利用
  - \* 不能再回收利用的内存——垃圾（Garbage）
- 某些编程语言提供了垃圾收集器（Garbage Collector）
  - \* 但C语言需要由程序员负责回收
  - \* 调用free函数来释放不再需要的动态内存





# 常见的内存错误-5

## ■ 起因（1）忘记回收内存

```
void Init(void)
{
    pszMyName=(char*)malloc(256);
    if (pszMyName == NULL)
    {
        return;
    }

    pszHerName=(char*)malloc(256);
    if (pszHerName == NULL)
    {
        return;
    }
    pszHisName=(char*)malloc(256);
```

```
    if (pszHisName == NULL)
    {
        return;
    }
    ... ..
    free(pszMyName);
    free(pszHerName);
    free(pszHisName);
    return;
}
```

严重程度取决于每次遗留内存垃圾的多少和代码被调用的次数

# 常见的内存错误-5

- 起因（1）忘记回收内存
- 问题：哪些程序对内存泄漏敏感？
  - \* 需长期稳定运行的服务程序
    - 如操作系统、网络服务、导弹防御等应用程序
  - \* 需要频繁对内存操作且消耗空间较大的程序
    - 如图形图像处理程序



# 常见的内存错误-5

- 解决对策：函数返回前检查是否有需要回收的内存

```
void Init(void)
{
    pszMyName=(char*)malloc(256);
    if (pszMyName == NULL)
    {
        return;
    }
    pszHerName=(char*)malloc(256);
    if (pszHerName == NULL)
    {
        free(pszMyName);
        return;
    }
    pszHisName=(char*)malloc(256);
```

```
    if (pszHisName == NULL)
    {
        free(pszMyName);
        free(pszHerName);
        return;
    }

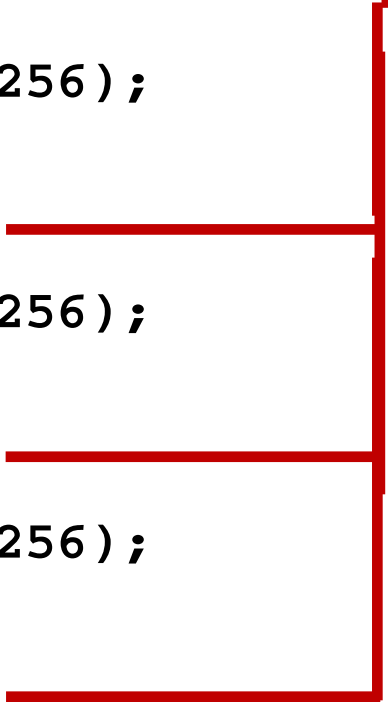
    ... ..

    free(pszMyName);
    free(pszHerName);
    free(pszHisName);
    return;
}
```

# 常见的内存错误-6

- 解决对策：用goto指向函数的出口，统一回收内存

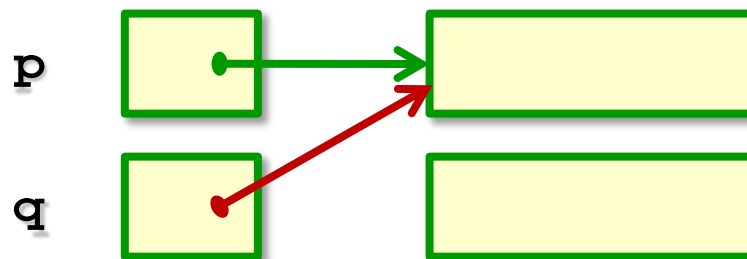
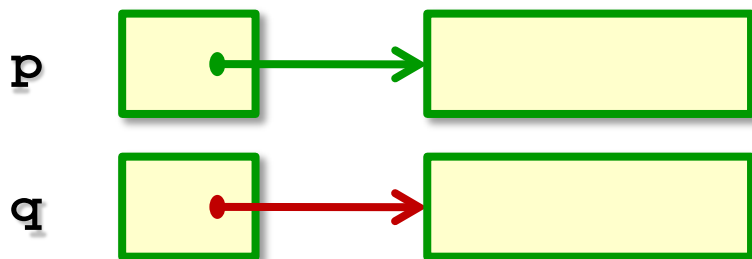
```
void Init(void)
{
    pszMyName=(char*)malloc(256);
    if (pszMyName == NULL)
        goto Exit;
    pszHeName=(char*)malloc(256);
    if (pszHeName == NULL)
        goto Exit;
    pszHiName=(char*)malloc(256);
    if (pszHiName == NULL)
        goto Exit;
    ... ..
Exit:
    if (pszMyName != NULL)
        free(pszMyName);
    if (pszHeName != NULL)
        free(pszHeName);
    if (pszHiName != NULL)
        free(pszHiName);
    return;
}
```

A diagram illustrating the use of goto for memory cleanup. On the left, the Init function contains three malloc calls followed by if checks. Each if check is followed by a 'goto Exit;' statement. Red horizontal lines connect each 'goto Exit;' to a central vertical blue line. From this line, a red arrow points to the 'Exit:' label in the cleanup section on the right. The cleanup section contains three 'free' calls for pszMyName, pszHeName, and pszHiName, followed by a 'return;' statement.

# 常见的内存错误-5

- 起因（2）分配了内存后，又丢失了对这块内存的追踪路径
  - 用realloc函数调整内存块大小时，若调用失败并返回NULL
  - 当调整指针指向的内存块时

```
char *p, *q;  
p = (char*)malloc(256);  
q = (char*)malloc(256);  
q = p; //使得q原来指向的内存块丢失追踪路径
```



# 常见的内存错误-5

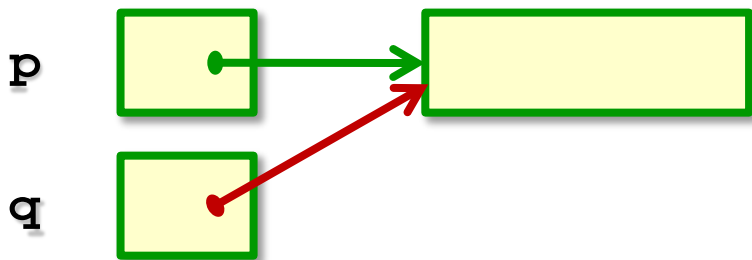
- 起因（3）误以为指针消亡了，它所指向的内存自然会被自动释放
  - \* 内存被释放并不表示指针会消亡或者成为空指针
  - \* 指针的消亡并不表示它所指向的内存会被自动释放
  - \* 问题：程序运行结束后所有内存都被系统回收，是不是不用释放内存了呢？

```
void Fun(void)
{
    char *p = (char*)malloc(256); //动态申请的内存会自动释放吗？
    .....
}
```

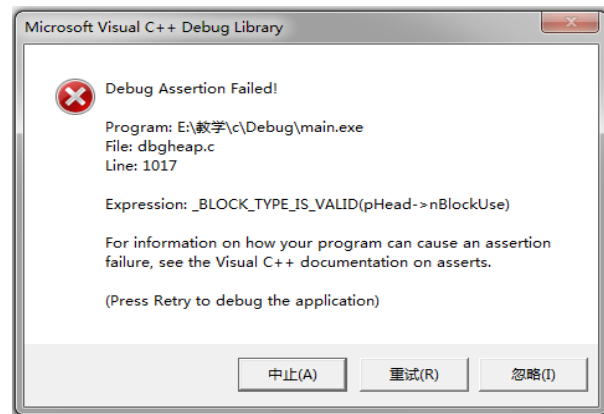
# 常见的内存错误-6

## ■ 矫枉过正，重复释放同一块内存

```
char *p, *q;  
p = (char*)malloc(256);  
q = p;  
...  
free(p);  
free(q); //重复释放了
```



VC6.0



# 小结

## ■ 使用动态内存分配函数时的注意事项

- \* 在需要时才malloc，并尽量减少malloc的次数
  - ∞ malloc的执行效率不高，过多使用使程序性能下降
  - ∞ 能用自动变量解决的问题，就不要用malloc来解决
  - ∞ malloc一般在大块内存分配和动态内存分配时使用
- \* 重复使用malloc申请到的内存
- \* 尽量让malloc和与之配套的free在一个函数或模块内
  - ∞ 尽量把malloc集中在函数入口处，free集中在出口



# 讨论

- 下面程序是否存在错误？如果存在，那么存在什么错误？

```
char *buf[30];
int i;
...
for (i=0; i<30; i++)
{
    buf[i] = (char*)malloc(20*sizeof(char));
    if (buf[i] == NULL)
    {
        printf("No enough memory!\n");
        exit(1);
    }
}
...
for (i=0; i<30; i++)
{
    free(buf[i]);
}
```



