

千锋嵌入式学院C语言培训

-动态分配内存

Author:Richard.zhang



源自清华 值得信赖

动态分配内存的使用场合

- ▶ 需要使用**规模较大**的内存，但是**不能预先确定**所需的容量
 - 规模较大：K、M级别
 - 不能预先确定：只有在运行时刻，才能根据要处理的数据内容确定其容量

静态、动态分配内存的比较

	静态分配	动态分配
适用场合	小内存，可以预先确定容量	大内存，不能以预先确定容量
一般规模	K级以内	K级以上
语法	<code>char buf[512];</code>	<code>char *p = malloc(2000000);</code>
内存位置	全局数据区 – 全局变量 栈 – 局部变量	堆
是否需要释放	不需要	需要
技术要求	简单	相对复杂

malloc函数

```
#include <stdlib.h>
```

```
void * malloc(size_t n);
```

参数：n是要申请的内存容量（字节）

返回值：所分配内存的首地址

例1：分配大数组

分配一个可以容纳1000000个整型数据的内存空间

```
int *array = (int *)malloc(1000000 * sizeof(int));
```

注意：

1. 内存容量 = 元素个数 * 元素容量
2. 为提高可移植性，元素容量使用sizeof获取

例2：大容量结构体数组

分配一个包含N个结构体变量的数组

```
struct test{  
    int a;  
    char b;  
    int c[10];  
};
```

```
struct test * p = (struct test *)malloc(N * sizeof(struct test));
```

free函数

在不再需要使用动态申请的内存时，将其及时释放。

```
#include <stdlib.h>
```

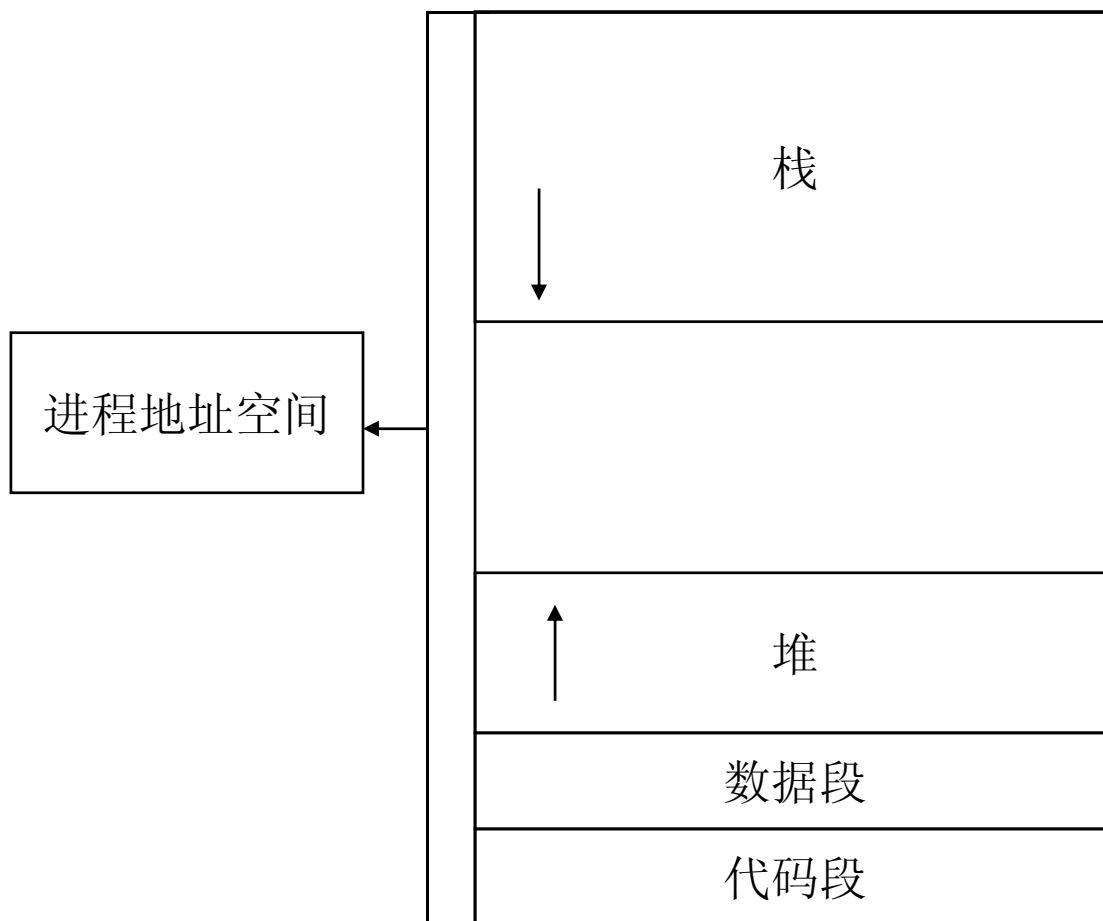
```
void free(void * p);
```

p是要释放的首地址

动态分配的补充说明

- ▶ 内存空间大小是在运行时确定的变量
- ▶ 动态分配的内存空间在未释放之前均可以被引用，其生命期在整个程序运行期间有效

堆和栈



malloc函数的实现机制

- ▶ 在堆中寻找一块空闲空间
- ▶ 分配原则——最先适合分配方法
- ▶ 一个进程使用一个堆，由操作系统管理

free机制总结

并不是真正的释放，只是将内存块标记为可用。

问题1：释放内存后，系统显示的可用内存数会发生改变吗？

问题2：释放的内存还可以引用吗？

非常规使用

(1)当申请0个字节时会出现什么情况

例如:

```
int *p;
```

```
p = (int *)malloc(0);
```

(2)释放一个非动态内存申请的空间

例如:

```
int array[10], *p;
```

```
p = array;
```

```
free(p);
```

两种内存分配的比较

▶ 动态分配内存和非动态分配内存的比较

非动态分配内存	动态分配内存
大小在编译时确定	大小在运行时确定
由编译器分配	由操作系统参与分配
分配在数据段和栈内	在堆内
由操作系统自动释放	手动显式释放

源自清华 值得信赖

memset函数概念

如果需要将一块内存设置为同一个值的时候，需要使用memset函数。

例如：

分配一个缓冲区，将该缓冲区内的值清零

memset函数原形

```
void memset(void *s, int n, size_t size);
```

s: 需要设置内存的首地址

n: 需要被设置的值

size: 需要设置的字节数

memset函数实例

```
#include <stdio.h>
int main()
{
    char s[10];
    memset((void *)s, 'a', 10);
    s[10] = '\0';
    printf("%s\n", s);
    return 0;
}
```

输出结构为：aaaaaaaaaa

综合实例

使用memset函数和malloc函数实现一个calloc函数

memset函数实例

```
#include <stdio.h>
int main()
{
    char s[10];
    memset((void *)s, 'a', 10);
    s[10] = '\0';
    printf("%s\n", s);
    return 0;
}
```

输出结果为：aaaaaaaaaa

memcpy函数概念

当需要在两块内存之间进行数据拷贝的时候需要使用memcpy函数

其原形为：

```
void * memcpy(void *dest, const void * src,  
size_t n);
```

dest: 复制到目的地址

src: 复制的源地址

n: 需要复制的字节数

memcpy函数实例

```
#include <stdio.h>
int main()
{
    char s[] = "hello", d[10];
    memcpy(d, s, 5);
    d[5] = '\0';
    printf("%s", d);
    return 0;
}
```

运行结果: hello

替代函数

```
void bzero(void *s, size_t n);
```

```
void bcopy(void * dest, const void * src, size_t  
n);
```

其它内存块操作的函数

memccpy（拷贝内存内容）

定义函数 `void * memccpy(void *dest, const void * src, int c, size_t n);`

函数说明 memccpy()用来拷贝src所指的内存内容前n个字节到dest所指的地址上。与memcpy()不同的是，memccpy()会在复制时检查参数c是否出现，若是则返回dest中值为c的下一个字节地址。

返回值为0表示在src所指内存前n个字节中没有值为c的字节。

其它内存块操作的函数

memcmp（比较内存内容） 相关函数 bcmp，
定义函数 `int memcmp (const void *s1, const void *s2, size_t n);`

函数说明 memcmp()用来比较s1和s2所指的内存区间前n个字符。字符串大小的比较是以ASCII码表上的顺序来决定，次顺序亦为字符的值。memcmp()首先将s1第一个字符值减去s2第一个字符的值，若差为0则再继续比较下个字符，若差值不为0则将差值返回。例如，字符串"Ac"和"ba"比较则会返回字符'A'(65)和'b'(98)的差值(-33)。返回值 若参数s1和s2所指的内存内容都完全相同则返回0值。s1若大于s2则返回大于0的值。s1若小于s2则返回小于0的值。

其它内存块操作的函数

memmove（拷贝内存内容）

定义函数 `void * memmove(void *dest,const void *src,size_t n);`

函数说明 memmove()与memcpy()一样都是用来拷贝src所指的内存内容前n个字节到dest所指的地址上。不同的是，当src和dest所指的内存区域重叠时，memmove()仍然可以正确的处理，不过执行效率上会比使用memcpy()略慢些。

需要注意的问题

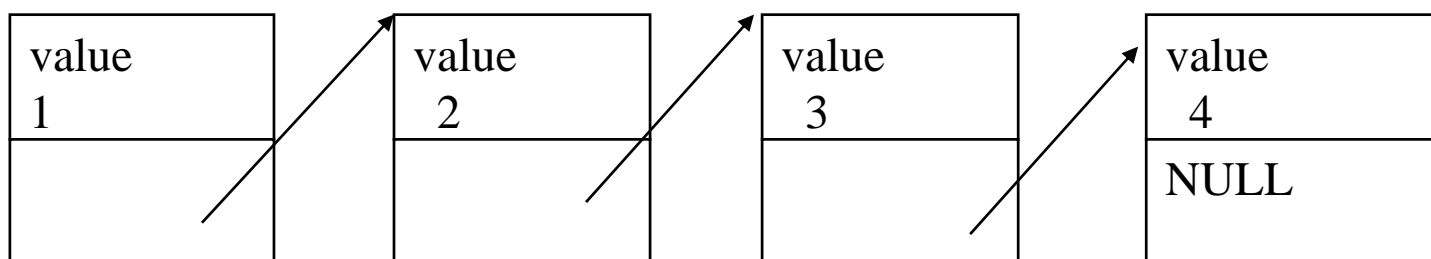
- ▶ 以上函数需要使用的头文件为string.h
- ▶ 进行内存块复制的时候要注意目的地址和源地址交叉的问题

Section 2: 链表

链表的概述

```
struct node
{
    int node; /* 数据域，存储结点的值 */
    struct node * next;
};
```

链表示意图



处理链表使用的函数

- ▶ 动态申请内存: `void * malloc(size_t n);`
- ▶ 释放动态内存: `void free(void *);`

插入一个结点

```
struct node *p = &b;  
a.next = p; /* 连接a结点和b结点 */  
b.next = &c; /* 连接b结点和c结点 */
```

删除一个结点

```
struct node *p = &b;  
a.next = b.next; /* 连接a结点和c结点 */  
free(p); /* 摘下的b结点一定要释放掉 */
```

Section 3: 动态内存分配实例

动态内存分配实例

设计一个学生链表，其每个结点是一个学生信息的集合。每个结点包含如下信息：学生姓名、学号、C语言成绩三项。初始时拥有3个学生，添加一个学生(使用一个函数实现此操作)，再删除一个学生(使用另一个函数实现此操作)，并打印该学生的信息。

实例关键点分析

结点结构:

```
struct info{  
    char name[10];  
    int id;  
    int score;  
};  
struct std{  
    struct info;  
    struct std * next;  
};
```

实例关键点分析

main函数:

```
int main(void)
{
    /* 初始化学生链表 */
    /* 插入一个学生信息结点 */
    /* 删除一个学生的信息，并且打印 */
    return 0;
}
```

实例关键点分析

```
int insert(char * name, int id, int score)
{
    /* 分配一个struct std结构对象 */
    /* 将参数赋值到结构对应的成员中 */
    return 1; /* 正确完成操作，返回1 */
}
```

实例关键点分析

```
int remove(int id,  struct std ** res)
{
    /* 根据id找到该学生的信息结点 */
    /* 将该结点从链表上取下 */
    /* 使用res保存该节点 */
    /* 释放该结点所占用的内存 */
    return 1; /* 成功操作返回1 */
}
```

综合实例

- (1) 实现print函数对其遍历打印链表
- (2) 实现destroy函数释放每一个链表节点
- (3) 实现search函数查找链表中的元素
- (4) 实现一个升级版的insert将元素按顺序插入
- (5) 实现一个升级版的search函数按顺序查找
- (6) 实现get_count函数得到链表元素个数

综合实例

两个扩展函数：

- (1) 实现一个链表排序函数，使用冒泡排序的方法。
- (2) 遍历一个链表，找到链表的中点节点。
- (3) 寻找某一个节点之前的那个节点

类malloc函数

- ▶ calloc函数

`void *calloc(size_t num, size_t size);`

- ▶ realloc函数

`void *realloc(void *mem_address, unsigned
int newsize);`

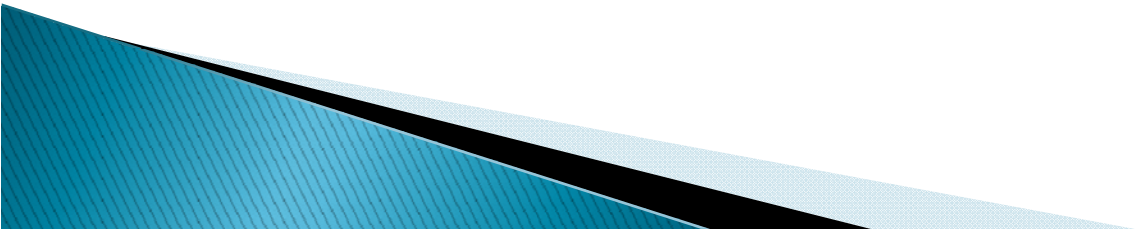
综合实例

实现一个可变的数组，从一个键盘输入若干个数字，以-1结尾。并将其逆序输出。

提示：作为数组的缓冲区的大小是固定的，当读取的数字的数目超过数组大小的时候需要使用realloc函数扩展缓冲区数组。

综合实例

实现一个realloc函数



源自清华 值得信赖

Thank you



源自清华 值得信赖