

第21讲：动态内存管理

目录

1. 为什么要有动态内存分配
2. malloc和free
3. calloc和realloc
4. 常见的动态内存的错误
5. 动态内存经典笔试题分析
6. 柔性数组
7. 总结C/C++中程序内存区域划分

正文开始

1. 为什么要有动态内存分配

我们已经掌握的内存开辟方式有：

```
1 int val = 20; //在栈空间上开辟四个字节
2 char arr[10] = {0}; //在栈空间上开辟10个字节的连续空间
```

但是上述的开辟空间的方式有两个特点：

- 空间开辟大小是固定的。
- 数组在申明的时候，必须指定数组的长度，数组空间一旦确定了大小不能调整

但是对于空间的需求，不仅仅是上述的情况。有时候我们需要的空间大小在程序运行的时候才能知道，那数组的编译时开辟空间的方式就不能满足了。

C语言引入了动态内存开辟，让程序员自己可以申请和释放空间，就比较灵活了。

2. malloc和free

2.1 malloc

C语言提供了一个动态内存开辟的函数：

```
1 void* malloc (size_t size);
```

这个函数向内存申请一块**连续可用**的空间，并返回指向这块空间的指针。

- 如果开辟成功，则返回一个指向开辟好空间的指针。
- 如果开辟失败，则返回一个 `NULL` 指针，因此malloc的返回值一定要做检查。
- 返回值的类型是 `void*`，所以malloc函数并不知道开辟空间的类型，具体在使用的时候使用者自己来决定。
- 如果参数 `size` 为0，malloc的行为是标准是未定义的，取决于编译器。

2.2 free

C语言提供了另外一个函数free，专门是用来做动态内存的释放和回收的，函数原型如下：

```
1 void free (void* ptr);
```

free函数用来释放动态开辟的内存。

- 如果参数 `ptr` 指向的空间不是动态开辟的，那free函数的行为是未定义的。
- 如果参数 `ptr` 是NULL指针，则函数什么事都不做。

malloc和free都声明在 `stdlib.h` 头文件中。

举个例子：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int num = 0;
7     scanf("%d", &num);
8     int arr[num] = {0};
9     int* ptr = NULL;
10    ptr = (int*)malloc(num*sizeof(int));
```

```

11     if(NULL != ptr) //判断ptr指针是否不为空
12     {
13         int i = 0;
14         for(i=0; i<num; i++)
15         {
16             *(ptr+i) = 0;
17         }
18     }
19     free(ptr); //释放ptr所指向的动态内存
20     ptr = NULL; //是否有必要?
21     return 0;
22 }

```

3. calloc和realloc

3.1 calloc

C语言还提供了一个函数叫 `calloc`，`calloc` 函数也用来动态内存分配。原型如下：

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

- 函数的功能是为 `num` 个大小为 `size` 的元素开辟一块空间，并且把空间的每个字节初始化为0。
- 与函数 `malloc` 的区别只在于 `calloc` 会在返回地址之前把申请的空间的每个字节初始化为全0。

举个例子：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p = (int*)calloc(10, sizeof(int));
7     if(NULL != p)
8     {
9         int i = 0;
10        for(i=0; i<10; i++)
11        {
12            printf("%d ", *(p+i));
13        }
14    }
15    free(p);

```

```
16     p = NULL;
17     return 0;
18 }
```

输出结果：

```
1 0 0 0 0 0 0 0 0 0 0
```

所以如果我们对申请的内存空间的内容要求初始化，那么可以很方便的使用calloc函数来完成任务。

3.2 realloc

- realloc函数的出现让动态内存管理更加灵活。
- 有时我们会发现过去申请的空间太小了，有时候我们又会觉得申请的空间过大了，那为了合理的使用内存，我们一定会对内存的大小做灵活的调整。那 `realloc` 函数就可以做到对动态开辟内存大小的调整。

函数原型如下：

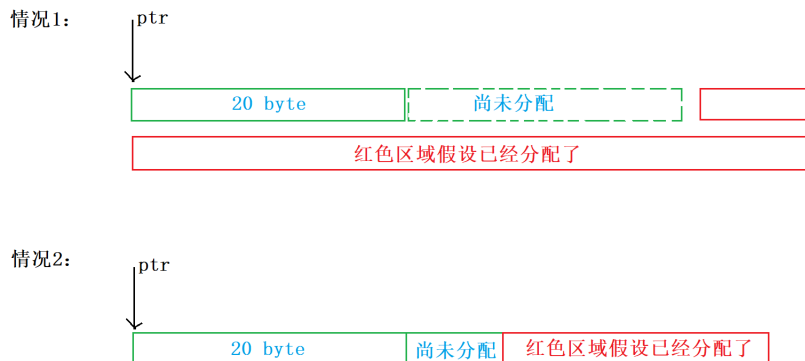
```
1 void* realloc (void* ptr, size_t size);
```

- `ptr` 是要调整的内存地址
- `size` 调整之后新大小
- 返回值为调整之后的内存起始位置。
- 这个函数调整原内存空间大小的基础上，还会将原来内存中的数据移动到 `新` 的空间。
- `realloc`在调整内存空间的是存在两种情况：
 - 情况1：原有空间之后有足够大的空间
 - 情况2：原有空间之后没有足够大的空间

```

int main()
{
    int* ptr = (int*)malloc(20);
    //....
    if (ptr != NULL)
    {
        int* tmp = realloc(ptr, 40);
        //....
    }
    return 0;
}

```



realloc在扩容时的2种常见情况

情况1

当是情况1 的时候，要扩展内存就直接原有内存之后直接追加空间，原来空间的数据不发生变化。

情况2

当是情况2 的时候，原有空间之后没有足够多的空间时，扩展的方法是：在堆空间上另找一个合适大小的连续空间来使用。这样函数返回的是一个新的内存地址。

由于上述的两种情况，realloc函数的使用就要注意一些。

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *ptr = (int*)malloc(100);
7      if(ptr != NULL)
8      {
9          //业务处理
10     }
11     else
12     {
13         return 1;
14     }
15     //扩展容量
16
17     //代码1 - 直接将realloc的返回值放到ptr中
18     ptr = (int*)realloc(ptr, 1000); //这样可以吗? (如果申请失败会如何?)
19
20     //代码2 - 先将realloc函数的返回值放在p中, 不为NULL, 在放ptr中
21     int*p = NULL;
22     p = realloc(ptr, 1000);
23     if(p != NULL)

```

```
24     {
25         ptr = p;
26     }
27     //业务处理
28     free(ptr);
29     return 0;
30 }
```

4. 常见的动态内存的错误

4.1 对NULL指针的解引用操作

```
1 void test()
2 {
3     int *p = (int *)malloc(INT_MAX/4);
4     *p = 20; //如果p的值是NULL，就会有问题
5     free(p);
6 }
```

4.2 对动态开辟空间的越界访问

```
1 void test()
2 {
3     int i = 0;
4     int *p = (int *)malloc(10*sizeof(int));
5     if(NULL == p)
6     {
7         exit(EXIT_FAILURE);
8     }
9     for(i=0; i<=10; i++)
10    {
11        *(p+i) = i; //当i是10的时候越界访问
12    }
13    free(p);
14 }
```

4.3 对非动态开辟内存使用free释放

```
1 void test()
```

```
2  {
3      int a = 10;
4      int *p = &a;
5      free(p); //ok?
6  }
```

4.4 使用free释放一块动态开辟内存的一部分

```
1  void test()
2  {
3      int *p = (int *)malloc(100);
4      p++;
5      free(p); //p不再指向动态内存的起始位置
6  }
```

4.5 对同一块动态内存多次释放

```
1  void test()
2  {
3      int *p = (int *)malloc(100);
4      free(p);
5      free(p); //重复释放
6  }
```

4.6 动态开辟内存忘记释放（内存泄漏）

```
1  void test()
2  {
3      int *p = (int *)malloc(100);
4      if(NULL != p)
5      {
6          *p = 20;
7      }
8  }
9
10 int main()
11 {
12     test();
13     while(1);
14 }
```

忘记释放不再使用的动态开辟的空间会造成内存泄漏。
比特就业课主页: <https://m.cctalk.com/inst/s9yewhfr>

切记: 动态开辟的空间一定要释放, 并且正确释放。

5. 动态内存经典笔试题分析

5.1 题目1:

```
1 void GetMemory(char *p)
2 {
3     p = (char *)malloc(100);
4 }
5 void Test(void)
6 {
7     char *str = NULL;
8     GetMemory(str);
9     strcpy(str, "hello world");
10    printf(str);
11 }
```

请问运行Test 函数会有什么样的结果?

5.2 题目2:

```
1 char *GetMemory(void)
2 {
3     char p[] = "hello world";
4     return p;
5 }
6 void Test(void)
7 {
8     char *str = NULL;
9     str = GetMemory();
10    printf(str);
11 }
```

请问运行Test 函数会有什么样的结果?

5.3 题目3:


```
1 void GetMemory(char **p, int num)
2 {
3     *p = (char *)malloc(num);
4 }
5 void Test(void)
6 {
7     char *str = NULL;
8     GetMemory(&str, 100);
9     strcpy(str, "hello");
10    printf(str);
11 }
```

请问运行Test 函数会有什么样的结果?

5.4 题目4:

```
1 void Test(void)
2 {
3     char *str = (char *) malloc(100);
4     strcpy(str, "hello");
5     free(str);
6     if(str != NULL)
7     {
8         strcpy(str, "world");
9         printf(str);
10    }
11 }
```

请问运行Test 函数会有什么样的结果?

6. 柔性数组

也许你从来没有听说过**柔性数组 (flexible array)** 这个概念,但是它确实是存在的。

C99 中,结构中的最后一个元素允许是未知大小的数组,这就叫做『柔性数组』成员。

例如:

```
1 struct st_type
2 {
3     int i;
```

```
4     int a[0]; //柔性数组成员
5 };
```

有些编译器会报错无法编译可以改成：

```
1 struct st_type
2 {
3     int i;
4     int a[]; //柔性数组成员
5 };
```

6.1 柔性数组的特点：

- 结构中的柔性数组成员前面必须至少一个其他成员。
- sizeof 返回的这种结构大小不包括柔性数组的内存。
- 包含柔性数组成员的结构用 malloc () 函数进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期大小。

例如：

```
1 typedef struct st_type
2 {
3     int i;
4     int a[0]; //柔性数组成员
5 } type_a;
6 int main()
7 {
8     printf("%d\n", sizeof(type_a)); //输出的是4
9     return 0;
10 }
```

6.2 柔性数组的使用

```
1 //代码1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main()
6 {
```

```

7     int i = 0;
8     type_a *p = (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
9     //业务处理
10    p->i = 100;
11    for(i=0; i<100; i++)
12    {
13        p->a[i] = i;
14    }
15    free(p);
16    return 0;
17 }

```

这样柔性数组成员a，相当于获得了100个整型元素的连续空间。

6.3 柔性数组的优势

上述的 `type_a` 结构也可以设计为下面的结构，也能完成同样的效果。

```

1 //代码2
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 typedef struct st_type
6 {
7     int i;
8     int *p_a;
9 }type_a;
10
11 int main()
12 {
13     type_a *p = (type_a *)malloc(sizeof(type_a));
14     p->i = 100;
15     p->p_a = (int *)malloc(p->i*sizeof(int));
16
17     //业务处理
18     for(i=0; i<100; i++)
19     {
20         p->p_a[i] = i;
21     }
22
23     //释放空间
24     free(p->p_a);
25     p->p_a = NULL;
26     free(p);

```

```
27     p = NULL;  
28     return 0;  
29 }
```

上述 代码1 和 代码2 可以完成同样的功能，但是 方法1 的实现有两个好处：

第一个好处是：方便内存释放

如果我们的代码是在一个给别人用的函数中，你在里面做了二次内存分配，并把整个结构体返回给用户。用户调用free可以释放结构体，但是用户并不知道这个结构体内的成员也需要free，所以你不能指望用户来发现这个事。所以，如果我们把结构体的内存以及其成员要的内存一次性分配好了，并返回给用户一个结构体指针，用户做一次free就可以把所有的内存也给释放掉。

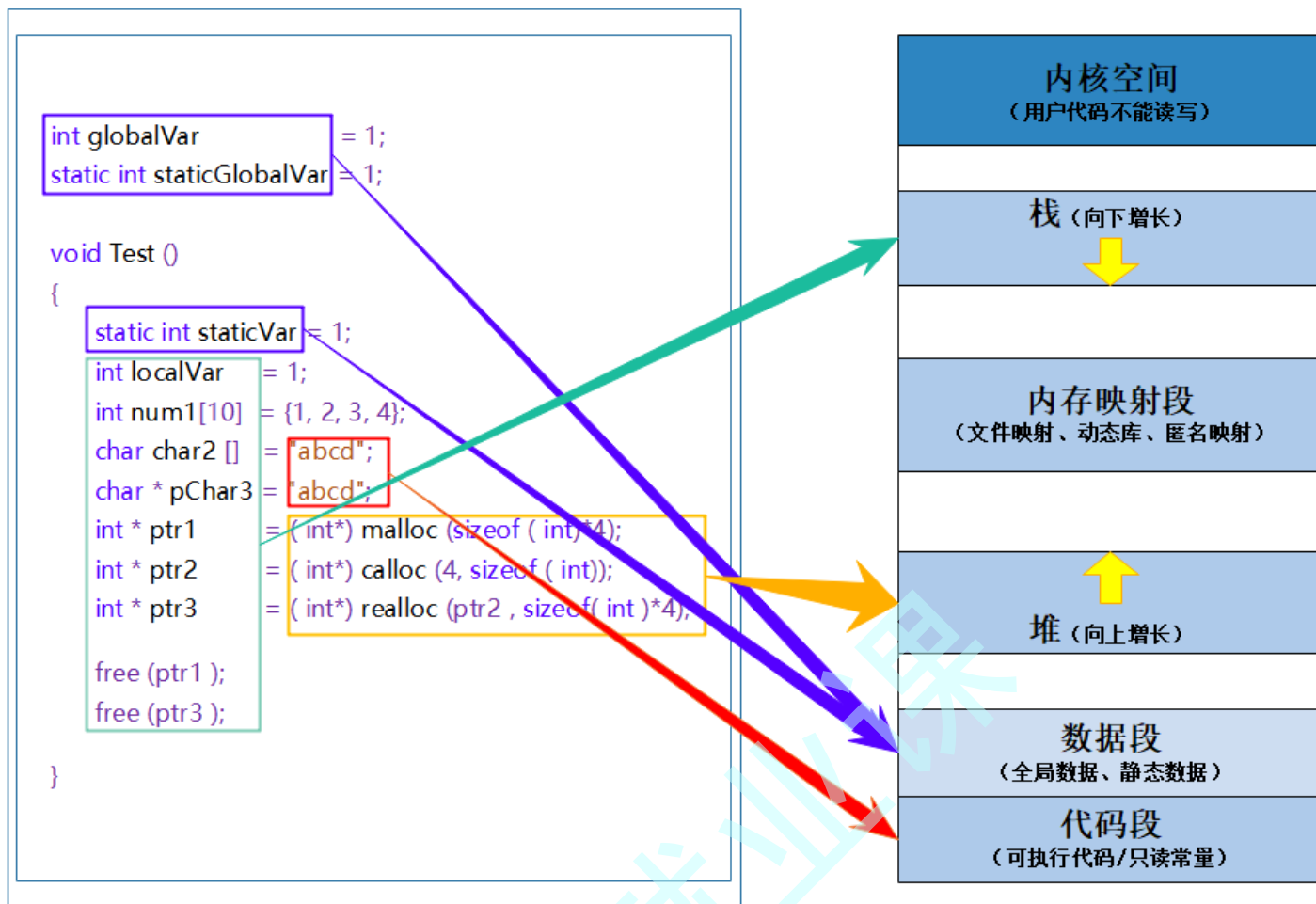
第二个好处是：这样有利于访问速度。

连续的内存有益于提高访问速度，也有益于减少内存碎片。（其实，我个人觉得也没多高了，反正你跑不了要用做偏移量的加法来寻址）

扩展阅读：

[C语言结构体里的数组和指针](#)

7. 总结C/C++中程序内存区域划分



C/C++程序内存分配的几个区域：

1. 栈区 (stack)：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。栈区主要存放运行函数而分配的局部变量、函数参数、返回数据、返回地址等。
《函数栈帧的创建和销毁》
2. 堆区 (heap)：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。分配方式类似于链表。
3. 数据段 (静态区)：(static) 存放全局变量、静态数据。程序结束后由系统释放。
4. 代码段：存放函数体 (类成员函数和全局函数) 的二进制代码。

完