

C++内存管理那些事

Vect. 于 2025-04-24 12:27:37 发布



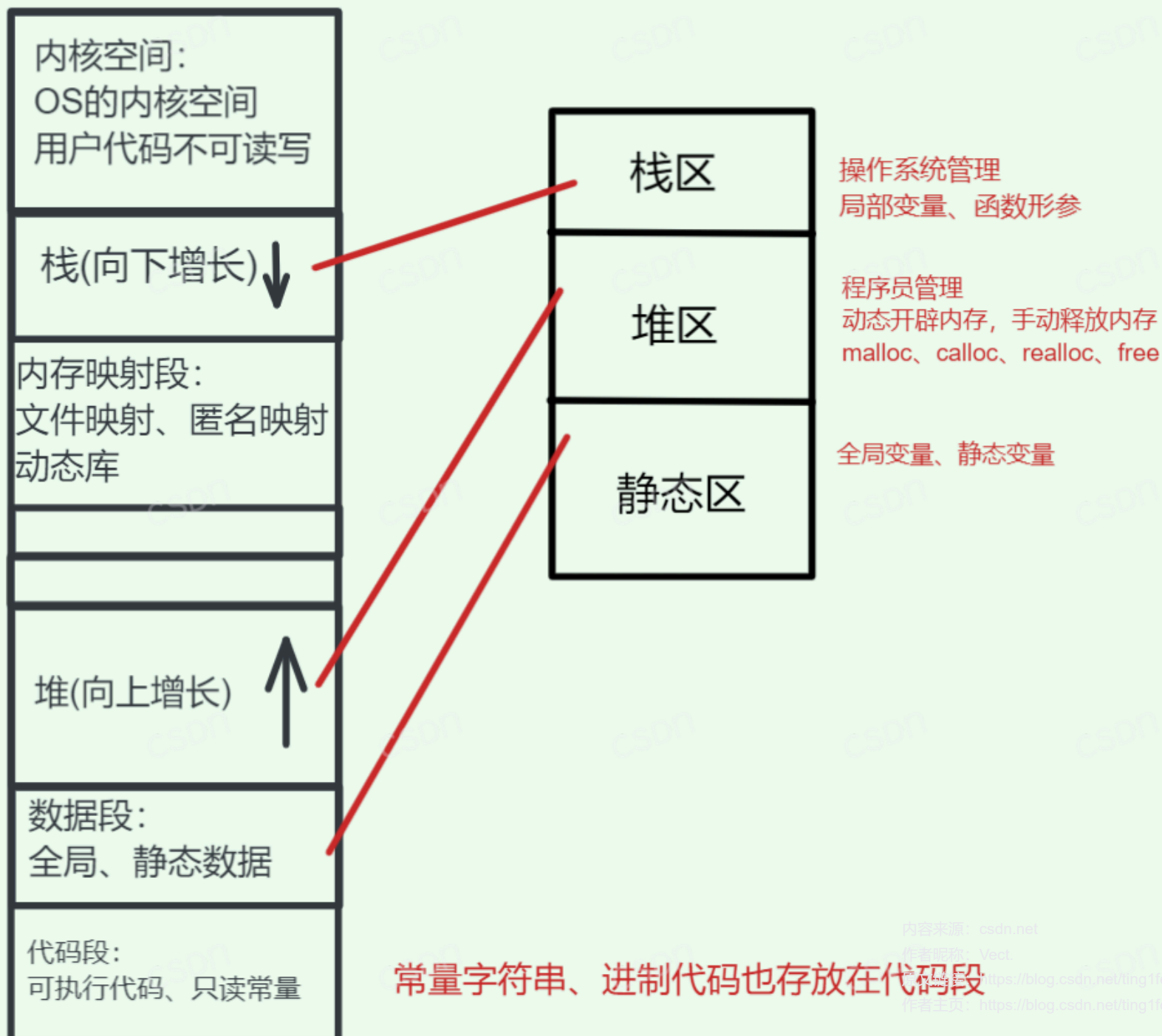
C/C++ 专栏收录该内容

6 篇文章

一、C/C++内存分布

内容来源: [csdn.net](https://blog.csdn.net/ting1fengyu1)
作者昵称: Vect.
原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>
作者主页: <https://blog.csdn.net/ting1fengyu1>

详细内存分布：



内容来源: csdn.net

作者昵称: Vect

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

【说明】：

1. **栈**又叫堆栈，是非静态局部变量、函数参数、返回值存放的区域，**栈**向下增长
2. **内存映射段**是高效的IO映射方式，用于装载一个共享的动态内存库。用户可以使用系统接口创建共享内存，做进程间的通信
3. **堆**是程序运行时动态分配的内存，**堆**是向上增长的
4. **数据段**是存储全局数据和静态数据的，即是我们以前所说的静态区
5. **代码段**存储可执行的代码、只读常量

我们看一段代码，回顾一下：

```
1 static int staticGlobalVar = 1; // 数据段
2 void Test()
3 {
4     static int staticVar = 1; //数据段
5     int localVar = 1; // 栈区
6     int num1[10] = { 1, 2, 3, 4 }; // num1栈区
7     char char2[] = "abcd"; // char2栈区 *char2存储{'a','b','c','d','\0'}这个数组的首地址 在栈区
8     const char* pChar3 = "abcd"; // 栈区 *pChar3存储常量字符"abcd\0"的首地址 在代码区（常量区）
9     int* ptr1 = (int*)malloc(sizeof(int) * 4); // ptr1栈区 *ptr1存储动态开辟的内存首地址 堆区
10    int* ptr2 = (int*)calloc(4, sizeof(int)); // ptr2栈区 *ptr2存储动态开辟的内存首地址 堆区
11    int* ptr3 = (int*)realloc(ptr2, sizeof(int) * 4); // ptr3 栈区 *ptr3存储动态开辟的内存首地址 堆区
12    free(ptr1);
13    free(ptr3);
14 }
15
```

AI写代码

内容来源：csdn.net

作者昵称：Vect

原文链接：https://blog.csdn.net/ting1fengyu1/article/details/147338081

作者主页：https://blog.csdn.net/ting1fengyu1

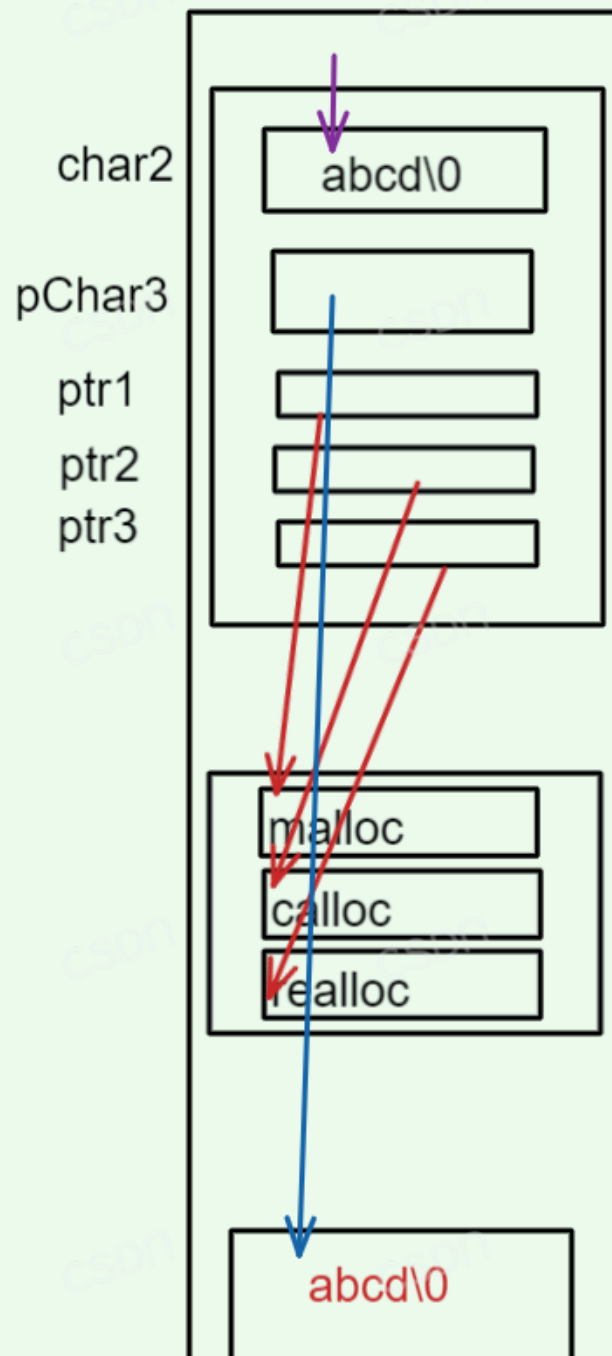
再画一张图理解一下：

内容来源：[csdn.net](https://blog.csdn.net/ting1fengyu1)
作者昵称：Vect.
原文链接：<https://blog.csdn.net/ting1fengyu1/article/details/147338081>
作者主页：<https://blog.csdn.net/ting1fengyu1>

栈区

堆区

代码段



内容来源: csdn.net

作者昵称: Vect.

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

二、C++ 内存管理^Q 方式

C++兼容C的内存管理方式，并且C++提出了新的内存管理方式：**new** 和 **delete** 操作符进行动态内存管理

1. new / delete 操作内置类型

```
1 int main() {  
2     // 动态申请一个int类型的空间  
3     int* p1 = new int;  
4  
5     // 动态申请一个int类型的空间，并初始化为0  
6     int* p2 = new int(0);  
7  
8     // 动态申请十个int类型的空间，并给部分初始化  
9     int* p3 = new int[10] {0, 1, 2, 3};  
10  
11     // 内存清理  
12     delete p1;  
13     delete p2;  
14     delete[] p3;  
15  
16 }
```

AI写代码

内容来源: [csdn.net](https://blog.csdn.net/ting1fengyu1)

作者昵称: Vect.

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

```

int main() {
    // 动态申请一个int类型的空间
    int* p1 = new int;

    // 动态申请一个int类型的空间, 并初始化为0
    int* p2 = new int(0);

    // 动态申请十个int类型的空间, 并给部分初始化
    int* p3 = new int[10] {0, 1, 2, 3};

    // 内存清理
    delete p1;
    delete p2;
    delete[] p3;
}

```

类型

初始化

对象个数

操作符

注意：申请和释放单个元素的空间，使用new和delete操作符，申请和释放连续的空间，使用new[]和delete[]

2. new / delete 操作自定义类型

```

1 class ShowClass {
2 public:
3     // 单参数默认构造
4     ShowClass(int val = 0) {
5         cout << "ShowClass(int val = 0)" << endl;
6     }
7 }

```

内容来源: csdn.net

作者昵称: Vect

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

```

7 // 多参数默认构造
8 //ShowClass(int val1 , int val2 ) {
9 //  cout << "ShowClass(int val1 = 0, int val2 = 1)" << endl;
10 //}
11
12 // 析构
13 ~ShowClass() {
14     cout << "~ShowClass()" << endl;
15 }
16 private:
17     int _val;
18 };
19
20 int main() {
21     // new/delete和malloc/free最大的区别就是前者会调用构造和析构，后者不会
22     ShowClass* p1 = (ShowClass*)malloc(sizeof(ShowClass)); // 不能初始化
23     ShowClass* p2 = new ShowClass(1); // int类型隐式转换成ShowClass类
24     free(p1);
25     delete p2;
26
27     // 内置类型是几乎是一样的
28     int* p3 = (int*)malloc(sizeof(int));
29     int* p4 = new int;
30     free(p3);
31     delete p4;
32
33     ShowClass* p5 = (ShowClass*)malloc(sizeof(ShowClass) * 10);
34     ShowClass* p6 = new ShowClass[10];
35     free(p5);
36     delete[] p6;
37     // 如果是多参数构造呢？
38     ShowClass* p7 = new ShowClass[10]{ 1,2,3,{7,8} };
39
40     delete[] p7;
41
42     return 0;
43 }

```

内容来源：csdn.net

作者昵称：Vect

原文链接：https://blog.csdn.net/ting1fengyu1/article/details/147338081

作者主页：https://blog.csdn.net/ting1fengyu1

输出结果:

```
41
42 int main() {
43     // new/delete和malloc/free最大的区别就是前者会调用构造和析构，后者不会
44     ShowClass* p1 = (ShowClass*)malloc(sizeof(ShowClass)); // 不能初始化
45     ShowClass* p2 = new ShowClass(1); // int类型隐式转换成ShowClass类
46     free(p1);
47     delete p2;
48
49     // 内置类型是几乎是一样的
50     int* p3 = (int*)malloc(sizeof(int));
51     int* p4 = new int;
52     free(p3);
53     delete p4;
54
55     ShowClass* p5 = (ShowClass*)malloc(sizeof(ShowClass) * 10);
56     ShowClass* p6 = new ShowClass[10];
57     free(p5);
58     delete[] p6;
59
60     return 0;
61 }
62
63 //int main() {
64 //    // 动态申请一个int类型的空间
```

Microsoft Visual Studio 调试器

```
ShowClass(int val = 0)
~ShowClass()
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
```

内容来源: csdn.net

作者昵称: Vect

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>作者主页: <https://blog.csdn.net/ting1fengyu1>

```

21
22 class ShowClass {
23     public:
24         // 单参数默认构造
25         ShowClass(int val = 0) {
26             cout << "ShowClass(int val = 0)" << endl;
27         }
28         // 多参数默认构造
29         ShowClass(int val1, int val2) {
30             cout << "ShowClass(int val1 = 0, int val2 = 1)" << endl;
31         }
32
33         // 析构
34         ~ShowClass() {
35             cout << "~ShowClass()" << endl;
36         }
37     private:
38         int _val;
39 };
40

```

这里防止重载歧义，单参数给缺省值，多参数不给缺省值

Microsoft Visual Studio 调试

```

ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val1 = 0, int val2 = 1)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
ShowClass(int val = 0)
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
~ShowClass()
D:\CODE\CPP\MemoryManagent\x64\Debug\MemoryM
按任意键关闭此窗口...

```

// 如果是多参数构造呢?

```

ShowClass* p7 = new ShowClass[10]{ 1,2,3,{7,8} };

delete[] p7;

return 0;

```

注意：在申请自定义类型的空间时，**new** 和 **delete** 会分别调用构造函数和析构函数，而 **malloc** 和 **free** 不会

3. operator new与operator delete函数

new 和 **delete** 是用户进行动态内存申请和释放的操作符
operator new 和 **operator delete** 是系统提供的全局函数

内容来源: csdn.net

作者昵称: VecL

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

new 在底层调用 **operator new** 全局函数来申请空间
delete 在底层通过 **operator delete** 全局函数来释放空间

3.1. 核心职责

- **operator new**

负责从堆中分配**原始内存块**（未初始化的连续字节）。

- 类似 **malloc**，但更智能：失败时默认抛出 **std::bad_alloc** 异常（可通过 **nothrow** 禁用抛异常）
- 示例：**void* p = operator new(100);** 分配 100 字节的原始内存

- **operator delete**

负责释放内存，将其归还给堆管理器。

- 类似 **free**，但专为 C++ 设计
- 示例：**operator delete(p);** 释放 **p** 指向的内存

2. 与 **new/delete** 的关系

- **new** 表达式的底层行为

```
1 ShowClass* p = new ShowClass();
2 // 等价于：
3 void* raw = operator new(sizeof(ShowClass)); // 1. 分配内存
4 p = static_cast<ShowClass*>(raw);
5 p->ShowClass::ShowClass(); // 2. 调用构造函数
AI写代码
```

- **delete** 表达式的底层行为

```
1 delete p;
2 // 等价于：
3 p->~ShowClass(); // 1. 调用析构函数
4 operator delete(p); // 2. 释放内存
```

内容来源：csdn.net

作者昵称：Vect

原文链接：<https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页：<https://blog.csdn.net/ting1fengyu1>

3. 关键区别

特性	<code>operator new/delete</code>	<code>new / delete</code>
内存管理	仅分配/释放原始内存	分配内存 + 构造（析构）对象
异常处理	可抛出 <code>bad_alloc</code>	自动处理构造/析构中的异常
重载用途	优化内存分配策略	通常不直接重载

总结： `operator new/delete` 是内存管理的“搬运工”，仅处理空间存取，而对象构造/析构由其他机制完成。

注意：

1. `malloc realloc calloc` 和 `free` 配套使用

2. `int* p4 = new int` 和 `delete p4` 配套使用

3. `ShowClass* p6 = new ShowClass[10]` 和 `delete[] p6` 配套使用

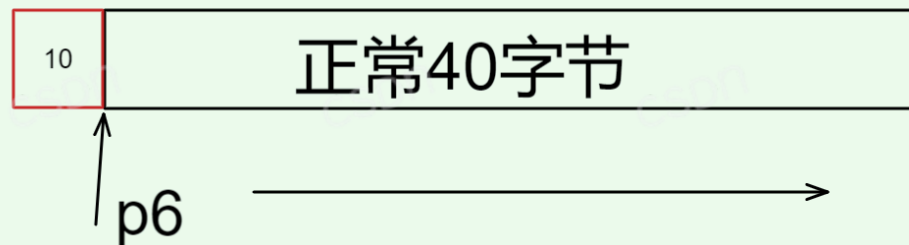
补充：

```
// 析构
~ShowClass() {
    cout << "~ShowClass()" << endl;
}
```

```
private:
    int _val;
```

```
ShowClass* p6 = new ShowClass[10];
free(p5);
delete[] p6;
```

若存在显式的析构函数，p6的大小为44字节，多出来的4字节内存是为了存储p6数组有效元素个数，使用delete[]时会先访问这个个数，之后就知道析构的次数了



若不存在显式的析构函数，p6的大小就为40字节

4. new和delete的实现原理

内容来源: [csdn.net](https://blog.csdn.net/ting1fengyu1/article/details/147338081)

作者昵称: Vect.

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>

4.1. 内置类型

如果申请的是内置类型的空间，`new` 和 `malloc`，`delete` 和 `free` 基本类似，不同的地方是：`new/delete` 申请和释放的是单个元素的空间，`new[]` 和 `delete[]` 申请的是连续空间，而且 `new` 在申请空间失败时会抛异常，`malloc` 会返回 `NULL`。

4.2. 自定义类型

- `new` 的原理

1. 调用 `operator new` 函数申请空间

```
ShowClass* p2 = new ShowClass(1); // int类型隐式转换成ShowClass类
00007FF67B267B35  mov     ecx,4
00007FF67B267B3A  call    operator new (07FF67B26104Bh)
```

2. 在申请的空间上执行构造函数，完成对象的构造

- `delete` 的原理

1. 在空间上执行析构函数，完成对象中资源的清理工作
2. 调用 `operator delete` 函数释放对象的空间

```
delete p4;
00007FF7769B2AA9  mov     rax,qword ptr [p4]
00007FF7769B2AAD  mov     qword ptr [rbp+2E8h],rax
00007FF7769B2AB4  mov     edx,4
00007FF7769B2AB9  mov     rcx,qword ptr [rbp+2E8h]
00007FF7769B2AC0  call    operator delete (07FF7769B141Ah)
```

- `new T[N]` 的原理

内容来源：csdn.net

作者昵称：Vect

原文链接：https://blog.csdn.net/ting1fengyu1/article/details/147338081

作者主页：https://blog.csdn.net/ting1fengyu1

1. 调用 `operator new[]` 函数，在 `operator new[]` 中实际调用 `operator new` 函数完成N个对象空间的申请

```
ShowClass* p6 = new ShowClass[10];
00007FF7769B2B01  mov     ecx, 30h
00007FF7769B2B06  call    operator new[] (07FF7769B1221h)
```

2. 在申请的空间上执行N次构造函数

- **`delete[]` 的原理**

1. 在释放的对象空间上执行N次析构函数，完成N个对象中资源的清理

2. 调用 `operator delete[]` 释放空间，实际在 `operator delete[]` 中调用 `operator delete` 来释放空间

5.面试题: **`malloc/free` 和 `new/delete` 的区别**

`malloc/free` 和 `new/delete` 的共同点是：都是从堆上申请空间，并且需要用户手动释放。

不同的地方是：

- `malloc` 和 `free` 是函数，`new` 和 `delete` 是操作符
- `malloc` 申请的空间不会初始化，`new` 可以初始化
- `malloc` 申请空间时，需要手动计算空间大小并传递，`new` 只需在其后跟上空间的类型即可，如果是多个对象，`[]` 中指定对象个数即可
- `malloc` 的返回值为 `void*`，在使用时必须强转，`new` 不需要，因为 `new` 后跟的是空间的类型
- `malloc` 申请空间失败时，返回的是 `NULL`，因此使用时必须判空，`new` 不需要，但是 `new` 需要捕获异常
- 申请自定义类型对象时，`malloc/free` 只会开辟空间，不会调用构造函数与析构函数，而 `new` 在申请空间后会调用构造函数完成对象的初始化，`delete` 在释放空间前会调用析构函数完成空间资源的清理释放

内容来源: [csdn.net](https://blog.csdn.net/ting1fengyu1)

作者昵称: Vect

原文链接: <https://blog.csdn.net/ting1fengyu1/article/details/147338081>

作者主页: <https://blog.csdn.net/ting1fengyu1>