



C++ 基础

第 8 章：动态内存管理

主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者





目录



1. 动态内存基础



2. 智能指针



3. 动态内存的相关问题



动态内存基础

有链接

• 栈内存 V.S. 堆内存

地址是由高到低生长 函数声明对象

- 栈内存的特点：更好的局部性，对象自动销毁

地址从低到高生长

- 堆内存的特点：运行期动态扩展，需要显式释放

• 在 C++ 中通常使用 new 与 delete 来构造、销毁对象

• 对象的构造分成两步：分配内存与在所分配的内存上构造对象；对象的销毁与之类似

• new 的几种常见形式

- 构造单一对象 / 对象数组

```
int *y = new int{2};
```

```
int* y = new int[5];
int* y = new int[5]{1,2,3,4,5};
delete[] y;
```

- nothrow new

```
#include <new>
int main() {
    int *y = new {std::nothrow} int[5]{};
    //如果失败了，系统不抛出异常，y指向一个nullptr的指针。
    if (y == nullptr) { // ... }
```

- placement new

```
// placement new
// 我现在已经有一块内存了。
// 我不需要你分配内存了，
// 我只需要你在上面构造对象。
// vector就有这个特性。就是
// 使用动态内存来实现动态增
// 长的特性。
// 典型做法：以*2的方式来进
// 行分配。
```

```
// placement new
char ch[sizeof(int)];
// 在栈上开辟了一个内存。
// 是一个char数组。
int* y = new (ch) int{4};
// 不需要在堆上开辟新的内存，
// 我在ch这里提供了一个内存。
// 这里使用ch作为首指针，来吧int
// 构造出来。ch要求：是个指针，
// 且地址是有效足够大的。
```

- new auto

```
// new auto
int *y = new int(3);
int *x = new auto(3);
```

• new 与对象对齐

```
// new与对象对齐
struct alignas(256) Str{};
// 开辟的地址一定得是256的整数倍
Str* ptr = new Str();
```

```
int* fun() {
    int *res = new int{2};
    return res;
}
int main() {
    int *y = fun();
    // 使用new分配了一个堆内存，然后
    // 用return返回回来。因为是堆内存，
    // 所以返回回来还是存在的
```

```
int* fun() {
    int res = 2;
    return &res;
} // 非常危险
int main() {
    int *y = fun();
    //非常危险，指向临时对象的指针
    // 在函数过后，栈内存会被释放。
```

```
int *y = new int{2};
std::cout << *y;
delete y;
```

内存分配不成功的两种情况：

1. 将所有free的空间都占满了
2. 每次分配很大的内存，但是每次都只释放其中一部分的内存。就造成了其中有很多小的可以使用的内存块（如只有8个字节），当我需要一个比较大的内存的时候，就发现无法分配了。这种情况就叫内存片段

如果无法分配，那么系统就会抛出异常。



动态内存基础（续）

- delete 的常见用法
 - 销毁单一对象 / 对象数组
 - placement delete
- 使用 new 与 delete 的注意事项
 - 根据分配的是单一对象还是数组，采用相应的方式销毁

- delete nullptr `int* x = 0; // nullptr`
`delete x; // 如果指针是nullptr, 那么c++什么也不做。`

- 不能 delete 一个非 new 返回的内存

```
// error
int x;
delete &x;
```

```
int* ptr = new int[5];
int* ptr2 = (ptr + 1);
delete[] ptr2; // 不能delete一个非new返回的内存
```

- 同一块内存不能 delete 多次
- 调整系统自身的 new / delete 行为
 - 不要轻易使用 `cppreference operator new`



智能指针

```
void dummy(int*) {}
std::shared_ptr<int> fun() {
    static int res = 3;
    return std::shared_ptr<int>(&res, dummy);}
int main() { std::shared_ptr<int> x = fun();}
通过引用一个自定义的delter来防止static int的值被删除
```

- 使用 new 与 delete 的问题：内存所有权不清晰，容易产生不销毁，多销毁的情况

- C++ 的解决方案：智能指针

- auto_ptr (C++17 删除)
- shared_ptr / unique_ptr / weak_ptr

- shared_ptr—— 基于引用计数的共享内存解决方案

- 基本用法 `std::shared_ptr<int> x(new int(3));`
`std::shared_ptr<int> y = x;`

- reset / get 方法 `std::shared_ptr<int> x = fun();`
`std::cout << *(x.get()) << std::endl;`
`// returns the stored pointer. 提供这个get()函数是为了和没有使用smart pointer的代码兼容`

- 指定内存回收逻辑 `void fun(int* ptr) { delete ptr; }`
`std::shared_ptr<int> x{new int{3}, fun}; //第二个参数就是指定的删除的指针的函数`

- `std::make_shared` `std::shared_ptr<int> x = new int(3);`
`std::shared_ptr<int> x = std::make_shared<int>(3); //和上面的一样，但是更建议使用make_shared`
`auto x = std::make_shared<int>(3); //上面的简洁版`

- 支持数组 (C++17 支持 `shared_ptr<T[]>` ; C++20 支持 `make_shared` 分配数组)

- 注意：shared_ptr 管理的对象不要调用 delete 销毁

```
std::shared_ptr<int> fun() {
    std::shared_ptr<int> res(new int(3));
    return res;
}
int main() {
    std::shared_ptr<int> x = fun();
}
```

```
void fun2(int* x) {
    std::cout << *x << std::endl;}
int main() {
    std::shared_ptr<int> x = fun();
    x.reset(new int(4)); // 原来的智能指针删除，并把新的关联
    fun2(x.get()); // 使得shared_ptr可以在普通指针的地方兼容
    x.reset((int*)nullptr); // reset一个空指针，指向0的指针}
```

reset: replaces
the managed
object



智能指针（续）

- `unique_ptr`—— 独占内存的解决方案
 - 基本用法
 - `unique_ptr` 不支持复制，但可以移动
 - 为 `unique_ptr` 指定内存回收逻辑
- `weak_ptr`—— 防止循环引用而引入的智能指针
 - 基于 `shared_ptr` 构造
 - `lock` 方法



动态内存的相关问题

- sizeof 不会返回动态分配的内存大小
- 使用分配器（ allocator ）来分配内存
- 使用 malloc / free 来管理内存
- 使用 aligned_alloc 来分配对齐内存
- 动态内存与异常安全
- C++ 对于垃圾回收的支持

感谢聆听 !
Thanks for Listening

