

01 | 堆、栈、RAII：C++里该如何管理资源？

2019-11-25 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 16:34 大小 11.39M



你好，我是吴咏炜。

今天我们就正式开启了 C++ 的学习之旅，作为第一讲，我想先带你把地基打牢。我们来学习一下内存管理的基本概念，大致的学习路径是：先讲堆和栈，然后讨论 C++ 的特色功能 RAII。掌握这些概念，是能够熟练运用 C++ 的基础。

基本概念

堆，英文是 heap，在内存管理的语境下，指的是动态分配内存的区域。这个堆跟数据结构里的堆不是一回事。这里的内存，被分配之后需要手工释放，否则，就会造成内存泄漏。

C++ 标准里一个相关概念是自由存储区，英文是 free store，特指使用 new 和 delete 来分配和释放内存的区域。一般而言，这是堆的一个子集：

new 和 delete 操作的区域是 free store

malloc 和 free 操作的区域是 heap

但 new 和 delete 通常底层使用 malloc 和 free 来实现，所以 free store 也是 heap。鉴于对其区分的实际意义并不大，在本专栏里，除非另有特殊说明，我会只使用堆这一术语。

栈，英文是 stack，在内存管理的语境下，指的是函数调用过程中产生的本地变量和调用数据的区域。这个栈和数据结构里的栈高度相似，都满足“后进先出”（last-in-first-out 或 LIFO）。

RAII，完整的英文是 Resource Acquisition Is Initialization，是 C++ 所特有的资源管理方式。有少量其他语言，如 D、Ada 和 Rust 也采纳了 RAII，但主流的编程语言中，C++ 是唯一一个依赖 RAII 来做资源管理的。

RAII 依托栈和析构函数，来对所有的资源——包括堆内存在内——进行管理。对 RAII 的使用，使得 C++ 不需要类似于 Java 那样的垃圾收集方法，也能有效地对内存进行管理。RAII 的存在，也是垃圾收集虽然理论上可以在 C++ 使用，但从来没有真正流行过的主要原因。


接下来，我将会对堆、栈和 RAII 进行深入的探讨。

堆


从现代编程的角度来看，使用堆，或者说使用动态内存分配，是一件再自然不过的事情了。下面这样的代码，都会导致在堆上分配内存（并构造对象）。

```
1 // C++
2 auto ptr = new std::vector<int>();
```

 复制代码

 复制代码

```
1 // Java
2 ArrayList<int> list = new ArrayList<int>();
```

 复制代码

```
1 # Python
2 lst = list()
```

从历史的角度，动态内存分配实际上是较晚出现的。由于动态内存带来的不确定性——内存分配耗时需要多久？失败了怎么办？等等——至今仍有很多场合会禁用动态内存，尤其在实时性要求比较高的场合，如飞行控制器和电信设备。不过，由于大家多半对这种用法比较熟悉，特别是从 C 和 C++ 以外的其他语言开始学习编程的程序员，所以提到内存管理，我们还是先讨论一下使用堆的编程方式。

在堆上分配内存，有些语言可能使用 `new` 这样的关键字，有些语言则是在对象的构造时隐式分配，不需要特殊关键字。不管哪种情况，程序通常需要牵涉到三个可能的内存管理器的操作：

1. 让内存管理器分配一个某个大小的内存块
2. 让内存管理器释放一个之前分配的内存块
3. 让内存管理器进行垃圾收集操作，寻找不再使用的内存块并予以释放

C++ 通常会做上面的操作 1 和 2。Java 会做上面的操作 1 和 3。而 Python 会做上面的操作 1、2、3。这是语言的特性和实现方式决定的。

需要略加说明的是，上面的三个操作都不简单，并且彼此之间是相关的。

第一，分配内存要考虑程序当前已经有多少未分配的内存。内存不足时要从操作系统申请新的内存。内存充足时，要从可用的内存里取出一块合适大小的内存，做簿记工作将其标记为已用，然后将其返回给要求内存的代码。

需要注意到，绝大部分情况下，可用内存都会比要求分配的内存大，所以代码只被允许使用其被分配的内存区域，而剩余的内存区域仍属于未分配状态，可以在后面的分配过程中使

用。另外，如果内存管理器支持垃圾收集的话，分配内存的操作还可能会触发垃圾收集。

第二，释放内存不只是简单地把内存标记为未使用。对于连续未使用的内存块，通常内存管理器需要将其合并成一块，以便可以满足后续的较大内存分配要求。毕竟，目前的编程模式都要求申请的内存块是连续的。

第三，垃圾收集操作有很多不同的策略和实现方式，以实现性能、实时性、额外开销等各方面的平衡。由于 C++ 里通常都不使用垃圾收集，所以就不是我们专栏的重点，不再展开讲解。

下面这张图展示了一个简单的分配过程：



图1a：一个长度为8的内存块



图1b：分配了1单位

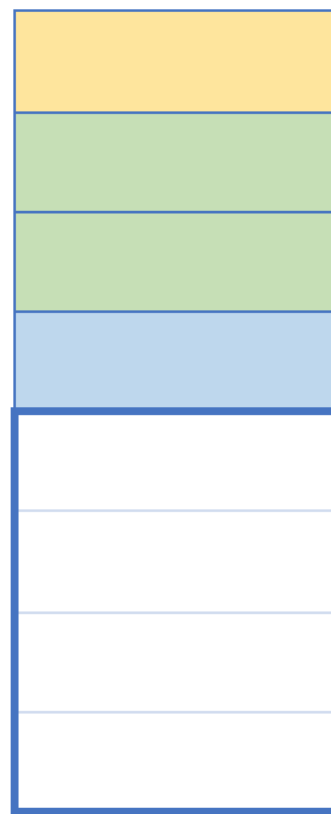


图1c：又分配了2单位和1单位

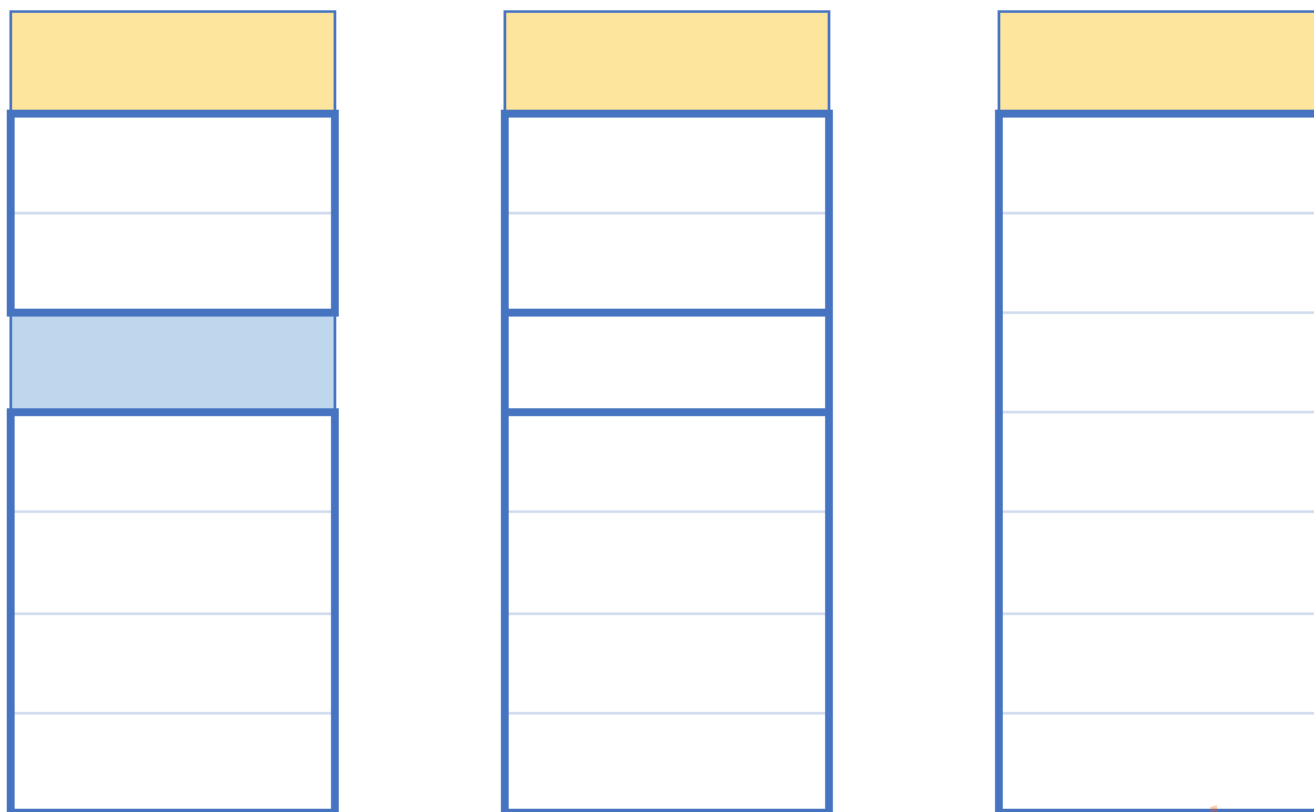


图1d：释放了中间的2单位

图1e：释放了末尾的1单位，未合并

图1f：空闲内存块合并

注意在图 1e 的状态下，内存管理器是满足不了长度大于 4 的内存分配要求的；而在图 1f 的状态，则长度小于等于 7 的单个内存要求都可以得到满足。


当然，这只是一个简单的示意，只是为了让你能够对这个过程有一个大概的感性认识。在不考虑垃圾收集的情况下，内存需要手工释放；在此过程中，内存可能有碎片化的情况。比如，在图 1d 的情况下，虽然总共剩余内存为 6，但却满足不了长度大于 4 的内存分配要求。

幸运的是，大部分软件开发人员都不需要担心这个问题。内存分配和释放的管理，是内存管理器的任务，一般情况下我们不需要介入。我们只需要正确地使用 `new` 和 `delete`。每个 `new` 出来的对象都应该用 `delete` 来释放，就是这么简单。

但真的很简单、可以高枕无忧了吗？

事实说明，漏掉 `delete` 是一种常见的情况，这叫“内存泄漏”——相信你一定听到过这个说法。为什么呢？

我们还是看一些代码例子。

 复制代码

```
1 void foo()
2 {
3     bar* ptr = new bar();
4     ...
5     delete ptr;
6 }
```

这个很简单吧，但是却存在两个问题：

1. 中间省略的代码部分也许会抛出异常，导致最后的 `delete ptr` 得不到执行。
2. 更重要的，这个代码不符合 C++ 的惯用法。在 C++ 里，这种情况下有 99% 的可能性不应该使用堆内存分配，而应使用栈内存分配。这样写代码的，估计可能是从 Java 转过来的（偷笑）——但我真见过这样的代码。

而更常见、也更合理的情况，是分配和释放不在一个函数里。比如下面这段示例代码：

 复制代码


```
1 bar* make_bar(...)
2 {
3     ...
4     try {
5         bar* ptr = new bar();
6         ...
7     }
8     catch (...) {
9         delete ptr;
10        throw;
11    }
12    return ptr;
13 }
14
15 void foo()
16 {
17     ...
18     bar* ptr = make_bar(...)
19     ...
20     delete ptr;
21 }
```


这样的话，会漏 `delete` 的可能性是不是大多了？有关这个问题的解决方法，我们在下一讲还会提到。

好，堆我们暂时就讨论到这儿。下面，我们看看更符合 C++ 特性的栈内存分配。

栈

我们先来看一段示例代码，来说明 C++ 里函数调用、本地变量是如何使用栈的。当然，这一过程取决于计算机的实际架构，具体细节可能有所不同，但原理上都是相通的，都会使用一个后进先出的结构。

 复制代码

```
1 void foo(int n)
2 {
3     ...
4 }
5
6 void bar(int n)
7 {
8     int a = n + 1;
9     foo(a);
10 }
11
12 int main()
13 {
14     ...
15     bar(42);
16     ...
17 }
```

这段代码执行过程中的栈变化，我画了下面这张图来表示：



图2a: 执行 bar 之前

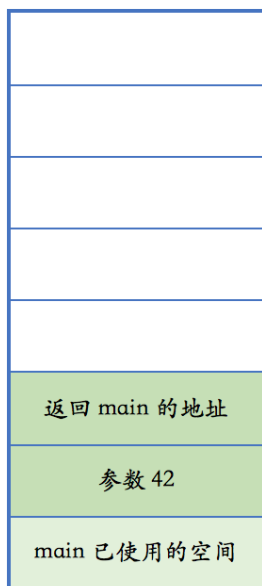


图2b: 调用 bar

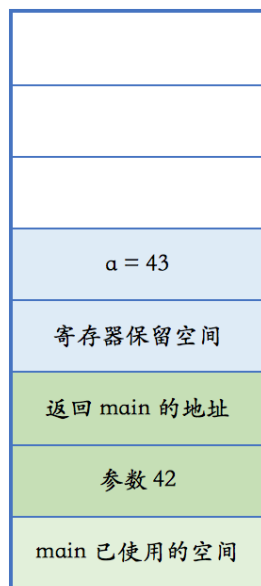


图2c: 执行 foo 之前

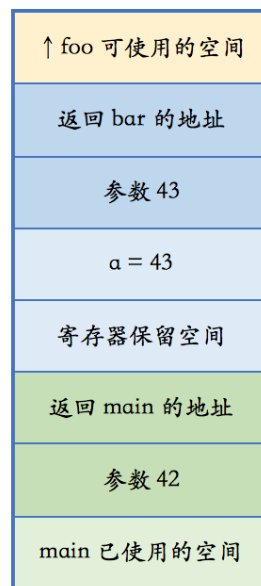


图2d: 调用 foo

在我们的示例中，栈是向上增长的。在包括 x86 在内的大部分计算机体系架构中，栈的增长方向是低地址，因而上方意味着低地址。任何一个函数，根据架构的约定，只能使用进入函数时栈指针向上部分的栈空间。当函数调用另外一个函数时，会把参数也压入栈里（我们此处忽略使用寄存器传递参数的情况），然后把下一行汇编指令的地址压入栈，并跳转到新的函数。新的函数进入后，首先做一些必须的保存工作，然后会调整栈指针，**分配出本地变量所需的空间**，随后执行函数中的代码，并在执行完毕之后，根据调用者压入栈的地址，返回到调用者未执行的代码中继续执行。

注意到了没有，本地变量所需的内存就在栈上，跟函数执行所需的其他数据在一起。当函数执行完成之后，这些内存也就自然而然释放掉了。我们可以看到：

栈上的分配极为简单，移动一下栈指针而已。

栈上的释放也极为简单，函数执行结束时移动一下栈指针即可。

由于后进先出的执行过程，不可能出现内存碎片。

顺便说一句，图 2 中每种颜色都表示某个函数占用的栈空间。这部分空间有个特定的术语，叫做栈帧（stack frame）。GCC 和 Clang 的命令行参数中提到 frame 的，如 `-fomit-frame-pointer`，一般就是指栈帧。

前面例子的本地变量是简单类型，C++ 里称之为 POD 类型（Plain Old Data）。对于有构造和析构函数的非 POD 类型，栈上的内存分配也同样有效，只不过 C++ 编译器会在生

成代码的合适位置，插入对构造和析构函数的调用。

这里尤其重要的是：编译器会自动调用析构函数，包括在函数执行发生异常的情况。在发生异常时对析构函数的调用，还有一个专门的术语，叫栈展开（stack unwinding）。事实上，如果你用 MSVC 编译含异常的 C++ 代码，但没有使用上一讲说过的 `/EHsc` 参数，编译器就会报告：

```
warning C4530: C++ exception handler used, but unwind semantics are not
enabled. Specify /EHsc
```

下面是一段简短的代码，可以演示栈展开：

 复制代码

```
1  #include <stdio.h>
2
3  class Obj {
4  public:
5      Obj() { puts("Obj()"); }
6      ~Obj() { puts("~Obj()"); }
7  };
8
9  void foo(int n)
10 {
11     Obj obj;
12     if (n == 42)
13         throw "life, the universe and everything";
14 }
15
16 int main()
17 {
18     try {
19         foo(41);
20         foo(42);
21     }
22     catch (const char* s) {
23         puts(s);
24     }
25 }
```

执行代码的结果是：

```
1 Obj()
2 ~Obj()
3 Obj()
4 ~Obj()
5 life, the universe and everything
```

也就是说，不管是否发生了异常，`obj` 的析构函数都会得到执行。

在 C++ 里，所有的变量缺省都是值语义——如果不使用 `*` 和 `&` 的话，变量不会像 Java 或 Python 一样引用一个堆上的对象。对于像智能指针这样的类型，你写 `ptr->call()` 和 `ptr.get()`，语法上都是对的，并且 `->` 和 `.` 有着不同的语法作用。而在大部分其他语言里，访问成员只用 `.`，但在作用上实际等价于 C++ 的 `->`。这种值语义和引用语义的区别，是 C++ 的特点，也是它的复杂性的一个来源。要用好 C++，就需要理解它的值语义的特点。

对堆和栈有了基本了解之后，我们继续往下，聊一聊 C++ 的重要特性 RAII。

RAII

C++ 支持将对象存储在栈上面。但是，在很多情况下，对象不能，或不应该，存储在栈上。比如：

对象很大；

对象的大小在编译时不能确定；

对象是函数的返回值，但由于特殊的原因，不应使用对象的值返回。

常见情况之一是，在工厂方法或其他面向对象编程的情况下，返回值类型是基类。下面的例子，是对工厂方法的简单演示：

```
1 enum class shape_type {
2     circle,
3     triangle,
4     rectangle,
5     ...
}
```

```

6  };
7
8  class shape { ... };
9  class circle : public shape { ... };
10 class triangle : public shape { ... };
11 class rectangle : public shape { ... };
12
13 shape* create_shape(shape_type type)
14 {
15     ...
16     switch (type) {
17     case shape_type::circle:
18         return new circle(...);
19     case shape_type::triangle:
20         return new triangle(...);
21     case shape_type::rectangle:
22         return new rectangle(...);
23     ...
24     }
25 }

```

这个 `create_shape` 方法会返回一个 `shape` 对象，对象的实际类型是某个 `shape` 的子类，圆啊，三角形啊，矩形啊，等等。这种情况下，函数的返回值只能是指针或其变体形式。如果返回类型是 `shape`，实际却返回一个 `circle`，编译器不会报错，但结果多半是错的。这种现象叫对象切片（object slicing），是 C++ 特有的一种编码错误。这种错误不是语法错误，而是一个对象复制相关的语义错误，也算是 C++ 的一个陷阱了，大家需要小心这个问题。

那么，我们怎样才能确保，在使用 `create_shape` 的返回值时不会发生内存泄漏呢？

答案就在析构函数和它的栈展开行为上。我们只需要把这个返回值放到一个本地变量里，并确保其析构函数会删除该对象即可。一个简单的实现如下所示：

```

1 class shape_wrapper {
2 public:
3     explicit shape_wrapper(
4         shape* ptr = nullptr)
5         : ptr_(ptr) {}
6     ~shape_wrapper()
7     {
8         delete ptr_;
9     }

```

 复制代码

```

10     shape* get() const { return ptr_; }
11 private:
12     shape* ptr_;
13 };
14
15 void foo()
16 {
17     ...
18     shape_wrapper ptr_wrapper(
19         create_shape(...));
20     ...
21 }

```

如果你好奇 `delete` 空指针会发生什么的话，那答案是，这是一个合法的空操作。在 `new` 一个对象和 `delete` 一个指针时编译器需要干不少活的，它们大致可以如下翻译：

 复制代码

```

1 // new circle(...)
2 {
3     void* temp = operator new(sizeof(circle));
4     try {
5         circle* ptr =
6             static_cast<circle*>(temp);
7         ptr->circle(...);
8         return ptr;
9     }
10    catch (...) {
11        operator delete(ptr);
12        throw;
13    }
14 }

```

 复制代码

```

1 if (ptr != nullptr) {
2     ptr->~shape();
3     operator delete(ptr);
4 }

```

也就是说，`new` 的时候先分配内存（失败时整个操作失败并向外抛出异常，通常是 `bad_alloc`），然后在这个结果指针上构造对象（注意上面示意中的调用构造函数并不是合法的 C++ 代码）；构造成功则 `new` 操作整体完成，否则释放刚分配的内存并继续向外

抛构造函数产生的异常。delete 时则判断指针是否为空，在指针不为空时调用析构函数并释放之前分配的内存。

回到 shape_wrapper 和它的析构行为。在析构函数里做必要的清理工作，这就是 RAI 的基本用法。这种清理并不限于释放内存，也可以是：

关闭文件（fstream 的析构就会这么做）

释放同步锁


释放其他重要的系统资源

例如，我们应该使用：

 复制代码

```
1  std::mutex mtx;
2
3  void some_func()
4  {
5      std::lock_guard<std::mutex> guard(mtx);
6      // 做需要同步的工作
7  }
```

而不是：

 复制代码

```
1  std::mutex mtx;
2
3  void some_func()
4  {
5      mtx.lock();
6      // 做需要同步的工作.....
7      // 如果发生异常或提前返回，
8      // 下面这句不会自动执行。
9      mtx.unlock();
10 }
```

顺便说一句，上面的 shape_wrapper 差不多就是个最简单的智能指针了。至于完整的智能指针，我们留到下一讲继续学习。

内容小结

本讲我们讨论了 C++ 里内存管理的一些基本概念，强调栈是 C++ 里最“自然”的内存使用方式，并且，使用基于栈和析构函数的 RAII，可以有效地对包括堆内存在内的系统资源进行统一管理。

课后思考

最后留给你一道思考题。shape_wrapper 和智能指针比起来，还缺了哪些功能？欢迎留言和我分享你的观点。

参考资料

[1] Wikipedia, “Memory management” .

[🔗 https://en.wikipedia.org/wiki/Memory_management](https://en.wikipedia.org/wiki/Memory_management)

[2] Wikipedia, “Stack-based memory allocation” .

[🔗 https://en.wikipedia.org/wiki/Stack-based_memory_allocation](https://en.wikipedia.org/wiki/Stack-based_memory_allocation)

[3] Wikipedia, “Resource acquisition is initialization” .

[🔗 https://en.wikipedia.org/wiki/RAII](https://en.wikipedia.org/wiki/RAII)

[3a] 维基百科, “RAII” . [🔗 https://zh.wikipedia.org/zh-cn/RAII](https://zh.wikipedia.org/zh-cn/RAII)

[4] Wikipedia, “Call stack” . [🔗 https://en.wikipedia.org/wiki/Call_stack](https://en.wikipedia.org/wiki/Call_stack)

[5] Wikipedia, “Object slicing” . [🔗 https://en.wikipedia.org/wiki/Object_slicing](https://en.wikipedia.org/wiki/Object_slicing)

[6] Stack Overflow, “Why does the stack address grow towards decreasing memory addresses?” [🔗 https://stackoverflow.com/questions/4560720/why-does-the-stack-address-grow-towards-decreasing-memory-addresses](https://stackoverflow.com/questions/4560720/why-does-the-stack-address-grow-towards-decreasing-memory-addresses)

注意：有些条目虽然有中文版，但内容太少；此处单独标出中文版条目的，则是内容比较全面、能够补充本专栏内容的情况。

点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 课前必读 | 有关术语发音及环境要求

下一篇 02 | 自己动手，实现C++的智能指针

精选留言

 写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。