01讲堆、栈、RAII：C++里该如何管理资源

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）。

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

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

```
# Python
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

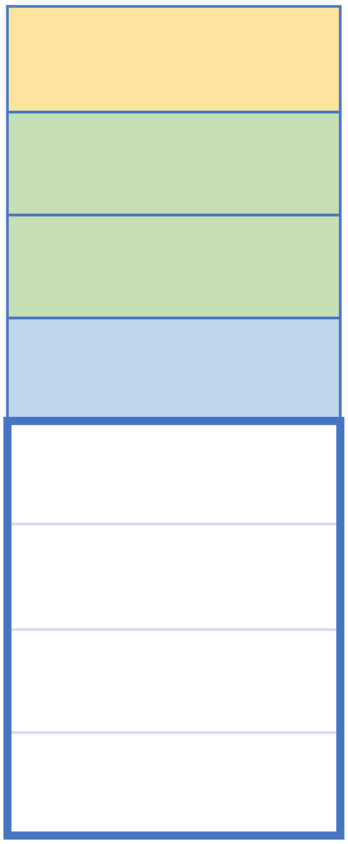

图1c：又分配了2单位和1单位

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。

```
void foo()
{
  bar* ptr = new bar();
  …
  delete ptr;
}
```

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：

```
bar* make_bar(…)
{
  …
  try {
    bar* ptr = new bar();
    …
  }
  catch (...) {
    delete ptr;
    throw;
  }
  return ptr;
}

void foo()
{
  …
  bar* ptr = make_bar(…)
  …
  delete ptr;
}
```

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。

```
void foo(int n)
{
  …
}

void bar(int n)
{
  int a = n + 1;
  foo(a);
}

int main()
{
  …
  bar(42);
  …
}
```

这段代码执行过程中的栈变化，我画了下面这张图来表示：

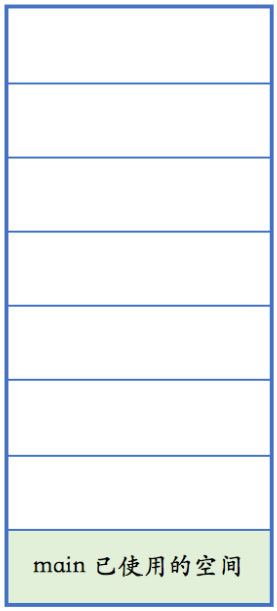

图2a：执行 bar 之前

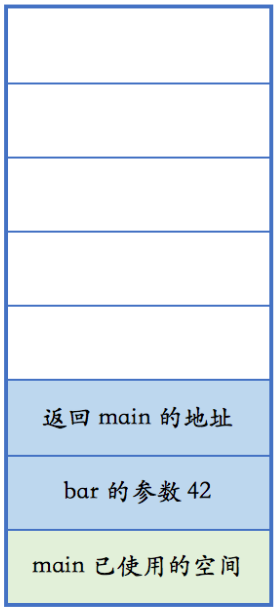

图2b：调用 bar

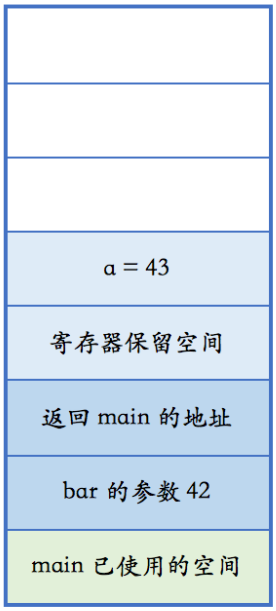

图2c：执行 foo 之前

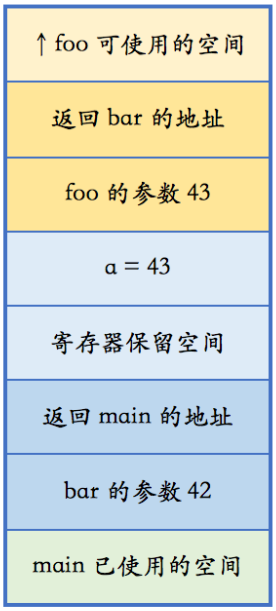

图2d：调用 foo

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：

```
#include <stdio.h>

class Obj {
public:
  Obj() { puts("Obj()"); }
  ~Obj() { puts("~Obj()"); }
};

void foo(int n)
{
  Obj obj;
  if (n == 42)
    throw "life, the universe and everything";
}

int main()
{
  try {
    foo(41);
    foo(42);
  }
  catch (const char* s) {
    puts(s);
  }
}
```

执行代码的结果是：

```
Obj()
~Obj()
Obj()
~Obj()
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：

```
enum class shape_type {
  circle,
  triangle,
  rectangle,
  …
};

class shape { … };
class circle : public shape { … };
class triangle : public shape { … };
class rectangle : public shape { … };

shape* create_shape(shape_type type)
{
  …
  switch (type) {
  case shape_type::circle:
    return new circle(…);
  case shape_type::triangle:
    return new triangle(…);
  case shape_type::rectangle:
    return new rectangle(…);
  …
  }
}
```

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：

```
class shape_wrapper {
public:
  explicit shape_wrapper(
    shape* ptr = nullptr)
    : ptr_(ptr) {}
  ~shape_wrapper()
  {
    delete ptr_;
  }
  shape* get() const { return ptr_; }
private:
  shape* ptr_;
};

void foo()
{
  …
  shape_wrapper ptr_wrapper(
    create_shape(…));
  …
}
```

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：

```
// new circle(…)
{
  void* temp = operator new(sizeof(circle));
  try {
    circle* ptr =
      static_cast<circle*>(temp);
    ptr->circle(…);
    return ptr;
  }
  catch (...) {
    operator delete(ptr);
    throw;
  }
}
```

```
if (ptr != nullptr) {
  ptr->~shape();
  operator delete(ptr);
}
```

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 RAII 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：

```
std::mutex mtx;

void some_func()
{
  std::lock_guard<std::mutex> guard(mtx);
  // 做需要同步的工作
}
```

而不是：

```
std::mutex mtx;

void some_func()
{
  mtx.lock();
  // 做需要同步的工作……
  // 如果发生异常或提前返回，
  // 下面这句不会自动执行。
  mtx.unlock();
}
```

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

[2] Wikipedia, “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

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

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

[5] Wikipedia, “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

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。

精选留言

Milittle

说实话，这个专栏对于我这个经常使用C++来做项目的人来讲，我认为不适合初学者，上车需要有过C++开发经验的。一般的小伙伴可能会有压力哒，但是如果想学，克服心里畏惧，从这个专栏出发可以迅速的深入。很好的专栏。

2019-11-26 08:33

作者回复

谢谢。这个专栏是要求之前学过、用过C++的。没学过的不合适。

2019-11-26 18:10

hello world

没有引用计数，没有拷贝和移动，没有线程安全，没有自定义delete函数，另外想请教老师一些问题.

1. 全局静态和局部静态的变量是存储在哪个区域？看很多书是静态存储区，但静态存储区又是什么区？堆？

2. thread local的变量存储在哪个区？因为线程是动态创建的，理解这个变量内存也应该动态分配的，线程结束内存自动释放？难道也是堆？

3. 类的大小是怎么定的呢？一般都是看类的成员变量占用字节数再根据是否虚类看是否加4字节，但是类里面有很多成员函数，这些成员函数不占空间吗，如果有静态成员变量或者静态成员函数呢？

谢谢老师!

2019-11-26 07:40

作者回复

其他都对，不过，自定义delete似乎目前没这个必要?

1. 好问题。静态存储区既不是堆也不是栈，而是……静态的。意思是，它们是在程序编译、链接时完全确定下来的，具有固定的存储位置（暂不考虑某些系统的地址扰乱机制）。堆和栈上的变量则都是动态的，地址无法确定。

2. thread_local和静态存储区类似，只不过不是整个程序统一一块，而是每个线程单独一块。用法上还是当成全局/静态变量来用，但不共享也就不需要同步了。

3. 非静态数据成员加上动态类型所需的空间。注意后者不一定是4，而一般是指针的大小，在64位系统上是8字节。还有，要考虑字节对齐的影响。静态数据成员和成员函数都不占个别对象的空间。

2019-11-26 09:57

bo

老师您好！工程的时候，具体怎么考虑在栈上分配还是在堆上分配，更合理些?

2019-11-26 10:17

作者回复

凡生命周期超出当前函数的，一般需要用堆（或者使用对象移动传递）。反之，生命周期在当前函数内的，就该用栈。

2019-11-26 18:03

hello world

话说一般delete.后需要把这个变量置成nullptr吗，我有时候这样写，不知道有没有必要

2019-11-26 07:41

作者回复

如果这个变量下面还有用到的地方，这是个好习惯。不过，这个习惯主要还是从C来的。现代C++不推荐一般代码里再使用裸指针和new/delete的。

2019-11-26 18:05

NEVER SETTLE

学习笔记：

1、概念
堆（heap）：在内存管理中，指的是动态分配内存的区域。当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（free store），特指使用 new 和 delete 来分配和释放内存的区域。
这是堆的一个子集：new 和 delete 操作的区域是 free store，而 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。

栈（stack）：在内存管理中，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。

RAII（Resource Acquisition Is Initialization）：C++ 所特有的资源管理方式。
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++ 不需要垃圾收集方法，也能有效地对内存进行管理。

2、堆
C++程序需要牵涉到两个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
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，并将其标记为已用，然后将其返回给要求内存的代码。

2). 让内存管理器释放一个之前分配的内存块
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。
目前的编程模式都要求申请的内存块是连续的。

从堆上申请的内存需要手工释放，但在此过程中，内存可能有碎片化的情况。
一般情况下不需要开发人员介入。因为内存分配和释放的管理，是内存管理器的任务。
开发人员只需要正确地使用 new 和 delete，即每个 new 出来的对象都应该用 delete 来释放。

3、栈
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，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。
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。

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。
栈上的内存分配，是移动一下栈指针。
栈上的内存释放，是函数执行结束时移动一下栈指针。
由于后进先出的执行过程，不可能出现内存碎片。

每个函数占用的栈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。

如果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。
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）。

在 C++ 里，所有的变量缺省都是值语义。
引用一个堆上的对象需要使用 * 和 & 。
对于像智能指针这样的类型，使用 ptr->call() 和 ptr.get()，语法上都是对的，并且 -> 和 . 有着不同的语法作用。
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。

2019-11-26 23:01

作者回复

认真记笔记非常好。

不过，建议笔记还是记关键字和要点，解释文字不用多。否则篇幅跟原文接近就意义不大了。

2019-11-27 08:12

yuchen

怕评论中您看不到，在此再问一下，麻烦您啦～
上个问题回顾：
对于图2d有疑惑，希望该图绘制中可以标明main函数占用的栈空间范围及其对应的栈帧，同理，对bar和foo也一样。如果将图2d从下到上每行编号为0，1，2，...，7，那么main、bar和foo对应的栈空间占用、栈帧分别是那几行呢?
您的回答：嗯，问得有道理。我的颜色选取不够好，回头改一下。按一般的栈帧定义，只有 0 属于 main，1–4 属于 bar。5 以上属于 foo。

首先，非常感谢您的回复～

然而，看到有人这样问您：“参数42”和“a=43”分别是函数调用的参数和函数局部变量，应该属于同一个栈帧，为什么这里不同?
您的回答是：同样，实际实现通常就是这个样子的。参数属于调用者而非被调用者，一般也是由调用者来释放——至少一般 x86 的实现是这个样子。

那么和您这里回答我的就不一致的呢。您这里回答我1-4属于bar，因此，那个人问的问题（“参数42”和“a=43”应该属于同一个栈帧）这句就是对的。另外您说“参数属于调用者而非被调用者”，这里1-4既然属于bar了，那么参数42不就属于了被调用者bar了吗？我理解的是main是调用者，main调用了bar，则bar是被调用者。

2019-11-26 21:13

作者回复

这里主要牵涉到“栈帧”是如何定义的。虽然“参数属于调用者而非被调用者，一般也是由调用者来释放”概念上没有错，但我当时

对“栈帧”的定义想当然了。我后来又查了一下定义（用词要以大家接受的用法为准），发现参数和局部变量应该算作一个栈帧里。也就是说，你们这儿的质疑是有道理的。所以，目前我已经把图修改了，这样应该就都没有疑问了。

2019-11-26 23:30

[?] [?]

nice，讲的很清楚

2019-11-27 15:36

作者回复

谢谢。

2019-11-27 20:31

蓝配鸡

栈展开那块的输出是下面这样吧?

Obj()
Obj()
~Obj()
~Obj()
life, the universe and everything

2019-11-27 11:57

作者回复

我感觉目前的结果是对的（也是从实际运行结果粘贴过来的）。你再看看、试验一下?

2019-11-27 20:30

虫 二

当使用shape_wrapper，内存是什么时候被释放的呢？也有用到引用计数吗?

2019-11-27 11:28

作者回复

对象超出作用域时被释放——一个shape_wrapper本地变量离开定义此变量的结束大括号时。当前的定义没有引用计数。

2019-11-27 20:28

Gerry

栈通常说是向下增长，从高地址到低地址。文中表述是向上增长感觉欠妥。

2019-11-27 09:43

作者回复

因你这句话，我特地又去查了一下，目前看到的图，开口永远是上方。中英文资料都是如此。

这个词的来源实际上可能是堆盘子。显然，你只能从上面取放盘子……

2019-11-27 20:22

robonix

老师，”编译器调用析构函数”是不是指编译器在二进制文件中插入析构函数的代码?

2019-11-27 08:43

作者回复

嗯，是的。

2019-11-27 20:16

小学生

关于对象切片的那个问题，因为使用到了继承，应该不会考虑返回普通对象吧？因为可能要需要基类或子类。
可以给出一些，使用了继承又返回普通对象（即非指针）的场景吗　经验真的少
create_sharp 里面如果new 了一块内存且用于保存该内存的对象在超过其作用范围内，对象被销毁，可是指向的内容没有手动delete，而后再也找不到该对象（也就是没有显式指针引用），这样就内存泄漏了，因为再也引用不到那块内存了。这是栈变量或者局部变量的作用域带来的，所以用类变量可以将其作用域延长至整个类的生命周期中

2019-11-26 23:39

作者回复

对，继承情况下，基类一般只用指针或引用，基本没例外。

跟局部变量对应的一般是全局/静态变量。C++ 里没有类变量的说法。要延长生命期，一般也不是用你说的方法，而是返回对象（指针）。

2019-11-27 08:18

Egos

老师的代码是标准的cpp code style 吗？看着class 第一个字母小写不习惯

2019-11-26 22:28

作者回复

代码风格有很多种，不同的项目都不一样。我现在做 Unix/Linux 项目比较多，一般全小写居多。

这个跟项目走，不用自己有很强的意见。

2019-11-27 08:10

yuchen

对于图2d有疑惑，希望该图绘制中可以标明main函数占用的栈空间范围及其对应的栈帧，同理，对bar和foo也一样。如果将图2d从下到上每行编号为0，1，2，...，7，那么main、bar和foo对应的栈空间占用、栈帧分别是那几行呢?

2019-11-26 19:43

作者回复

嗯，问得有道理。我的颜色选取不够好，回头改一下。按一般的栈帧定义，只有 0 属于 main，1–4 属于 bar。5 以上属于 foo。

2019-11-26 20:57

%;

找到了a tour of c++ 学习先

2019-11-26 16:04

xjtu_ss_70309

缺了引用计数

2019-11-26 12:12

frazer

有点看不懂了，还得回去恶补CPP的语法知识

2019-11-26 11:00

作者回复

那就多读几遍　。

2019-11-26 18:14

流浪地球

老师您好，请问 “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” 这段讲述中的delete是重载过的吗？全局的::operator delete是做判空操作吗?

谢谢

2019-11-26 10:33

作者回复

没有重载。全局的delete（和free）都可以接受空指针的。

2019-11-26 17:59

李良川

智能指针还少了对*和->符号的支持，为支持所有指针，需要用类模版。

2019-11-26 00:12

作者回复

嗯，是的。这个也是目前这个类缺失的地方。

2019-11-26 09:36

小林coding

老师，文稿中的这句话：

[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]

这里想说明的是返回类型是普通 shape ， 而不是 shape* 或 shape& 是吗?

2019-11-25 23:09

作者回复

是的。

2019-11-26 00:04