RAII与智能指针

直接使用指针

- 问题
  - 忘记 delete 堆内存指针
  - 程序异常导致无法执行 delete 语句

```
// 工厂函数返回堆内存指针
Buz *buz = create_buz(args, ...);

// do something, but throws exception

// 此时将无法执行该语句，内存泄漏
delete buz;
```

- 解决
  - 不要忘记 delete 任何函数返回的指针，以及 new 产生的指针
  - 编写完整的异常处理，但是 C++ 中并没有 `finally` 这样的关键字，多少有些不便
- 结论
  - "不要忘记" = "一定会忘记"
  - 在大型项目中排查是否有堆指针内存泄漏不是一件容易的事情，除非使用诸如 MFC、Valgrind 等框架

RAII

- 关于 new
  - 对于 Buz *buz = new Buz(); 这条调用语句可以看成是下面 3 行语句的执行

```
void *mem = operator new(sizeof(Buz));  // 内存分配，大多时刻调用 malloc
buz = static_cast<Buz *>(mem);          // 显式类型转换，将 void * 转换为 Buz *
buz->Buz::Buz();                        // 调用构造函数初始化对象（伪代码）
```

  - 可以看到，构造函数是由编译器帮我们调用的，不需要开发者介入。但是 delete 却必须由开发者手动调用—编译器不知道该对象的生命周期
- 关于栈内存对象
  - 既然 Buz *buz = new Buz(); 是编译器在分配内存后调用对象，那么在函数中执行的 Buz buz(); 又是谁调用构造函数呢?
  - **当我们的对象（带有构造和析构函数，一般称之为非 Plain Old Data 对象）分配在栈内存时，编译器会在合适的地方插入对构造函数和析构函数的调用**
  - 为什么分配在栈内存的对象编译器会主动的调用析构函数呢?
    - 因为编译器非常清楚的明白栈对象的生命周期就是在当前作用域中
    - 编译器将自动调用栈对象的析构函数，即使是在有异常发生时。并且，在发生异常时对析构函数的调用也称之为栈展开（Stack Unwinding）
- RAII
  - **在有了栈对象自动调用构造函数和析构函数的特性以后，结合析构函数，就可以实现 RAII，其实也就是使用局部对象来管理资源。**

```
class BuzWrapper {
private:
    Buz *m_buz;
public:
    BuzWrapper(Buz *buz) : m_buz(buz) {}
    ~BuzWrapper() {delete m_buz;}
};


// 某一个作用域
{
    Buz *buz = new Buz();
    BuzWrapper wrapper(buz);

    // do something, may be throw exception
}
```

    - 使用 BuzWrapper 对 Buz 对象进行了一层封装，接收一个指向堆内存的指针
    - 当 wrapper 离开作用域时，将自动调用其析构函数，而 BuzWrapper 的析构函数即删除所管理的堆内存指针。这个过程由于有栈展开的存在，在异常发生时也将会进行
    - 这其实就是使用局部对象 wrapper 来管理堆内存，使得开发者不需要再手动执行 delete ptr; 而是将这个过程交给了一个局部对象，局部对象是可以被自动管理的

智能指针

- 我们在使用资源时大致可以分为 3 部分: 1. 获取资源，2.使用资源，3.归还资源
  - 大部分的内存泄漏均发生在第 3 步，也就是归还资源上
- 因此，基于栈和析构函数实现的 RAII 就变成了 C++ 语言中及其重要的内存管理手段，可以有效地对包括堆内存在内的系统资源进行统一管理
- 智能指针的原理和上面写的 BuzWrapper 基本一样，只不过 BuzWrapper 缺少了引用计数、拷贝构造、拷贝赋值等一些辅助功能，以及不支持泛型、自定义删除器和移动语义。尽管缺失了这么多功能，但是智能指针最为核心的功能还是自动管理内存。