shared_ptr

基本概念

- 基本原理
  - 前面我们提到了 RAII 与智能指针的关系，智能指针其实就是利用基于栈和析构函数的 RAII 而工作的
  - 我们将一个堆内存指针交给局部对象进行管理，当该局部对象离开其作用域后，系统将自动调用该对象的析构函数，而在该析构函数中，实际上是释放了所管理的堆内存指针
  - 因为栈展开的存在，使得这种方式在异常发生时也不会造成内存泄漏，这就是 C++ 比较独特的管理堆内存方式
- 特性
  - shared_ptr 非常类似于 Python 中通过引用计数实现的 GC
  - **多个 shared_ptr 可共享同一块堆内存，当我们对 shared_ptr 进行赋值初始化、拷贝初始化时将自动增加其引用计数 当 shared_ptr 离开作用域时将自动减少其引用计数，当引用计数减至为 0 时，将自动地销毁所管理的堆内存**
- 基本使用
  - 初始化
    - shared_ptr 的构造函数被声明为 explict，故我们必须使用显式初始化的方式对其进行初始化
    - shared_ptr<int> ptr = new int(1024); — 错误，不允许进行隐式地类型转换
    - shared_ptr<int> ptr{new int(1024)}; — 正确，圆括号和大括号初始化都可以
  - **尽量不要使用已存在的堆内存指针初始化 shared_ptr**
    - ```
      int *k = new int(1024);
      shared_ptr<int> ptr{k};
      ```
    - 这么写编译的确不会有任何问题，但是此时程序中既可以使用指针 k，也可以使用智能指针 ptr，那么就会存在裸指针和智能指针混用的情况，首先是混乱，其次可能会带来错误
    - ```
      int *k = new int(1024);
      shared_ptr<int> ptr{k};
      // do something
      delete k;
      ```
      — 如果我们手动的释放了指针 k，那么当 ptr 的引用计数减至为 0 并销毁对象时，将会因为对同一块内存 delete 两次而报错
- make_shared
  - 在实际应用中，我们可能更多的是使用 make_sharded 函数模板来创建并初始化一个智能指针
  - shared_ptr<int> ptr = make_shared<int>(1024);
  - 函数模板中的函数参数即为构造对象所需要的参数，也就是 args

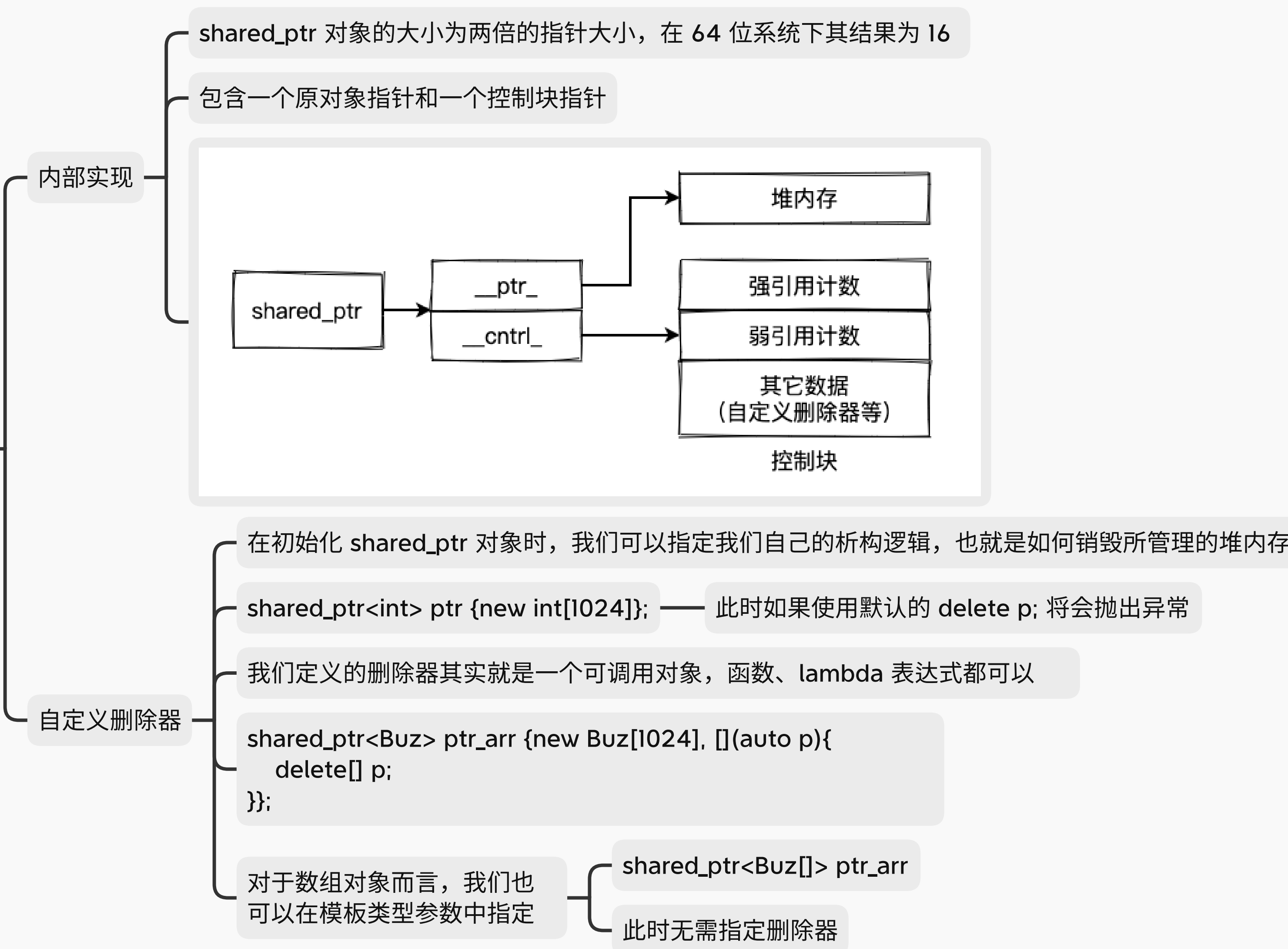
引用计数的增加与减少

- 引用计数的增加
  - 显式拷贝构造、拷贝赋值
    ```
    shared_ptr<int> ptr = make_shared<int>(1024);
    shared_ptr<int> ptr2 {ptr};          // 拷贝构造

    shared_ptr<int> ptr3;
    ptr3 = ptr2;                         // 拷贝赋值

    cout<<ptr.use_count()<<endl;    // 3
    ```
  - 值类型作为函数形参
    - ```
      void foo(shared_ptr<int> ptr) {
          cout<<ptr.use_count()<<endl;
      }
      ```
    - 当我们使用值类型作为函数形参时，函数调用将使用实参初始化形参，此时将调用拷贝构造函数，使得引用计数加 1。函数调用结束时引用计数减 1，引用计数将恢复到调用函数之前
- 引用计数的减少 — 通常发生在局部变量离开作用域时

内部实现与指定删除器

- 内部实现
  - shared_ptr 对象的大小为两倍的指针大小，在 64 位系统下其结果为 16
  - 包含一个原对象指针和一个控制块指针
  - 

- 自定义删除器
  - 在初始化 shared_ptr 对象时，我们可以指定我们自己的析构逻辑，也就是如何销毁所管理的堆内存
  - shared_ptr<int> ptr {new int[1024]}; — 此时如果使用默认的 delete p; 将会抛出异常
  - 我们定义的删除器其实就是一个可调用对象，函数、lambda 表达式都可以
  - ```
    shared_ptr<Buz> ptr_arr {new Buz[1024], [](auto p){
        delete[] p;
    }};
    ```
  - 对于数组对象而言，我们也可以在模板类型参数中指定
    - shared_ptr<Buz[]> ptr_arr
    - 此时无需指定删除器

常用操作

- use_count()
  - 返回和当前 shared_ptr 对象（包含自身）指向相同堆内存的所有 shared_ptr 对象的数量
  - ```
    shared_ptr<int> ptr = make_shared<int>(1024);
    shared_ptr<int> ptr1 {ptr};
    shared_ptr<int> ptr2 {ptr1};

    cout<<ptr.use_count()<<endl;
    ```
  - 目前有 3 个 shared_ptr 对象指向内容为 1024 的堆内存，故输出结果为 3
- unique() — 判断是否只有当前 shared_ptr 指向某一个堆内存，其实就是判断 user_count() 结果是否为 1
- reset() — 重置或者是复位
  - 不带参数 — 使当前 shared_ptr 放弃对某一个堆指针的管理，引用计数减一
  - 带参数 — 使当前 shared_ptr 管理一个新的堆内存
- 解引用 — 获得所管理的堆指针的实际对象
- get
  - 获取原有堆指针，或者说返回裸指针
  - 慎用该方法，除非真的有必要。例如一些老的函数库只能接收裸指针，无法接收智能指针
- swap — 交换 2 个相同类型 shared_ptr 智能指针的内容
    ```