Modern C++ Camp

现代C++系统研发骨干特训营

李建忠 Boolan

Boolan
高端IT咨询与教育平台

现代C++系统研发骨干特训营

模块三、C++内存管理

RAII 习语——资源获取即初始化

- RAII (Resource Acquisition Is Initialization) 是C++内存和资源管理最重要的机制之一。
- RAII通过三个环节来保证内存或资源得到确定性释放
  - 1、构造器中获取内存或资源
  - 2、析构器中释放内存或资源
  - 3、栈对象在作用域结束，即确定性调用析构器、回收内存
- RAII的析构机制是由编译器根据对象生命周期销毁机制自动确保的，无需手工干预，得到确保执行。
- RAII 机制 完善了 C++对值语义的坚持。
- 标准库中RAII无处不在(string,thread,ifstream,unique_ptr……)

RAII 的核心优势

- 1. 异常免疫，即使出现异常，也确保执行析构。
- 2. 同时管理内存与非内存资源（如文件句柄、锁、网络IO...）
- 3. 不仅管理栈对象，同时管理堆对象
- 4. 失效即释放，时间确定性（而非主流垃圾收集器的非确定性）
- 5. RAII 针对对象嵌套结构是递归进行的（依赖析构器的递归）。
- 6. RAII和移动语义、智能指针结合之后达成了资源管理正确性和性能的双重保障。
- 7. RAII 和 C++对值语义的彻底支持可谓珠联璧合。
- 8. 遵从好的RAII设计规范，是C++实现内存和资源安全的坦途。

性能指南——对象与资源管理

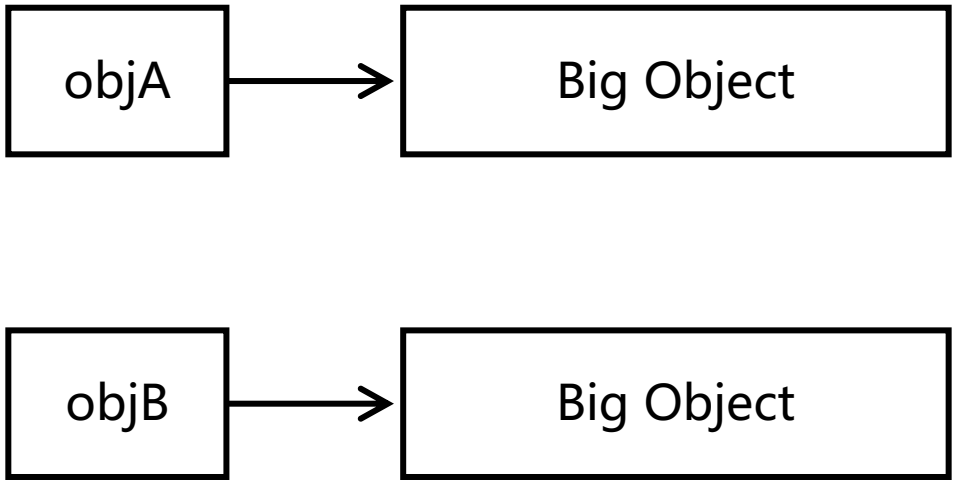
- 利用RAII（资源获取即初始化）自动管理资源
- 优先采用有作用域的栈对象，避免不必要的堆分配
- 保持作用域尽量小，最小化资源持有时间
- 尽量避免全局变量/静态变量
- 严格避免 malloc() 和 free()（无对象状态管理）
- 尽量避免显式调用 new 和 delete
- 同时重载相匹配的分配、回收函数对

对象拷贝的代价

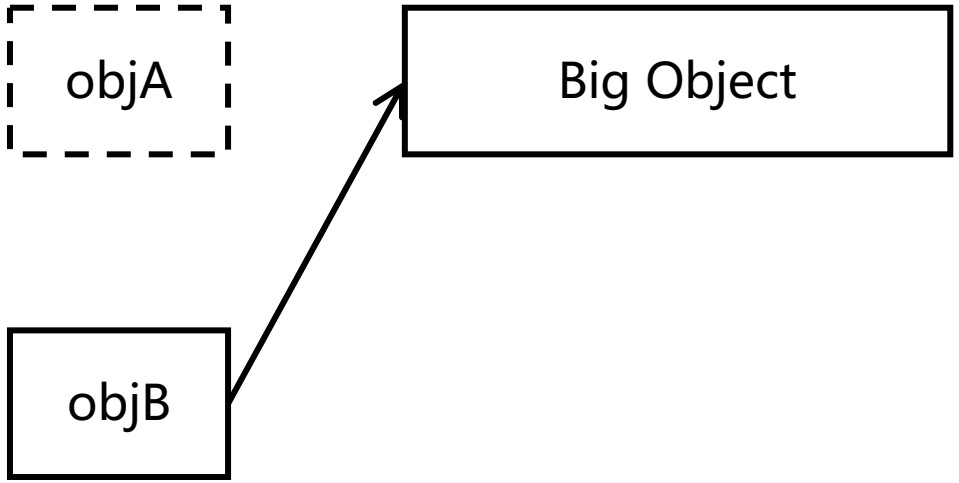
- 内含动态内存的对象，值语义下要求执行深拷贝
- 对象拷贝发生的时机有很多，而且有很多隐藏在其他操作之中
  - 赋值
  - 初始化
  - 传参
  - 返回值
  - 交换 swap
  - 容器扩容
  - .....
- 手工编码避免大对象的拷贝，生命周期管理极易出错，因此C++11引入了移动操作(move)

使用对象移动降低拷贝代价

- 如果被拷贝的对象objA，在拷贝之后确定不再使用，那么直接将目标对象objB的大对象指针指向objA内的大对象指针将大大提升拷贝性能。这就是移动操作的来源。



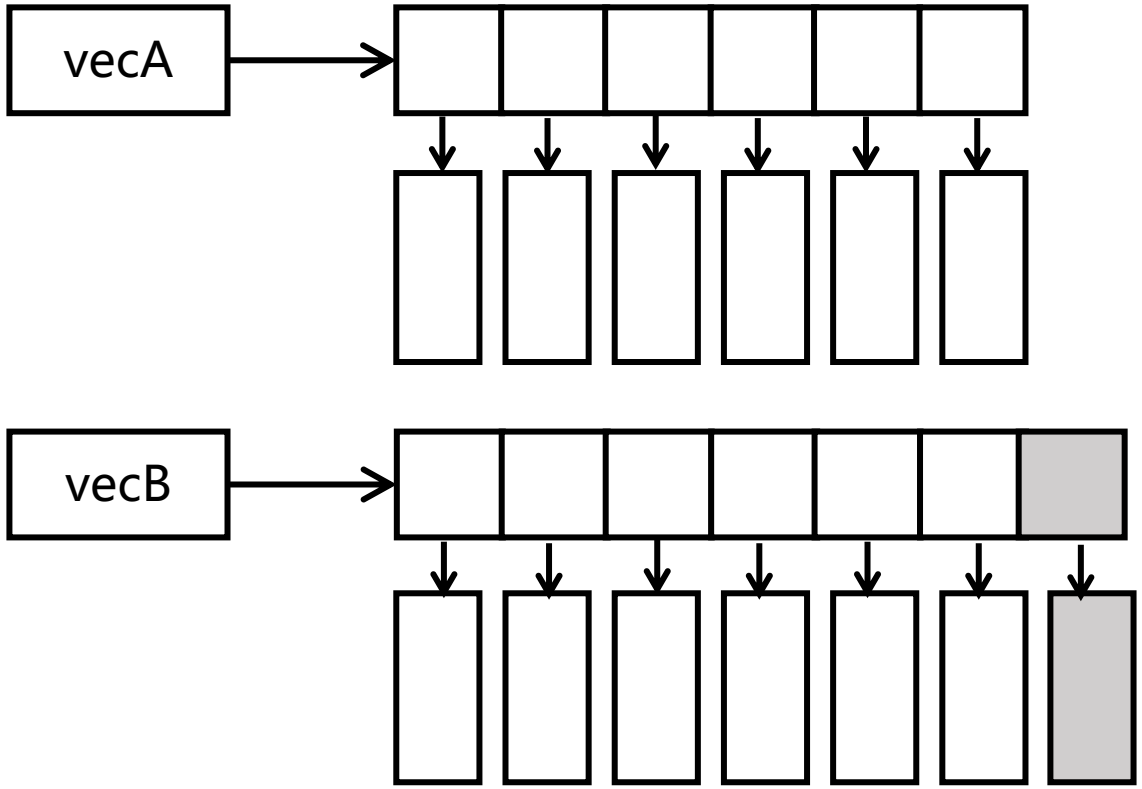
objB=objA;



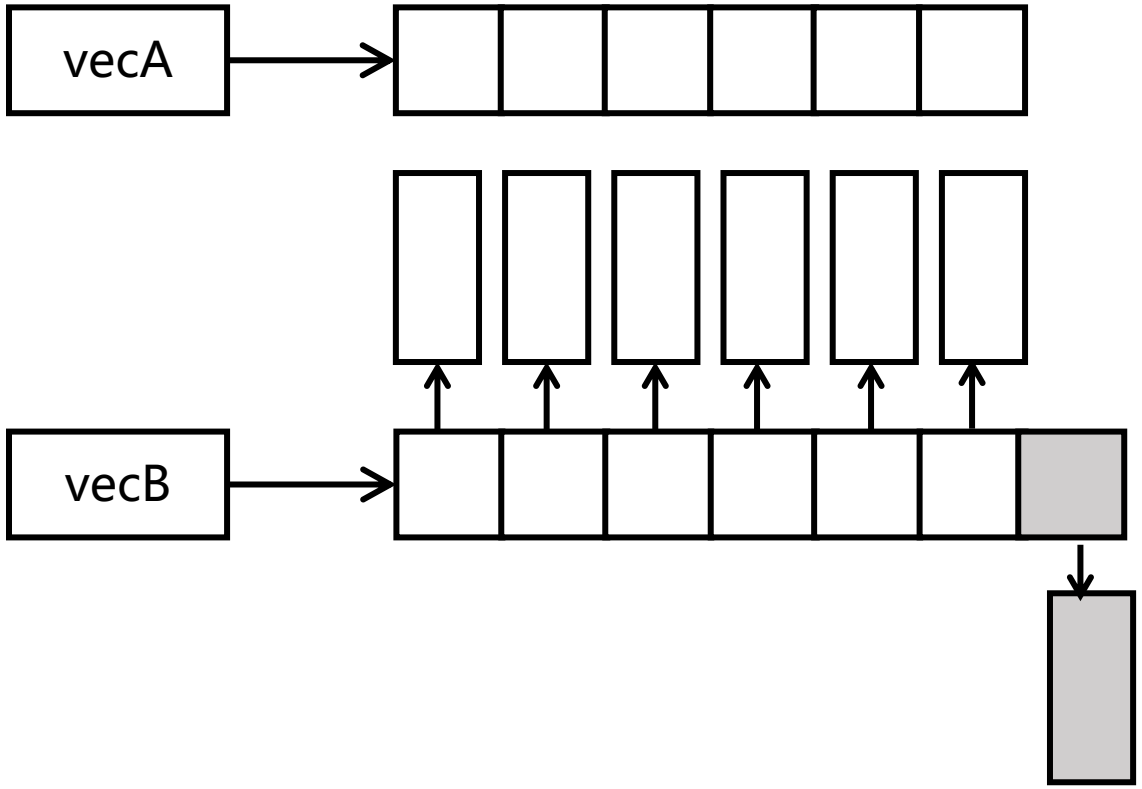
objB=std::move(objA);

使用对象移动降低集合变更代价

- 各种集合的push_back, insert, emplace, resize, erase...变更操作



vecA.push_back(obj);
假设导致扩容，不支持move



vecA.push_back(obj);
假设导致扩容，支持move

移动语义的主要价值

- 移动发挥最大价值的场景：
  - 对象内部又有分离内存（通常是指针指向的堆内存）
  - 对象拷贝是深拷贝
- 移动仅复制对象内本身的数据，不复制分离内存
- 而拷贝既复制对象内本身的数据，也复制分离内存
- 移动永远不会比拷贝慢，通常更快。
- C++ 通过两个操作来支持移动语义：
  - 移动构造函数
  - 移动赋值操作符
- 对象可被移动的前提是——移动之后不再使用！此乃右值的来源

右值引用

- C++ 11引入右值引用语法：T&&
- 通常的引用T&现在被认为是左值引用

- 右值引用在某些方面和左值引用有类似行为
  - 必须被初始化，不能被重新绑定

- 右值引用表示对象可以“从这里移动到别的对象”

左值Lvalues与右值Rvalues

- 左值：命名对象、可取地址&，可赋值
  - 基本类型变量、数组、数组元素
  - 字符串字面量，如“CppCamp”
  - 对象变量、对象成员
  - 指针、指针解引用后的对象/变量
  - 函数（可取地址）
  - 返回左值的表达式
- 右值：无名、无法取地址，不可赋值
  - 各种临时对象（函数返回值）、字符串临时对象
  - 除字符串外的其他基本类型字面量
  - lambda 表达式
  - 运算符表达式

左值、右值与移动

- 左值（lvalue，left value）：有身份、不能移动
- 纯右值(prvalue，“pure” rvalue)：没身份、可移动
- 将亡值(xvalue，“eXpiring” value）：有身份、可移动

- 当源对象是一个左值，移动左值并不安全，因为左值后续持续存在，可能被引用，虽然可以将左值强制转换为右值，但是要自负安全
- 当源对象是一个右值，移动很安全。
- void func(Widget&& v) 函数形参v，到底是右值、还是左值？
  - 对函数调用者来说，v 是个右值引用参数（要传递右值给它）
  - 对函数内部来说，v 是个地道的左值引用（可以取地址）

实现移动支持

- 移动拷贝构造函数
  - 1. 窃取源对象内指针指向的值
  - 2. 将源对象内的指针值设为有效状态
- 移动赋值操作符
  - 1. 删除当前对象内指针指向的值
  - 2. 窃取源对象内指针指向的值
  - 3. 将源对象内的指针值设为有效状态
- 类内的对象成员处理、基类对象的处理
  - 不要直接拷贝，要使用std::move，从而调用它们的移动操作
  - 右值引用参数传递给其他函数被认为是左值，如要移动需要std::move
- 除了移动构造和赋值操作，还有
  - 赋值型函数（setXXX)也建议支持移动操作

绑定规则

- 左值（ Lvalues ）可以绑定到左值引用（ lvalue references ）
- 左值不可以绑定到右值引用（ rvalue references ）
- 右值（ Rvalues ）可以绑定到左值常量引用（ lvalue references to const ）
- 右值（ Rvalues ）可以绑定到右值非常量引用（ rvalue references to non-const ）

	&	const&	&&	const&&
左值	可以	可以		
const 左值		可以		
右值		可以	可以	可以
const 右值		可以		可以

理解std::move操作

- **不是移动**：move函数并不做具体移动操作，其目的只是告诉编译器当前对象具备可移动条件
- **类型转换**：本质是一种强制类型转换，将参数转换为右值，可以理解为“右值类型转换”（rvalue_case）编译时特征，对运行期无影响
- **不保证**：并不必然导致移动构造或赋值发生，还要看参数是否符合其他条件
  - 如果参数本身不支持移动构造和赋值
  - 如果是对const左值参数使用std::move，移动构造不接受常量性参数
- **退化拷贝**：如果不能满足移动的条件，对移动的请求最后还会退回为拷贝操作。

特殊成员函数的五法则

- 回顾三法则：析构函数、拷贝构造函数、赋值操作符 三者自定义其一，则需要同时定义另外两个（编译器自动生成的一般语义错误）
- 如果没有自定义析构函数、拷贝构造函数、赋值操作符任何其一，编译器也会自动生成移动构造和移动赋值操作符，生成的是按成员进行实例成员的移动操作请求（如果不支持，则退化为拷贝）。
- 如果自定义了析构函数、拷贝构造函数、赋值操作符任何其一，那么移动构造和移动赋值操作符，都需要自定义，编译器将不再自动生成（常见陷阱的由来！）
- 如果自定义了移动构造、移动赋值操作符任何其一，编译器将不再自动生成另外一个（注意 和三法则的编译自动生成规则不同）和对应的拷贝构造、或赋值操作符。
- 简单规则：五大特殊成员函数要么全部自定义（指针指向动态数据成员），要么全交给编译器自动生成（基本类型或对象成员）。

转发引用 与 右值引用


```
template < typename T>
void func(T&& obj)
```

```
auto&& obj2=obj1;
```

- 两种引用都不一定是右值引用。它们可能是左值，也可能是右值，具体要通过绑定参数，来进行类型推导，因此被称为转发引用
- 如果使用右值绑定转发引用，则得到右值引用；如果使用左值绑定转发引用，则得到左值引用。
- 如果没有进行类型推导，则代表右值引用。
- 如果是const T&&，也是地道的右值。

理解std::forward操作

- 应用于转发引用
  - forward只有应用于转发引用，才有意义。
- 有条件的编译时类型转换，没有任何运行时计算
  - 当传入的参数是右值，forward将类似std::move函数，转换为右值（注意形参默认是左值），从而保留参数的右值特性。
  - 当传入的参数是左值，forward将什么都不做，继续保留参数的左值特性。
- 不要对转发引用，调用std::move，因为可能是左值。
- 如果没有forward，很多函数需要同时提供两种重载（传入左值时，使用左值引用；传入右值时，使用右值引用），代码重复且易错。

了解引用折叠


```
template < typename T>
void func(T&& obj)

func(obj1)
auto&& obj2=obj1;
```

- 如果obj1为左值（假设： int ），其类型辨析为int&，T&&的结果为int & &&，引用折叠后结果为int &
- 假设obj1为int右值（假设： int ），其类型辨析为int，T&&的结果为int&&，不发生引用折叠，结果仍为int &&

Move语义相关操作最佳实践（1）

- 当对象内含指针指向的动态内存（这时移动操作通常比拷贝更快），提供移动操作（移动构造和移动赋值）
- 当对象内不含指针指向的动态内存，这时移动不会比拷贝更快。
- 尽可能为移动操作加上noexcept，否则某些强异常安全保证的容器操作会寻求使用拷贝代替移动。
- 为动态容器(大对象)提供移动操作，因为拷贝非常昂贵
- 移动操作应当进行移动，并使原对象处于有效状态
- 不要假设对象执行移动操作后，内部指针一定是nullptr

Move语义相关操作最佳实践（2）

- 对于没有提供移动支持的类型请求move是安全的，使用拷贝替换
- 仅当需要明确移动某个对象到别的作用域时，使用 std::move()
- 对于“将被移动”参数，按 X&& 进行传递并对参数 std::move
- 如果局部对象会被返回值优化，则不要对返回值调用std::move
- 不要在 const 对象上调用 std::move()