|  |  |
| --- | --- |
| 1. 移动构造函数和移动赋值运算符 2. 移动操作、标准库容器和异常 3. 移动赋值运算符 4. 移后源对象必须可析构 5. 合成的移动操作 6. 移动右值，拷贝左值 7. 没有移动构造函数，右值也被拷贝 8. 拷贝并交换 赋值运算符和移动操作 9. Message类的移动操作 10. 移动迭代器 | Strvec::StrVec(StrVec &&s) noexcept  :element(s.elements),first\_free(s.first\_free),cap(s.cap)  {  s.element=s.first\_free=s.cap=nullptr  }  Strvec::StrVec(StrVec &&s) noexcept  {  If(this != &rhs){  …  Rhs.element=nullptr;  }  Return \*this;  }  Struct X{  Int I;//内置类型可以移动  Std::string s;//string定义了自己的移动操作  }  Struct hasX{  X mem;  }  X x,x2=std::move(x);  hasX hx,hx2 = std::move(hx);  StrVec v1,v2;  V1 = v2;  StrVec getVec(istream&)  V2 = getVec(cin);//getVec返回右值，使用移动赋值  Class Foo{  Public:  Foo() = default;  Foo(const Foo&);  }  Foo x;  Foo z(std::move(x));//也是执行拷贝构造函数  HasPtr& operator=(HasPtr rhs)  Hp = std::move(hp2);//拷贝并交换  Uninitialized\_copy(make\_move\_iterator(begin()),make\_move\_iterator(edn()),first) |
| 1. 移动构造函数和移动赋值运算符 2. 我们的类也同时支持移动和拷贝 3. 是窃取资源 不是拷贝资源 4. 写法形式与拷贝的差不多，不过是右引用，并且const不同 5. 必须确保移后源对象处于销毁无害的。源对象不再指向被移动的资源 6. 因为移动赋值后，若源对象被销毁，并且还指向被移动的资源，那么就会释放掉我们刚刚移动的内存 7. 移动操作、标准库容器和异常 8. 因为是窃取，通常来说，移动操作不会抛出任何异常，应该通知标准库，不然标准库认为移动我们的类对象可能会抛出异常，并会作出一些额外的工作。一种是声明noexcept。 9. 声明和定义的地方都需要noexcept，在参数列表之后 10. noexcept的作用   可以帮助我们深入理解标准库与我们自定义的类型交互的  移动操作可以发生异常  标准库容器能对异常发生时其自身的行为提供保障，如调用push\_back时发生异常，vector自身不会改变  1.如果重新分配过程中使用了移动构造函数，移动了部分时发生了异常，让移动源元素已经改变,而新空间中未构造的元素可能尚不存在，vector将不能满足自身保持不变的要求  2.但是如果使用了拷贝构造，即使发生异常，释放新分配的内存并返回，vector原元素仍然存在。  结合1,2说明总结：  vector如果知道元素类型的移动构造函数不会抛出异常，在重新分配内存的过程中，就使用移动构造函数，否则就使用拷贝构造函数   1. 移动赋值运算符 2. 处理析构函数和移动构造函数相同的工作， 必须处理自赋值 3. 直接先检查this与rhs是否同一个对象 4. 移后源对象必须可析构 5. 从一个对象移动数据并不会销毁此对象，但有时在移动操作完成后，源对象会销毁。   所以必须移后源对象必须可析构   1. 还需要对象仍然有效的，指：可以赋予新值或者可以安全地使用而不依赖其当前值。程序不应该依赖移后源对象中的数据 2. 为了让移后源对象有效，不同类有不同的行为，像StrVec类是将成员为默认初始化时一样的状态，为nullptr   Note：移后源对象必须保持有效的、可析构的状态，但用户不能对其值进行任何假设   1. 合成的移动操作 2. 编译器根本不会为"某些类"合成移动操作，特别是一个类定义了自己的拷贝构造函数 拷贝赋值运算符或者析构函数。则不会为它合成移动，那么就是删除的移动操作。那么类会使用拷贝构造操作来代替移动操作，因为const int&可以绑定右值 3. 只有当类没有定义任何自己版本的拷贝控制成员，且类的每个非static数据成员都可以移动时 才会合成（内置类型可以移动，成员为标准库类对象且有对应的移动操作）   删除的移动操作   1. 移动操作永远不会隐式定义为删除的函数。 2. 显式的指定default 并且编译器不能移动所有成员，则定义为删除的   除了一个重要例外，合成的移动操作定义为删除的函数遵循与定义删除的合成拷贝操作类型的原则   1. 类成员定义了拷贝构造未定义移动构造，或类成员为定义拷贝构造不能为其合成移动构造 2. 类成员的移动构造被定义为删除或不可访问 3. 类的析构函数被定义为删除或不可访问的 4. 类成员是const的或引用，类的移动赋值运算符为删除，但移动拷贝构造可以   5) 最后一个 移动操作和合成的拷贝控制成员关系  一个类定义了一个移动构造函数和/或一个移动赋值运算符，则该类的合成构造函数和拷贝赋值运算符会被定义为删除的  Note:定义了一个移动构造函数或移动赋值运算符的类必须也定义自己的拷贝操作。否则，这些成员默认地被定义为删除的   1. 移动右值，拷贝左值 2. 既有拷贝构造又有移动构造，编译器使用普通的函数匹配规则来确定 3. 右值引用只接受实参是非static右值的情形 4. 因为赋值运算符，右值可以用const 左引用引用，或者直接右引用引用，所以右值存在两个匹配 5. 没有移动构造函数，右值也被拷贝 6. 若类有一个拷贝构造函数并没有移动构造函数。函数匹配规则保证该类型的对象会被拷贝，即使通过调用move来移动也是如此 7. 可以将 int&& 转换为const int& 8. 拷贝构造函数代替移动构造函数总是安全的。符合：拷贝给定对象，将源对象置于有效状态。   Noe：移动赋值运算符类似   1. 拷贝并交换 赋值运算符和移动操作 2. 定义拷贝并交换赋值运算符，添加一个移动构造函数，则也会获得一个移动赋值运算符 3. 结合临时对象和swap创建一个即是移动赋值运算符 也可以是拷贝赋值运算符 4. 如果传的拷贝并交换形参是右值，则调用移动构造函数，执行移动赋值运算符操作   如果传的拷贝并交换形参是左值，则调用拷贝构造函数，执行拷贝赋值运算符操作   1. Swap交换两个运算对象的状态。Rhs离开作用域时，这个string被销毁   建议：更新三/五法则  五个拷贝控制成员应该看做一个整体,如果一个类定义了任何一个拷贝操作，它就应该定义所有五个操作。通常一个类有一个资源成员那么需要拷贝成员，但是拷贝成员额外开销，所以定义移动的就可以避免   1. Message类的移动操作 2. 就是一个概念，使用移动操作是将右侧的移动到左侧后，右侧运算对象称为移后源对象就不会再在原先的floder集合中了 3. 并且减少不必要的消耗 4. 移动迭代器 5. for循环construct很慢，用uninitialized\_copy来简单，需使用移动迭代器适配器 6. 一个移动迭代器通过改变给定迭代器的解引用运算符的行为来适配此迭代器   解引用移动迭代器生成一个右值引用   1. make\_move\_iterator移动迭代器支持正常迭代器的操作，可以传给算法 2. 由于移动一个对象可能销毁掉原对象，只有在确信算法在为一个元素赋值或将其传递给一个用户定义的函数后不再访问它时，才能将移动迭代器传递给算法   建议：不要随意使用移动操作  note:确信move是安全的再使用。不然莫名其妙，难以查找的错误 | |