当我们定义一个类时，我们明确或隐含地指定当该类类型的对象被

复制、移动、赋值和销毁时会发生什么。类通过定义五个特殊的成员函数来控制这些操作：

复制构造函数、复制赋值操作符、移动构造函数、移动赋值操作符和析构函数。

复制和移动构造函数定义了当一个对象从另一个相同类型的对象初始化时会发生什么。复制和移动赋值操作符定义了当我们将一个类类型的对象赋值给同一个类类型的另一个对象时会发生什么。析构函数定义了当该类型的对象不再存在时会发生什么。总的来说，我们将把这些操作称为复制控制。

如果类没有定义所有的复制控制成员，编译器会自动定义缺少的操作。因此，许多类可以忽略复制控制（第7.1.5节）。然而，对于某些类，依赖默认定义会导致灾难。通常，实现复制控制操作最困难的部分是识别何时需要首先定义它们。

复制控制是定义任何C++类的基本部分。刚接触C++的程序员经常被定义当对象被复制、移动、赋值或销毁时会发生什么所迷惑。这种混淆是复杂的，因为如果我们不显式定义这些操作，编译器会为我们定义它们-尽管编译器定义的版本可能不会像我们预期的那样运行。

该参数几乎总是对const的引用，尽管我们可以定义复制构造函数来引用nonconst。复制构造函数在几种情况下隐式使用。因此，复制构造函数通常不应该是显式的（第7.5.4节）。

当我们使用直接初始化时，我们要求编译器使用普通的函数匹配（第6.4节）来选择与我们提供的参数最匹配的构造函数。当我们使用复制初始化时，我们要求编译器将右边的操作数复制到正在创建的对象中，如果需要的话，转换该操作数（§ 7.5.4，p.294）。

将对象作为参数传递给非引用类型的形参

从具有非引用返回类型的函数返回对象

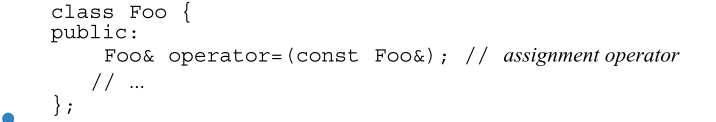
当使用花括号 {} 初始化数组元素或聚合类（Aggregate Class）的成员时，也会发生拷贝初始化。

MyClass obj1;

MyClass arr[] = {obj1, MyClass()};

复制构造函数用于初始化类类型的非引用参数这一事实解释了为什么复制构造函数自己的参数必须是引用。

然而，即使编译器省略了对复制/移动构造函数的调用，复制/移动构造函数也必须存在并且在程序中的该点必须是可访问的（例如，不是私有的）。



* **复制构造函数**：用于初始化一个新对象，函数签名是 ClassName(const ClassName& other)。
* **复制赋值运算符**：用于将一个对象的值赋给另一个已经存在的对象，函数签名是 ClassName& operator=(const ClassName& other)。

在构造函数中，成员在函数体执行之前被初始化，并且成员的初始化顺序与它们在类中出现的顺序相同。 在析构函数中，首先执行函数体，然后销毁成员。 成员按与初始化顺序相反的顺序销毁。

内置指针类型成员的隐式销毁不会删除该指针所指向的对象

与普通指针不同，智能指针（第 12.1.1 节，第 452 页）是类类型并具有析构函数。 因此，与普通指针不同，智能指针的成员在销毁阶段会自动销毁。

析构函数在以下情况下自动调用：

1. 变量离开作用域时。
2. 对象的成员销毁时。
3. 容器销毁时，容器中的元素也会销毁。
4. 动态分配的对象被 delete 时。
5. 临时对象在完整表达式结束时销毁。

当对象的引用或指针超出范围时，析构函数不会运行。.

当您决定一个类是否需要定义其自己的复制控制成员版本时，可以使用的一条经验法则是首先决定该类是否需要析构函数。 通常，对析构函数的需求比对复制构造函数或赋值运算符的需求更为明显。 如果类需要析构函数，那么它几乎肯定也需要复制构造函数和复制赋值运算符。

如果一个类需要析构函数，那么它几乎肯定还需要复制赋值运算符和复制构造函数。

如果一个类需要一个复制构造函数，那么它几乎肯定需要一个复制赋值运算符。 反之亦然——如果类需要赋值运算符，那么它几乎肯定也需要复制构造函数。

大多数类应该隐式或显式定义默认构造函数、复制构造函数以及复制赋值运算符。

**当类的某个成员无法被拷贝、赋值或销毁时，编译器会将合成的拷贝控制成员（如拷贝构造函数、拷贝赋值运算符、析构函数等）定义为删除的（deleted）**。

行为类似于值的类有自己的状态。 当我们复制一个类似值的对象时，副本和原始对象是相互独立的。 对副本所做的更改不会影响原始版本，反之亦然。

行为类似于指针的类共享状态。 当我们复制此类的对象时，副本和原始对象使用相同的基础数据。 对副本所做的更改也会更改原始版本，反之亦然。

关键概念：赋值运算符 编写赋值运算符时需要记住两点： • 如果对象被赋值给自身，则赋值运算符必须正确工作。 • 大多数赋值运算符与析构函数和复制构造函数共享工作。

编写赋值运算符时使用的一个好模式是首先将右侧操作数复制到本地临时值中。 复制完成后，可以安全地销毁左侧操作数的现有成员。 一旦左侧操作数被销毁，将数据从临时数据复制到左侧操作数的成员中。

即使将对象分配给自身，赋值运算符的正确工作也至关重要。 一个好方法是在销毁左侧操作数之前复制右侧操作数。

与复制控制成员不同，交换从来都不是必需的。 然而，定义交换对于分配资源的类来说可能是一个重要的优化。

* 自定义 swap 函数时，应该调用类型特定的 swap 函数，而不是直接调用 std::swap。
* 使用 using std::swap 允许编译器选择最合适的 swap 函数。
* 如果类的成员变量有自己的 swap 函数，调用这些特定的 swap 函数可以避免不必要的性能开销。
* 使用复制和交换的赋值运算符会自动实现异常安全并正确处理自赋值。

复制赋值运算符通常执行与复制构造函数和析构函数中所需的相同的工作。 在这种情况下，公共工作应该放在私有实用函数中。.

**移动构造函数**

* **作用**：将一个对象的资源“移动”到新创建的对象中。
* **语法**：

cpp

复制

ClassName(ClassName&& other) noexcept;

* **特点**：
  + 参数是一个右值引用（&&）。
  + 通常标记为 noexcept，表示不会抛出异常。
  + 移动后，原对象的资源被置为空（如指针置为 nullptr），以确保析构时不会释放资源。

**移动赋值函数**

* **作用**：将一个对象的资源“移动”到另一个已存在的对象中。
* **语法**：

cpp

复制

ClassName& operator=(ClassName&& other) noexcept;

* **特点**：
  + 参数是一个右值引用（&&）。
  + 通常标记为 noexcept，表示不会抛出异常。
  + 移动后，原对象的资源被置为空（如指针置为 nullptr），以确保析构时不会释放资源。

库容器、字符串和shared\_ptr 类支持移动和复制。 IO 和 unique\_ptr 类可以移动但不能复制。

新标准引入了一种新的引用，即右值引用。 右值引用是必须绑定到右值的引用。 右值引用是通过使用 && 而不是 & 获得的。 正如我们将看到的，右值引用有一个重要的属性，即它们只能绑定到即将被销毁的对象。 因此，我们可以自由地将资源从右值引用“移动”到另一个对象。

查看左值和右值表达式的列表，应该清楚左值和右值在一个重要方面彼此不同：

左值具有持久状态，而右值是在计算表达式过程中创建的文字或临时对象。 因为右值引用只能绑定到临时对象，所以我们知道：

• 所引用的对象即将被销毁

• 该对象不能有其他用户

变量是左值； 即使该变量被定义为右值引用类型，我们也不能直接将右值引用绑定到变量。

使用 move 的代码应该使用 std::move，而不是 move。 这样做可以避免潜在的名称冲突。

通知库的一种方法是在我们的构造函数上指定 noexcept。 我们将在第 18.1.4 节（第 779 页）中详细介绍新标准引入的 noexcept。 现在重要的是要知道 noexcept 是我们承诺函数不会抛出任何异常的一种方式。 我们在函数的参数列表之后指定 noexcept

不能抛出异常的移动构造函数和移动赋值运算符应标记为 noexcept。

**移动构造函数的工作原理**

* **接管资源**：移动构造函数通过成员初始化列表接管源对象 s 的资源（如指针 elements、first\_free 和 cap）。
* **置空源对象**：在构造函数体内，将源对象 s 的指针置为 nullptr，以确保源对象在析构时不会释放资源。

**标准库容器的异常安全保证**

* **std::vector 的 push\_back**：
  + 当 push\_back 需要重新分配内存时，它会尝试移动现有元素到新内存中。
  + 如果移动操作抛出异常，std::vector 会保证容器本身的状态不变。
* **noexcept 的作用**：
  + 如果移动操作标记为 noexcept，std::vector 会使用移动操作来转移元素。
  + 如果移动操作可能抛出异常，std::vector 会使用拷贝操作来转移元素，以确保异常安全。

移动操作后，“移出”对象必须仍然是有效的、可破坏的对象，但用户不能对其值做出任何假设。

与复制操作不同，对于某些类，编译器根本不综合移动操作。 特别是，如果类定义了自己的复制构造函数、复制赋值运算符或析构函数，则不会合成移动构造函数和移动赋值运算符。

仅当类未定义其自己的任何复制控制成员并且仅当所有数据成员均可分别进行移动构造和移动赋值时，编译器才会综合移动构造函数和移动赋值。

定义移动构造函数或移动赋值运算符的类还必须定义自己的复制操作。 否则，这些成员将被默认删除。

如果一个类有一个可用的复制构造函数但没有移动构造函数，则对象将被复制构造函数“移动”。 对于复制赋值运算符和移动赋值也是如此。

所有五个复制控制成员都应该被视为一个单元：通常，如果一个类定义了这些操作中的任何一个，那么它通常应该定义所有这些操作。 正如我们所看到的，某些类必须定义复制构造函数、复制赋值运算符和析构函数才能正常工作（第 13.1.4 节，第 504 页）。 此类类通常具有复制成员必须复制的资源。 通常，复制资源需要一定的开销。 在不需要复制的情况下，定义移动构造函数和移动赋值运算符的类可以避免这种开销。

在类实现代码（例如移动构造函数或移动赋值运算符）之外，仅当您确定需要执行移动并且保证移动安全时才使用 std::move。

区分移动和复制参数的重载函数通常有一个采用 const T& 的版本和一个采用 T&& 的版本。

**1. 左值引用成员函数**

* **左值引用成员函数**只能被左值对象调用。
* 通过在成员函数后添加 & 来限定该函数只能被左值对象调用。

**2. 右值引用成员函数**

* **右值引用成员函数**只能被右值对象调用。
* 通过在成员函数后添加 && 来限定该函数只能被右值对象调用。

如果成员函数具有引用限定符，则具有相同参数列表的该成员的所有版本都必须具有引用限定符。

每个类控制当我们复制、移动、分配或销毁其类型的对象时会发生什么。 特殊成员函数（复制构造函数、移动构造函数、复制赋值运算符、移动赋值运算符和析构函数）定义了这些操作。 移动构造函数和移动赋值运算符采用（通常是非常量）右值引用； 复制版本采用（通常是 const）普通左值引用。

如果类没有声明这些操作，编译器将自动定义它们。 如果未定义为已删除，则这些操作按成员初始化、移动、分配或销毁对象：依次采用每个非静态数据成员，合成操作会执行适合成员类型的任何操作来移动、复制、分配或销毁该成员 。

分配内存或其他资源的类几乎总是要求该类定义复制控制成员来管理分配的资源。 如果一个类需要析构函数，那么它几乎肯定需要定义移动和复制构造函数以及移动和复制赋值运算符。