如果运算符函数是成员函数，则第一个（左侧）操作数将绑定到隐式 this 指针（第 7.1.2 节，第 257 页）。 由于第一个操作数隐式绑定到此，因此成员运算符函数的参数比该运算符的操作数少一个（显式）参数。

通常，逗号、地址、逻辑 AND 和逻辑 OR 运算符不应重载。

当你设计一个类时，你应该首先考虑该类将提供哪些操作。 只有在知道需要哪些操作之后，您才应该考虑是将每个操作定义为普通函数还是重载运算符。 具有到运算符的逻辑映射的那些操作是定义为重载运算符的良好候选者：

• 如果类执行 IO，请定义移位运算符，使其与内置类型的 IO 执行方式保持一致。 • 如果类具有测试相等性的操作，请定义operator==。 如果类有operator==，它通常也应该有operator!=。

• 如果该类具有单个自然排序操作，请定义operator<。 如果类有operator<，它可能应该有所有的关系运算符。

• 重载运算符的返回类型通常应与内置版本的运算符的返回兼容：逻辑和关系运算符应返回 bool，算术运算符应返回类类型的值，赋值和复合运算符应返回 赋值应返回对左侧操作数的引用。

赋值运算符的行为应与合成运算符类似：赋值后，左侧和右侧操作数中的值应具有相同的值，并且运算符应返回对其左侧操作数的引用。

每个运算符在内置类型上的使用都有一个相关的含义。 例如，二进制 + 与加法密切相关。 将二进制 + 映射到类类型的类似操作可以提供方便的符号速记。 例如，库字符串类型遵循许多编程语言通用的约定，使用 + 来表示串联——将一个字符串“添加”到另一个字符串。

当存在内置运算符到我们类型上的操作的逻辑映射时，运算符重载最有用。 使用重载运算符而不是发明命名操作可以使我们的程序更加自然和直观。 过度使用或彻底滥用运算符重载可能会使我们的类难以理解。

在实践中很少发生明显滥用运算符重载的情况。 例如，没有负责任的程序员会定义operator+来执行减法。 更常见但仍然不可取的是，使用扭曲运算符的“正常”含义来强制适合给定类型。 运算符应仅用于用户可能明确的操作。 如果一个运算符似乎有不止一种解释，则该运算符的含义不明确。

* **必须定义为成员函数的运算符**：=, [], (), ->。
* **通常应定义为成员函数的运算符**：复合赋值运算符 (+=, -= 等)、自增 (++)、自减 (--)、解引用 (\*)。
* **通常应定义为非成员函数的运算符**：对称运算符 (+, -, ==, < 等)、流运算符 (<<, >>)。

在C++中，**是否返回一个新的对象**以及**是否修改当前对象的状态**是决定运算符应该定义为**类外非成员函数**还是**类内成员函数**的重要因素。

通常，输出运算符应该以最少的格式打印对象的内容。 他们不应该打印换行符。

输入操作员应该决定如何处理错误恢复（如果有的话）。

同时定义算术运算符和相关复合赋值的类通常应该使用复合赋值来实现算术运算符。

具有相等逻辑意义的类通常应该定义operator==。 定义 == 的类使用户可以更轻松地将类与库算法一起使用。

如果 < 存在单个逻辑定义，类通常应该定义 < 运算符。 但是，如果该类还具有 ==，则仅当 < 和 == 的定义产生一致的结果时才定义 <。

赋值运算符必须（通常复合赋值运算符应该）定义为成员。 这些运算符应返回对左侧操作数的引用。

如果一个类有下标运算符，它通常应该定义两个版本：一个返回普通引用，另一个是 const 成员并返回对 const 的引用。

定义自增或自减运算符的类应同时定义前缀和后缀版本。 这些运算符通常应定义为成员。

为了与内置运算符保持一致，后缀运算符应返回旧值（未递增或未递减）。 该值作为值而不是引用返回。

解引用 (\*) 和箭头 (->) 运算符通常用在表示迭代器的类和智能指针类中（第 12.1 节，第 450 页）。 我们也可以从逻辑上将这些运算符添加到 StrBlobPtr 类中：

操作员箭头必须是成员。 解引用运算符不需要是成员，但通常也应该是成员。

。 如果 point 是指针，则应用内置箭头运算符，这意味着该表达式是 (\*point).mem 的同义词。 指针被取消引用，并从结果对象中获取指示的成员。 如果point指向的类型没有名为mem的成员，那么代码是错误的。

2. 如果point是定义operator->的类的对象，则point.operator->()的结果用于获取mem。 如果该结果是一个指针，则对该指针执行步骤 1。 如果结果是一个本身具有重载运算符->() 的对象，则对该对象重复此步骤。 此过程将继续，直到返回指向具有指定成员的对象的指针或返回某个其他值，在这种情况下，代码将出错。

重载的箭头运算符必须返回指向类类型的指针或定义其自己的运算符箭头的类类型的对象。

函数调用运算符必须是成员函数。 一个类可以定义多个版本的调用运算符，每个版本的参数数量或类型都必须有所不同。

**1. Lambda表达式生成的类**

* 当你编写一个lambda表达式时，编译器会将其转换为一个**未命名的类**（类似于你提到的 SizeComp 类）。
* 这个类包含：
  + 一个**构造函数**，用于初始化捕获的变量。
  + 一个**调用运算符 (operator())**，其参数和返回类型与lambda表达式一致。
  + **数据成员**，用于存储lambda表达式捕获的变量。
* Lambda表达式会被编译器转换为一个类，该类包含：
  + 一个构造函数（用于初始化捕获的变量）。
  + 一个调用运算符（operator()）。
  + 数据成员（用于存储捕获的变量）。
* 生成的类没有默认构造函数和赋值运算符，但有一个默认的析构函数。
* 是否生成拷贝/移动构造函数取决于捕获的变量类型。
* 通过理解lambda表达式生成的类的行为，可以更好地使用lambda表达式和函数对象。

**4. 调用签名的重要性**

* 调用签名是std::function能够统一存储不同类型可调用对象的关键。
* 只要调用签名匹配，std::function就可以存储：
  + 普通函数
  + 函数指针
  + Lambda表达式
  + 函数对象（仿函数）
  + 绑定表达式（std::bind的结果）
  + 其他可调用对象

**1. const成员函数的作用**

* **const修饰的成员函数**：
  + 表示该函数不会修改类的任何成员变量（除非成员变量被声明为mutable）。
  + 在const成员函数中，所有成员变量都被视为const，即不可修改。

实际上，类很少提供转换运算符。 很多时候，如果转换自动发生，用户更有可能感到惊讶，而不是因为转换的存在而得到帮助。

转换为 bool 通常用于条件。 因此，bool 运算符通常应定义为显式的。

重载运算符必须是类的成员，或者至少具有一个类类型的操作数。 当应用于构建时，重载运算符具有与相应运算符相同的操作数数、结合性和优先级。

在类型中。 当运算符被定义为成员时，其隐式 this 指针将绑定到第一个操作数。 赋值、下标、函数调用和箭头运算符必须是类成员。

重载函数调用运算符operator()的类对象称为“函数对象”。 此类对象通常与标准算法结合使用。 Lambda 表达式是定义简单函数对象类的简洁方法。

类可以定义自动使用的与其类型之间的转换。 可以使用单个参数调用的非显式构造函数定义从参数类型到类类型的转换； 非显式转换运算符定义从类类型到其他类型的转换。