图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

1：构造函数声明为显示。

2：构造函数拷贝。

3：主要是为了深拷贝指针变量。



有新对象必须调用构造函数，无新对象赋值就行，不涉及构造函数。

文本

描述已自动生成

文本, 信件

描述已自动生成

文本, 信件

中度可信度描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

不要产生未定义的行为。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

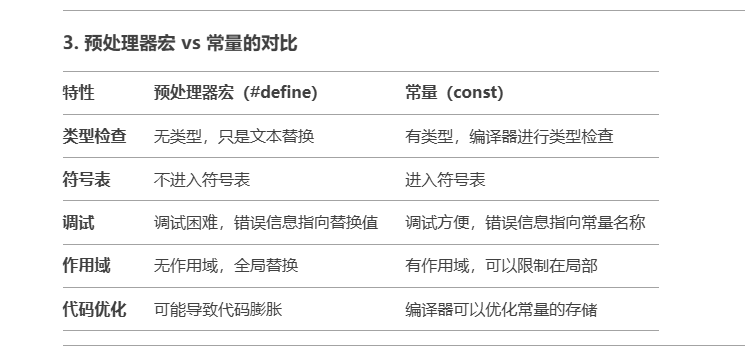
命名规范，指针通常是p包含大写字母。

一些文字和图片的手机截图

描述已自动生成

在c的模式上增加了对象，模版，标准库。







常量指针，指向和内容都不能改变。

文本

描述已自动生成

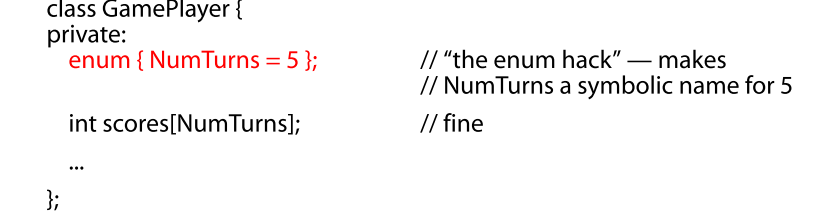
用string代替char文本

描述已自动生成

类中常量不能变，并且只有一个副本。

文本, 电子邮件

描述已自动生成



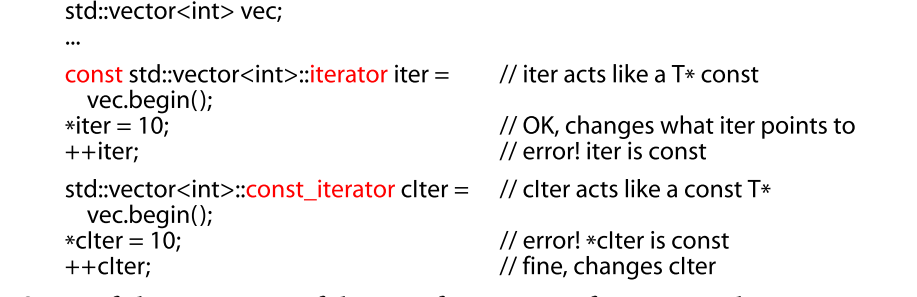
利用枚举，更像define，因为取地址非法。

文本

描述已自动生成

文本

描述已自动生成



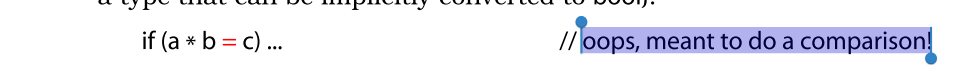
可以修改对象，可以修改指针。

图片包含 文本

描述已自动生成文本

低可信度描述已自动生成

人都要笑死了，暴行。



const Member Functions

文本, 应用程序

中度可信度描述已自动生成

提供不同的重载方式，

文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

# 确保对象在使用之前已经初始化

图形用户界面, 文本, 应用程序

描述已自动生成

**利用成员初始化列表构造。**

**赋值构造的话，会先调用默认构造函数**

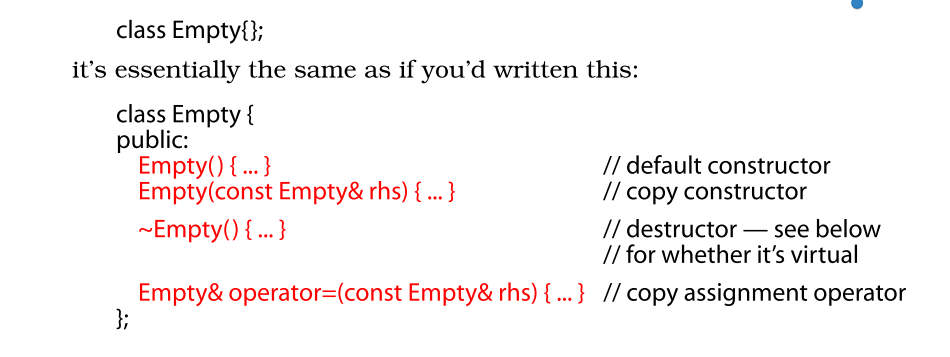
**始终按照声明来初始化**

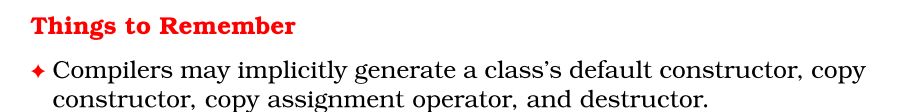
文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成





**编译器可以隐式生成类的默认构造函数、复制构造函数、复制赋值运算符和析构函数。**

# explanatory不允许使用编译器生成的函数。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

**私有声明构造函数，防止外人调用，同时不需要写函数名，不用实现。**

图形用户界面, 文本, 应用程序, 信件

描述已自动生成

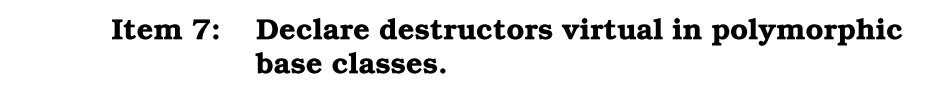
图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

**私有化编译器提供的功能，赋值和copy。这样防止乱用。**



图形用户界面, 文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

**任何有虚函数的类，都应该有一个虚析构函数。**

**如果一个类不包含虚函数，这通常表明它不打算用作基类。当一个类不打算作为基类时，使析构函数为虚通常是一个坏主意**

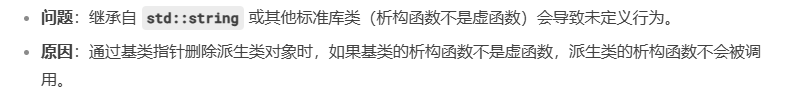
**虚函数的实现要求对象携带可在运行时用于确定应在对象上调用哪些虚函数的信息。这些信息通常采用称为vptr（“虚拟表指针”）的指针的形式。vptr指向一个称为vtbl（“虚拟表”）的函数指针数组;每个具有虚拟函数的类都有一个关联的vtbl。当对对象调用虚函数时，通过跟随对象的vptr到达vtBL，然后在vtBL中查找适当的函数指针来确定调用的实际函数。**

**在32位体系结构上，它们将从64位（对于两个int）到96位（对于int加上vptr）;在64位体系结构上，它们可能从64位到128位，因为这种体系结构上的指针大小为64位。**

**事实上，许多人都这样总结这种情况：当且仅当类包含至少一个虚函数时，才在类中声明虚析构函数。**

**有虚函数，就代表有继承，然后就要有虚析构函数。**

**添加虚析构函数会增加内存消耗。**



**(e.g., vector,**

**list, set,**

**tr1::unordered\_map (see Item 54), etc.).**

**这些都不要继承，没有虚析构函数。**

图形用户界面, 文本, 信件, 电子邮件

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

**多态基类应该声明虚析构函数。如果一个类有任何虚函数，它应该有一个虚析构函数。类不应该被设计为基类或不应该被设计为多态使用，不应该声明虚析构函数。**

**、**图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

**析构函数不应该发出异常。如果在析构函数中调用的函数可能抛出异常，析构函数应该捕获任何异常，然后吞下它们或终止程序。如果类客户端需要能够对操作期间抛出的异常做出反应，则类应该提供常规（即，非析构函数（non-destructor）执行该操作。**文本

中度可信度描述已自动生成第9条：在构造或销毁过程中永远不要调用虚函数

**因为基类构造函数在派生类构造函数之前执行，所以在基类构造函数运行时派生类数据成员尚未初始化。如果在基类构造期间调用的虚函数向下到派生类，则派生类函数几乎肯定会引用本地数据成员，但这些数据成员尚未初始化。**

文本

描述已自动生成

文本

描述已自动生成

**由于在构造过程中不能使用虚函数从基类向下调用，因此可以通过让派生类将必要的构造信息传递给基类构造函数来进行补偿。**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本

描述已自动生成

**不要在构造或析构过程中调用虚函数，因为这样的调用永远不会转到比当前执行的构造函数或析构函数更派生的类**



**让赋值运算符返回对 \*this的引用**

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图片包含 图形用户界面

描述已自动生成

**赋值运算符返回当前对象的引用。**



# 第11项：在operator=中处理对self的赋值。

文本

描述已自动生成

确保operator=在对象被赋给自身时行为良好。这些技术包括比较源对象和目标对象的地址、仔细的语句排序以及复制和交换。如果两个或多个对象相同，请确保在多个对象上操作的任何函数都能正确运行



# 第12项：复制对象的所有部分。

本项目标题中“复制所有部分”的含义现在应该清楚了。当你写一个复制函数时，一定要（1）复制所有的本地数据成员，（2）在所有的基类中调用适当的复制函数。

文本

描述已自动生成

文本

描述已自动生成.复制函数应该确保复制对象的所有数据成员及其所有基类部分。不要试图根据另一个来实现一个复制功能。相反，将公共功能放在第三个函数中，两者都调用。

当我们定义一个引用时，我们不是复制初始化器的值，而是将引用绑定到它的初始化器。初始化后，引用保持绑定到其初始对象。没有办法重新绑定一个引用来引用不同的对象。因为没有办法重新绑定引用，所以必须初始化引用。

引用不是对象。相反，引用只是一个已经存在的对象的另一个名称。

定义引用后，对该引用的所有操作实际上都是对该引用绑定到的对象的操作：

当我们赋值给一个引用时，我们是在赋值给该引用所绑定的对象。当我们获取一个引用的值时，我们实际上是在获取该引用所绑定的对象的值。

除了我们将在§ 2.4.1（p.61）和§ 15.2.3（p.601）中介绍的两个例外，引用的类型和引用所引用的对象必须完全匹配。此外，由于我们将在第2.4.1节中探讨的原因，引用可以仅绑定到对象，而不是文本或更一般表达式的结果

图片包含 文本

描述已自动生成

指针是一种“指向”另一种类型的复合类型。与引用一样，指针用于间接访问其他对象。与引用不同，指针本身就是一个对象。指针可以被赋值和复制;单个指针可以在其生命周期内指向多个不同的对象。与引用不同，指针不需要在定义时初始化。与其他内置类型一样，在块作用域定义的指针如果没有初始化，则具有未定义的值。

复制

指针通常很难理解。由于指针错误引起的编译问题甚至使有经验的程序员也感到苦恼

除了我们在§ 2.4.2（p.62）和§ 15.2.3（p.601）中介绍的两个例外，指针和它所指向的对象的类型必须匹配：

复制或尝试访问无效指针的值是错误的。当我们使用一个未初始化的变量时，这个错误是编译器不太可能检测到的。访问无效指针的结果是未定义的。因此，我们必须始终知道给定的指针是否有效。

当一个指针指向一个对象时，我们可以使用解引用操作符（\* 操作符）来访问该对象：

\*在c++中有不同的含义，可以代表这是一个指针，也可以代表解引用。&可以代表引用，也能代表取地址对吗

有些符号，如&和 \*，既用作表达式中的运算符，也用作声明的一部分。使用符号的上下文决定了符号的含义：

在声明中，&和 \* 用于形成复合类型。在表达式中，这些相同的符号用于表示运算符。

最直接的方法是使用新标准引入的文本nullptr初始化指针。nullptr是一个具有特殊类型的文字，可以转换（第2.1.2节）为任何其他指针类型。

现代C++程序通常应该避免使用NULL，而使用nullptr

建议：初始化所有指针

与其他未初始化的变量一样，当我们使用未初始化的指针时会发生什么是未定义的。使用未初始化的指针几乎总是会导致运行时崩溃。然而，调试由此产生的崩溃可能会令人惊讶地困难。

在大多数编译器下，当我们使用未初始化的指针时，指针所在的内存中的位被用作地址。使用一个未初始化的指针是一个访问假定位置上的假定对象的请求。没有办法区分一个有效的地址和一个无效的地址，无效的地址是由碰巧在分配指针的内存中的位形成的。

我们建议初始化所有变量，这对于指针尤其重要。如果可能的话，只在定义了它应该指向的对象之后才定义指针。如果没有对象绑定到指针，则将指针初始化为nullptr或零。这样，程序就可以检测到指针没有指向对象。

需要记住的重要一点是赋值会改变它的左操作数

类型void\* 是一种特殊的指针类型，可以保存任何对象的地址。像任何其他指针一样，void\* 指针持有一个地址，但该地址处的对象类型是未知的：

我们不能使用void\* 对它所寻址的对象进行操作--我们不知道该对象的类型，而类型决定了我们可以对该对象执行什么操作。

理解复合类型声明

正如我们所见，变量定义由基类型和声明符列表组成。每个声明符都可以将其变量与基类型相关联，这与同一定义中的其他声明符不同。因此，单个定义可以定义不同类型的变量：

int \*p1, \*p2; // both p1 and p2 are pointers to int

将变量和声明符放在一起。

声明类型符是可以叠加的。

我们用它自己的 \* 来表示每个指针级别。也就是说，我们写 \*\* 表示一个指向一个指针的指针，\* 表示一个指向一个指针的指针，等等：.

int \*&r = p;

理解r类型的最简单方法是从右向左阅读定义。最接近变量名的符号（在本例中是& in &r）是对变量类型有最直接影响的符号。

了支持这种用法，同时避免同一变量的多个定义，const变量被定义为文件的本地变量。当我们在多个文件中定义具有相同名称的const时，就好像我们在每个文件中为单独的变量编写了定义。

因为我们不能在创建const对象后改变它的值，所以它必须被初始化。

要在多个文件之间共享const对象，必须将变量定义为extern。

因为我们不能直接赋值给ci，所以我们也不能使用引用来改变ci。

文本

描述已自动生成

重要的是要认识到，对const的引用只限制了我们通过该引用可以做的事情。将对const的引用绑定到一个对象并不能说明底层对象本身是否是const

将指针定义为指向const的指针只影响我们可以使用指针做什么。重要的是要记住，不能保证const指针指向的对象不会改变

将const的指针和引用看作“认为它们指向或引用const”的指针或引用可能会有所帮助。

一个常量，定义的时候，必须初始化。

**定义**：顶层 const 表示对象本身是常量，不能被修改

图形用户界面, 文本

描述已自动生成

* **定义**：底层 const 表示指针或引用所指向的对象是常量，不能通过指针或引用修改对象的值。

文本

描述已自动生成

另一方面，低级const永远不会被忽略。当我们复制一个对象时，两个对象必须有相同的低级别const限定，或者两个对象的类型之间必须有转换。一般来说，我们可以将nonconst转换为const，但不能反过来

常量表达式是其值不能更改并且可以在编译时计算的表达式。字面量是一个常量表达式。

一般来说，对于要用作常量表达式的变量，最好使用constexpr

文本

中度可信度描述已自动生成

文本

描述已自动生成

每当更新头文件时，必须重新编译使用该头文件的源文件，以获取新的或更改的声明。

读者应该意识到，在编译之前，他们必须在我们的示例中添加适当的#include和using声明

vector是一个模板，而不是类型。从vector生成的类型必须包括元素类型，例如vector<int>。

文本, 信件

描述已自动生成

（）代表一组，{}代表几个。

文本, 信件

描述已自动生成