基础议题

Item 1 仔细区别pointers和references

- 在语言的表现形式上，pointers和references是不一样的，但是从比较深层的东西来说，他们似乎又是一样的东西（都是内存地址完成传递）。
- pointers可以指向空指针，但是references必须绑定一个左值。通常这会带来两个问题
  - pointers会要求使用它的函数（方法）对其进行`nullptr`的测试，这可能会带来编码上的很大麻烦。但是pointers可以随时随地转移绑定的对象。
  - references不允许转移绑定的对象，转移操作会被认为是对原来对象的修改。但是这也可以减少编码的复杂度，因为不需要更多的分类讨论。
- `operator[]`通常返回一个reference，而不是pointer，因为在大多数情况下，它更符合数组的行为。

Item 2 最好使用C++转型操作符

- C语言的转型操作在C++种仍然成立，因为C++喜欢普渡众生。
  - 但是C++的普渡众生隐性给程序员的标准提升了一大截，因为大多数人会在抉择上花费很多时间，而且，看出转换背后的东西总是费神。
- 在彻底理解四大转型之前，需要了解一些名词。
  - cv限定：`const`和`volatile`作为变量类型的限定词。`const`绝大部分人都比较了解，但是`volatile`不是很了解。实际上`volatile`作为一种不准优化的动作，减少编译器的优化，来确保对于不稳定接口的正确性或者多线程下的正确性。

四大转型

`static_cast`

`static_cast`主要用于良性转换，这种转换的危险性不大。

- 内置类型转换、向上兼容、指针向下兼容、`enum`转换，添加cv限定。
- `void*`转其他指针，注意其他指针是不可以互转的（自己转自己除外）。
- 构造类型专门指定转换方法也可以。比如`Complex`转`double`（也有`asDouble`骚操作）或者`vector<T>`的构造函数有单独的`int`。
- 弃值表达式。`static_cast<void>(v.size())`

`const_cast`

下述转换能用`const_cast`执行，特别是，只有`const_cast`才可以去掉cv限定`。

- 两个指向**同一类型**的指针可以互相转换，cv限定对转换结果无关。指针可以是多层的。
- `T`类型**左值**可以转换为**同一类型**`T`的**左值或右值引用**，cv限定符可以修改。如果是**右值**则转换为**右值引用**。
- 空指针可以随意转换到其他类型。
- 对于数据成员的指针也是一样的规则。请注意，不在成员函数的指针上工作。

`dynamic_cast`

下述转换可以使用`dynamic_cast`，如果要求去除cv限定则无法转换（编译报错）。如果编译通过，转型失败，新类型是指针类型，那么返回。

- 和`static_cast`一样，这种方法可以添加cv限定。
- 如果表达式是空指针，那么可以转换。
- 继承体系中的向上继承。和隐式转换、`static_cast`一样。请注意，即便指针是有问题的也会强制按照新类型来理解，如果其中存在虚函数，找不到虚函数表可不要怪编译器，因为这说明你之前犯下的错误非常大。
- 继承体系中的向下继承。借助RTTI进行检测，确定安全的才会成功，否则失败。
    - 对于成功的定义：如果存在$A \to B \to C$这样的链条，那么

```
A* pa=new C();
// A* pa=new A(); 这样的代码不可以
dynamic_cast<B*>(pa);
```

    这样的代码是可以运行的

`reinterpret_cast`

在某种程度上这是一种非常不安全的转型，因为这玩意可以把一个类型的东西强制理解成另一个类型，而且在不同平台上的处理是不一样的，因此绝大部分应用层面都在其他三种转换不能处理的情况，比如函数指针。

Item 3 绝对不要以多态处理数组

- 从某种程度上说，数组是按照指针的方式来处理数据的，在存在$A \to B \to C$中，如果把B类型数组作为传入参数，结果拿到了C的数组，那么在数组运作的时候，如果选择`tmp[i]`，那么极有可能会指向一个内存错误。
- 这里指明了一个绝对真理：不要对自己不知道的东西随意折腾，在这里，每一个元素的大小和类的继承关系就是你不知道的东西。

Item 4 非必要不提供default constructor

关于一个类是否有必要提供默认构造函数，这里一般会有两派进行争论。

数组初始化

数组初始化，一般来说定义数组通常是`T A[10]`，如果类型`T`不提供初始化函数的话，那么这里就会提供一个编译错误。反过来如果一定不需要默认构造函数的话，一般会带来很多编码上的麻烦，或者需要用指针来完成这一系列操作。

关于数组初始化的问题，如果不需要对`template`类初始化，可以考虑使用`operator new`进行操作，之后在进行placement new的初始化方式。

但是这会带来一个比较大的问题，就是通过`operator new`进行初始化的对象不能通过`delete`简单的清理掉，而是必须使用`operator delete`，这对于知识储备有一定要求。

模板类

在模板类中，如果不提供默认构造函数，那么会带来很神奇的景象：

```
template<class T>
class Array
```

```
{
public:
    Array(int size);
    //...
private:
    T* data;
    //...
};
Array<T>::Array(int size)
{
    data=new T[size];
    //...
}
```

这不还是数组初始化的问题吗？但是问题在包装一层以后就会产生新的问题。相对而言，STL模板库的设计会相对严谨一些，当然很多大型企业的开源库也是为了解决STL的痛处进行的。

> 很多templates的设计什么都有，独缺谨慎。

虚基类

如果虚基类不存在默认构造函数，那么感受简直是吃屎一样。虚基类的构造函数自变量必须要求产生对象的派生层次最深的类提供，但是实际上，你永远不知道你的是不是最后一层。

设计上的麻烦

和指针可以nullptr一样，默认构造函数相当于允许一个不合法的对象存在，这对于设计会带来一个比较大的问题，毕竟能少一层if就能少一些读代码的时间，少一点代码执行的时间，也少一点程序打包的空间。