基础议题

该部分内容主要涉及到了以下几个议题：

- pinters
- reference
- casts
- arrays
- constructors

条款1：仔细区别 pointer 和 reference

pointer 和 reference 看起来不同，但是做着类似的事情——间接参考其他对象。在 pointer 和 ref- erence 之间进行选择需要被仔细考虑。

1. 没有 null reference

这一点意味着，一个 reference 必须总是用来代表将某个对象，而 pointer 没有这点要求，可以指向空。

- 如果用来指向某个对象或者不指向任何对象——使用 pointer
- 如果总是必须代表一个对象——使用 reference

> [!warning] 拒绝以任何形式使得 reference 指向空，例如：

```
char *pc = 0;
char &rc = *pc;
```

没有 null reference 也代表着另外的两件事：

> [!tips] reference 必须有初值

```
    string &rc; // ! 错误，需要设定初值
    string s("hello");
    string &rc = s; // * 正确
```

> [!tips] reference 效率更高，因为不用检查是否有效

```
void printDouble(const double &rd)
{
  std::cout << rd << std::endl;
}
```

```
void printDouble(const double *pd)
{
  if (pd)
    std::cout << pd << std::endl;
}
```

2. reference 总是指向最初的那个对象

pointers 可以被重新赋值，指向另一个对象，而 reference 不可以。

```
std::string s1("Nancy");
std::string s2("Clancy");

std::string &rs = s1;
std::string *ps = &s1;

ps = &s2; // ps 指向了另一个对象
std::cout << "s1: " << s1 << std::endl; // 原有的值没有变化
std::cout << "s2: " << *ps << std::endl;
rs = s2; // rs 所代表的对象存储的值被 s2 内容替换
std::cout << "s1: " << s1 << std::endl; // s1 的值被改变了
```

3. 必须返回某种“能够被当作” assignment 赋值对象的东西时选择 reference

最常见的例子是 operator[] 需要返回 reference，否则返回 pointer 会产生下列的语句：

```
vector<int> v(10);
*v[5] = 10;
```

条款2：最好使用 c++ 转型操作符

相较于旧式的 c 转型方式，c++ 转型方案更加精确和明了。

> [!info] 旧式的转型方案有两点问题：
>
> 1. 无法区分不同的转型(const-to-non-const/base-to-derived)
> 2. 使用()，在代码中难以辨识

为此，c++ 引入了 4 个新的转型操作符：

- `static_cast`
- `const_cast`
- `dynamiv_cast`
- `reinterpret_cast`

static_cast

static_cast 基本上和 c 的旧式转型相同，主要用于 **不涉及继承机制** 的类型执行转型动作。

```
int a;
static_cast<double>(a);
```

const_cast

const_cast 用来改变某个对象的**常量性**。

```
void update(int *p);
int a;
const int *pa = &a;
update(const_cast<int *>(pa));
```

dynamic_cast

dynamic_cast 主要用来实现**继承体系**中的安全 **"向下转型 或 跨系转型动作"**。即，将base class objects 转型为 derived 或者 sibling base class object。**dynamic_cast 无法对 缺乏虚函数的类型进行转型**

```
class Widget {}
class SpecialWidget : public Widget {};

void update(SpecialWidget *p);

Widget *pw;
update(dynamic_cast<SpecialWidget *>(pw));
```

reinterpret_cast

reinterpret_cast 总是和编译平台息息相关，因此不具备移植性。它可以将"函数指针"类型进行转 换。这种转换过程并不总是结果正确的，应该尽量避免使用这种转型方案。

```
typedef void (*FuncPtr)(); // 定义一个 函数指针
FuncPtr funcPtrArray[10];

int func();
funcPtrArrary[0] = &func; // 错误，因为类型不对
funcPtrArrary[0] = reinterpret_cast<FuncPtr>(&func); // 正确，强制编译器理解
```

旧式实现

如果无法使用新式转型动作，可以使用旧式转型方案来实现一些简单但无安全性可言的转型函数来增强程序的可维护性。

```
#define static_cast(TYPE, EXPR) ((TYPE) (EXPR))
#define const_cast(TYPE, EXPR) ((TYPE) (EXPR))
#define reinterpret_cast(TYPE, EXPR) ((TYPE) (EXPR))
```

条款3：绝对不要以多态方式处理数组

继承的一个特性是：通过指向 base class 的 pointer 或 reference，可以操作 derived class 对象。这种行为被称为多态。

但是多态不能同**指针算术**混用，因此也不应当和数组进行混用。因为数组总会运用到指针算术。

```
class BST {};
class BalancedBST {};

void printBSTArray(ostream &os, const BST array[], int num) {
  for (int i = 0; i < num; ++i) {
    s << array[i];
  }
}

BalancedBST array[10];
printBSTArray(cout, array, 10);
```

上述代码会出现问题，因为"指针算术表达式"的含义是对指针进行后移，而后移的距离与数组中对象的大小有关系。当 BST 与 BalancedBST 的对象大小不一致时就会发生错误。但编译时，这并不会被检查出来。同样 的情况在 数组的删除过程中 也同样会存在，因为实际执行的是循环调用每个对象的 destructor。

想要避免 多态方式处理数组 的一个简单手段是，不要让 "具体类继承另一个具体类"，类似于 BST 和 BalancedBST。

条款4：非必要不提供 default constructor

如果实现的对象不能进行 "从无到有" 的转变，则不应该提供一个 default constructor。**添加无意义的 default constructor 会影响 class 的效率。**

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
}
```

对于上面这个类，显然不应该存在 ID 为空的仪器识别码，因此不提供 default constructor。但这种行为会造成其他的问题。

1. 无法产生数组

```
EquipmentPiece bestPieces[10];
EquipmentPiece *pBestPieces = new EquipmentPiece[10];
```

上面这两种行为都是错误的，因为这必将调用 EquipmentPiece() 这个构造函数，而由于在实现过程中没 有使用 default constructor 所以会出问题。

想要解决这个束缚，存在3个方法。

a. 使用 non-heap 数组

```
int ID1, ID2, ID3, ID4;
EquipmentPiece bestPieces[] = {
  EquipmentPiece(ID1),
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  EquipmentPiece(ID4)
}
```

这种方法的问题在于无法延伸至 heap 数组，也就是说无法使用 new。

b. 使用 “指针数组” 而非 “对象数组”

```
typedef EquipmentPiece* PEP;

PEP bestPieces[10];
PEP *pBestPieces = new PEP[10];

for (int i = 0; i < 10; ++i) {
  bestPieces[i] = new EquipmentPiece(ID Number);
}
```

这种方法的缺点在于：

- 必须记得将所有对象使用 delete 删除，否则会发生 resource leak 问题。
- 需要的内存总量比较大，因为需要一些空间来放置指针，还需要一些空间来放置 对象。

c. 使用 placement new 来避免过度使用内存

```
// 1. 使用 raw memory 进行内存分配
void *rawMemory = operator new[](10 * sizeof (EquipmentPiece));

// 2. 让 bestPieces 指向这块内存空间
EquipmentPiece *bestPieces = static_cast<EquipmentPiece*>(rawMemory);

// 3. 利用 placement new 来构造
```

```
for (int i = 0; i < 10; ++i) {
  new (&bestPieces[i]) EquipmentPiece(ID Number);
}
```

placement new 的缺点在于：

- 大部分程序开发者不熟悉，维护比较困难
- 需要手动调用 destructor 并最终使用 operator delete[] 方式释放 raw memory，而非普通的 delete [] 方式

```
for (int i = 0; i > -1; --i) {
  bestPieces[i].~EquipmentPiece();
}

operator delete[](rawMemory);
```

不能使用`delete [] bestPieces`，因为 bestPieces 并非由 new 直接创建。

2. 可能将不适用于许多 template-based container

因为大多数 templates 内几乎总是会产生一个以 “template 类型参数” 作为类型而架构起来的数组。 通过谨慎设计 template 可以避免这类事情的发生，但是需要时间来理解这个过程。

是否应该提供 default constructors

尽管不提供 default constructors 会带来许多束缚，但是添加无意义的 default constructors 会影响 classes 的效率。因为这意味着，使用者需要对某些内容进行测试以确定对象的合法性。

因此，应当谨慎的提供 default constructors，非必要，不要提供无意义的 default constructors。