Recitation of C++ Part

felomid

`iostream`

`std::cin` & `std::cout`

`std::cin` 和 `std::cout` 是定义在 `<iostream>` 中的两个**对象**，分别表示标准输入流和标准输出流。

`std::endl;`

- `std::endl` 是一个“操纵符”。`std::cout << std::endl` 的含义是**输出换行符，并刷新输出缓冲区**。

如果你不手动刷新缓冲区，`std::cout` 自有规则在特定情况下刷新缓冲区。（C `stdout` 也是一样）

`std::string`

字符串的连接

`+` 两侧至少有一个得是 `std::string` 对象, `+` 是**左结合**的.

引用“reference”

引用即别名

引用必须初始化（即在定义时就指明它绑定到谁），并且这个绑定关系不可修改。

引用只能绑定到实际的对象（左值）

变量、数组、指针等都是实际存在的对象，但“引用”本身只是一个傀儡，是一个实际存在的对象的“别名”。

- 不能定义绑定到引用的引用。
- 也不能定义指向引用的指针，因为指针也必须指向实际的对象。

传引用参数 (pass-by-reference)

可以定义绑定到数组的引用：

```
void print_array(int (&array)[10]) {
  for (int x : array)           // 基于范围的 for 语句可以用来遍历数组!
    std::cout << x << ' ';
  std::cout << std::endl;
}
int a[10], ival = 42, b[20];
print_array(a);     // Correct
print_array(&ival); // Error!
print_array(b);     // Error!
```

这个 `array` 真的是（绑定到了）一个数组，而不是指向数组首元素的指针。

- `array` 只能绑定到 `int[10]`，其它牛鬼蛇神都会 Error（C 做不到这一点）。
  而这意味着不会退化成 `int *`.

传引用参数 (pass-by-reference)

以**引用**的方式传参，避免拷贝：如果传入一个很长的`std::string`或者`std::vector`类型，拷贝将效率很低。

注意:如果参数定义为引用，这意味着你在调用函数的时候并不能在对应的实参处传入**字面值**!

但这样有问题：

```
print_upper("Hello"); // Error
```

需要注意！

引用是实际对象的别名，所以引用也是左值

但是引用不是对象！！！因为它仅仅是一个别名，没有对应的具体对象

Reference-to-`const`

类似于“指向常量的指针”（即带有“底层 `const` ”的指针），我们也有“绑定到常量的引用”

一个 reference-to-`const` **自认为自己绑定到** `const` **对象**，所以不允许通过它修改它所绑定的对象的值，也不能让一个不带 `const` 的引用绑定到它。（不允许“去除底层 `const` ”）

对于一个`const`类型的变量，只能用`const &`的引用来绑定它（**不能去除底层**const）

Reference-to-`const`

特殊规则：常量引用可以绑定到右值：

```
const int &cref = 42; // Correct
int fun();
const int &cref2 = fun(); // Correct
int &ref = fun(); // Error
```

当一个常量引用被绑定到右值时，实际上就是让它绑定到了一个临时对象。

- 这是合理的，反正你也不能通过常量引用修改那个对象的值

Pass-by-reference-to-`const` 非常重要

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
print_upper("Hello"); // Error
```

当我们传递 `"Hello"` 给 `std::string` 参数时，实际上发生了一个由 `const char [6]` 到 `std::string` 的**隐式转换**，这个隐式转换产生**右值**，无法被 `std::string&` 绑定。

```
const std::string s = "hello";
print_upper(s); // Error: Casting-away low-level const
```

不带底层 `const` 的引用无法绑定到 `const` 对象。

Pass-by-reference-to-`const`

将参数声明为**常量引用**，既可以避免拷贝，又可以允许传递右值，也可以传递常量对象，也可以**防止你不小心修改了它**。

在 C++ 中声明函数的参数时，**尽可能使用常量引用**（如果你不需要修改它）。

（如果仅仅是 `int` 或者指针这样的内置类型，可以不需要常量引用）

真正的“值类别”

（语言律师需要掌握）

C++ 中的表达式依值类别被划分为如下三种：

英文	中文	has identity?	can be moved from?
lvalue	左值	yes	no
xvalue (expired value)	亡值	yes	yes
prvalue (pure rvalue)	纯右值	no	yes

lvalue + xvalue = glvalue（广义左值），xvalue + prvalue = rvalue（右值）

- 所以实际上“左值是实际的对象”是不严谨的，右值也可能是实际的对象（xvalue）

`std::vector`

定义在标准库文件 `<vector>` 中

真正好用的“动态数组”

C++17 CTAD

Class Template Argument Deduction：只要你给出了足够的信息，编译器可以自动推导元素的类型！

```
std::vector v{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42); // vector<int>
std::vector v4(10); // Error: cannot deduce template argument type
```

清空 `std::vector`

`v.clear()`。不要写愚蠢的 `while (!v.empty()) v.pop_back();`

`v.back()` 和 `v.front()`

分别获得最后一个元素、第一个元素的**引用**。

`v.back()`，`v.front()`，`v.pop_back()` 在 `v` 为空的情况下是 undefined behavior，而且实际上是**严重的运行时错误**。

`std::vector` 的增长策略

假设现在有一片动态分配的内存，长度为 `i`。

当第 `i+1` 个元素到来时，朴素做法：

1. 分配一片长度为 `i+1` 的内存
2. 将原有的 `i` 个元素拷贝过来
3. 将新的元素放在后面
4. 释放原来的那片内存

但这需要拷贝 `i` 个元素。`n` 次 `push_back` 总共就需要 $\sum_{i=0}^{n-1} i = O\left(n^2\right)$ 次拷贝！

`std::vector` 的增长策略

假设现在有一片动态分配的内存，长度为 `i`。

当第 `i+1` 个元素到来时，

1. 分配一片长度为 `2*i` 的内存
2. 将原有的 `i` 个元素拷贝过来
3. 将新的元素放在后面
4. 释放原来的那片内存

而当第 `i+2`，`i+3`，…，`2*i` 个元素到来时，我们不需要分配新的内存，也不需要拷贝任何对象！

`std::vector` 的增长策略

$0 \to 1 \to 2 \to 4 \to 8 \to 16 \to \cdots$

假设 $n = 2^m$，那么总的拷贝次数就是 $\sum_{i=0}^{m-1} 2^i = O(n)$，**平均**（“均摊”）一次 `push_back` 的耗时是 $O(1)$（常数），可以接受。

使用 `v.capacity()` 来获得它目前所分配的内存的实际容量，看看是不是真的这样。

- 注意：这仅仅是一种可能的策略，标准并未对此有所规定。

* **看看 hw4/prob2** `attachments/testcases/vector.c`

`std::vector` 动态增长带来的影响

我们已经看到，改变 `vector` 的大小可能会导致它所保存的元素“搬家”，这会使得所有指针、引用、迭代器失效。

不要在用基于范围的 `for` 语句遍历容器的同时改变容器的大小！ (undefined behavior)

`new` 和 `delete`

`new` 表达式

动态分配内存，**并构造对象**

```
int *pi1 = new int;      // 动态创建一个默认初始化的 int
int *pi2 = new int();    // 动态创建一个值初始化的 int
int *pi3 = new int{};    // 同上，但是更 modern
int *pi4 = new int(42); // 动态创建一个 int，并初始化为 42
int *pi5 = new int{42}; // 同上，但是更 modern
```

需要注意的是，`new [0]` 是会分配内存的。（implementation-defined）

对于内置类型：

- **默认初始化** (default-initialization)：就是未初始化，具有未定义的值
- **值初始化** (value-initialization)：类似于 C 中的“空初始化”，是各种零。

`delete` 和 `delete[]` 表达式

销毁动态创建的对象，并释放其内存

- `new` 必须对应 `delete`，`new[]` 必须对应 `delete[]`，否则是 **undefined behavior**
- 忘记 `delete`：内存泄漏

一一对应，不得混用

违反下列规则的一律是 undefined behavior:

- `delete ptr` 中的 `ptr` 必须等于某个先前由 `new` 返回的地址
- `delete[] ptr` 中的 `ptr` 必须等于某个先前由 `new[]` 返回的地址
- `free(ptr)` 中的 `ptr` 必须等于某个先前由 `malloc`, `calloc`, `realloc` 或 `aligned_alloc` 返回的地址。

`new` 和 `delete` 的具体行为

- `new` / `new[]` 表达式会**先分配内存，然后构造对象**。对于类类型的对象，它可能会调用一个合适的**构造函数**。
- `delete` / `delete[]` 表达式会**先销毁对象，然后释放内存**。对于类类型的对象，它会调用**析构函数**。
  - 如果是nullptr，首先不会调用析构函数，其次释放内存是implementation defined(如果调用了，保证什么事情都不做)

Move 移动

Motivation: copy is slow

```
s = a + b;
```

1. Evaluate `a + b` and store the result in a temporary object, say `tmp`.
2. Perform the assignment `s = tmp`.
3. The temporary object `tmp`

Can we make this faster ?

- The assignment `s = tmp` is done by **copying** the contents of `tmp`.
- But `tmp` is about to die! Why can't we just *steal* contents from it?

Distinguish between the different kinds of assignments

```
s = a;
```

```
s = a + b;
```

What is the key difference between them?

- `s = a` is an assignment from an **lvalue**,
- while `s = a + b` is an assignment from an **rvalue**.

If we only have the copy assignment operator, there is no way we can distinguish them.

* **Define two different assignment operators, one accepting an lvalue and the other accepting an rvalue?**

`std::move`

`std::move`

Defined in `<utility>`

`std::move(x)` performs and **lvalue to rvalue cast**:

```
int ival = 42;
int &&rref = ival; // Error
int &&rref2 = std::move(ival); // Correct
```

Calling `std::move(x)` tells the compiler that:

- `x` is an lvalue, but
- we want to treat `x` as an **rvalue**.

`std::move`

`std::move(x)` indicates that we want to treat `x` as an **rvalue**, which means that `x` will be *moved from*.

The call to `std::move` **promises** that we do not intend to use `x` again,

- except to assign to it or to destroy it.

After a call to `std::move`, **we cannot make any assumptions about the value of the moved-from object.**

" `std::move` does not *move* anything. It just makes a *promise*"

NRVO, Move and Copy Elision

Returning a Temporary(pure rvalue)

```
std::string foo(const std::string &a, const std::string &b) {
  return a + b; // a temporary
}
std::string s = foo(a, b);
```

- First, a temporary is generated to store the result of `a + b`.
- How is this temporary returned?

Since C++17, **no copy or move** is made here. The initialization of `s` is the same as

```
std::string s(a + b);
```

This is called **copy elision**.

Returning a Named Object

```
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());
  for (std::size_t i = 0; i != a.size(); ++i)
    result.at(i) = a.at(i);
  for (std::size_t i = 0; i != b.size(); ++i)
    result.at(a.size() + i) = b.at(i);
  return result;
}
a = concat(b, c);
```

- `result` is a local object of `concat`.
- Since C++11, `return result` performs a **move initialization** of a temporary object, say `tmp`.
- Then a **move assignment** to `a` is performed

Named Return Value Optimization, NRVO 命名返回值优化

```
Dynarray concat(const Dynarray &a, const Dynarray &b) {
  Dynarray result(a.size() + b.size());// ...
  return result;
}
Dynarray a = concat(b, c); // Initialization
```

NRVO transforms this code to

```
// Pseudo C++ code.
void concat(Dynarray &result, const Dynarray &a, const Dynarray &b) {
  // Pseudo C++ code. For demonstration only.
  result.Dynarray::Dynarray(a.size() + b.size()); // construct in-place
  // ...
}
  Dynarray a@; // Uninitialized.
  concat(a@, b, c);
```

so that no copy or move is needed.

`class` 初步

访问限制

- `private`：外部代码不能访问，只有类内和 `friend` 代码可以访问。
- `public`：所有代码都可以访问。
- 为何需要将数据成员“藏起来”？
  - 只有`private(封装)`和 `public(不封装)` 的区别

构造函数

构造函数 (constructors, ctors) 负责对象的初始化方式。

- 构造函数往往是**重载** (overloaded) 的，因为一个对象很可能具有多种合理的初始化方式。
- 构造函数的函数名就是类名本身：`Student`
- 构造函数**不声明返回值类型**，可以含有 `return;` 但不能返回一个值，**但不能认为它的返回值类型是** `void` **。**

构造函数初始值列表

构造函数负责这个对象的初始化，包括**所有成员**的初始化。

一切成员的初始化过程都在**进入函数体之前结束**，它们的初始化方式（部分是）由**构造函数初始值列表** (constructor initializer list) 决定：

```
class Student {
  // ...
 public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
```

构造函数初始值列表由冒号 `:` 开头，依次给出各个数据成员的初始化器，用 `,` 隔开。

构造函数初始值列表

数据成员按照**它们在类内声明的顺序**进行初始化，而非它们在构造函数初始值列表中出现的顺序。

- 如果你的构造函数初始值列表中的成员的顺序和它们在类内的声明顺序不同，编译器会以 warning 的方式提醒你。
- 未在初始值列表中出现的成员：
  - 如果有**类内初始值**（稍后再说），则按照类内初始值初始化；
  - 否则，**默认初始化**。

默认初始化对于内置类型来说就是*不初始化*，但是对于类类型呢？

构造函数初始值列表

“先默认初始化，后赋值”有什么坏处？

- 不是所有类型都能默认初始化，不是所有类型都能赋值。
- 引用无法被默认初始化，无法被赋值。`const` 对象无法被赋值。
  - （内置类型的）`const` 对象无法被默认初始化。
  - 我们将在后面看到，类类型可以拒绝支持默认初始化和赋值，这取决于设计。
- 如果你把能默认初始化、能赋值的成员在函数体里赋值，其它成员在初始值列表里初始化，**你的成员初始化顺序将会非常混乱**，很容易导致错误。

特殊的构造函数：不接受参数。

- 专门负责对象的**默认初始化**，因为调用它时不需要传递参数，相当于不需要任何初始化器。
- 特别地，类类型的**值初始化**就是**默认初始化**。

```
class Point2d {
  double x, y;
 public:
  Point2d() : x(0), y(0) {}
  Point2d(double x_, double y_) : x(x_), y(y_) {}
};
Point2d p1; // 调用默认构造函数，(0, 0)
Point2d p2(3, 4); // 调用 Point2d(double, double), (3, 4)
Point2d p3(); // 这是在调用默认构造函数吗?
```

- 注意，`Point2d p3()` 并不是调用默认初始化，而是调用了一个名为`p3`，不接受任何参数的以`Point2d` 为返回型的**函数**！！

你的类需要一个默认构造函数吗?

- 首先，如果需要开数组，你几乎肯定需要默认构造函数：
- 但关键问题是**它是否具有一个合理的定义?**

构造函数

如果一个类没有 user-declared 的构造函数，编译器会试图帮你合成一个默认构造函数。

编译器合成的默认构造函数的行为非常简单：逐个成员地进行默认初始化

- 如果有成员有类内初始值，则使用类内初始值。
- 如果有成员没有类内初始值，但又无法默认初始化，则编译器会放弃合成这个默认构造函数。

注意： 如果一个类有**至少一个** user-declared 的构造函数，但没有默认构造函数，则编译器**不会**帮你合成默认构造函数。

- 在它看来，这个类的所有合理的初始化行为就是你给出的那几个构造函数，
- 因此不应该再画蛇添足地定义一个默认行为。

你可以通过 `= default;` 来显式地要求编译器合成一个具有默认行为的默认构造函数：

你的类需要一个默认构造函数吗?

宁缺毋滥：如果它没有一个合理的默认构造行为，就不应该有默认构造函数!

- 而不是胡乱定义一个或者 `=defualt`，这会留下隐患。
- 对默认构造函数的调用应当引发**编译错误**，而不是随意地允许。

《Effective C++》条款 6：如果编译器合成的函数你不需要，应当显式地禁止。

类内初始值

可以在声明数据成员的时候顺便给它一个初始值。

但是类内初始值不能用圆括号...

如果这个成员在某个构造函数初始值列表里没有出现，它会采用类内初始值，而不是默认初始化。

```
struct X {
  // 错误：受限于编译器的设计，它会在语法分析阶段被认为是函数声明
  std::vector<int> v(10);

  // 这毫无疑问是声明一个函数，而非设定类内初始值
  std::string s();
};
```

析构函数 (destructor, dtor)

析构函数是在这个对象被销毁的时候自动调用的函数。

- 它通常用来完成一些最后的清理
- 特别地，那些**拥有一定资源**的类通常在析构函数里释放它们所拥有的资源
- 析构函数的名字是 `~ClassName`
- 析构函数不接受参数，不声明返回值类型
- 一个类只能有一个析构函数 (until C++20)

`const` 成员函数：复习

- `const` 写在哪？——参数列表后，函数体之前
- `const` 成员函数的 `const` 是作用于谁的？
  - 加在隐式的 `this` 指针上的**底层** **`const`**
  - 表示当前对象是 `const`，其所有数据成员也都是 `const`
- 在什么对象上能调用 `const` 成员函数？
  - 什么对象上都可以，因为添加底层 `const` 永远没问题。
- `const` 成员函数能做哪些事？不能做哪些事？
  - 不能修改数据成员，不能调用数据成员的 non-`const` 成员函数
  - 不能调用自身的 non-`const` 成员函数

在 `const` vs non-`const` 重载中避免重复

将一模一样的代码编写两遍实在是太麻烦了... 有没有办法避免重复?

- C++23 deducing-this 能帮得上忙

假如没有额外的语法特性... 能不能让一个函数调用另一个?

假如我们让 non-`const` 版本的函数调用 `const` 版本的函数:

- 首先，我们需要显式地为 `this` 添加底层 `const`。
- `const` 版本的函数返回的是 `const int &`，我们得把它的底层 `const` 去除。

在 `const` vs non-`const` 重载中避免重复

- 先用 `static_cast<const Dynarray *>(this)` 为 `this` 添加底层 `const`
- 这时调用 `->at(n)`，就会匹配 `const` 版本的 `at`
- 将返回的 `const int &` 的底层 `const` 用 `const_cast` 去除

```
class Dynarray {
 public:
  const int &at(std::size_t n) const {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return const_cast<int &>(static_cast<const Dynarray *>(this)->at(n));
  }
};
```

- 千万不能让non-`const`函数调用`const`函数，这相当于人为去除low-level `const`

`static` 成员

`static` 数据成员

* **将这个计数器限定在类的作用域内**：将它定义为类的 `static` 成员。

```
class Dynarray {
  static int s_cnt; // !!!
  int *m_storage;
  std::size_t m_length;
  int m_id;
 public:
  Dynarray(std::size_t n)
      : m_storage(new int[n]{}), m_length(n), m_id(s_cnt++) {}
  Dynarray() : m_storage(nullptr), m_length(0), m_id(s_cnt++) {}
  // ...
};
```

根据 C++ 的规则，你还需要在类外初始化一下它：

```
int Dynarray::s_cnt = 0;
```

`static` 数据成员

`static` 数据成员：一个限定在类的作用域内的“全局变量”

- 限定在类的作用域内：受访问限制说明符的影响，如果是 `private`，那么外部代码不可访问。
- “全局变量”：每个程序只有一个 `s_cnt`，这个 `s_cnt` 不属于任何的对象，而是属于这个类。

访问： `A::s_cnt`，而非 `a.s_cnt`。

- 实际上 `a.s_cnt` 也可以，但是 `a.s_cnt` 和 `b.s_cnt` 访问的是同一个 `s_cnt`。

单例 (singleton) 模式

假如我们定义了一个类 `Widget`。如果我们希望这个世界上只有一个 `Widget` 类型的对象，怎么办？

- 只能我们自己创建这个对象：所有构造函数都是 `private` 的
- 禁止拷贝
- 注：此处的 `static Widget` 的类型与上文的类内 `static` 类型并不同，而是在局部作用域内定义的“全局变量”，生命期为编译器第一次执行到这个语句开始一直到整个程序结束。

```
class Widget {
  // 构造函数是 private 的
  Widget();
  // ...
 public:
  Widget(const Widget &) = delete;
  Widget &operator=(const Widget &) = delete;
  // Magic happens here!!
  static Widget &get_instance() {
    static Widget w; // 这个 static 是什么意思?
    return w;
  }
};
```

外部代码获得这个对象的唯一方式是 `Widget::get_instance()`。

拷贝控制：总结

此处非常建议再仔细阅读r10 by Mr._Gkxx

拷贝控制成员

- 拷贝构造函数 copy constructor **（参数表必须是引用，否则无穷递归）**
- 拷贝赋值运算符 copy assignment operator （返回等号左边对象的引用，self safe）
- 移动构造函数 move constructor
- 移动赋值运算符 move assignment operator
- 析构函数 destructor

虽然后三个名字里没有“拷贝”，但也属于“copy control members”。

两个移动操作是 C++11 开始有的。

何时需要

首先，**分清初始化和赋值**。

- 初始化是在变量声明语句中的，它必然调用构造函数。
- 赋值是一个**运算符**，它必然在**表达式**中。

“拷贝”是传统艺能：对于左值必然是拷贝，右值在移动操作存在的情况下被移动。但如果移动操作不存在，右值也被拷贝。（通常情况下）

析构函数的调用意味着对象生命期的结束。

- 超出作用域时，程序结束时，以及 `delete` / `delete[]` 表达式

默认行为

在某些情况下（包括用 `= default` 显式要求时），编译器会合成一个具有默认行为的拷贝控制成员。

- 默认行为：**先父类，后自己的成员**，且成员按**声明顺序**，逐个执行对应的操作。
  - 析构顺序相反。
- 默认的移动行为：等同于将 `std::move` 作用于每个成员。
  - 并不是苛求每个成员或父类都采用移动操作。
  - 能移动就移动，不能移动就拷贝。

如果默认行为中涉及的任何一个操作无法正常进行（不存在或不可访问），这个函数就是删除的 (deleted function)。

为何需要 `std::move`

```
struct X {
  std::string s;
  X(X &&other) noexcept : s(other.s) {}
};
```

右值引用是左值：Anything that has a **name** is an lvalue, even if it is an rvalue reference.

从生命期的角度理解：右值引用延长了右值的生命期，使用右值引用时就如同在使用一个普通的（左值）变量。

`other` 是左值，`other.s` 自然是左值，`s(other.s)` 是拷贝而非移动。

`= delete`

删除的函数 (deleted function)

- **仍然参与重载决议，**
- **但如果被匹配到，就是 error。**
- 任何函数都可以是删除的。

特别例外：如果编译器合成了一个删除的移动操作，它不会参与重载决议，这是为了让右值被拷贝。[CWG1402]

- 《C++ Primer》在这个问题上说的是不对的。

三/五法则

C++11 以前是“三”，C++11 以后是“五”。

如果你认为有必要自定义这五个函数中的任何一个，通常意味着这五个你都应该定义。

"Define zero or five or them."

根据“五法则”：如果五个函数中的**任何一个具有用户自定义** (user-provided) 的版本，编译器就**不应该再合成其它**那些用户没有定义的函数

- 重要例外：一个类不能没有析构函数 ~~就像...~~
- 另一个例外：兼容旧的代码
  - 在 C++98 时代，“三法则”并未在编译器的行为上予以体现。
  - 如果一个类有自定义的拷贝构造函数或析构函数，而没有自定义拷贝赋值运算符，C++98 编译器会合成这个拷贝赋值运算符。（拷贝构造函数同理）
  - 为了兼容旧的代码，不能直接禁止这种行为，只能将它判定为 deprecated。

Copy-and-swap

能不能写出一个简单的 `swap` 函数，交换两个 `Dynarray` 对象的值？

```
class Dynarray {
 public:
  void swap(Dynarray &other) noexcept {
    std::swap(m_storage, other.m_storage);
    std::swap(m_length, other.m_length);
  }
};
```

直接交换 `m_storage` 指针，就可以快速交换两个“动态数组”。这个 `swap` 甚至是 `noexcept` 的，它远远好过传统的 `auto tmp = a; a = b; b = tmp;` 写法。

Copy-and-swap

赋值 = 拷贝构造 + 析构：拷贝新的数据，销毁原有的数据。

所以我们可以用拷贝构造函数和析构写出一个拷贝赋值运算符

为 `other` 建立一个拷贝 `tmp`，直接将自己和 `tmp` 交换！

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    auto tmp = other;
    swap(tmp);
    return *this;
  }
};
```

- 拷贝构造函数会负责正确拷贝 `other`。
- `tmp` 的析构函数会正确销毁旧的数据。

Copy-and-swap

更简洁些：

```
class Dynarray {
 public:
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this); // C++23: auto{other}.swap(*this);
    return *this;
  }
};
```

自我赋值安全吗？

不仅好写，还自我赋值安全，还提供强异常安全保证！

"Copy"-and-swap

更进一步，直接在传参的时候做好“拷贝”。

```
class Dynarray {
 public:
  Dynarray &operator=(Dynarray other) noexcept {
    swap(other);
    return *this;
  }
};
```

如果参数是右值，`other` 将被**移动初始化**，而不是**拷贝初始化**。

也就是说，这个赋值运算符**既是一个拷贝赋值运算符，又是一个移动赋值运算符！**

`std::string` 拷贝还是移动?

我们已经习惯于将不修改的参数声明为 reference-to-`const`：

```
class Message {
 public:
  Message(const std::string &contents)
      : m_contents{contents} {}
};
```

但是如果传进来的字符串是右值，能不能直接移动给 `m_contents`？

`std::string` 拷贝还是移动?

如果参数是左值，将它拷贝给 `m_contents`；否则，将它移动给 `m_contents`。

```
class Message {
 public:
  Message(const std::string &contents) : m_contents{contents} {}
  Message(std::string &&contents) : m_contents{std::move(contents)} {}
};
```

`std::string` 拷贝还是移动?

如果参数是左值，将它拷贝给 `m_contents`；否则，将它移动给 `m_contents`。

直接按值传递不就行了?

```
class Message {
 public:
  Message(std::string contents) : m_contents{std::move(contents)} {}
};
```

拷贝/移动会在对参数 `contents` 的初始化中自动决定，而我们只需要把 `contents` 移给 `m_contents`。

`friend`

```
class Message {
  friend class Folder;
};
```

可以将类 `Folder` 声明为 `Message` 的 `friend`，则 `Folder` 的代码可以访问 `Message` 的 `private` 和 `protected` 成员。

- 这个 `friend class Folder;` **不是**成员声明，**不受访问限制修饰符的作用**。
- `friend` 声明可以放在任何位置，通常在一个类的开头或末尾集中声明所有 `friend`。

`friend`

也可以声明某个单独的函数

```
class Message {
  friend class Folder;
  friend void modify(Message &m, const std::string &s);
};
void modify(Message &m, const std::string &s) {
  // 在这里可以直接访问、修改 m.m_contents
}
```

- 这个 `friend void modify(...);` **不是**成员函数声明，`friend` 函数**不是**成员函数，**不受访问限制修饰符的作用。**

`friend`

`friend` 函数也可以在类内定义，但它们仍然不是成员函数。

```
class Message {
  friend class Folder;
  friend void modify(Message &m, const std::string &s) { /* ... */ }
};
```

但是将 `friend` 函数定义在类内会引发特殊的名字查找问题：

```
struct X {
  friend int add(int x, int y) {
    return x + y;
  }
};
```

```
int main() {
  auto x = add(1, 2);
  // Error: `add` was not declared
  // in this scope.
}
```

个人不推荐将 `friend` 函数定义在类内，除了一个和模板有关的特殊情况（以后再说）

Smart Pointers

Smart Pointers

`<memory>` provides two types of smart pointers:

- `std::unique_ptr<T>`, which uniquely **owns** an object of type `T`.
  - No other smart pointer pointing to the same object is allowed.
  - Disposes of the object(calls its destructor) once this `unique_ptr` gets destroyed or assigned a new value.
- `std::shared_ptr<T>`, which **shares** ownership of an object of type `T`.
  - Multiple `shared_ptr`s pointing to a same object is allowed.
  - Disposes of the object (calls its destructor) when the last `shared_ptr` pointing to that object gets destroyed or assigned a new value.

Creating an `std::unique_ptr`

Use `std::unique_ptr` to create an object in dynamic memory,

- if no other pointer to this object is needed

Tow ways of creating an `std::unique_ptr` :

- passing a pointer created by `new` in the constructor.

```
std::unique_ptr<Student> p(new Student("Bob", 2020123123));
```

- use `std::make_unique<T>`, pass initializers to it:

```
std::unique_ptr<Student> p1 = std::make_unique<Student>("Bob", 2020123123);
auto p2 = std::make_unique<Student>("Alice", 2020321321);
```

Using `auto` here does not reduce readability, because `std::make_unique<Student>` clearly hints the type.

`std::unique_ptr`: Automatic Memory Management

An `std::unique_ptr` automatically calls the destructor once it gets destroyed or assigned a new value.

- No manual `delete` needed!

`std::unique_ptr`: Move-only

```
auto p = std::make_unique<std::string>("Hello");
std::cout << *p << std::endl; // Prints "Hello".
std::unique_ptr<std::string> q = p; // Error, copy is not allowed.
std::unique_ptr<std::string> r = std::move(p); // Correct.
// The ownership of this std::string is transferred to r.
std::cout << *r << std::endl; // Prints "Hello".
assert(!p); // p is now invalid
```

An `std::unique_ptr` cannot be copied, but only moved.

- Remember, only one `std::unique_ptr can own the managed object.
- A move operation transfers its ownership.

Move assignment

`std::unique_ptr is only move-assignable, not copy-assignable.

```
std::unique_ptr<T> p = some_value(), q = some_other_value();
p = q; // Error
p = std::move(q); // OK.
```

The assignment `p = std::move(q)` does the following:

- `p` releases the object it used to manage. Destructor is called and memory is deallocated.
- Then, the object that `q` manages is transferred to `p`. `q` no longer owns an object.

`unique_ptr` for array type

A *template specialization*: `std::unique_ptr<T[]>

- Specially designed to represent pointers that point ot a "dynamic array" of objects.
- Has some array-specific operators, e.g. `operator[]`.
- Does not support `operator*` and `operator->`.
- Uses `delete[]` instead of `delete`.

`shared pointer`

Idea: Reference counting

```
class CountedObject {
  Object the_object;
  int ref_cnt = 1;
};

class SharedPtr {
  CountedObject *m_ptr;
 public:
  SharedPtr(const SharedPtr &other)
    : m_ptr(other.m_ptr) { ++m_ptr->ref_cnt; }
  SharedPtr &operator=(const SharedPtr &other) {// Pay attention to self-assignment safe!!!!
    ++other.m_ptr->ref_cnt;
    if(--m_ptr->ref_cnt == 0)
      delete m_ptr;
    m_ptr = other.m_ptr;
    return *this;
  }
  ~SharedPtr() {
    if(--m_ptr->ref_cnt == 0)
      delete m_ptr;
  }
};
```

Create a `shared_ptr`

For `std::shared_ptr`, **`std::make_shared`** is preferable to directly using `new`

```
auto sp = std::make_shared<Type>(args); //preferable
std::shared_ptr<Type> sp2(new Type(args)); // Ok, but less preferred
```

Read *Effective Modern C++* Item 21.

Operations

`*` and `->` can be used as if it is a raw pointer.

`sp.use_count()` : The value of the reference counter.

```
auto sp = std::make_shared<std::string>(10, 'c');
{
  auto sp2 = sp;
  std::cout << sp.use_count() << std::endl; //2
}//sp2 is destroyed
std::cout << sp.use_count() << std::endl; //1
```

Notes:

- Both of the two smart pointers support `.get()`, which returns a raw pointer to the managed object.
    - This is useful when some old interfaces only accept raw pointers, e.g. `glfwSwapBuffers`.
- **Don't** delete the raw pointer you get from `.get()` !!!!

参数转发

回顾 `std::make_shared` 和 `std::make_unique`：

```
auto sp = std::make_shared<std::string>(10, 'c');  // "cccccccccc"
auto sp2 = std::make_shared<std::string>("hello"); // "hello"
auto up = std::make_unique<Student>("Alice", "2020123123");
```

甚至，如果传入右值，它们会移动构造那个对象：

```
auto sp3 = std::make_shared<std::string>(std::move(*sp));
std::cout << *sp << std::endl; // empty string
```

这种将参数转发给另一个函数，又能保持它们的值类别的操作叫做**完美转发** (perfect forwarding)

参数转发

`std::make_shared/unique<T>(...)` 可以接受任意多个任意类型的参数，并将它们原封不动地转发给 `T` 的构造函数，不丢失值类别，不丢失 `const`。

等学了模板，就知道是咋回事了。

标准库很多函数都支持这样的操作，其中非常典型的是**容器的 `emplace` 系列操作**：

```
std::vector<Student> students;
students.emplace_back("Alice", "2020123123");
std::vector<std::string> words;
words.emplace_back(10, 'c');
```

标准库容器的 `emplace`

```
std::vector<Student> students;
students.emplace_back("Alice", "2020123123");
std::vector<std::string> words;
words.emplace_back(10, 'c');
```

`emplace` 系列操作利用传入的参数直接原地构造出那个对象，而不是将构造好的对象拷贝/移动进去。

- 提高效率。
- 对所存储的数据类型的要求进一步降低。尤其是 `std::list<T>`（链表）自 C++11 起不需要 `T` 具备任何拷贝/移动操作，只要有办法构造和析构即可。
- `vector` 由于需要搬家（增长时重新分配内存），无法存储不可拷贝、不可移动的元素，除非你不需要它搬家。

多文件和动态链接

`#include`

`#include` 机制：在其它真正的编译过程开始之前，由**预处理器**（preprocessor）将被 `#include` 的文件原封不动地复制过来。

所以如果一个文件被 `#include` 两遍，它所包含的内容就会出现两次。

- 如果不加以保护，就会出现重复定义符号的问题。

Include guard

```
#ifndef MY_FILE_HPP
#define MY_FILE_HPP

// 将头文件的内容放在这里

#endif // MY_FILE_HPP
```

- `#ifndef X` … `#endif`：如果宏 `X` 在此前没有定义过，那么编译这部分代码。
- 如果 `MY_FILE_HPP` 此前没有定义过：首先在这里定义它，然后编译下面的代码。
- 当这段代码再次出现时，宏 `MY_FILE_HPP` 已经被定义过了！

分离式编译

例：将函数的声明写在头文件里，定义写在另一个 `.cpp` 文件里。

在 `a.cpp` 里调用函数 `sum` 时，**只要 `sum` 的声明已经出现，其实就可以编译，只是暂时无法生成可执行文件。**

- 因此只要在 `a.cpp` 里 `#include "sum.hpp"` 。

编译： `g++ a.cpp sum.cpp -o a`

- 编译器会将 `a.cpp` 中对于 `sum` 的调用和在 `sum.cpp` 中的定义**链接**起来。

更进一步：可以单独编译 `sum.cpp`，生成一个中间文件，再在编译 `a.cpp` 时链接它：

```
g++ -shared sum.cpp -o libsum.so
```

这时生成了 `libsum.so`：**动态链接库文件**（有点儿像 Windows 上的 `.dll`，见过吗？）

编译 `a.cpp`： `g++ a.cpp -o a -Wl,-rpath . -L. -lsum`

- `-Wl,-rpath .` 告知链接器在当前目录（`.`）下寻找 `.so` 文件
- `-L. -lsum` 链接到 `libsum.so`。

`inline`

`inline` 关键字：**向编译器发出一个请求**，希望将这个函数在调用点内联展开。

编译器可以拒绝某些 `inline` 请求，也可能自动为某些没有显式 `inline` 的函数 inline。

- 递归函数是难以 inline 的：编译器怎么知道它要展开多少层呢?
- 太过复杂的函数的 inline 请求也会被拒绝。

拒绝并不会导致报错或 warning。

任何写在类内的函数都是隐式 `inline` 的： `static` members, non-`static` members

继承 (Inheritance)、多态 (Polymorphism)

`protected` members

- 除了 ctor 和 dtor **以外的所有成员**，子类都会全部继承，无论访问级别！
  - 父类中的 `protected` 成员级别是 `private`，但是它可以在子类中访问。
  - 父类的 `private` 成员子类可以继承，但是子类不能直接访问！

继承

- 一个子类对象里一定有一个完整的父类对象*
  - 父类的**所有成员**（除了构造函数和析构函数）都被继承下来，无论能否访问
  - 子类的构造函数必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员
  - 子类的析构函数在析构了自己的成员之后，必然调用父类的析构函数
- 禁止“坑爹”：继承不能破坏父类的封装性
  - 子类不可能改变继承自父类成员的访问限制级别
  - 子类不能随意初始化父类成员，必须经过父类的构造函数
    - 先默认初始化后赋值是可以的

* 除非父类是空的，这时编译器会做“空基类优化 (Empty Base Optimization, EBO)”（空类大小不为0）。

子类的构造函数

必然先调用父类的构造函数来初始化父类的部分，然后再初始化自己的成员

```
class DiscountedItem : public Item {
 public:
  DiscountedItem(const std::string &name, double price,
                 int minQ, double discount)
      : Item(name, price), m_minQuantity(minQ), m_discount(discount) {}
};
```

可以在初始值列表里调用父类的构造函数

- 如果没有调用，则自动调用父类的默认构造函数
    - 如果父类不存在默认构造函数，则报错。

合成的默认构造函数：先调用父类的默认构造函数，再 ……

子类的析构函数

析构函数在执行完函数体之后：

- 先按成员的声明顺序倒序销毁所有成员。对于含有 non-trivial destructor 的成员，调用其 destructor。
- 然后调用父类的析构函数销毁父类的部分。

子类的拷贝控制

自定义：不要忘记拷贝/移动父类的部分

```
class Derived : public Base {
 public:
  Derived(const Derived &other) : Base(other), /* ... */ { /* ... */ }
  Derived &operator=(const Derived &other) {
    Base::operator=(other);
    // ...
    return *this;
  }
};
```

合成的拷贝控制成员（不算析构）的行为?
合成的拷贝控制成员（不算析构）：先父类，后子类自己的成员。

- 如果这个过程中调用了任何不存在/不可访问的函数（比如父类的构造函数），则合成为 implicitly deleted

动态绑定

向上转型：

- 一个 `Base *` 可以指向一个 `Derived` 类型的对象
  - `Derived *` 可以向 `Base *` 类型转换
- 一个 `Base &` 可以绑定到一个 `Derived` 类型的对象
- 一个 `std::shared/unique_ptr<Base>` 可以指向一个 `Derived` 类型的对象
  - `std::shared/unique_ptr<Derived>` 可以向 `std::shared/unique_ptr<Base>` 类型转换

注意！：这个时候只有父类中的成员才可以通过 `Base *(&)` 来访问（包括构造和析构函数）。

虚函数

继承父类的某个函数 `foo` 时，我们可能希望在子类提供一个新版本（override）。

我们希望在一个 `Base *`，`Base &` 或 `std::shared/unique_ptr<Base>` 上调用 `foo` 时，可以根据**动态类型**来选择正确的版本，而不是根据静态类型调用 `Base::foo`。

在子类里 override 一个虚函数时，函数名、参数列表、 `const` ness 必须和父类的那个函数**完全相同**。
子类中的 override 函数**不可以扩大**父类中的对应函数

- 返回值类型必须**完全相同**或者随类型*协变* (covariant)。

加上 `override` 关键字：让编译器帮你检查它是否真的构成了 override

注意： `auto` 不能用来推测 `virtual` 函数的返回值。

`virtual` - `override`

想要去 override（覆盖/覆写）一个 `virtual` 函数，

- 函数的参数列表必须和父类中的版本的参数列表一样
- 函数的返回值和父类函数中对应函数的返回值是 **identical to or covariant with** 的.
  （详情可以参考Hw7文字题和*More Effective C++* Item 25，有关 `virtual` constructor 的内容）
- `const` ness **不能被改变**！！！
- 相比父类中对应的函数**不能扩大**访问级别。

尽管没有显式声明，但是一个 overriding 函数仍然是一个虚函数。
`virtual` 和 `override` 都可以被省略，但是**建议一直加上他们**.

虚函数

- 对于一个多态
- 默认生成的（compiler-generated）的析构函数是 **non-`virtual`** 的！
  - 所以我们必须显示声明其为 `virtual` .
- 如果父类的 dtor 是虚函数，那么子类默认生成的 dtor 也是虚函数。

虚函数

除了 override，不要以其它任何方式定义和父类中某个成员同名的成员。

- 阅读以下章节，你会看到违反这条规则带来的后果。

《Effective C++》条款 33：避免遮掩继承而来的名称

《Effective C++》条款 36：绝不重新定义继承而来的 non-`virtual` 函数

《Effective C++》条款 37：绝不重新定义继承而来的缺省参数值

条款33：避免遮掩继承而来的名称 - Avoid hiding inherited names.

我们知道在诸如这般的代码中：

```
int x;              //global变量
void someFunc() {
  double x;         //local变量
  std::cin >> x;    //读一个新值赋予local变量x
}
```

编译器在local作用域内查找x名称，找不到再找其他作用域。
名称遮掩规则（name-hiding rules）做的唯一事情就是：遮掩名称

- 与此相似的，我们知道当 `derived class` 成员指涉到 `base class` 中的某物，编译器可以找到我们所指涉的东西
- 实际原理是： `derived class` 作用域被嵌套在`base calss` 作用域内

条款33：避免遮掩继承而来的名称 - Avoid hiding inherited names.

- 解决办法：

1. using声明式：

```
class Base {
 private:
  int x;
 public:
  virtual void mf1() = 0;
  virtual void mf1(int);
  virtual void mf2();
  void mf3();
  void mf3(double);
};
```

```
class Derived: public Base {
 public:
  using Base::mf1;  //让Base class内名为mf1和mf3
                    //的所有东西在Derived作用域
                    //内都可见（并且public）
  using Base::mf3;
  virtual void mf1();
  void mf3();
  void mf4();
};
```

条款33：避免遮掩继承而来的名称 - Avoid hiding inherited names.

- 解决办法：

2. 转交函数（forwarding functions）

```
class Derived: private Base {
 public:
  virtual void mf1() {  //转交函数 (forwarding function)
    Base::mf1();        //暗自成为inline
  }
};
```

- **注**：inline 转交函数的另一个用途是为那些不支持 `using` 的老旧编译器另辟一条新路。**（这并非正确行为）**

条款36：绝不重新定义继承而来的 `non-virtual` 函数

- 适用于B对象的每一件事，也适用于D对象，因为每个D对象都是一个B对象；
- B的 derived classes 一定会继承func的接口和实现，因为func是B的一个 non-virtual 函数。

所以，如果D重新定义func，设计便出现矛盾：

- 每个D都是一个B不为真
- func无法为B反映出“不变性凌驾特异性”的性质

条款 37：绝不重新定义继承而来的缺省参数值

理由：virtual函数系动态绑定（dynamically bound），而缺省参数值确实静态绑定（statically bound）。

```
class Shape {
 public:
  enum ShapeColor { Red, Green, Blue };
  virtual void draw(ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape {
 public: //注意，赋予不同的缺省参数值。这真糟糕!
  virtual void draw(ShapeColor color = Green) const;
};
class Circle: public Shape {
 public:
  virtual void draw(ShapeColor color) const;
  //请注意，以上这么写则当客户以对象调用此函数，一定要指定参数值。
  //因为静态绑定下这个函数并不从其base继承缺省参数值。
  //但若以指针（或reference）调用此函数，可以不指定参数值，
  //因为动态绑定下这个函数会从其base继承缺省参数值
}
```

条款 37：绝不重新定义继承而来的缺省参数值

现在考虑这些指针

```
Shape *ps;                                  //静态类型为Shape*
Shape *pc = new Circle;                     //静态类型为Shape*
Shape *pr = new Rectangle;                  //静态类型为Shape*
```

对象所谓的动态类型（dynamic type）则是指“目前所指对象的类型”。也就是说，动态类型可以表现出一个对象将会有什么行为。以上例而言，pc的动态类型是 `Circle*` ，pr的动态类型是 `Rectangle*` 。ps没有动态类型，因为他尚未指向任何对象。

`Virtual` 函数系动态绑定而来，意思是调用一个 `virtual` 函数时，究竟调用哪一份函数实现代码，取决于发出调用得分那个对象的动态类型。

```
pc->draw(Shape::Red);                       //调用Circle::draw(Shape::Red)
pr->draw(Shape::Red);                   //调用Rectangle::draw(Shape::Red)
```

条款 37：绝不重新定义继承而来的缺省参数值

- virtual 函数是动态绑定，而缺省参数值却是静态绑定。意思是说：
- 调用一个定义于 derived class 内的 virtual函数的同时，却使用 base class 为它所指定的缺省参数值。

```
pc->draw();         \\调用Rectangle::draw(Shape::Red)!
```

调用的是 Rectangle 的 virtual函数，但是缺省参数值来自 Shape class 而非 Rectangle class。

- **同样的问题在把指针换成 reference 时依然存在！**

原因？

- 可以提升编译器效率，不需要在运行时为 virtual函数决定参数值。

纯虚函数

通过将一个函数声明为 `=0`，它就是一个**纯虚函数** (pure virtual function)。

一个类如果有某个成员函数是纯虚函数，它就是一个**抽象类**。

- 不能定义抽象类的对象，不能调用无定义的纯虚函数*。

* 事实上一个纯虚函数仍然可以拥有一份定义，阅读《Effective C++》条款 34（必读，HW7 的客观题会涉及此条款的内容）。

纯虚函数通常用来定义**接口**：这个函数在所有子类里都应该有一份自己的实现。

如果一个子类继承了某个纯虚函数而没有 override 它，这个成员函数就仍然是纯虚的，这个类仍然是抽象类，无法被实例化。

运行时类型识别 (RTTI)

`dynamic_cast` 可以做到“向下转型”：

- 它会在运行时检测这个转型是否能成功
- 如果不能成功，`dynamic_cast<T*>` 返回空指针，`dynamic_cast<T&>` 抛出 `std::bad_cast` 异常。
- **非常非常慢**，你几乎总是应该先考虑用一组虚函数来完成你想要做的事。

`typeid(x)` 可以获取表达式 `x` （忽略顶层 `const` 和引用后）的动态类型信息

- 通常用 `if (typeid(*ptr) == typeid(A))` 来判断这个动态类型是否是 `A`。

https://quick-bench.com/q/E0LS3gJgAHlQK0Em_6XzkRzEjnE

根据经验，如果你需要获取某个对象的动态类型，通常意味着设计上的缺陷，你应当修改设计而不是硬着头皮做 RTTI。

继承的访问权限

```
class DiscountedItem : public Item {};
```

继承的访问权限：**这个继承关系（“父子关系”）是否对外公开。**

如果采用 `private` 继承，则在外人眼里他们不是父子，任何依赖于这一父子关系的行为都将失败。

- 访问继承而来的成员（本质上也是向上转型）
- 向上转型（包括动态绑定等等）、向下转型

`private` 继承：建模“is-implemented-in-terms-of”。阅读《Effective C++》条款 38、39。

条款38：通过复合塑模出 has-a 或“根据某物实现出”

- 复合（composition）是类型之间的关系，当某种类型的对象内含它种类型的对象，便是这种关系。

```
class Address {};                                   //某人的住址
class PhoneNumber {};

class Person {
 public:
  ...
 private:
  std::string name;                                 //合成成分物（composed object）
  Address address;                                  //同上
  PhoneNumber voiceNumber;                          //同上
  PhoneNumber faxNumber;                            //同上
}
```

继承的访问权限

```
struct A : B {}; // public inheritance
class C : B {};  // private inheritance
```

`struct` 和 `class` 仅有两个区别：

- 默认的成员访问权限是 `public` / `private` 。
- 默认的继承访问权限是 `public` / `private` 。

重载运算符 (Operator Overloading)

总结：技术

不能发明新的运算符。不能为内置类型定义运算符。

不能被重载：

- `?:`，`.`，`::` 等，具有特别意义的运算符
- `?:` 无法被重载，因为它有一个运算对象不被求值，这一点用函数传参不可能做到

不建议被重载：（为什么？）

- `&&`，`||`，`,`，`&` (一元取地址运算符)
- `&&` 和 `||` 的短路求值会失效。`,` 和 `&` 本来就有特殊含义

重载的行为需要和内置的行为保持某种程度上的一致，除非你有很好的理由不这么做

- `++i` 返回 `i` 的引用，`i++` 返回递增前的 `i` 的一份拷贝。
- `=` 赋值、复合赋值返回左侧运算对象的引用
- 解引用 `*p` 通常返回左值引用

总结：技术

如果需要左侧运算对象也能隐式转换，则它**不可以是成员**

- `r == 1` 被视为 `r.operator==(1)`，即 `r.operator==(Rational(1))`
- 但 `1 == r` 无法被视为 `1.operator==(r)`

不要漏 `const`！

```
class Rational {
 public:
  bool operator==(const Rational &) const;
};
bool operator!=(const Rational &, const Rational &);
```

非成员有两个 `const`，成员也必然有两个 `const`，只是其中一个变成了这个成员函数的 qualifier。

特殊的运算符：后置递增 `i++`

```
class Rational {
 public:
  Rational &operator++();
  // 90% 的后置递增运算符都应该这样写
  Rational operator++(int) {
    auto tmp = *this; // 拷贝原来的对象
    ++*this;          // 真正的“递增”由前置版本完成
    return tmp;       // 返回递增前的拷贝
  }
};
```

- `int` 参数：**仅仅是为了区分前置版本和后置版本**，没有实际意义，也不需要用到，自然也就没有名字。
- `++i` 被视为 `i.operator++()`，`i++` 被视为 `i.operator++(0)`。

特殊的运算符：`->`

```
class SharedPtr {
 public:
  Object &operator*() const;
  Object *operator->() const {
    return std::addressof(this->operator*());
  }
};
```

- 为了让 `p->mem` 和 `(*p).mem` 等价，`operator->` 几乎总是应该这样定义。
- 注意在本例中 `operator*` 和 `operator->` 都是 `const`：它们都允许在 `const SharedPtr` 上调用。

输入、输出运算符

输入运算符需要考虑输入错误的情况：见客观题 18。

```
struct Vec3 {
  double x_, y_, z_;
  double l2_norm_;
};
std::istream &operator>>(std::istream &is, Vec3 &v) {
  is >> v.x_ >> v.y_ >> v.z_;
  if (!is) // 如果输入发生错误，要将对象置于有效的状态。
    v.x_ = v.y_ = v.z_ = 0;
  v.l2_norm_ = std::sqrt(v.x_ * v.x_ + v.y_ * v.y_ + v.z_ * v.z_);
  return is;
}
```