让自己习惯c++

条款01: 视c++ 为一个语言联邦

如今的 c++ 已经是一个 **多重泛型编程语言**，一个同时支持过程形式、面向对象形式、函数形式、泛型形式、元编程形式 的语言。将 c++ 看作一个联邦，主要包括四个部分：

- C: c++ 是 c 的继承，但是包含很多 c 语言以外的特性
- Object-Oriented c++: 这部分就是 c with class
- Template C++: 这是 c++ 的泛型编程部分
- STL: STL 是 template 程序库

条款02: 尽量以 const, enum, inline 替换掉 #define

- 对于单纯常量，最好使用 const 对象或者 enums 来替代 #define
- 对于形似函数的宏，最好使用 inline 函数来替代 #define

这条条款实际上意味着，“宁可用编译器替换预编译器”。

```
#define ASPECT_RATIO 1.6
```

上面的这种语句当发生错误时，编译器产生的错误信息可能提到的时 1.6 而非 ASPECT_RATIO。当你查找错误时，这个举动 会徒增难度。

解决的方法是 使用常量替代 上述的宏定义：

```
const double AspectRatio = 1.6;
```

这种方式不仅解决了 编译器无法找到对应的记号的问题，同时也解决了宏定义可能产生多份复制的情况。

常量替换 #define

使用常量替换 #define 还需要讨论两个小问题。

1. 常量指针

因为常量定义通常被放在头文件中，所以为了防止程序某处对其内容进行修改，需要将指针声明为 const，不仅是所指之物。

```
const char* const authorName = "hello world";
```

或者你也可以选择使用某些对象进行替换，比如上例就可以使用 string 对象来实现

```
const std::string authorName("hello world");
```

2. class 专属常量

为了将常量限制在 class scoop 中，且限制只存在一个常量。需要使用 static 关键字来实现。

```
class GamePlayer {
private:
    static const int NumTurns = 5; // 常量声明式
    int scores[NumTurns];
};

const int GamePlayer::NumTurns; // 常量定义式
```

c++ 要求你对你所使用的任何东西提供一个定义式。特别是 class 专属常量，又是 static 且为整数类型。这就需要特殊 处理。只要不取他们的地址，你可以只声明他们而不用定义。而如果你需要获取某个 class 专属常量的地址，或者编译器坚 持要看到一个定义式，你就必须提供上面最后这样类似的定义式。其中的定义式部分并没有提供特定的初值，因为在声明中已 经给出了初值，因此在定义式中不能再设置。

但是有时候，编译器可能不允许 “static 整型 class 常量完成 in class 初值设定”。而如果你又坚持要在编译期间知道 数组的大小，你可以改用所谓的 “the enum 可权充 ints 使用” 的技巧。

```
class GamePlayer {
private:
    enum { NumTurns = 5 };
    int scores[NumTurns];
};
```

这种 enum hack 方式值得学习，它具有几点好处：

1. enum hack 行为类似 #define，不能取用地址。当你不希望别人通过 pointer 或者 reference 来获取该常量可以使用。
2. 实用主义，很多代码使用了它。enum hack 实际上是 “模板元编程” 的基础技术。

inline 函数替换 #define

通常希望使用 宏定义 的方式来实现一个看起来像函数的东西，因为不会产生调用带来的额外开销。但是有时候会产生更加难 以控制的局面。

```
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 6;
CALL_WITH_MAX(++a, b); // 比较完成后，a = 7
CALL_WITH_MAX(++a, b + 10); // 比较完成后，a = 8
```

上面这段代码显然不会按照原有的设想运行，但是如果你使用 template inline 函数来书写则不会产生这种结果。

```
template <typename T>
inline void callWithMax(const T& a, const T& b) {
    f(a > b ? a : b);
}
```

除了上面这点保证，inline 也会保证遵守作用域原则，而 #define 则不会。

条款03: 尽可能使用 const

- 将某些东西声明为 const 可以帮助编译器侦测出错误用法。
- 编译器强制实施 bitwise constness，但你编写程序时应该使用 logical constness
- 当 const 和 non-const 成员函数有着实质等价的实现时，令 non-const 版本调用 const 版本可避免代码重复。

---

const 允许你指定一个语义约束，而编译器会强制实施这项约束。

关于指针的 const 使用：

- const 在 * 的左边，意味着被指物是常量
- const 在 * 的右边，意味着指针自身是常量

```
int a = 10;
int b = 11;
const int* p = &a;
int* const q = &a;
*p = 12; // 错误
p = &b;  // 正确
*q = 12; // 正确
q = &b;  // 错误
```

同样的,使用 STL 迭代器，也需要区分不同的 const：

```
std::vector<int> vec;

const std::vector<int>::iterator iter = vec.begin();
std::vector<int>::const_iterator cIter = vec.begin();
iter++; // 错误
cIter++; // 正确
*iter = 12; // 正确
*cIter = 12; // 错误
```

上述代码中前者实际相当于 iter 是不可更改的，即 `int* const` 类型，而后者则是 `const int*` 类型，即被指物是 不可更改的。

在函数声明中使用 const 进行限制，包括函数返回值、参数、函数自身，可以降低因使用者错误使用而造成的意外。

```
class Rational{
public:
    const Rational operator*(const Rational& lhs, const Rational& rhs);
};

Rational a, b, c;
...
if (a * b = c) {...} // == 意外书写为了 =
```

在上述的代码中，使用 const 来限定返回值可以防止 `(a * b) = c` 这类代码通过编译，因为这显然是不合理的。

const 成员函数

将 const 实施于成员函数，是为了确认该成员函数可作用 const 对象。

1. const 成员函数使得 class 接口比较容易理解
2. 他们使得 “操作 const 对象” 成为可能。

> [!tip] 两个成员函数，如果只是常量性不同，可以被重载

```
class TextBlock {
public:
    const char& operator[](std::size_t position) const {
        return text[position];
    }
    char& operator[](std::size_t position) {
        return text[position];
    }
private:
    std::string text;
};

void print(const TextBlock& ctb) {
    std::cout << ctb[0];
}

TextBlock tb("hello");
TextBlock ctb("hello");
tb[0] = 'x';  // 正确
ctb[0] = 'x'; // 错误
```

成员函数是 const 的含义

对于这个观点，存在两个流行概念：bitwise constess 和 logical constness

bitwise const

此阵营的人认为，**成员函数只有不更改对象之任何成员变量时才可以说是 const 的。**但是这也存在漏洞，比如将数据 存储为 char* 而非 string 的话就可能会产生问题。

```
class CTextBlock {
public:
    char& operator[] (std::size_t position) const {
        return pText[position];
    }

private:
    char* pText;
};
```

尽管 operator[] 中并没有对任何数据进行更改，但是却返回了一个指向对象内部值的 reference。如果调用者在后续对返 回值进行更改，仍然是可行的，但是却违背了 const 的约定。

logical constness

这一派则主张，一个 const 成员函数可以修改它所处理的对象内的某些 bits，但是只有在用户侦测不出的情况下才可以。

```
class CTextBlock {
public:
    std::size_t length() const;
private:
    char* pText;
    mutable std::size_t textLength;
    mutable bool lengthIsValid;
};

std::size_t CTextBlock::length() const {
    if(!lengthIsValid) {
        textLength = std::strlen(pText);
        lengthIsValid = true;
    }
    return textLength;
}
```

上面这段代码显然不符合 bitwise const，因为 length 函数改变了其中的值，但是却符合 logical constness。此外， 其中使用 mutable 关键字进行限定，其目的是当编译器坚持 bitwise const 时，使用该关键字允许释放掉 non-static 成员变量的 bitwise constness 约束。

在 const 和 non-const 成员函数中避免重复

随着代码越来越复杂，const 与 non-const 代码之间可能出现大量的重复内容，比如：

```
class TextBlock {
public:
    const char& operator[] (std::size_t position) const {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }

    char& operator[] (std::size_t position) {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }
private:
    std::string text;
};
```

上面这段代码通过使用 non-const 版本调用 const 函数的方式来进行简化，能够大大降低代码的复杂度。

```
class TextBlock {
public:
    const char& operator[] (std::size_t position) const {
        // 边界检查
        // 日志记录
        // 数据完整性校验
        return text[position];
    }

    char& operator[] (std::size_t position) {
        return const_cast<char&>(
            static_cast<const TextBlock&> (*this)[position]
        );
    }
private:
    std::string text;
};
```

如果反过来使用 const 函数调用 non-const 函数，怎么样？不要这么做，因为 const 成员函数承诺不对其中的对象进行 逻辑状态的更改，如果反向调用则会产生修改的风险。

条款04: 确定对象被使用前已经被初始化

- 为内置型对象进行手工初始化，因为 c++ 不保证初始化他们。

- 构造函数最好使用成员初值列，而不要在构造函数中使用赋值操作。初始化顺序于声明次序相同，而与初值列顺序无关。
- 为免除"跨编译单元之初始化次序"问题，使用 local static 对象替换 non-local static 对象。

关于 "将对象初始化"，c++ 似乎反复无常。但是现在，我们有一些规则，描述"对象的初始化动作何时一定发生，何时不一定 发生"。

通常，如果使用 c part of c++ 而且初始化可能招致运行期成本，那么就不会保证发生初始化。但是当你进入 non-c part of c++，则具有保证。

这似乎是个无法决定的状态，最佳的处理方法就是永远在使用对象之前先将它初始化。

对于内置类型以外的其他东西，初始化责任则落在了构造函数身上。就需要确保每一个构造函数都将对象的每一个成员初始化。

```
class PhoneNumber {};
class ABEntry {
public:
    ABEntry(const string& name, const string& address, const list<PhoneNumber>&
phones);
private:
    string theName;
    string theAddress;
    list<PhoneNumber> thePhones;
    int numTimesConsulted;
};

ABEntry::ABEntry (const string& name, const string& address, const
list<PhoneNumber>& phones)
    : theName(name), theAddress(address), thePhones(phones), numTimesConsulted(0)
{}
```

上面使用 member initialization list 来实现初始化，而非在 constructor 内使用赋值来进行初始化操作。因为，前 者发生在 constructor 开始执行本体和 default constructor 之前。这种方式效率更高。

许多 classes 拥有多个构造函数，每个构造函数都有自己的成员初值列。如果这种 classes 存在许多成员变量，多份成员 初值列的存在就会导致重复。这种情况下，可以将部分 "赋值表现和初始化一样好" 的成员变量放置在 constructor 进行 进行操作。并且可以将重合部分提取为 private 的"伪初始化"函数。

关于 "成员初始化次序"，c++ 中有着严格的次序规定：base classes 更早于 derived classes 被初始化。成员变量总是 以声明次序进行初始化，无论成员初值列的次序如何。

不同编译单元内定义之 non-local static 对象

如果你能够按照上面的这些约束进行编码，那么只剩最后一件值得关心的事情 "不同编译单元内定义之 non-local static 对象" 的初始化次序。

static 对象是指 global 对象，定义于 namespace 作用域内的对象，在 class 内、函数内、以及在 file 作用域内被声 明为 static 的对象。他们的寿命从构造出来直至程序结束为止。其中在函数内的 static 对象被称为 local

static 对象。 而其他则都被称为 non-local static 对象。

编译单元是指产生单一目标文件的那些源码。

```
// 1.cpp
class FileSystem {
public:
    std::size_t numDisk() const;
};

extern FileSystem tfs;
```

```
// 2.cpp
class Director {
public:
    Directory(params) {
        std::size_t disks = tfs.numDisks();
    }
};

Directory tempDir(params);
```

上面这两段代码，其中 tfs 以及tempDir 是 non-local static 对象，而且存放在不同的源码文件中。从中可以看到二者 存在直接关系，即 tempDir 需要借助 tfs 完成初始化操作。但是 c++ 没有规定 non-local static 对象在不同编译单元 内定义初始化的顺序，因此很容易产生错误。

解决办法十分简单，通过 singleton 模式，将 non-local static 对象转换成 local static 对象，即将前者放入一个专属函数中实现。

```
// 1.cpp
class FileSystem {
public:
    std::size_t numDisk() const;
};

FileSystem& tfs() {
    static FileSystem fs;
    return fs;
}
```

```
// 2.cpp
class Director {
public:
    Directory(params) {
        std::size_t disks = tfs().numDisks();
    }
```

```
};

Directory& tempDir() {
    static Directory td;
    return td;
}
```

这种结构下，函数十分淡出，可以通过 inline 来实现，尤其是如果他们被频繁调用的话。

> [!note] 在 more effective c++ 中，作者在条款26中提到，不要产生内含 local static 对象的 inline non-member functions。 主要是因为 inline non-member functions 存在内部连接，可能会将 local static 对象复制多份。但是，其中注释 也解释道自 1996年7月，ISO/ANSI 委员会便将 inline 函数的默认连接由内部改为了外部。因此现在可以使用 inline 来标记存在 local static 对象的 non-member functions。