条款 1：尽量用 const 和 inline 而不用#define

- #define 的问题

```
#define max(a,b) ((a) > (b) ? (a) : (b))
int a = 5, b = 0;
max(++a, b);// a 的值增加了 2 次
```

条款 2：尽量用<iostream>而不用<stdio.h>

- 优点：类型安全和可扩展性
- 增加 “<<”支持

```
class Rational {
public:
    Rational(int numerator = 0, int denominator = 1);
    ...
private:
    int n, d;// 分子，分母
    friend ostream& operator<<(ostream& s, const Rational& );
};
ostream& operator<<(ostream& s, const Rational& r) {
    s<< r.n << '/' << r.d;
    return s;
}
```

- 如果编译器同时支持 <iostream>和<iostream.h>，那头文件名的使用会很微妙。例如，如果使用了#include <iostream>，得到的是置于名字空间 std 下的 iostream 库的元素；如果使用#include <iostream.h>，得到的是置于全局空间的同样的元素。

条款 14: 确定基类有虚析构函数

- 当通过基类的指针去删除派生类的对象，而基类又没有虚析构函数时，结果将是不可确定的，派生类的析构函数肯定没有调用

条款 15: 让 operator=返回*this 的引用

- 这样可以支持赋值链操作 w = x = y = z = "hello";

条款 16：在 operator=中对所有数据成员赋值

- 继承关系下正确的赋值运算符

```
derived& derived::operator=(const derived& rhs) {
    if (this == &rhs) return *this;
    base::operator=(rhs);   // 调用 this->base::operator=
    y = rhs.y;
    return *this;
}
```

- 继承关系下正确的拷贝构造函数

```
class derived: public base {
public:
    derived(const derived& rhs): base(rhs), y(rhs.y) {}
    ...
}
```

条款 17: 在 operator=中检查给自己赋值的情况

```
c& c::operator=(const c& rhs) {
    if (*this == rhs) return *this;
    ...
}
```

条款 20: 避免 public 接口出现数据成员

- 在 public 接口里放上数据成员无异于自找麻烦，所以要把数据成员安全地隐藏在与功能分离的高墙后。如果现在就开始这么做，那我们就可以无需任何代价地换来一致性和精确的访问控制。

条款 21：尽可能使用 const

- 以下两函数可以重载，const 函数内部不能显示的修改成员变量，申明为 mutable 的成员变量除外

```
int &f()
int &f() const
```

- const_cast<char*>const_buf 同等于 c 方式的(char *)const_buf

条款 23：必须返回一个对象时不要试图返回一个引用

- 引用函数外部的对象除外

条款 25：避免对指针和数字类型重载

- void * const null = 0; //可能的 null 定义

条款 26：当心潜在的二义性

```
class B;                    //对类 B 提前声明
class A {
public:
    A(const B&);            //可以从 B 构造出类 A
};
class B {
public:
    operator A() const;     //可以从 B 转换出类 A，此为类型转换，同
operator char*()，仿函数格式为 int operator()(int x)
};
```

```
void f(const A&) {
    …
}
B b;
f(b);// 错误!——二义
```

条款 27：如果不想使用隐式生成的函数就要显式地禁止它

```
template<class T>
class Array {
private:
    //只申明，不要定义这个函数!
    Array& operator=(const Array& rhs);
    ...
};
```

条款 30：避免这样的成员函数：其返回值是指向成员的非 const 指针或引用，但成员的访问级比这个函数要低

- 劳累的编译器要费九牛二虎之力来确保你设置的访问限制不被破坏，你也不要糟蹋编译器的努力结果，返回 const 的引用是例外

条款 31：千万不要返回局部对象的引用，也不要返回函数内部用 new 初始化的指针的引用

- 写一个返回废弃指针的函数无异于坐等内存泄漏的来临。

条款 33：明智地使用内联

- 一般来说，实际编程时最初的原则是不要内联任何函数，除非函数确实很小很简单
- 慎重地使用内联，不但给了调试器更多发挥作用的机会，还将内联的作用定位到了正确的位置：它是一个根据需要而使用的优化工具。一个程序往往花 80%的时间来执行程序中 20%的代码。这是一条很重要的定律，因为它提醒你，作为程序员的一个很重要的目标，就是找出这 20%能够真正提高整个程序性能的代码。一旦找出了程序中那些重要的函数，以及那些内联后可以确实提高程序性能的函数（这些函数本身依赖于所在系统的体系结构），就要毫不犹豫地声明为 inline。同时，要注意代码膨胀带来的问题，并监视编译器的警告信息（参见条款 48），看看是否有内联函数没有被编译器内联。

条款 34：将文件间的编译依赖性降至最低

- 如果可以使用对象的引用和指针，就要避免使用对象本身。定义某个类型的引用和指针只会涉及到这个类型的声明。定义此类型的对象则需要类型定义的参与。
- 尽可能使用类的声明，而不使用类的定义。因为在声明一个函数时，如果用到某个类，是绝对不需要这个类的定义的，即使函数是通过传值来传递和返回这个类。

- 不要在头文件中再#include 其它头文件，除非缺少了它们就不能编译。相反，要一个一个地声明所需要的类，让使用这个头文件的用户自己（通过#include 指令）去包含其它的头文件，以使用户代码最终得以通过编译。一些用户会抱怨这样做对他们来说很不方便，但实际上你为他们避免了许多你曾饱受的痛苦。事实上，这种技术很受推崇，并被运用到 C++标准库。
- 使用代理（包含一个指向具体实现类的指针，并转发调用，参与的类：句炳/信封类、主体/信件类）和接口（参与的类：抽象基类、派生的实现类）可以降低依赖

条款 35: 使公有继承体现 "是一个" 的含义

- 反例：正方形是一个矩形吗？对矩形适用的规则（宽度的改变和高度没关系）不适用于正方形（宽度和高度必须相同）。但公有继承声称：对基类对象适用的任何东西也适用于派生类对象

条款 36: 区分接口继承和实现继承

- 定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。
- 声明一般虚函数的目的在于，使派生类继承函数的接口和缺省实现。（同时提供函数接口和缺省实现是很危险的，子类可能由于疏忽没定义而继承默认行文。不被推荐使用）
- 定义纯虚函数，同时实现它（做到了接口和实现的分离，子类必需重新实现，推荐使用）

```
class Airplane {
public:
    virtual void fly(const Airport& destination) = 0;
    ...
};
void Airplane::fly(const Airport& destination) {
    ...
}
```

- 声明非虚函数的目的在于，使派生类继承函数的接口和强制性实现。

条款 37: 决不要重新定义继承而来的非虚函数

```
class B {
public:
    void mf();
    ...
};
class D: public B {
public:
    void mf(); // 隐藏了 B::mf;
    ...
};
```

- 技术上：派生类会隐藏基类的函数，调用基类函数还是派生类函数取决于指向它的指针所声明的类型

- 理论上：如果 D 重新定义了 mf，设计中就会产生矛盾。如果 D 真的需要实现和 B 不同的 mf，那么每个 D 将不 "是一个" B。相反，如果 D 真的必须从 B 公有继承，而且 D 真的需要和 B 不同的 mf 的实现，那么，mf 就没有为 B 反映出特殊性上的不变性。

条款 38: 决不要重新定义继承而来的缺省参数值

- 虚函数是动态绑定而缺省参数值是静态绑定的。

条款 39: 避免 "向下转换" 继承层次

- "向下转换" 可以通过几种方法来消除。最好的方法是将这种转换用虚函数调用（建议纯虚）来代替，同时，它可能对有些类不适用，所以要使这些类的每个虚函数成为一个空操作。第二个方法是加强类型约束，使得指针的声明类型和你所知道的真的指针类型之间没有出入。
- 有些情况下，比如只读的第三方库，真的不得不执行向下转换。
- 任何时候发现自己写出 "如果对象属于类型 T1，做某事；但如果属于类型 T2，做另外某事" 之类的代码，就要扇自己一个耳光。

条款 40: 通过分层来体现 "有一个" 或 "用...来实现"

条款 41: 区分继承和模板

- 当对象的类型不影响类中函数的行为时，就要使用模板来生成这样一组类。
- 当对象的类型影响类中函数的行为时，就要使用继承来得到这样一组类。
- 模板导致的 "代码膨胀"。

条款 42: 明智地使用私有继承

- 私有继承意味着只是继承实现，接口会被忽略。私有继承意味着 "用...来实现"。分层" 也具有相同的含义。怎么在二者之间进行选择呢？答案很简单：尽可能地使用分层，有保护成员或虚函数介入的时候，私有继承是表达类之间 "用...来实现" 关系的唯一有效途径。
- 这是一段令人惊叹的代码，虽然你可能一时还没意识到。因为这是一个模板，编译器将根据你的需要自动生成所有的接口类。因为这些类是类型安全的，用户类型错误在编译期间就能发现。因为 GenericStack 的成员函数是保护类型，并且接口类把 GenericStack 作为私有基类来使用，用户将不可能绕过接口类。因为每个接口类成员函数被（隐式）声明为 inline，使用这些类型安全的类时不会带来运行开销；生成的代码就象用户直接使用 GenericStack 来编写的一样（假设编译器满足了 inline 请求 ---- 参见条款 33）。因为 GenericStack 使用了 void*指针，操作堆栈的代码就只需要一份，而不管程序中使用了多少不同类型的堆栈。简而言之，这个设计使代码达到了最高的效率和最高的类型安全。很难做得比这更好。

```
class GenericStack {
protected:
    GenericStack();
```

```
    ~GenericStack();
    void push(void *object);  //因为使用了 void*指针，操作堆栈的代码
就只需要一份
    void * pop();             //同上
    bool empty() const;
private:
    struct StackNode {
        void *data; // 节点数据
        StackNode *next;  // 下一节点
        StackNode(void *newData, StackNode *nextNode)
            : data(newData), next(nextNode) {}
    };
    StackNode *top; // 栈顶
    GenericStack(const GenericStack& rhs);  // 防止拷贝和赋值
    GenericStack&  operator=(const GenericStack& rhs);
};
template<class T>
class Stack: private GenericStack {
public:
    void push(T *objectPtr) {          //代码为隐形 inline
        GenericStack::push(objectPtr);
    }
    T * pop() {                       //代码为隐形 inline
        return static_cast<T*>(GenericStack::pop());
    }
    bool empty() const {              //代码为隐形 inline
        return GenericStack::empty();
    }
};
```

条款 43：明智地使用多继承

- 单继承的层次结构只需要非虚基类，继承层次结构中参数的向上传递采用的是一种很自然的方式：第 n 层的类将参数传给第 n-1 层的类。但是，虚基类的构造函数则不同，它的参数是由继承结构中最底层派生类的成员初始化列表指定的。这就造成，负责初始化虚基类的那个类可能在继承图中和它相距很远；如果有新类增加到继承结构中，执行初始化的类还可能改变。（避免这个问题的一个好办法是：消除对虚基类传递构造函数参数的需要。最简单的做法是避免在虚基类中放入数据成员）。

条款 44：说你想说的；理解你所说的

- 共同的基类意味着共同的特性。
- 公有继承意味着 "是一个"。
- 私有继承意味着 "用...来实现"。
- 分层意味着 "有一个" 或 "用...来实现"。

下面的对应关系只适用于公有继承的情况

- 纯虚函数意味着仅仅继承函数的接口。

- 简单虚函数意味着继承函数的接口加上一个缺省实现。
- 非虚函数意味着继承函数的接口加上一个强制实现。

条款 45: 弄清 C++在幕后为你所写、所调用的函数

- 针对空类，编译器自动生成的代码

```
class Empty {}; //该定义和以下定义一样
class Empty {
public:
    Empty();                    // 缺省构造函数
    Empty(const Empty& rhs);    // 拷贝构造函数
    ~Empty();                   // 析构函数，是否为虚函数看下文说明
    Empty&
    operator=(const Empty& rhs);// 赋值运算符
    Empty* operator&();         // 取址运算符
    const Empty* operator&() const;
};
inline Empty::Empty() {}
inline Empty::~Empty() {}
inline Empty * Empty::operator&() {
    return this;
}
inline const Empty * Empty::operator&() const {
    return this;
}
```

至于拷贝构造函数和赋值运算符，官方的规则是：缺省拷贝构造函数/赋值运算符对类的非静态数据成员进行 "以成员为单位的" 逐一拷贝构造/赋值。

条款 47：确保非局部静态对象在使用前被初始化

- 如果你让对象 A 必须在对象 B 之前初始化，同时又让 A 的初始化依赖于 B 已经被初始化，可以使用函数内部静态变量。
- 对于函数内部静态对象什么时候被初始化，C++却明确指出：它们在函数调用过程中初次碰到对象的定义时被初始化。
- 避免多线程环境下使用

条款 49: 熟悉标准库

- 旧的 C++头文件名如<iostream.h>将会继续被支持，尽管它们不在官方标准中。这些头文件的内容不在名字空间 std 中。(GCC4.4.0 以后版本已不支持)
- 新的 C++头文件如<iostream>包含的基本功能和对应的旧头文件相同，但头文件的内容在名字空间 std 中。
- 标准 C 头文件如<stdio.h>继续被支持。头文件的内容不在 std 中。
- 具有 C 库功能的新 C++头文件具有如<cstdio>这样的名字。它们提供的内容和相应的旧 C 头文件相同，只是内容在 std 中。

- 标准库大量的使用模版，因此不要手工声明 string（或标准库中其它任何部分）。相反，只用包含一个适当的头文件，如<string>。
- 支持国际化最主要的构件是 facets 和 locales。

条款 50: 提高对 C++的认识

```
class Base {
public:
    virtual void f(int x);
};
class Derived: public Base {
public:
    virtual void f(double *pd);
};
Derived *pd = new Derived;
pd->f(10);          // 错误!

class Derived: public Base {
public:
    using Base::f;        // 将 Base::f 引入 Derived 的空间范围
    virtual void f(double *pd);
};
Derived *pd = new Derived;
pd->f(10); // 正确，调用 Base::f
```