

C++ 高质量编程



可维护性

- 减少预编译和宏的使用

```
// Bad Idea  
#define PI 3.14159;
```

使用预编译的宏会导致你和编译器看到的代码是不同的，尤其是宏是具有全局作用域的，这很可能导致一些异常产生。

```
// Good Idea  
namespace my_project {  
    class Constants {  
    public:  
        // if the above macro would be expanded, then the following line would be:  
        // static const double 3.14159 = 3.14159;  
        // which leads to a compile-time error. Sometimes such errors are hard to understand.  
        static const double PI ;  
    };  
    const double constants::PI = 3.14159265;  
}
```

- 编译器运行之前实现，难以debug
- 使用namespace



可维护性

- 减少预编译和宏的使用

```
// Bad Idea
#define PI 3.14159;

// Good Idea
namespace my_project {
    class Constants {
    public:
        // if the above macro would be expanded here
        // static const double 3.14159 = 3.14159;
        // which leads to a compile-time error.
        static const double PI = 3.14159;
    };
}
```



此类变量只有整形或枚举可以类内初始化

- 编译器运行之前实现，难以debug
- 使用namespace



可维护性

- 改进for循环的写法
 - 传统for循环

```
for (auto itr = vec1.begin();  
     itr != vec1.end();  
     ++itr) {  
    // do something with itr  
}
```

- 需构建迭代器，需在循环的每一次做判断



可维护性

- 改进for循环的写法
 - 错误难以发现

```
for (auto itr = vec1.begin();  
     itr != vec2.end();  
     ++itr) {  
    // do something with itr  
}
```



可维护性

- 改进for循环的写法
 - 改进的写法

```
for (const auto &i : vec1) {  
    // do something with i  
    // i does not have to be dereferenced  
}
```

- 有什么缺点？



可维护性

- override 和 final

```
class B
{
public:
    virtual void f(short) {std::cout << "B::f" << std::endl;}
};
```

```
class D : public B
{
public:
    virtual void f(int) {std::cout << "D::f" << std::endl;}
};
```



可维护性

- **override** 和 **final**

- **override**, 表示函数应当重写基类中的虚函数。

```
class B
```

```
{
```

```
public:
```

```
    virtual void f(short) {std::cout << "B::f" << std::endl;}  
};
```

```
class D : public B
```

```
{
```

```
public:
```

```
    virtual void f(int) override {std::cout << "D::f" << std::endl;}  
};
```

编译器此时会帮助报错！



可维护性

- `override` 和 `final`
 - `final`, 表示派生类不应当重写这个虚函数

```
class B
```

```
{...};
```

```
class D : public B
```

```
{
```

```
public:
```

```
    virtual void f(int) override final {std::cout << "D::f" << std::endl;}  
};
```

```
class F : public D
```

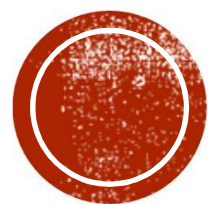
```
{
```

```
public:
```

```
    virtual void f(int) override {std::cout << "F::f" << std::endl;}  
};
```

编译器此时会帮助报错！





良好的可扩展性



案例5：24点程序设计

- 题目：给定1~10的4个整数，列出所有由这4个数通过加减乘除得到24的表达式
- 例如：
 - 输入：1 2 3 4
 - 输出： $1 * 2 * 3 * 4 = 24$
 - 输入：4 4 10 10
 - 输出： $(10 * 10 - 4) / 4 = 24$
- 思路1：
 - 枚举所有可能的表达式模板，如：
 - $A + B + C + D = 24$
 - $A * B * C * D = 24$
 - $A * (B + C) + D = 24$
 -



案例5：24点程序设计（方案1）

```
// @return: a +/- b +/- c +/- d
// @param isPlus1, isPlus2, isPlus3: true if the
operation is plus,
// e.g. a + b - c + d then (true, false ,true).
const double eps = 0.000001;
void TryFuncType1(int a, int b, int c, int d,
bool isPlus1, bool isPlus2, bool isPlus3)
{
    double sum = a;
    sum = isPlus1 ? sum + b : sum - b;
    sum = isPlus2 ? sum + c : sum - c;
    sum = isPlus3 ? sum + d : sum - d;

    ...
}
```



案例5：24点程序设计（方案1）

...

```
if (abs(sum - 24) < eps)
{
    printf("%d %c %d %c %d %c %d = 24",
        a, isPlus1 ? '+' : '-',
        b, isPlus2 ? '+' : '-',
        c, isPlus3 ? '+' : '-', d);
}
```

}

还有 TryFuncType2,3,4,...



案例5：续

```
void Foo(int numbers[])
```

```
{
```

```
    for (int i0 = 0; i0 < 4; i0++)
```

```
    for (int i1 = 0; i1 < 4; i1++)
```

```
    for (int i2 = 0; i2 < 4; i2++)
```

```
    for (int i3 = 0; i3 < 4; i3++)
```

```
    for (int o0 = 0; o0 < 2; o0++)
```

```
    for (int o1 = 0; o1 < 2; o1++)
```

```
    for (int o2 = 0; o2 < 2; o2++)
```

```
    {
```

```
        if (!AllDistinct(i0, i1, i2, i3)) continue;
```

```
        TryFuncType1(numbers[i0], numbers[i1],
```

```
numbers[i2], numbers[i3],
```

```
            o0, o1, o2);
```

```
        .....
```

```
    }
```

```
}
```

最大的缺点：
无法扩展！！



可扩展的代码

- 如果需求变了呢？
 - 输入的整数个数不限制为4
 - 输入目标值不是24
 - 运算符不限于加减乘除
 - 要求输出不重复的所有表达式
- 不要局限在实现细节上，先把问题理解成抽象的模型



把运算过程抽象化

- $1 + 2 * 3 - 4 = ?$
- 运算符
 - 给两个数，返回一个数
 - 有优先级，优先度高的先计算
 - 是否可交换
- 计算表达式的过程
 - 找优先度最高的一个运算符做一次运算，并用结果替代原表达式，以此类推
$$1 + 2 * 3 - 4$$
$$1 + 6 - 4$$
$$7 - 4$$
$$3$$



课后练习：函数接口的设计

- 需求是什么？
 - 给定任意个数
 - 目标值为任意数
 - 可以自定义运算符
 - 输出所有可以通过自定义的运算由给定的几个数算出目标值的计算表达式
- 例如：`void Foo(std::vector<double> numbers, double target, ?, ...)`
- 具体题目以网络学堂为准



设计模式

- 设计模式（Design pattern）是一套（23种）被反复使用的代码设计经验的总结。目的是为了设计可复用代码，可以让代码更容易被理解，也保证了代码的可靠性。
- 设计模式的应用发自具体的设计需求，不是为了用设计模式而用设计模式。
- 几个原则：
 - 开闭原则
 - 里氏代换原则
 - 依赖倒转原则
 - 接口隔离原则
 - 迪米特法则
 - ...
- 常用的几种设计模式（观察者模式、策略模式、命令模式、工厂模式）



案例6:

- 设计一个监视器类，当监视器检测到门被打开的时候发出通知
 - 假设已有函数 `bool IsDoorOpened()` 可以判断当前时刻门是否被打开
- 主要逻辑：

```
while (true)
{
    if (IsDoorOpened())
    {
        // Notify
    }
}
```



实现1

```
void DoorMonitor::Monitor()  
{  
    while (true)  
    {  
        if (IsDoorOpened())  
        {  
            SendMail("A", "Door is  
opened");  
        }  
    }  
}
```

问题：现在B也想被通知，怎么办？



实现2

```
void DoorMonitor::Monitor(std::vector<std::string>
toList)
{
    while (true)
    {
        if (IsDoorOpened())
        {
            for (auto name : toList)
            {
                SendMail(name, "Door is opened");
            }
        }
    }
}
```

问题： C希望被打电话， D希望被发短信通知



实现3

```
void DoorMonitor::Monitor(std::vector<std::string> toList,
                          std::vector<std::string> callList,
                          std::vector<std::string> smsList)
{
    while (true)
    {
        if (IsDoorOpened())
        {
            std::for_each(toList.begin(), toList.end(),
                [](std::string name) { SendMail(name, "Door is opened"); });

            std::for_each(callList.begin(), callList.end(),
                [](std::string name) { Call(name); });

            std::for_each(smsList.begin(), smsList.end(),
                [](std::string name) { SendSMS(name); });
        }
    }
}
```

问题：又来了一个人，想用微信通知。 T_T



观察者模式

- 观察者(人) -> 观察物(监视器+门) -> 变化事件(门开了)
- 观察者关心事件、观察物产生事件
- 观察物预先不知道且不想知道有哪些观察者和通知方式

```
class IObserver
{
public:
    virtual void
Update() = 0;
};
```

```
class MailObserver: public
IObserver
{
public:
    virtual void Update()
    {
        SendMail(name, "Door is
opened!");
    }
private:
    std::string name;
};
```



实现4

```
void DoorMonitor::Monitor(std::vector<IObserver*>
observers)
{
    while (true)
    {
        if (IsDoorOpened())
        {
            for (auto pObserver : observers)
            {
                pObserver->Update();
            }
        }
    }
}
```

问题：监视途中，B觉得太烦，不想收通知了。

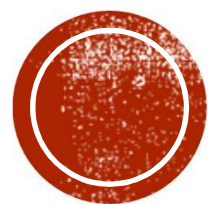


观察者模式（续）

```
class Observable
{
public:
    virtual void RegisterObserver(IObserver*
pObserver);
    virtual void UnregisterObserver(IObserver*
pObserver);
    virtual void Monitor();

private:
    std::list<IObserver*> observers;
};
```

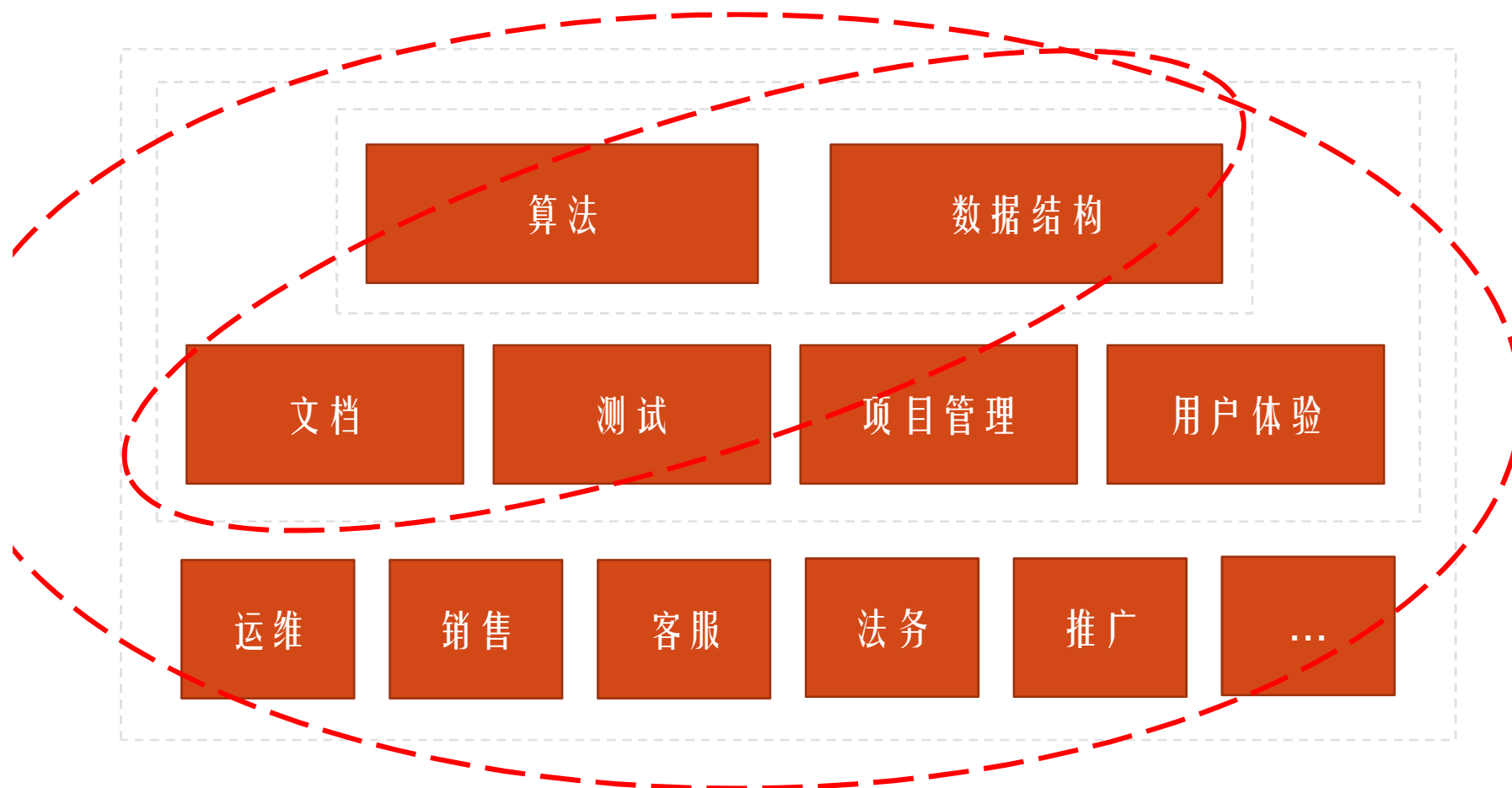




程序的安全性



算法之外?



注：个人意见， 仅供

“我的代码是对的”

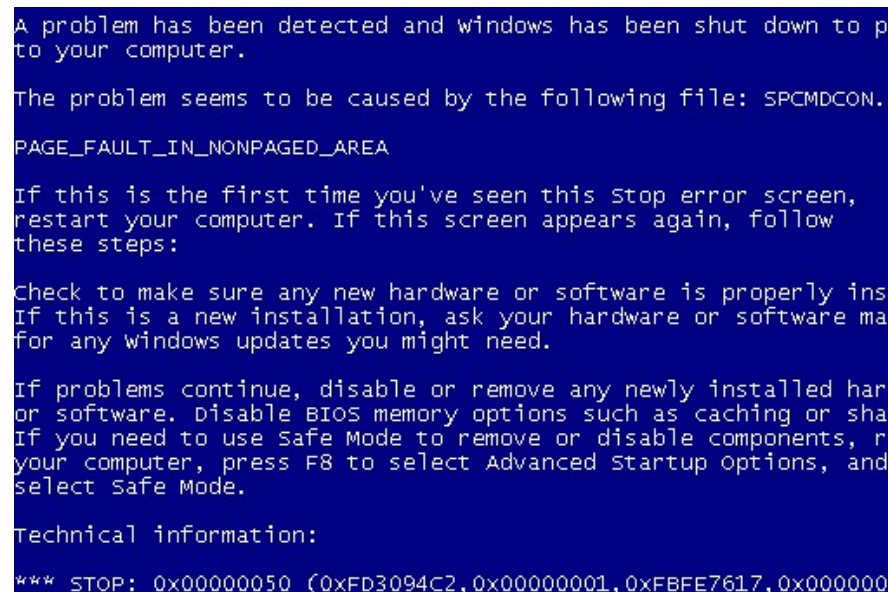
- 程序在正确的输入时候输出正确
- 程序在错误的输入时候仍能正常处理



程序出错之后...



崩溃



蓝屏



“所有输入都是邪恶的”

- 所有用户输入都需要 **足够 & 正确** 地验证

```
int GetAt(int pos)
{
    if (pos < 0 || pos > _size)
        return -1;
    ...
}
```



异常处理

- 输入的判断
- 运行中的数据检验
 - 数据类型，数据范围
- 运行结果的判断



CASE: 文档中的问题

```
CreateProcess(..., "C:\\Program Files  
(x86)\\A.exe", ...);
```

<= "C:\\Program Files (x86)\\A.exe" ?

<= "C:\\Program" "Files (x86)\\A.exe" ?

"c:\\program files\\sub dir\\program name"

c:\\program.exe files\\sub dir\\program name

c:\\program files\\sub.exe dir\\program name

c:\\program files\\sub dir\\program.exe name

c:\\program files\\sub dir\\program name.exe



其他安全性

- 仔细定义函数的返回值
 - 类的成员变量
 - 使用&或const &可减小程序开销
 - 值返回有助于thread safety
 - 对类的成员变量，如果在成员函数中修改，则函数对数被的多线程调用，就会有危险，因此用值返回修改，
 - 局部变量
 - 必须及时值返回



其他安全性

- 不要对内置类型使用**const &**: 引用对应指针操作, 较慢

```
// Very Bad Idea
class MyClass
{
public:
    explicit MyClass(const int& t_int_value)
        : m_int_value(t_int_value)
    {
    }

    const int& get_int_value() const
    {
        return m_int_value;
    }

private:
    int m_int_value;
}
```

```
// Good Idea
class MyClass
{
public:
    explicit MyClass(const int t_int_value)
        : m_int_value(t_int_value)
    {
    }

    int get_int_value() const
    {
        return m_int_value;
    }

private:
    int m_int_value;
}
```



其他安全性

- 避免直接的内存访问

```
// Bad Idea
MyClass *myobj = new MyClass;

// ...
delete myobj;

// Good Idea
auto myobj = std::make_unique<MyClass>(constructor_param1, constructor_param2); // C++14
auto myobj = std::unique_ptr<MyClass>(new MyClass(constructor_param1, constructor_param2)); // C++11
auto mybuffer = std::make_unique<char[]>(length); // C++14
auto mybuffer = std::unique_ptr<char[]>(new char[length]); // C++11

// or for reference counted objects
auto myobj = std::make_shared<MyClass>();

// ...
// myobj is automatically freed for you whenever it is no longer used.
```

其他安全性

- 使用std::array（定长数组）和std::vector（变长数组）
 - 变长数组并不是C++标准中所给的；
 - 容易访问到无效的地址引起崩溃，是非常危险的使用法



其他安全性

- 使用C++风格的类型转换，而非C风格

```
// Bad Idea  
double x = getX();  
int i = (int) x;
```

`static_cast`更安全一些，对于指针操作的话，多了一些检查，例如无关指针就无法转换，父类指针向孩子指针，无法转换，常指针向非常指针无法转换。

```
// Not a Bad Idea  
int i = static_cast<int>(x);
```

- 自学其他的类型转换方式



线程控制

- 减少全局变量的使用
 - 可见性大就意味着可能在很多地方被读写
 - 静态变量 `static`
 - 共享指针 `std::shared_ptr` (线程安全)
 - 单例模式 (Singleton)
 - 一个类只有一个对象
 - 该对象可全局访问



强类型枚举

- 传统枚举
 - 暴露在外层作用域中
 - 这样若是同一作用域下有两个不同的枚举类型，但含有相同的枚举常量也是不可的
 - 无法拥有特定的用户定义类型
 - `enum class Color:char{RED, GREEN, BLACK, WHITE};`
- 强类型枚举

```
enum class Options: unsigned int {None, One, All};  
Options o = Options::All;
```



其 他

- 智能指针
- 匿名函数lambda
- ○ ○ ○



单元测试

- 测试自己所写的代码，每个函数和代码单元都进行测试
- 测试驱动开发 (Test-driven development) :
 - 先编写测试代码
 - 只编写使测试通过的代码
- 开源测试框架： gtest, cppunit



代码评审 (CODE REVIEW)

- 更好地发现BUG
- 提高代码的质量
- 提高开发速度
- 控制评审时间
 - 只看一个小单元，这样能够提高效率



参 考 资 料

- Google Style Guide
 - <https://google.github.io/styleguide/cppguide.html>
 - <https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/contents/> (中文翻译版)
- `cpplint`: 可以自动识别代码是否符合google style c++的编程规范



总结

- 良好的可维护性
 - 统一代码风格
 - 命名规范
 - `const` 关键词
 - 注释
 - 传值和引用
 - 资源的释放
 - 编程规范
 - 其他可维护性
- 良好的可扩展性
 - 代码抽象
 - 设计模式
- 安全性
 - 输入判断
 - 异常处理
 - 路径的写法
- 其他

