C++高质量编程

徐枫

清华大学软件学院

feng-xu@tsinghua.edu.cn

可维护性

- 减少预编译和宏的使用

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。

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

// 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 understa
nd.
    static const double PI ;
  };
    const double constants::PI = 3.14159265;
}
```

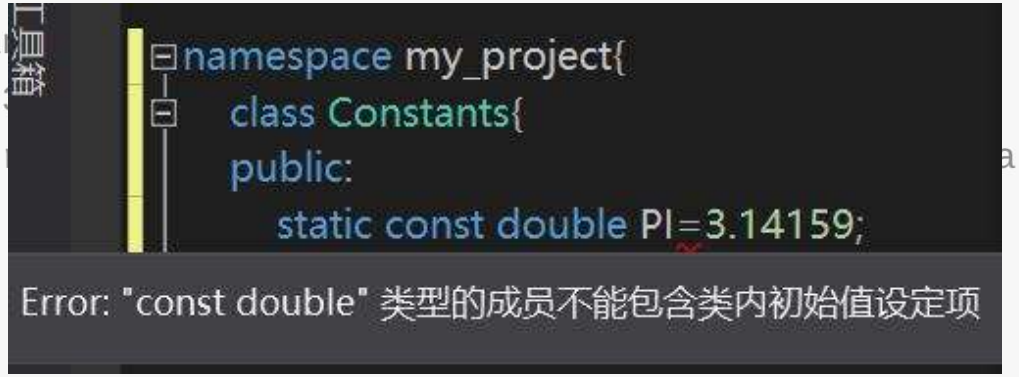
- 编译器运行之前实现，难以debug
- 使用namespace

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 expa
    //   static const double 3.14159 =
    // which leads to a compile-time er
nd.
    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?

范围 FOR 迭代

- 从 C++ 11 引入
- 支持迭代的元素只读或可修改

```
for (auto element : vec)
    std::cout << element << std::endl; // read only
for (auto &element : vec) {
    element += 1;                      // writeable
}
```

STL容器

- std::array
  - 定长数组，比std::vector高效
  - 代替传统数组，可以查询大小size()和容量capacity()
- std::tuple
  - 存放多个、不同类型的数据
  - 例如，让函数能够返回多个不同类型的值

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;
};
```

抽象类
纯虚函数

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

实现4

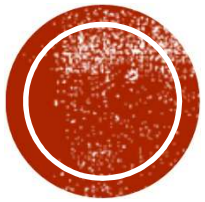
```
void  DoorMonitor::Monitor(std::vector<IObserver*>
observers)
{
    while (true)
    {
        if (IsDoorOpened())
        {
            for (auto pObserver : observers)
            {
                pObserver->Update();
            }
        }
    }
}
```

问题：监视途中，B觉得太烦，不想收通知了。

观察者模式（续）

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

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



程序的安全性

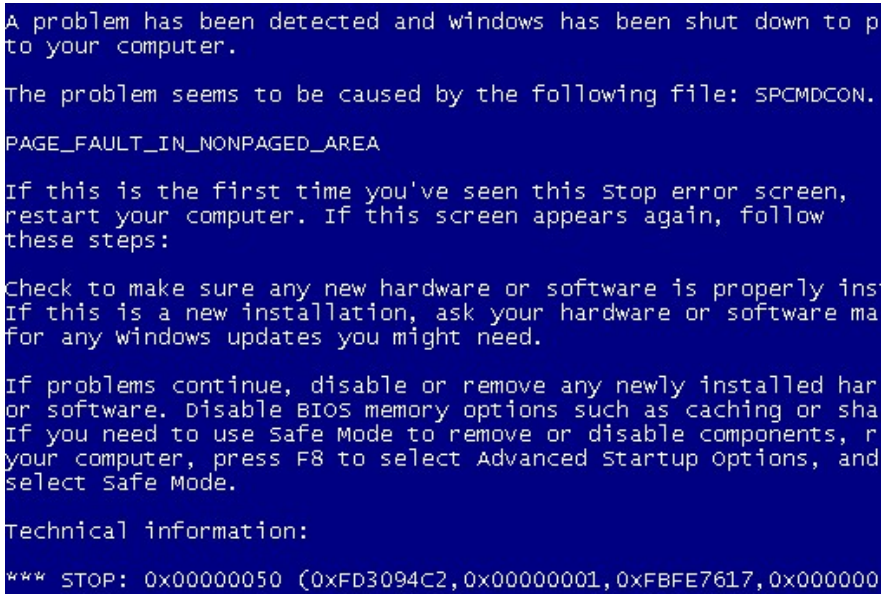
“我的代码是对的”

- 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...


nternet Information Services
Internet Information Services has encountered a problem and needs to close. We are sorry for the inconvenience.
If you were in the middle of something, the information you were working on might be lost.
Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Internet Information Services. We will treat this report as confidential and anonymous.
To see what data this error report contains, click here.
Debug
Send Error Report
Don't Send

崩溃


A problem has been detected and windows has been shut down to p
to your computer.
The problem seems to be caused by the following file: SPCMDCON.
PAGE_FAULT_IN_NONPAGED_AREA
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:
Check to make sure any new hardware or software is properly ins
If this is a new installation, ask your hardware or software ma
for any Windows updates you might need.
If problems continue, disable or remove any newly installed har
or software. Disable BIOS memory options such as caching or sha
If you need to use Safe Mode to remove or disable components, r
your computer, press F8 to select Advanced Startup Options, and
select Safe Mode.
Technical information:
*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x000000

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
      - 否则，返回的引用可以修改成员变量，某些成员函数也可以，多线程可能会冲突。
  - 局部变量
    - 只能返回值，不能返回引用或指针

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_para
m2)); // 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C++风格的类型转换，而非C风格

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

// Not a Bad Idea
int i = static_cast<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。

- 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（枚举类）

- 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
- 。。。

函数对象容器 STD::FUNCTION

- 类似函数指针
- Lambda 表达式的本质是闭包对象（类似函数对象），可以转换为函数指针值作为另一个函数的参数，例如：

```
using foo = void(int); // 定义函数类型，using 的使用见上一节中的别名语法
void functional(foo f) { // 参数列表中定义的函数类型 foo 被视为退化后的函数指针类型 foo*
    f(1); // 通过函数指针调用函数
}

int main() {
    auto f = [](int value) {
        std::cout << value << std::endl;
    };
    functional(f); // 传递闭包对象，隐式转换为 foo* 类型的函数指针值
```

STD::BIND 和 STD::PLACEHOLDER

- std::bind 则是用来绑定函数调用的参数的，
- 它解决的需求是我们有时候可能并不一定能够一次性获得调用某个函数的全部参数，通过这个函数，我们可以将部分调用参数提前绑定到函数身上成为一个新的对象，然后在参数齐全后，完成调用。例如：

```
int foo(int a, int b, int c) {
    ;
}
int main() {
    // 将参数1,2绑定到函数 foo 上，
    // 但使用 std::placeholders::_1 来对第一个参数进行占位
    auto bindFoo = std::bind(foo, std::placeholders::_1, 1,2);
    // 这时调用 bindFoo 时，只需要提供第一个参数即可
    bindFoo(1);
}
```

初始化列表

- C++11 引入 std::initializer_list，允许构造函数或其他函数像参数一样使用初始化列表，可以传入任意数量的参数

```
class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
             it != list.end(); ++it)
            vec.push_back(*it);
    }
};
int main() {
    // after C++11
    MagicFoo magicFoo = {1, 2, 3, 4, 5};
```

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