# Modern C++ Coding

answeror@gmail.com

2013-11-30

*A cynical answer is that many people program in C++, but do not understand and/or use the higher level functionality. Sometimes it is because they are not allowed, but many simply do not try (or even understand).*

*As a non-boost example: how many folks use functionality found in* `<algorithm>`*?*

*In other words, many C++ programmers are simply C programmers using C++ compilers, and perhaps* `std::vector` *and* `std::list`*. That is one reason why the use of* `boost::tuple` *is not more common.*[1]

[1] `http://stackoverflow.com/a/855478`

# Outline

内存管理

抽象

*C++ is the best language for garbage collection*
*principally because it creates less garbage.*

— Bjarne Stroustrup

# 智能指针 (smart pointer)

less pointer

# 智能指针 (smart pointer)

less pointer

less new

# 智能指针 (smart pointer)

less pointer

less new

never delete

# 智能指针 (smart pointer)

- 共享所有权, 共享使用权: `shared_ptr`

[2] 良好初始化和管理的裸指针被广泛使用, 但不建议新手使用.

# 智能指针 (smart pointer)

- 共享所有权, 共享使用权: `shared_ptr`
- 独占所有权, 独占使用权: `unique_ptr`, `boost::scoped_ptr`

[2] 良好初始化和管理的裸指针被广泛使用, 但不建议新手使用.

# 智能指针 (smart pointer)

- 共享所有权, 共享使用权: `shared_ptr`
- 独占所有权, 独占使用权: `unique_ptr`, `boost::scoped_ptr`
- 没有所有权, 具有使用权: 引用[2], `ref`, `boost::optional`

[2] 良好初始化和管理的裸指针被广泛使用, 但不建议新手使用.

# 智能指针 (smart pointer)

- 共享所有权, 共享使用权: `shared_ptr`
- 独占所有权, 独占使用权: `unique_ptr`, `boost::scoped_ptr`
- 没有所有权, 具有使用权: 引用[2], `ref`, `boost::optional`

共享权指生命周期的控制, 使用权指可拷贝性.

[2] 良好初始化和管理的裸指针被广泛使用, 但不建议新手使用.

# 共享所有权, 共享使用权

```
std::shared_ptr<model> load_model(const std::string &path);
...
auto m = load_model("learnt-model.xml");
output_model_info(m);
algo1.set_model(m);
algo2.set_model(m);
```

Python, Java 等语言里的绝大多数变量均共享所有权和使用权. 对应到 C++ 里一般都可以用`shared_ptr`处理. 原则上被共享的类型最好是不可变的, 即仅包含const public 成员.

# 独占所有权, 独占使用权

```
{
    typedef some_large_class_not_sutable_on_stack type;
    boost::scoped_ptr<type> foo(new type);
    // use foo
}
// foo being released automatically
```

在任何情况下, 都应优先使用独占所有权和使用权的智能指针. 因为共享通常意味着可控性的降低, 意味着更多的 bug.

# 没有所有权, 具有使用权

```
struct wrapper {
    const foo &base;
    wrapper(const foo &base) : base(base) {}
    std::string name() const {
        return "wrapped" + base.name()
    }
};
```

托管生命周期的共享是 C++ 内存管理的精髓. 引用和 ref 都是为了减少裸指针的误用而存在的. boost 库本身极少使用shared_ptr, 而是把对象生命周期托管给库的使用者.

# 没有所有权, 具有使用权

```
using boost::irange;
using boost::adaptors::filtered;

auto even = irange<int>(0, 10) | filtered([](int x){
    return x & 1 == 0;
});
```

内存管理责任的移交意味着高效和易错. 上述代码中even区间在构造完毕时就已经失效了.

# 句柄 (handle)[3]

less `shared_ptr`

[3]参见 Accelerated C++, 不要与操作系统里的 (文件, 窗口等) 句柄混淆.

# 句柄的两种用途

- 隐藏实现.

# 句柄的两种用途

- 隐藏实现.
- 将 (共享) 语义与类型关联.

# 例子

```
// qda.hpp
class QDA {
public:
    QDA();
    ~QDA();

    QDA fit(const np::mat &X, const np::ivec &y);
    np::ivec predict(const np::mat &X) const;

protected:
    struct impl;
    yapimpl::shared<impl> m;
};
```

```cpp
// qda.cpp
struct qda::impl {
    np::mat internal_data;
    struct method : qda {
        np::mat decision_function(const np::mat &X);
    };
};
qda qda::fit(const np::mat &X, const np::ivec &y) {
    // some calculation
    m->internal_data = // something
    return this;
}
np::ivec qda::predict(const np::mat &X) const {
    auto a = m(this)->decision_function(X);
    // manupulate a
}
```

```
auto clf = load_qda(); // copy returned qda to clf
auto X = load_data(); // (rvalue) copy returned data
auto y = clf.predict(X);
```

用户不必关心使用哪种智能指针来保存 qda, 同时 QDA 的设计者可以决定类型的共享语义, 避免误用.

用户只需要知道类型的 (值/共享) 语义, 可以用同一套语法来操作其对象 (使用点号调用成员, 而不是指针的箭头符号).

对于值语义的对象, 比如X, 可以通过`std::move`避免数据拷贝[4].

[4]但仍需要指针拷贝

# Outline

# 一切皆为抽象

- 编程语言和数学系统的核心皆为抽象.

# 一切皆为抽象

- 编程语言和数学系统的核心皆为抽象.
- 变量, 函数, OOP 都是针对各自领域问题的抽象手段.

# 一切皆为抽象

- 编程语言和数学系统的核心皆为抽象.
- 变量, 函数, OOP 都是针对各自领域问题的抽象手段.
- OOP 经常在不合适的领域被极度滥用.

# 函数

- 在科学计算领域, 函数是最好的抽象手段.[5]

[5]个人认为

[6]pure function

# 函数

- 在科学计算领域, 函数是最好的抽象手段.[5]
- (纯) 函数[6]易于描述 (输入, 输出, 前条件, 后条件).

[5]个人认为

[6]pure function

# 函数

- 在科学计算领域, 函数是最好的抽象手段.[5]
- (纯) 函数[6]易于描述 (输入, 输出, 前条件, 后条件).
- (纯) 函数易于测试.

[5]个人认为

[6]pure function

# 编码过程

1. 首先明确问题 (函数) 的输入输出.

[7]大于 7, 参见代码大全.

# 编码过程

1. 首先明确问题 (函数) 的输入输出.
2. 构造数据, 写针对函数的单元测试.

[7]大于 7, 参见代码大全.

# 编码过程

1. 首先明确问题 (函数) 的输入输出.
2. 构造数据, 写针对函数的单元测试.
3. 分解主函数为多个子函数, 写子函数的单元测试.

[7]大于 7, 参见代码大全.

# 编码过程

1. 首先明确问题 (函数) 的输入输出.
2. 构造数据, 写针对函数的单元测试.
3. 分解主函数为多个子函数, 写子函数的单元测试.
4. 子函数之间通过函数参数传递状态.

[7] 大于 7, 参见代码大全.

# 编码过程

1. 首先明确问题 (函数) 的输入输出.
2. 构造数据, 写针对函数的单元测试.
3. 分解主函数为多个子函数, 写子函数的单元测试.
4. 子函数之间通过函数参数传递状态.
5. 若函数参数列表太长[7], 则将共享参数的函数打包成类.

[7]大于 7, 参见代码大全.

# template

- duck typing: If it looks like a duck and quacks like a duck, it's a duck.

# template

- duck typing: If it looks like a duck and quacks like a duck, it's a duck.
- C++ 模板提供了编译期的 duck typing 手段.

# duck typing in Python

```
def mean(a):
    # a must be acceptable by len
    assert len(a) > 0
    # a must be acceptable by sum
    return sum(a) / len(a)
```

# duck typing in C++

```
template<class Range>
typename boost::range_value<Range>::type
    mean(const Range &a)
{
    BOOST_ASSERT(std::size(a) > 0);
    // a must be acceptable by free function sum
    return sum(a) / std::size(a);
}
```

# duck typing in C++

```
np::mat a = ...;
np::vec mean_of_each_row = np::map(np::mean, np::rows(a));
np::vec mean_of_each_col = np::map(np::mean, np::cols(a));
```

# 类型擦除 (type erase)

- 有时候模板带来的编译负担太重[8], 需要借助虚函数减少代码生成.

[8]还有源码保密等因素

# 类型擦除 (type erase)

- 有时候模板带来的编译负担太重[8], 需要借助虚函数减少代码生成.
- 但是我们通常需要组合式的接口 (例如 sum, size), 使用 OOP 会造成组合爆炸以及侵入式继承体系.

[8]还有源码保密等因素

# C style optimization

```
void optimize(double(*f)(double*), double *x);
```

# C style optimization

```
void optimize(double(*f)(double*), double *x);
```

- 只能传递函数指针

# C style optimization

```
void optimize(double(*f)(double*), double *x);
```

- 只能传递函数指针
- 容易诱使全局变量产生

# C style optimization

```
void optimize(double(*f)(double*), double *x);
```

- 只能传递函数指针
- 容易诱使全局变量产生
- 难以并行化

# C style optimization

```
void optimize(double(*f)(double*), double *x);
```

- 只能传递函数指针
- 容易诱使全局变量产生
- 难以并行化
- 解决办法一般是在 f 签名中增加表示数据的 void 指针

# old and bad C++ style optimization

```
struct fn {
    virtual double call(const np::vec &x) const = 0;
};
np::vec optimize(const fn &f, const np::vec &initx);
```

# old and bad C++ style optimization

```
struct fn {
    virtual double call(const np::vec &x) const = 0;
};
np::vec optimize(const fn &f, const np::vec &initx);
```

- 侵入式, 要求继承fn.

# old and bad C++ style optimization

```
struct fn {
    virtual double call(const np::vec &x) const = 0;
};
np::vec optimize(const fn &f, const np::vec &initx);
```

- 侵入式, 要求继承`fn`.
- 代码长 (Keep it simple, stupid!).

# template C++ style optimization

```
template<class F>
np::vec optimize(const F &f, const np::vec &initx);
```

# template C++ style optimization

```
template<class F>
np::vec optimize(const F &f, const np::vec &initx);
```

- 所有实现必须写在头文件里, 编译慢.

# template C++ style optimization

```
template<class F>
np::vec optimize(const F &f, const np::vec &initx);
```

- 所有实现必须写在头文件里, 编译慢.
- 源码泄漏.

# modern C++ style optimization

```
np::vec optimize(
    std::function<double(const np::vec &)> f,
    const np::vec &initx
);
```

# the core of type erase

```
struct classifier {
    virtual np::ivec predict(const np::mat &X) const = 0;
};

template<class T>
struct classifier_impl : classifier {
    T impl;
    np::ivec predict(const np::mat &X) const {
        return impl.predict(X);
    }
};
```

# the core of type erase

```
np::real score(
    const classifier &clf,
    const np::mat &X,
    const np::ivec &y_true
) {
    auto y_pred = clf.predict(X);
    // compare y_pred and y_true
}
...
score(QDA(), X, y_true);
```

# Boost.TypeErasure

```
BOOST_TYPE_ERASURE_MEMBER((has_predict), predict, 1)
namespace concept {
    typedef mpl::vector<
        copy_constructible<>,
        relaxed,
        has_predict<np::ivec(const np::mat&), const _self>
    > classifier;
}
typedef any<concept::classifier> classifier;
```

Boost.TypeErasure 提供了组合接口的各种组件. 上面的 classifier 与之前手工定义的 classifier 效果完全一样.

# Thanks