Modern C++ Coding

abstraction

answeror@gmail.com

2014-02-21

Slides

https://github.com/Answeror/moderncppcoding

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.

Outline

函数

- 在科学计算领域, 函数是最好的抽象手段.[1]

[1]个人认为

[2]pure function

函数

- 在科学计算领域, 函数是最好的抽象手段.[1]
- (纯) 函数[2]易于描述 (输入, 输出, 前条件, 后条件).

[1]个人认为

[2]pure function

函数

- 在科学计算领域, 函数是最好的抽象手段.[1]
- (纯) 函数[2]易于描述 (输入, 输出, 前条件, 后条件).
- (纯) 函数易于测试.

[1]个人认为

[2]pure function

编码过程

1. 首先明确问题 (函数) 的输入输出.

[3]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]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.

[3]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.

[3]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[3], 则将共享参数的函数打包成类.

[3]大于 7, 参见代码大全.

Outline

模板 (template)

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

模板 (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));
```

Outline

类型擦除 (type erasure)

- 有时候模板带来的编译负担太重[4], 需要借助虚函数减少代码生成.

[4]还有源码保密等因素

类型擦除 (type erasure)

- 有时候模板带来的编译负担太重[4], 需要借助虚函数减少代码生成.
- 但是很多时候用户无法控制库的类层次结构.

[4]还有源码保密等因素

类型擦除 (type erasure)

将具有一个共通接口的形形色色的类型变成具有相同接口的“一个”类型[5].

[5]“C++ 模板元编程”9.7.5 节.

假设机器学习库 ml 提供了 QDA, SVM 等具有相同接口的分类器 (fit, predict):

```
// 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;
};
```

另一个机器学习库 yaml 提供了 AdaBoost, CART 等分类器, 但是接口与 ml 不同:

- `fit` $\rightarrow$ `train`
- `predict` $\rightarrow$ `transform`

现在我们需要写一个函数对分类器的性能做评估:

```
np::real score(
    const classifier &clf,
    const np::mat &X,
    const np::ivec &y_true
);
```

How to write classifier?

the core of type erasure

ml 和 yaml 库内部的类之间可能并不存在继承体系. 我们需要一个额外的基类.

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

the core of type erasure

利用模板和虚函数包装具有相同接口的类, 即 “擦除” 了真实类型, 对外仅公布classifier的接口.

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

the core of type erasure

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

the core of type erasure

```
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(classifier_ml<QDA>(qda), X, y_true);
score(classifier_yaml<AdaBoost>(adaboost), 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_ml`具有相似的功能.

组合爆炸

- 不同的模块可能需要功能上不完全相同, 却又包含一定重叠的接口. 功能重叠的 “类型擦除器” 会导致 “组合爆炸”.

组合爆炸

- 不同的模块可能需要功能上不完全相同, 却又包含一定重叠的接口. 功能重叠的 “类型擦除器” 会导致 “组合爆炸”.
- Boost.TypeErasure 提供了一种通过模板定制 “类型擦除器” 的接口的手段. 类似的还有 Boost.Function(已纳入 C++11 标准), Boost.Range 和 Boost.Any.

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

OOP style optimization

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

OOP 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.

OOP 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 style optimization

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

template 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 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
);
```

Thanks