Modern C++ Coding

memory management

answeror@gmail.com

2013-11-30

Slides

https://github.com/Answeror/moderncppcoding

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`

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

— Bjarne Stroustrup

Outline

智能指针 (smart pointer)

句柄 (handle)

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.

Outline

智能指针 (smart pointer)

句柄 (handle)

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

Thanks