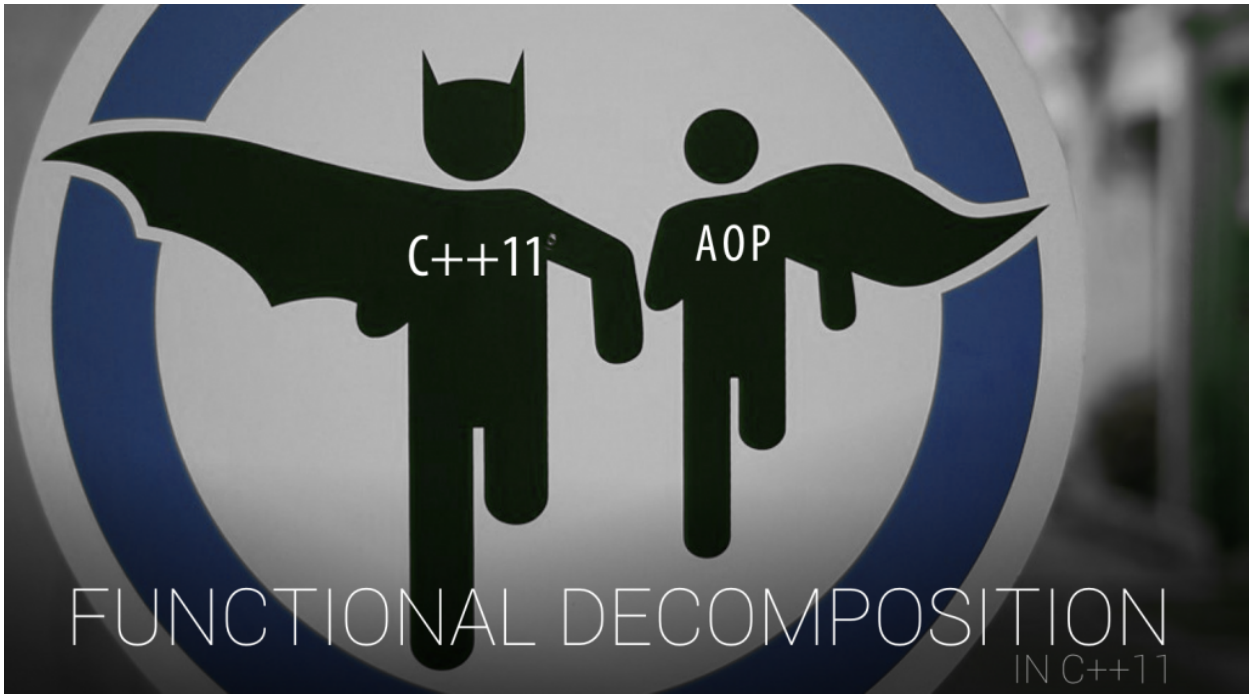
Victor Laskin's Blog

Programming, architecture and design (C++, QT, .Net/WPF, Android, iOS, NoSQL, distributed systems, mobile development, image processing, etc...)

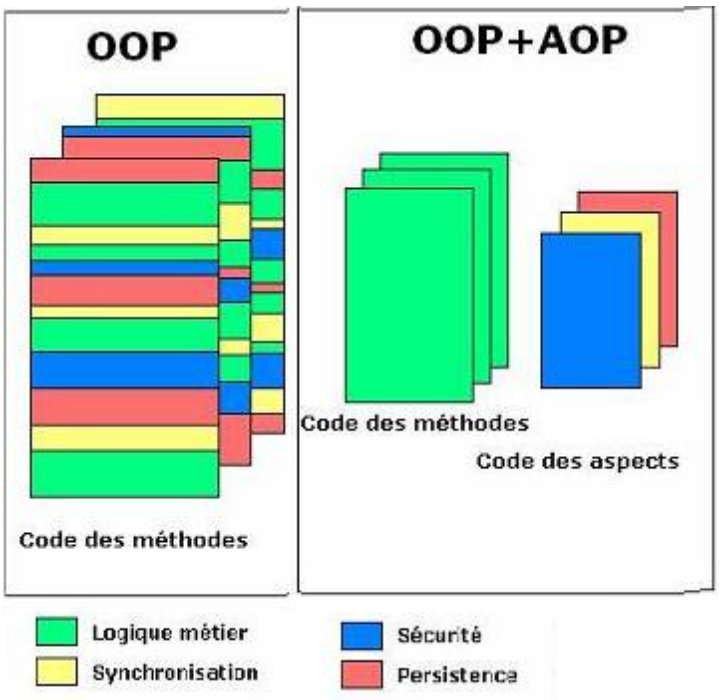
C++11 functional decomposition – easy way to do AOP



This post is about making functional decomposition from perspective of Aspect Oriented Programming using C++11. If you are not familiar with ideas of AOP don't be afraid – it's rather simple concept, and by the end of this post you will understand the benefits of it.

You also can treat this post just as example how to use high-order functions in C++11.

In short – AOP tries to perform decomposition of every business function into orthogonal parts called aspects such as security, logging, error handling, etc. The separation of crosscutting concerns. It looks like:



Old picture about AOP

Since C++11 supports high-order functions now we can implement factorization without any additional tools and frameworks (like PostSharp for C#).

You can scroll down to 'what for' chapter to check out the result to get more motivated.

PART 1 – TRIVIAL SAMPLE

Let's start from something simple – one aspect and one function.

Here is simple lambda with trivial computation inside:

```
1 auto plus = [](int a, int b) { LOG << a + b << NL; };
```

I want to add some logging before and after computation. Instead of just adding this boilerplate code into function body let's go other way. In C++11 we just can write high-order function which will take function as argument and return new function as result:

```
1 template <typename ...Args>
2 std::function<void(Args...)> wrapLog(std::function<void(Args...)> f)
3 {
4     return [f](Args... args){
5         LOG << "start" << NL;
6         f(args...);
```

```
7         LOG << "finish" << NL;
8     };
9 }
```

Here we used *std::function,* variadic templates and lambda as result. (*LOG, NL* – my own logging stream and you can just change it with *std::cout* , *std::endl* or your another logging lib).

As i hoped to achieve the most simple and compact solution, i expected to use it like this:

```
1 auto loggedPlus = wrapLog(plus);
```

Unfortunately this will not compile. 'no matching function to call ....' The reason is that lambda is not *std::function* and automatic type conversion can't be done. Of cause we can write something like this:

```
1 auto loggedPlus = wrapLog(static_cast<std::function<void(int,int)>>(plus));
```

This line will compile, but this is ugly... I hope cpp committee will fix this casting issue. Meanwhile, the best solution i found so far is the following:

```
1  template <typename Function>
2  struct function_traits
3  : public function_traits<decltype(&Function::operator())>
4  {};
5
6  template <typename ClassType, typename ReturnType, typename... Args>
7  struct function_traits<ReturnType(ClassType::*)(Args...) const>
8  {
9      typedef ReturnType (*pointer)(Args...);
10     typedef std::function<ReturnType(Args...)> function;
11 };
12
13 template <typename Function>
14 typename function_traits<Function>::function
15 to_function (Function& lambda)
16 {
17     return typename function_traits<Function>::function(lambda);
18 }
```

This code is using type traits to **convert anonymous lambda into *std::function*** of same type. We can use it like this:

```
1 auto loggedPlus = wrapLog(to_function(plus));
```

Not perfect but much better. Finally we can call functional composition and get the result.

```
1 loggedPlus(2,3);
2
3 // Result:
4 // start
5 // 5
```

```
6 // finish
```

Note: if we had declared aspect function without variadic template we could compose functions without to_function() conversion, but this would kill the benefit from writing universal aspects discussed further.

PART 2 – REALISTIC EXAMPLE

Introduction is over, let's start some more real-life coding here. Let's assume we want to find some user inside database by id. And while doing that we also want log the process duration, check that requesting party is authorised to perform such request (security check), check for database request fail, and, finally, check in local cache for instant results.

And one more thing – i don't want to rewrite such additional aspects for every function type. So let's write them using variadic templates and get as universal methods as possible.

Ok, let's start. I will create some dummy implementation for additional classes like *User, etc. Such classes are only for example and actual production classes might be completely different, like user id should not be int, etc.*

Sample *User* class as immutable data:

```
1  // Simple immutable data
2  class UserData {
3  public:
4      const int id;
5      const string name;
6      UserData(int id, string name) : id(id), name(name) {}
7  };
8
9  // Shared pointer to immutable data
10 using User = std::shared_ptr<UserData>;
```

Let's emulate database as simple vector of users and create one method to work with it (find user by id):

```
1 vector<User> users {make<User>(1, "John"), make<User>(2, "Bob"), make<User>(3, "Ma
  x")};
2
3 auto findUser = [&users](int id) -> Maybe<User> {
4     for (User user : users) {
5         if (user->id == id)
6             return user;
7     }
8     return nullptr;
9 };
```

make<> here is just shortcut for *make_shared<>*, nothing special.

Maybe<> monad

You, probably, noticed that return type of request function contains something called *Maybe<T>*. This class is inspired by Haskell maybe monad, with one major addition. Instead of just saving *Nothing* state and *Content* state, it also might contain *Error* state.

At first, here is sample type for error description:

```
/// Error type - int code + description
class Error {
public:
    Error(int code, string message) : code(code), message(message) {}
    Error(const Error& e) : code(e.code), message(e.message) {}

    const int code;
    const string message;
};
```

Here is minimalistic implementation of *Maybe*:

```
template < typename T >
class Maybe {
private:
    const T data;
    const shared_ptr<Error> error;
public:
    Maybe(T data) : data(std::forward<T>(data)), error(nullptr) {}
    Maybe() : data(nullptr), error(nullptr) {}
    Maybe(decltype(nullptr) nothing) : data(nullptr), error(nullptr) {}
    Maybe(Error&& error) : data(nullptr), error(make_shared<Error>(error)) {}

    bool isEmpty() { return (data == nullptr); };
    bool hasError() { return (error != nullptr); };
    T operator()(){ return data; };
    shared_ptr<Error> getError(){ return error; };
};

template <class T>
Maybe<T> just(T t)
{
    return Maybe<T>(t);
}
```

Note, that you don't have to use Maybe<> and here it's used only for example.

Here we also use the fact that *nullptr* in C++11 has it's own type. Maybe has defined constructor from that type producing *nothing* state. So when you return result from *findUser* function, there is no need

for explicit conversion into *Maybe<>* – you can just return *User* or *nullptr*, and proper constructor will be called.

Operator *()* returns possible value without any checks, and *getError()* returns possible error.

Function *just()* is used for explicit *Maybe<T>* construction (this is standard name).

Logging aspect

First, let's rewrite log aspect so it will calculate execution time using *std::chrono*. Also let's add new string parameter as name for called function which will be printed to log.

```
template <typename R, typename ...Args>
std::function<R(Args...)> logged(string name, std::function<R(Args...)> f)
{
        return [f,name](Args... args){

            LOG << name << " start" << NL;
            auto start = std::chrono::high_resolution_clock::now();

            R result = f(std::forward<Args>(args)...);

            auto end = std::chrono::high_resolution_clock::now();
            auto total = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();
            LOG << "Elapsed: " << total << "us" << NL;

            return result;
        };
}
```

Note *std::forward* here for passing arguments more clean way. We don't need to specify return type as *Maybe<R>* because we don't need to perform any specific action like error checking here.

'Try again' aspect

What if we have failed to get data (for example, in case of disconnect). Let's create aspect which will in case of error perform same query one more time to be sure.

```
// If there was error - try again
template <typename R, typename ...Args>
std::function<Maybe<R>(Args...)> triesTwice(std::function<Maybe<R>(Args...)> f)
{
        return [f](Args... args){
            Maybe<R> result = f(std::forward<Args>(args)...);
            if (result.hasError())
                return f(std::forward<Args>(args)...);
```

```
 9              return result;
10          };
11  }
```

Maybe<> is used here to identify error state. This method can be extended – we could check error code and decide is there any sense to perform second request (was it network problem or database reported some format error).

Cache aspect

Next thing – let's add client side cache and check inside it before performing actual server-side request (in functional world this is called memoization). To emulate cache here we can just use std::map:

```
 1  map<int,User> userCache;
 2
 3  // Use local cache (memoize)
 4  template <typename R, typename C, typename K, typename ...Args>
 5  std::function<Maybe<R>(K,Args...)> cached(C & cache, std::function<Maybe<R>(K,Arg
    s...)> f)
 6  {
 7          return [f,&cache](K key, Args... args){
 8              // get key as first argument
 9
10              if (cache.count(key) > 0)
11                  return just(cache[key]);
12              else
13              {
14                  Maybe<R> result = f(std::forward<K>(key), std::forward<Args>(args
    )...);
15                  if (!result.hasError())
16                      cache.insert(std::pair<int, R>(key, result())); //add to cach
17  e
18                  return result;
19              }
20          };
    }
```

This function will insert element into cache if it was not there. Here we used that knowledge that cache is std::map, but it can be changed to any key-value container hidden behind some interface.

Second important part, we used only first function argument here as key. If you have complex request where all parameters should act as composite key – what to do? It's still possible and there are a lot of ways to make it. First way is just to use *std::tuple* as key (see below). Second way is to create cache class which will allow several key parameters. Third way is to combine arguments into single string cache using variadic templates. Using *tuple* approach we can rewrite it like this:

```
 1  map<tuple<int>,User> userCache;
```

```
2
3  // Use local cache (memoize)
4  template <typename R, typename C, typename ...Args>
5  std::function<Maybe<R>(Args...)> cached(C & cache, std::function<Maybe<R>(Args...
   )> f)
6  {
7          return [f,&cache](Args... args){
8
9              // get key as tuple of arguments
10             auto key = make_tuple(args...);
11
12             if (cache.count(key) > 0)
13                 return just(cache[key]);
14             else
15             {
16                 Maybe<R> result = f(std::forward<Args>(args)...);
17                 if (!result.hasError())
18                     cache.insert(std::pair<decltype(key), R>(key, result())); //a
   dd to cache
19                 return result;
20             }
21         };
22 }
```

Now it's much more universal.

Security aspect

Never forget about security. Let's emulate user session with some dummy class –

```
1 class Session {
2 public:
3     bool isValid() { return true; }
4 } session;
```

Security checking high-order function will have additional parameter – session. Checking will only verify that *isValid()* field is true:

```
1  // Security checking
2  template <typename R, typename ...Args, typename S>
3  std::function<Maybe<R>(Args...)> secured(S session, std::function<Maybe<R>(Args..
   .)> f)
4  {
5          // if user is not valid - return nothing
6          return [f, &session](Args... args) -> Maybe<R> {
7              if (session.isValid())
8                  return f(std::forward<Args>(args)...);
9              else
10                 return Error(403, "Forbidden");
11         };
12 }
```

'Not empty' aspect

Last thing in this example – let's treat not found user as error.

```
// Treat empty state as error
template <typename R, typename ...Args>
std::function<Maybe<R>(Args...)> notEmpty(std::function<Maybe<R>(Args...)> f)
{
        return [f](Args... args) -> Maybe<R> {
            Maybe<R> result = f(std::forward<Args>(args)...);
            if ((!result.hasError()) && (result.isEmpty()))
                return Error(404, "Not Found");
            return result;
        };
}
```

Im not writing here about error handling aspect, but it's also can be implemented via same approach. Note that using error propagation inside *Maybe*<> monad you can avoid using exceptions and define your error processing logic different way.

Multithread lock aspect

```
template <typename R, typename ...Args>
std::function<R(Args...)> locked(std::mutex& m, std::function<R(Args...)> f)
{
    return [f,&m](Args... args){
        std::unique_lock<std::mutex> lock(m);
        return f(std::forward<Args>(args)...);
    };
}
```

No comments.

FINALLY

Finally, what for was all this madness? **FOR THIS LINE:**

```
// Aspect factorization

auto findUserFinal = secured(session, notEmpty( cached(userCache, triesTwice( logg
ed("findUser", to_function(findUser))))));
```

Checking (let's find user with id 2):

```
auto user = findUserFinal(2);
LOG << (user.hasError() ? user.getError()->message : user()->name) << NL;

// output:
// 2015-02-02 18:11:52.025 [83151:10571630] findUser start
// 2015-02-02 18:11:52.025 [83151:10571630] Elapsed: 0us
// 2015-02-02 18:11:52.025 [83151:10571630] Bob
```

Ok, let's perform tests for several users (here we will request same user twice and one non-existing user):

```
auto testUser = [&](int id) {
    auto user = findUserFinal(id);
    LOG << (user.hasError() ? "ERROR: " + user.getError()->message : "NAME:" + user()->name) << NL;
};

for_each_argument(testUser, 2, 30, 2, 1);

//2015-02-02 18:32:41.283 [83858:10583917] findUser start
//2015-02-02 18:32:41.284 [83858:10583917] Elapsed: 0us
//2015-02-02 18:32:41.284 [83858:10583917] NAME:Bob
//2015-02-02 18:32:41.284 [83858:10583917] findUser start
//2015-02-02 18:32:41.284 [83858:10583917] Elapsed: 0us
// error:
//2015-02-02 18:32:41.284 [83858:10583917] ERROR: Not Found
// from cache:
//2015-02-02 18:32:41.284 [83858:10583917] NAME:Bob
//2015-02-02 18:32:41.284 [83858:10583917] findUser start
//2015-02-02 18:32:41.284 [83858:10583917] Elapsed: 0us
//2015-02-02 18:32:41.284 [83858:10583917] NAME:John
```

As you can see it's working as intended. It's obvious that we got a lot of benefits from such decomposition. Factorisation leads to **decoupling of functionality**, more modular structure and so on. You gain more focus on actual business logic as result.

We can change order of aspects as we like. And as we made aspect functions rather universal we can reuse them **avoiding a lot of code duplication**.

Instead of functions we can use more sophisticated functors (with inheritance), and instead of *Maybe*<> also could be more complex structure to hold some additional info. So whole scheme is extendable.

Note also, that you can pass lambdas as additional aspect parameters.

Working sample to play with: github gist or ideone

Ps. BONUS:

```
1 template <class F, class... Args>
2 void for_each_argument(F f, Args&&... args) {
3     (void)(int[]){(f(forward<Args>(args)), 0)...};
4 }
```

This entry was posted in AOP, Architecture, C++, C++11, Coding, Functional programming on 3 February, 2015 [http://vitiy.info/c11-functional-decomposition-easy-way-to-do-aop/] .

Scott Prager • 3 years ago

Hi, in to_function, "static_cast<typename function_traits<function="">::function> (lambda);" should really just be "typename function_traits<function>::function(lambda);". A static cast adds no real safety here.

1 • Reply • Share ›

Victor Laskin Mod ➚ Scott Prager • 3 years ago

Thanks, you are right, its working without cast. I will change in text accordingly.

• Reply • Share ›

Kapil Dhaimade • 17 days ago

Hi. It's a great article! I'm interested in applying this to some of our code.

How should we factorize non-static methods of a class? If it were usersManager.findUser(), would the final call always be a global function like findUserFinal() which wraps instance of UserManager?

Also, wrapping all these functions should be done during app initialization in wiring up stage and there should be some service/factory to provide all factorized functions? Otherwise, the code would become polluted with all the repeated wrapping calls, right?

• Reply • Share ›

Joe Cool • 7 months ago

Hi, great article!

I am new to C++ and have a problem compiling your example with Visual Studio 2017. The for_each_argument function (see below) creates an compiler error: C4576 'a parenthesized type followed by an initializer list is a non-standard explicit type conversion syntax'.