The Impossibly Fast C++ Delegates, Fixed

Sergey Alexandrovich Kryukov, 11 Mar 2017

4.73 (40 votes) Rate this:

Derived work based on the article by Sergey Ryazanov "The Impossibly Fast C++ Delegates": this good solution is fixed and further developed using C++11.

Download source code — 12.6 KB

This is a derived work based on the ideas published by Sergey Ryazanov in his article "The Impossibly Fast C++ Delegates", 18 Jul 2005.

I found the idea interesting, especially after reading the criticism in the discussion section. Even though the code and some approached do have certain problems, if not defeat the purpose of the delegates, at least partially, the idea of making the delegate very fast seems very fruitful, as well as the main technique. I hope I re-worked it into something really usable.

What is Added?

The code is written from scratch, based exclusively on the text of the original article where the idea is clearly expressed: the use of function stubs and filling the stub pointer from the template parameters of the factory methods. It is obvious that the types of the call parameters and the return type should be further abstracted out as template parameters. Naturally, it requires arbitrary number of parameters.

So, as a first step, I added the generalization for the class templates based on C++/11 variadic templates, combined with partial template specialization, to create what Sergey called "preferred syntax", `<RET(PARAMS...)>`, completely eliminating all the preprocessor code.

Notably, in general case, there are two levels of template and template instantiation: first, the delegate profile is instantiated by specifying the return and parameter types. On top of that, a desired *factory function* is instantiated by specifying the class type and its static or instance function. This way, the instantiation of a delegate profile and the target of invocation are carried apart in a near-optimal way.

As to the *factory functions*, first thing to do was to note that the can be given the same name (so called "overloading"), in my choice, "`create`".

I also added the practically important support of lambda expressions in the style similar to `std::function`. The delegate of the same time could be assigned to an instance of a lambda expression and called later. Importantly, the closure *capture* performed by a lambda expression is preserved in the delegate instance.

All of the above is covered by a single `delegate` template class. I also added one more template, `multicast delegate`, in .NET style.

I also added *semantic* comparison between delegate instances of both types and comparison with null pointer, assignment operators giving the compiler the possibility to implicitly instantiate correspondent method templates through type *inference*, and similar small features.

Notably, the instances of both delegate types can be created as "empty". It reflects the main paradigm of the delegate usage, when the delegate instance is hosted by some class; and the code using the class sets or adds its own handlers in the course of this usage.

Performance comparison was done with the use of `std::function`. In all cases where the time measurements might be considered as relatively valid (very roughly, starting from a hundred of delegate instances each called a hundred of times), the `delegate` type has shown superior performance compared with `std::function`. Roughly, depending on many factors, the gain in `delegate/std::function` creation time was from 10 to 60 times, and performance gain in the call operations was from 1.1 to 3 times. I performed the measurements on Windows in two platforms, x86 (IA-32) and x86-64, with Clang, GCC and Microsoft compilers — see also Compatibility and Build.

Usage By Examples

Hide Shrink Copy Code

```
class Sample {
public:
    double InstanceFunction(int, char, const char*) { return 0.1; }
    double ConstInstanceFunction(int, char, const char*) const { return 0.2; }
    static double StaticFunction(int, char, const char*) { return 0.3; }
}; //class Sample

//...

Sample sample;
```

```
delegate<double(int, char, const char*)> d;

auto dInstance = decltype(d)::create<Sample, &Sample::InstanceFunction>(&sample);
auto dConst = decltype(d)::create<Sample, &Sample::ConstInstanceFunction>(&sample);
auto dFunc = decltype(d)::create<&Sample::StaticFunction>();
// same thing with non-class functions

dInstance(0, 'A', "Instance method call");
dConst(1, 'B', "Constant instance method call");
dFunc(2, 'C', "Static function call");

int touchPoint = 1;
auto lambda = [&touchPoint](int i, char c, const char* msg) -> double {
    std::cout << msg << std::endl;
    // touch point is captured by ref, can change:
    return (++touchPoint + (int)c) * 0.1 - i;
}; //lambda

decltype(d) dLambda = lambda; // lambda to delegate
// or:
//decltype(d) dLambda(lambda);

if (d == nullptr) // true
    d(1, '1', "lambda call"); //won't

d = dLambda; // delegate to delegate

if (d == dLambda) // true, and also d != nullptr
    d(1, '1', "lambda call"); //will be called
```

By the way, compare this usage in the cases of class/struct instance function with the same thing using `std::function`:

Hide Copy Code

```
//...

Sample sample;
using namespace std::placeholders;
std::function<double(double(int, char, const char*))> f
    = std::bind(&Sample::InstanceFunction, &sample, _1);
```

The confusing part is the use of `std::placeholders` and `_1` (which is used to express the notion of the instance pointer passed as the first implicit parameter to the instance function call), which hardly looks obvious.

Multicast Delegate Usage

Hide Copy Code

```
multicast_delegate<double(int, char, const char*)> md;
multicast_delegate<double(int, char, const char*)> mdSecond;
if (md == nullptr) // true
    md(5, '6', "zero calls"); //won't
// add some of the delegate instances:
md += mdSecond; // nothing happens to md
md += d; // invocation list: size=1
md += dLambda; // invocation list: size=2
if (md == dLambda) //false
    std::cout << "md == dLambda" << std::endl;
if (dLambda == md) //false
    std::cout << "dLambda == md" << std::endl;
if (md == mdSecond) //false
    std::cout << "md == mdSecond" << std::endl;
//adding lambda directly:
md += lambda; // invocation list: size=3
    md(7, '8', "call them all");
```

The above examples of multicast delegate usage discard the objects returned from each operation. All the code samples shown above can work with `void` return type. What to do with the return objects? They can be somehow accumulated; some of the values can be discarded, and so on. So, what solution would be the universal? It can be done by supplying a handler called on each return.

Multicast Invocation with Return Object Handlers

In addition to the `operator()`, the invocation with handling can be performed with a separate invocation template `operator()` with additional parameter — a handler for the returned objects:

Hide Copy Code

```
double total = 0;
md(9, 'a', "handling the return values:",
    [&total](size_t index, double* ret) -> void {
        std::cout << "\t"
            << "index: "
            << index
            << "; returned: " << *ret
            << std::endl;
            total += *ret;
});
```

Note that the handler shown here is the instance of the `delegate` template class created from a lambda expression. There is another form of the handler based on `std::function`. Interestingly, no function template instantiation is required, as the type of the handler can be *deduced* (*inferred*) by a compiler from the context.

This code sample demonstrates the effect of enclosure capture used to accumulate the returned values in `total`. Of course, any other logic can be used; see also "DelegateDemo.h".

Why it's Fast?

The basic idea is explained in the article by S. Ryazanov.

The mechanism is fast because good part of work is delegated to the compile-time phase of the product life time.

The delegate invocation call, `operator ()`, has only one level of indirection, a call to one of three *stub* functions (I added one more to support *lambda*). The address of an original method is not stored in the delegate class instance; instead, it is created during compile time, template instantiation. This way, each fragment of code creating an instance of a delegate based on a distinct function, instantiates a separate version of the stub. In each stub, the address of the function to be called is presented to the compiler as an *immediate constant*. Only one pointer is passed during run time: the pointer to the object used as "`this`" call argument (with the overhead of passing `nullptr` for static functions), which I later started to reuse as a pointer to a *lambda expression* instance, to support lambda.

Important performance factor here is the absence of any runtime check which could be used to distinguish all four cases: instance function of a class, constant instance function of a class, static function and lambda expression. According to my brief and not very accurate research, such check would almost double the execution time related to the invocation mechanism overhead.

Variadic Templates and Parsing of Template Parameters

This is the technique used to provide the structured function-like syntax for template parameters:

Hide Copy Code

```
template <typename T> class delegate;

template<typename RET, typename ...PARAMS>
class delegate<RET(PARAMS...)> final : private delegate_base<RET(PARAMS...)> {
    //...
};
```

The specialization `delegate<RET(PARAMS…)>` creates the convenient profile for template instantiation, similar to the template `std::function`:

Hide Copy Code

```
delegate<double(int, const string*)> del;
delegate<void(double&)> byRefVoidDel;
// ...and the like
```

Lambda Expressions

The idea behind the lambda is to reuse the "`this`" pointer in the delegate instance data. In other cases, this pointer is used to hold the instance pointer of the class which instance method is used in the delegate, to pass it as the first (normally implicit) parameter of the method call.

To perform the lambda expression call, I added one more stub function, `lambda_stub` (see "Delegate.h"):

Hide Copy Code

```
template <typename LAMBDA>
static RET lambda_stub(void* this_ptr, PARAMS... arg) {
    LAMBDA* p = static_cast<LAMBDA*>(this_ptr);
    return (p->operator())(arg...);
}
```

The reason for such solution is this is related to what I already explained above — too expensive overhead of possible check of the different cases.

In passing the lambda instance to the delegate instance, the most important problem is the support of the major lambda, closure capture. Notably, this feature is lost when a lambda instance is copied by value (if the capture set is not empty, the copy throws exception at its call). To me, such lambda design is questionable, but this is a standard behavior. Passing the instance by pointer created by the caller works, but this hardly could be a viable design, due to the need of dealing with null pointers. So, the only reasonable solution is to pass the lambda instance by constant reference and creation of the pointer inside the factory function.

This is how the factory function looks:

Hide Copy Code

```
template <typename LAMBDA>
static delegate create(const LAMBDA & instance) {
    return delegate((void*)(&instance), lambda_stub<LAMBDA>);
}
```

This is the formal way to instantiate the template for this function:

Hide Copy Code

```
auto d = delegate<double(int, char, const char*)>
    ::create<decltype(lambda)>(lambda);
```

(See the declaration of `d` and `lambda` above.)

However, there is no a need to do it, because the lambda type will be deduced (inferred) by a compiler from the delegate type. So, it will work if `<decltype(lambda)>` is omitted from the code fragment shown above. Moreover, it could be done through the assignment operator defined for the `delegate` class:

Hide Copy Code

```
delegate<double(int, char, const char*)> dl = lambda;
```

This is possible because of the template copy constructor and template assignment operator. Again, the template instantiation is not needed (and not possible for a constructor), because the template parameters are deduced (inferred) from delegate and lambda types.

Multicast Delegate

First of all, as multicast delegate represent the set of handlers in the form of *invocation list*, and the heap is used (this is the only piece of code where heap is used), much higher intrinsic performance cost of the list operation support and the list iteration totally absorbs fine performance detail of the list item call. And yet, the invocation list item holds the same pair of `this/stub` pointers as in `delegate`, not the pointer to a `delegate` instance.

A `multicast_delegate` instance is created with an empty invocation list, which can be then populated with items using a set of "`+=`" operators from other instances of `multicast_delegate`, `delegate` and from lambda expressions of matching profiles. When existing list items used, they are cloned.

Are the Delegates Comparable?

Yes, they are, despite the statements made in Sergey's article. "Comparison" is not a fully accurate term in this case; all the talking is actually about the *equality* or *identity* relation. As soon as this is meant by "comparison", delegates are "comparable" — please see the set of "`==`" and "`!=`" operators. Several operators represent all cases of equality checks: each of the classes `delegate` and `multicast_delegate` can be equal or not equal to the instance of its own or another type, additionally, an instance of each type can be equal or not equal to `nullptr`. Taking into account *commutativity*, it gives 12 cases. For all of the purposes of such relation, this is a perfectly valid set of operators.

Sergey argued that "a delegate doesn't contain a pointer to method". But why? It actually *does* contain such pointer, only this pointer is not passed to a delegate instance during run time. Instead, all the template instantiations generate the all the method addresses in the form of *immediate constants*, so the comparison of such pointers is done correctly but indirectly, through the comparison of different stubs. If two stub pointers are different, it always means that underlying function pointers are different, and visa versa.

Two delegate instances created from the same function or the same class and same "`this`" pointer compare as identical. The same goes with the delegate instances created from some lambda expression instance. If, by some reason, someone manages to compile some fragment of source code independently in different compilation units and make those units link together successfully, it yields *different* classes and function pointers, not the same. Likewise, two separate lambda expressions in the same stack frame with identical code still produce two different types, which can be easily checked up; and this is done for a reason. In both cases, the delegate instances instantiated from these *not-really-identical* objects simply *must* compare as *not* identical.

After all, the set of "`==`" and "`!=`" operators defines the relationship on the set of delegate instances which matches the definition of equivalence relation: it is *reflective*, *symmetric* (*commutative*) and *transitive* — all that matters.

Compatibility and Build

All the `delegate/multicast_delegate` solution is contained in just three files:

"DelegateBase.h",
"Delegate.h",
"MultiCastDelegate.h";

they include each other in the given order and can be added to any project.

The compiler should support C++11 or later standard. For GCC, this is an option which should be set to `-std=c++11` or, say, `-std=c++14`.

The demo and benchmark project is provided in two forms: 1) Visual Studio 2015 solution and project using Microsoft C++ compiler and Clang — see "CppDelegates.sln" and 2) Code::Blocks project using GCC — " CPPDelegates.cbp". For all other options, one can assemble a project or a make file by adding all "*.h" and "*.cpp" files in the code directory "CppDelegates".

I tested the code with Visual Studio 2015, Clang 4.0.0, GCC 5.1.0.

The C++ options included "disable language extensions" (`/Za` for Microsoft and Clang), which seems to be essential for Microsoft. However, with this option, one weird Microsoft problem is the failure to compile "`//`" comments at the end of a file; the problem can be solved, for example, by adding an empty line at the end of the file; I set up a "Microsoft guard" in the form of "`/* ... */`" at the end of each file.

License

Share

invocation : 기도, 기원, 간구, 탄원, 청원

About the Author

Sergey Alexandrovich Kryukov
Architect
United States

No Biography provided

You may also be interested in...

The Impossibly Fast C++ Delegates

Modernize Your C# Code - Part III: Values

Fast C++ Delegate

Delegates: C++11 vs. Impossibly Fast - A Quick and Dirty Comparison

O(n)CacheXX_RU Series Caching Algorithms

Fast C++ Delegate: Boost.Function 'drop-in' replacement and multicast

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Compile time instance address
John Wellbelove **12-May-19 21:24**

Re: Compile time instance address
John Wellbelove 13-May-19 0:18

Virtual method problem
Member 13744291 **29-Aug-18 2:03**

/Za
jogomu **26-Sep-17 7:48**

Re: /Za
Sergey Alexandrovich Kryukov 26-Sep-17 10:29

delegate referring to lambda as return value
jogomu **5-Sep-17 12:19**

Re: delegate referring to lambda as return value
Sergey Alexandrovich Kryukov 5-Sep-17 13:11

Re: delegate referring to lambda as return value
jogomu 5-Sep-17 13:29

That's right
Sergey Alexandrovich Kryukov 5-Sep-17 16:16

Re: That's right
jogomu 11-Sep-17 15:31

delegate nullptr testing
Member 13120502 **11-Apr-17 23:28**

Re: delegate nullptr testing
Sergey Alexandrovich Kryukov 12-Apr-17 3:40

Re: delegate nullptr testing
Member 13120502 13-Apr-17 0:30

Re: delegate nullptr testing
Sergey Alexandrovich Kryukov 13-Apr-17 5:26

Re: delegate nullptr testing
earlnsk 28-Jul-18 1:45

Re: delegate nullptr testing
Sergey Alexandrovich Kryukov 29-Jul-18 19:27

Nice.
Pete O'Hanlon **20-Mar-17 22:35**

Re: Nice.
Sergey Alexandrovich Kryukov 21-Mar-17 2:46

Perfect forwarding
Raúl Bocanegra Algarra **20-Mar-17 12:58**

Re: Perfect forwarding
Sergey Alexandrovich Kryukov 20-Mar-17 13:30

Re: Perfect forwarding
Raúl Bocanegra Algarra 24-Mar-17 15:47

Parameter pack
Sergey Alexandrovich Kryukov 24-Mar-17 21:35

Re: Parameter pack
Raúl Bocanegra Algarra 25-Mar-17 2:03

My vote of 5
Maciej Los **20-Mar-17 2:16**

Re: My vote of 5
Sergey Alexandrovich Kryukov 20-Mar-17 6:11

Refresh

1 2 Next »

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink | Advertise | Privacy | Cookies | Terms of Use | Mobile
Web01 | 2.8.190518.1 | Last Updated 11 Mar 2017

언어 선택

Layout: fixed | fluid