Generic, type-safe delegates and events in C++

While other languages such as C# offer type-safe callbacks/delegates out-of-the-box, C++ unfortunately does not offer such features. But delegates and events provide a nice way of “coupling” completely unrelated classes without having to write a lot of boilerplate code.

This blog post describes a generic, type-safe implementation of such delegates and events using advanced C++ features such as non-type template arguments and partial template specialization, supporting both free functions and member functions alike, without any restrictions or dynamic memory allocation.

Again, let’s start with a simple example of how we want to use delegates. Imagine that **MyDelegate** is a delegate accepting an integer, returning nothing (how such a delegate can be declared will be shown later):

```
void FreeFunction(int)
{
  // do something
}

class Class
{
public:
  void MemberFunction(int)
  {
    // do something
  }
};

MyDelegate delegate;
delegate.Bind(&FreeFunction);   // free function
delegate.Invoke(10);            // calls the bound function

Class c;
delegate.Bind(&Class::MemberFunction, &c);
delegate.Invoke(10);            // calls the bound function
```

As can be seen in the example, we want to be able to bind arbitrary free functions as well as member functions to the delegate, as long as they match the delegate’s signature – **void (int)** in this case. The only difference between those two is that member functions can only be called on an instance of the corresponding class, hence we need to pass a **Class** object to the delegate.

But how do we store pointers to either free functions and member functions internally?

Pointers to functions in C++

Storing pointers-to-functions inside the delegate class is easy:

```
class Delegate
{
  typedef void (*FreeFunction)(int);

public:
  void Bind(FreeFunction function)
  {
    m_function = function;
  }

  void Invoke(int ARG0)
  {
    (m_function)(ARG0);
  }

private:
  FreeFunction m_function;
};
```

In case you have never used pointers to functions in C++ before, the typedef defines a type **FreeFunction**, which is a pointer to any free function accepting an integer, returning void. Such a free function can be any global function, a function inside a namespace, or a class-static function – but not a member function.

Pointers to member functions

Pointers to member functions in C++ are an entirely different beast altogether.

First, their type differs from ordinary pointers to functions:

```
void FreeFunction(int);        // type: void (*)(int)

class Class
{
  void MemberFunction(int);    // type: void (Class::*)(int)
};
```

As can be seen, a pointer to a member function also "carries" the type of the class, hence pointers to member functions cannot be interchanged between different classes!

Secondly, while pointers to free functions have the same size as **void***, pointers to member functions do not. That means that pointers to different member functions can have different sizes:

```
struct Base
{
  virtual void Do(int) {}
};

struct Derived1 : public Base
{
  virtual void Do(int);
};

struct Derived2 : public Base
{
  virtual void Do(int);
};

struct MultipleInheritance : public Derived1, public Derived2
{
  virtual void Do(int);
};

struct VirtualMultipleInheritance : virtual public Derived1, virtual publ
{
  virtual void Do(int);
};

ME_LOG0("size: %d", sizeof(&Base::Do));
ME_LOG0("size: %d", sizeof(&Derived1::Do));
ME_LOG0("size: %d", sizeof(&Derived2::Do));
ME_LOG0("size: %d", sizeof(&MultipleInheritance::Do));
ME_LOG0("size: %d", sizeof(&VirtualMultipleInheritance::Do));
```

On my machine using Visual Studio 2010, I get the following output:

```
size: 4

size: 4

size: 4

size: 8

size: 12
```

That may come as a surprise, but the standard does not dictate strict sizes for pointers to member functions. Generally, in Visual Studio pointers to member functions will occupy either 4, 8 or 12 bytes (for single inheritance, multiple inheritance, and virtual inheritance, respectively). A more thorough discussion can be read here.

The above is also a very strong argument against casting pointers to member functions into **void***, ever – they do not fit, and your program is broken (even though it might work on your machine).

Back to our delegates, how can we turn any pointer to a member function into the same type? One possibility is to use type erasure in conjunction with an abstract base class:

```
class AbstractBase
{
public:
  virtual void CallFunction(int ARG0) = 0;
};

```

```
template <class C>
class MemberFunctionWrapper : public AbstractBase
{
  typedef void (C::*MemberFunction)(int);

public:
  MemberFunctionWrapper(MemberFunction memberFunction)
  {
    m_memberFunction = memberFunction;
  }

  virtual void CallFunction(int ARG0)
  {
    (m_instance->*m_memberFunction)(ARG0);
  }

private:
  C* m_instance;
  MemberFunction m_memberFunction;
};
```

Using type erasure, concrete implementations of the class **AbstractBase** still know about the type of class they're dealing with (**template <class C>**), and can call member functions without any nasty casting or other illegal statements. And instances of **AbstractBase** just use C++'s virtual function mechanism for dispatching the function call to the underlying implementation. This means that we could store pointers to member functions by storing instances of **AbstractBase**:

```
AbstractBase* function = new MemberFunctionWrapper<Class>(&Class::Do);
```

This is certainly a valid and legal approach, but it has the drawbacks of using dynamic memory allocations, and adding another level of indirection via the virtual function.

Let us go back to the original delegate, and see whether there's an alternative solution. If we somehow could turn pointers to member functions into ordinary pointers to functions, we only need to store the latter, and not care about member functions at all.

We could try to write such a wrapper function the following way:

```
template <class C>
void WrapMemberFunction(C* instance, void (C::*memberFunction)(int), int A
{
  (instance->*memberFunction)(ARG0);
}
```

The above works, but we cannot store it inside our Delegate class, because **WrapMemberFunction** has a different signature than an ordinary free function:

```
typedef void (*FreeFunction)(int);
typedef void (*WrapperFunction)(C*, void (C::*)(int), int);
```

Still, the type of the **class C** is lurking around in the typedefs, and we need an additional **C*** argument which is the instance the pointer to member function is to be called upon.

Introducing an additional, unused argument to the free function, and using plain, old **void*** pointers halfway gets us there:

```
void WrapFreeFunction(void* instance, void (*freeFunction)(int), int ARG0
{
  // instance is unused
  (freeFunction)(ARG0);
}

template <class C>
void WrapMemberFunction(C* instance, void (C::*memberFunction)(int), int
{
  (static_cast<C*>(instance)->*memberFunction)(ARG0);
}
```

Note that we are **not** sacrificing type-safety by using a **void*** here – the function template always knows the original type, and correctly casts it back again (casting from any pointer to **void*** and back is safe according to the standard).

But the second argument still does not match. We need to get rid of it altogether, and we can do that by using one of C++'s advanced template features – passing pointers to free functions and pointers to member functions as template arguments, using non-type template parameters.

That means we can turn our delegate into the following:

```
Delegate

def void* InstancePtr;
def void (*InternalFunction)(InstancePtr, int)
def std::pair<InstancePtr, InternalFunction> Stub;

urns a free function into our internal function stub
late <void (*Function)(int)>
ic ME_INLINE void FunctionStub(InstancePtr, int ARG0)

 we don't need the instance pointer because we're dealing with free functions
turn (Function)(ARG0);


urns a member function into our internal function stub
late <class C, void (C::*Function)(int)>
ic ME_INLINE void ClassMethodStub(InstancePtr instance, int ARG0)

 cast the instance pointer back into the original class instance
turn (static_cast<C*>(instance)->*Function)(ARG0);


:
gate(void)
n_stub(nullptr, nullptr)



Binds a free function
late <void (*Function)(int)>
 Bind(void)
```

```
stub.first = nullptr;
stub.second = &FunctionStub<Function>;


Binds a class method
late <class C, void (C::*Function)(int)>
 Bind(C* instance)

stub.first = instance;
stub.second = &ClassMethodStub<C, Function>;


Invokes the delegate
 Invoke(int ARG0) const

_ASSERT(m_stub.second != nullptr, "Cannot invoke unbound delegate. Call Bind() f
turn m_stub.second(m_stub.first, ARG0);


e:
 m_stub;
```

Voila!

Both free function and member function "wrappers" now share exactly the same signature, and can thus be stored inside the **Delegate** class. The delegate can now be used for both free functions and member functions:

```
void FreeFunction(int)
{
  // do something
}

class Class
{
public:
  void MemberFunction(int)
  {
    // do something
  }
};

MyDelegate delegate;
delegate.Bind<&FreeFunction>();  // free function
delegate.Invoke(10);             // calls the bound function

Class c;
delegate.Bind<Class, &Class::MemberFunction>(&c);
delegate.Invoke(10);             // calls the bound function
```

All that is left to do is turn the delegate into a class template, so it can be used for arbitrary return types and arguments.

This can easily be done by using partial template specialization:

```
// base template
template <typename T>
class Delegate {};
```

```cpp

template <typename R>
class Delegate<R ()>
{
  // implementation for zero arguments, omitted
};

template <typename R, typename ARG0>
class Delegate<R (ARG0)>
{
  // implementation for one argument, omitted
};

template <typename R, typename ARG0, typename ARG1>
class Delegate<R (ARG0, ARG1)>
{
  // implementation for two arguments, omitted
};

// other partial template specializations
// ...

// defining any delegate in user-code
typedef Delegate<void (int, const char*) MyDelegate;
typedef Delegate<bool (const Vec2&)> IntersectionDelegate;
// etc.
```

Personally, I did not write all the implementations by hand, but rather used a macro-based approach, in conjunction with a pre-build step inside Visual Studio. That way, you only have to write the implementation once, but keep all the benefits (readability, debuggability) as if you had actually written all implementations by hand.

Regarding performance, the delegate implementation offers exactly the same performance as simple free function pointers, because the pointers are passed as template arguments, and the stub functions are marked ME_INLINE (__forceinline), forcing the compiler to inline the call (profiled using Visual Studio 2010) at the calling site.

Events

Using the same approach, we can easily extend the above delegate implementation into a full-blown event system by storing an array of Stubs rather than just a single one.

In Molecule, the event system is split into an **Event** and a corresponding **Event::Sink**. The **Event::Sink** takes care of storing all the listeners to a certain event, and such a sink can be bound to an **Event** using **Event::Bind**(). Using **Event::Signal**(), all listeners inside a bound sink will be notified of the event.

Splitting the event system into those two parts allows me to store an **Event** somewhere as member in a class, allowing different sinks/listeners to be notified while never exposing the **Signal**() functionality to the user of the class.

Use cases

As an example, Molecule already offers a completely generic hot-reload system, which makes heavy use of the event system. Just define a **DirectoryWatcher** somewhere, and hook up some event listeners:

```
// store some event listeners in a sink
DirectoryWatcher::ActionEvent::Sink eventSink;
eventSink.AddListener<&ConfigHotReloader::OnAction>();
eventSink.AddListener<&TextureReloader::OnAction>();
eventSink.AddListener<&ResourceManager::OnAction>();
eventSink.AddListener<&User_OnFileAction>();  // some user-defined function

// hook them to the directory watcher
DirectoryWatcher directoryWatcher(currentDirectory);
directoryWatcher.Bind(&eventSink);

// tick the directory watcher once a frame
directoryWatcher.Tick();
```

Another example of coupling unrelated classes using the delegate and/or event system is binding low-level input from physical devices to high-level logical devices, but that will be discussed in a separate post.

September 19, 2011 **55** Replies

« Previous

Next »

Pingback: Wiring physical devices to abstract inputs | Molecular Musings

Brandon K on October 10, 2011 at 12:37 pm

Just curious, but is there a reason you elected not to use either boost::function or c++11's std::function?

Reply

Stefan Reinalter on October 10, 2011 at 1:49 pm

Good question!

I chose not to use boost because I didn't want my clients to depend on boost headers, and furthermore my implementation has a smaller memory footprint (8 bytes assuming 32-bit) and less overhead (no memory allocations done; member function calls are resolved at compile-time and are as fast as free function calls).

Unfortunately, C++11 is not available on many platforms yet (certainly not consoles), and judging from experience, it will take a long time until those features are supported on their respective compilers.

Reply

Brandon K on October 10, 2011 at 2:04 pm

Valid points indeed 🙂 The only weakness I can see is that this (intentionally) doesn't support currying. I wonder how GCC and MSVC handle lambdas / std::functions without any bound parameters; that is, in situations where you would want to use this technique. Something to investigate!

GCC and MSVC already have decent support for some of c++11's major features, hopefully other important compilers don't lag too far behind — lambdas are just too useful to do without anymore.

Keep up the great articles! I found your blog when I too was investigating the raw input madness for keyboards. Very few good articles on that particular API along with the "Text Services Framework" api for working with IME input (international input).

Stefan Reinalter on October 10, 2011 at 2:14 pm

I haven't looked into lambdas yet, I'll probably still wait a few more months (and see what the console compilers support then) until I start using more C++11 stuff.

And thanks for the positive feedback! The keyboard APIs in Windows certainly triggered some "WTF?"-moments for me, so I can understand where you're coming from :).

Reply

anticop on November 13, 2011 at 3:05 am

Very nice article. Even though I already use a delegate system based on these techniques, it was very educational to see how it's done under the hood.

Reply

Michael Herzog on February 14, 2012 at 8:17 am

Very nice implementation! Are your event methods static? (e.g. ConfigHotReloader::OnAction()).
I Just wonder how your event class and sink is implemented? Does your event sink contain a fixed array of stubs which size you can specifiy with a template parameter or are they dynamically allocated?

Reply

Stefan Reinalter on February 14, 2012 at 10:12 am

Thanks!
No, they are member functions. The above was only a simplified example. The real code actually looks more like this:

// someplace else:
ConfigHotReloader configReloader;

```
// setting up the sink
DirectoryWatcher::ActionEvent::Sink eventSink(applicationAllocator, 16);
eventSink.AddListener(&configReloader);
```

This is written from the top of my head, but is reasonably close to the real code. I don't use singletons/statics anywhere, rather instances (mostly on the stack) instead.

Regarding your second question: As you can see, the event sink takes an allocator as parameter, and a maximum number of listeners to be expected. The sink then just stores an array of listeners as member, with the array being allocated by the given allocator – I want to be able to exactly specify where in memory those listeners end up.

Reply

Michael Herzog on February 14, 2012 at 1:08 pm

Thanks for your fast answer!
Now everything is very clear for me.
Regarding "instances on the stack"… This has smth to do with the "Singleton-Anti-Pattern" I think and this question don't belong to this article but why e.g. a Logger isn't allowed to be a global var? (e.g. g_logManager).
For me this is a situation where a singleton is "allowed", because I want fast access to the log manager and don't want too many different log files. So in my engine I've calls like this: g_logManager.Info(, ,)

How you would solve this without global vars or a singleton and also keep usability in mind?

Stefan Reinalter on February 14, 2012 at 2:50 pm

Good question.

The way I do it with loggers is simple – I create them on the stack, and each logger adds itself to an intrusive list in the constructor, and removes itself in the destructor.

Additionally, a logDispatch namespace takes care of iterating the list, calling e.g. Log(), Warning() or Error() functions on the loggers currently contained in the list. With a simple macro like ME_LOG/ME_WARNING/ME_ERROR, I can easily dispatch whatever I want to whomever is currently registered (the macros do nothing more than calling logDispatch::Log with the correct arguments).

In code, this looks something like this:

```
// somewhere at the beginning of main
Console console(200, 80);
ConsoleLogger consoleLogger(&console);
FileLogger fileLogger("log.txt");
TcpIpLogger networkLogger;
```

And somewhere else, completely unrelated, I can simply create another logger on the stack (for debugging of certain engine parts, gameplay code, etc.) and can be sure that logs will be dispatched to all loggers automatically.

There's no singleton involved, no dynamic memory allocation – just a simple intrusive linked-list. Nobody needs to make sure that singletons get constructed before main because somebody wants to print something from inside the memory system (which is also initialized before main, leading to the "static initialization order fiasco", yada-yada). Living without pre-main stuff and without singletons has been a big win for me.

Reply

Michael Herzog on February 14, 2012 at 3:54 pm

Intrusive lists are a great aproach, because global static vars will be stored in the .BSS or the .SDATA segment and so they get initialized long before any pre-main stuff. It seems that I've to reconsider my logging class and adapt your ideas…
The macros around the logDispatcher allow you to completely ignore all log statements by a given log level and also they are nicer to handle, am I right?
So e.g. a call to ME_WARNING(2, "number %d", 2) will allow you to log a warning statement to the second channel of your LogManager which maybe a tcp logger or smth….
Very great idea indeed! Thanks for explanation!

Reply

bitwize93 on July 7, 2016 at 6:59 pm

If you are containing an intrusively-linked-list of different logger types, are you using virtual pointers and inheritance to keep them all within the same list? Or do you have multiple intrusively-linked-lists that all get iterated from your call to logDispatch::Log?

Reply

Stefan Reinalter on July 10, 2016 at 2:39 pm

In this case I use a base class and one intrusive linked list.

Stefan Reinalter on February 14, 2012 at 5:06 pm

@.BSS/.SDATA: Exactly! One of the main reasons I do it this way :).

@Macros: The macros exist for the sole reason of getting rid of all logging in e.g. retail builds by way of a simple #define. Log levels are handled by the log dispatcher and loggers internally – log levels can be changed at run-time (via config-files, scripts, or the ingame console).

Reply

Thomas Francis on February 14, 2012 at 10:44 pm

Hey, I have a couple questions about a few things…
Could you explain to me this line:
typedef void (*InternalFunction, int)(InstancePtr)
What exactly is it saying?

Also, what is ME_INLINE and R? I'm guess that ME_INLINE is a constant for inlining, but what is R?

-Tom

Reply

Stefan Reinalter on February 14, 2012 at 11:24 pm

In fact, the line
typedef void (*InternalFunction, int)(InstancePtr)
is wrong and should read
typedef void (*InternalFunction)(InstancePtr, int).
Thanks for spotting the mistake!

It defines a function pointer named InternalFunction, taking two arguments (InstancePtr and int).

R is simply one of the template arguments in the template specialization, used to indicate the return value of the delegate:

```
// base template
template <typename T>
class Delegate {};

template <typename R>
class Delegate<R ()>
```

```
{
  // implementation for zero arguments, omitted
};

template <typename R, typename ARG0>
class Delegate<R (ARG0)>
{
  // implementation for one argument, omitted
};
```

ME_INLINE is one of the many internal Molecule Engine macros (hence they all start with ME_). It forces the compiler to inline a function, and expands to e.g. __forceinline on MSVC and __attribute__((always_inline)) on GCC.

Reply

Michael Herzog on February 15, 2012 at 8:01 am

WordPress doesn't like you! :p

It eats up some angle brackets…

billbasher on April 22, 2012 at 6:05 pm

Could you maybe post a working example? I have not used non-type template parameters so far and I am wondering about the R parameter.

In the class definition of delegate what does on the second line mean?

template
class Delegate

Reply

Stefan Reinalter on April 22, 2012 at 6:37 pm

WordPress ate your brackets :).

The R parameter is simply the return value, which itself is a template parameter (see the example in the comments above).

Reply

nate@createdimage.com on May 4, 2012 at 9:14 pm

Thanks for the post. I have implemented similar systems in the past. I like your implementation for its run-time efficiency. In game development, we usually care more about speed than static code size, but that is not always the case, just as inlining is not always a good thing.

If you declare a template function using a function pointer type, then the compiler must generate a unique instantiation for each type. In other words, the generated code can be shared for all functions matching the same signature. If you declare a template function using a function pointer value, on the other hand, there is no sharing. As I said, this cost is probably acceptable, especially if the template function is very simple, but it is worth considering.

Your biggest argument against the MemberFunctionWrapper method was the need for dynamic memory allocation. For the sake of argument, that dynamic allocation isn't strictly necessary either. Consider the following…

// Using dynamic memory
AbstractBase* function = new MemberFunctionWrapper(&Class::Do);

// Using static memory
template inline AbstractBase* GetMemberFunctionWrapper()
{
// The compiler must generate a unique instance of this function-local static for each T.
static MemberFunctionWrapper s_wrapper;
return s_wrapper;
}

AbstractBase* function = GetMemberFunctionWrapper(&Class::Do);

This example obviously isn't complete. You would have to have a way to bind an actual instance of "Class" along with the member function pointer, but you get the idea. This may or may not be a better way to implement delegates, but I mention it because it is a useful technique in general. It is certainly a technique that I have used many times, and to good results. Thanks again.

Reply

nate@createdimage.com on May 4, 2012 at 9:34 pm

Wow. I guess WordPress didn't like my template brackets. Also, I see now that there are inconsistencies in the parameters to GetMemberFunctionWrapper. I guess I typed it too quickly. Hopefully you do get the idea. I actually implemented a system like this once, but the object instance and the member function pointer were both passed as argument to the wrapper's "Invoke" function. You are correct that member function pointers can't be passed by void*, but they can be passed by another wrapper of sufficient size. The correctness of "sufficient size" can be guaranteed using static asserts. Doing it that way meant a lot less code generation, and resulted in a measurable reduction in code size and compile time. Runtime invocation was fast enough, but not quite as fast as you have done. Which is better? Like everything, it all depends on how the system is used in your code base. Thanks again.

Reply

Stefan Reinalter on May 4, 2012 at 10:28 pm

Thanks for the comments!

You are absolutely right about code size, templates generally can bloat your code if you're not careful. But since the wrapper is really small (essentially just a function call and passing arguments), the generated code should be small as well, albeit I didn't check.
Since delegates are used in specific cases only, there's maybe a hundred unique instantiations, hence I'm not too concerned about code size, but it is something to keep in mind in general. I would be concerned about hundred different instantiations for a std::vector instead, for example :).

Regarding dynamic memory allocations you could also add e.g. a char[16] member to the base class, and construct the MemberFunctionWrapper by using placement-new. That still leaves you with the virtual function call overhead, though.

As you said (and I totally agree), it absolutely depends how a system is used in an engine/codebase. People shouldn't just blindly use something without checking the advantages/disadvantages of a specific implementation first.

Dieter on June 18, 2012 at 10:30 pm

Hi Stefan,

How do you handle const member function ?
I'm currently looking at different delegate implementations and while I really like your solution, it doesn't seem to take const member functions like these :
class foo
{
void Bar(int i) const {printf("Yellow !");}
};
is this a limitation of your system, am I doing it wrong or is this something you omitted from this post ?

↪ Reply

Stefan Reinalter on June 19, 2012 at 12:05 pm

Hi Dieter,

It is not an inherent limitation of the system, but I omitted it for clarity because it complicates matters. If you want to use const member functions you will notice that the respective Delegate::Bind method is missing – the one I posted will only take non-const member functions as non-type template arguments.

First, you need to use a type which is able to hold both const and non-const typeless pointers. A union fits the bill, because we will only use one pointer at a time, never both. This new type becomes InstancePtr:

```
union InstancePtr
{
    InstancePtr(void) : as_void(nullptr) {}

    void* as_void;
    const void* as_const_void;
};
```

Now, in addition to the original Delegate::Bind method you need another method which takes const-member functions as template parameter:

```
template <class C, R (C::*Function)(ARG0)>
void Bind(C* instance)
{
    m_stub.first.as_void = instance;
    m_stub.second = &ClassMethodStub<C, Function>;
}

template <class C, R (C::*Function)(ARG0) const>
void BindConst(const C* instance)
{
    m_stub.first.as_const_void = instance;
    m_stub.second = &ConstClassMethodStub<C, Function>;
}
```

Note both the **const** in the template argument, and the **const** in the function argument (const C* instance). I've named the method BindConst for now – we'll come back to that later.
Of course now you need stubs for both const and non-const member functions, which is easy to come up with:

```
template <class C, R (C::*Function)(ARG0)>
static ME_INLINE R ClassMethodStub(InstancePtr instance, ARG0 ar
{
    return (static_cast<C*>(instance.as_void)->*Function)(arg0);
}

```

```
template <class C, R (C::*Function)(ARG0) const>
static ME_INLINE R ConstClassMethodStub(InstancePtr instance, AR
{
    return (static_cast<const C*>(instance.as_const_void)->*Func
}
```

Again, note the **const** in the template argument, the static_cast, and when using the union (instance.as_const_void). All of this is important in order to make the code compile, and keep the same semantics as if you were calling those functions on an instance directly.

If you want to go one step further and rename *BindConst* to *Bind*, you have to employ another trick to make it work. The reason you cannot call both methods *Bind* as of now is due to overload resolution. Each instance can implicitly be converted into a *const C**, so the compiler will never try to use the first overload. We need to force the compiler into only using the non-const overload if the template argument is a non-const member function, and vice versa.

This can be achieved with the following:

```
template <class C, R (C::*Function)(ARG0)>
struct NonConstWrapper
{
    NonConstWrapper(C* instance)
        : m_instance(instance)
    {
    }

    C* m_instance;
};

template <class C, R (C::*Function)(ARG0) const>
struct ConstWrapper
{
    ConstWrapper(const C* instance)
        : m_instance(instance)
    {
    }

    const C* m_instance;
};

template <class C, R (C::*Function)(ARG0)>
void Bind(NonConstWrapper<C, Function> wrapper)
{
    m_stub.first.as_void = wrapper.m_instance;
    m_stub.second = &ClassMethodStub<C, Function>;
}

template <class C, R (C::*Function)(ARG0) const>
void Bind(ConstWrapper<C, Function> wrapper)
{
    m_stub.first.as_const_void = wrapper.m_instance;
    m_stub.second = &ConstClassMethodStub<C, Function>;
}
```

By introducing two helpers called NonConstWrapper and ConstWrapper, we've achieved two things:

1) Both Bind methods are now truly different overloads, taking different arguments.
2) By using non-explicit constructors in our helpers, the arguments need to always undergo implicit conversion.

Because the template argument *Function* is part of the function argument (so to say), only non-const member functions can be converted into a NonConstWrapper temporary, and const member functions need to take the ConstWrapper overload.
It's not super-nice, but it gets the job done and works.

Reply

Dieter on June 19, 2012 at 9:21 pm

Thanks for the detailed reply 🙂
Upon the part of the NonConstWrapper and ConstWrapper, I figured it out myself but didn't like the fact that I had to use 2 bind functions (a const and a non-const bind).
looks like you are smarter than me XD

Stefan Reinalter on June 20, 2012 at 10:00 am

Not smarter, but maybe having more experience "bending" the language in certain ways to make corner-cases work :).

Viktor on July 25, 2016 at 8:58 pm

Unfortunately, your solution with the Non-/ConstWrapper doesn't work in VS 2015, giving an ambiguity error when trying to bind a non-/const member function when the tested object itself isn't const. Is there any other way to workaround the implicit const conversion?

Stefan Reinalter on July 26, 2016 at 8:16 am

What is the bug you're seeing?
I filed a bug report with Microsoft for Visual Studio 2013 which was fixed in Update 1.

Anyway, the Const/NonConstWrapper was a work-around for older compiler versions anyway. If you're on a somewhat recent compiler (>= VS 2013 should do), you can use my solution presented in Game Engine Gems 3.

Henri Korpela on June 25, 2012 at 4:07 pm

Thanks a lot for this excellent solution. Clean, easy-to-use and simple altogether. I can only thank God for finding this implementation. Highly recommended for anybody wanting to use delegates.

Reply

eduwushu on May 2, 2013 at 9:30 pm

First of all Stefan, congratulations for this excellent blog which now is one of my favourites. I've learnt so much in this single post about template programming and macro programming that I cannot do other thing but to thank you for the effort and time spent on it (I know writing and having a blog up to date its an overwhelming task!)

I've used a pseudo-delegate system in other applications using Qt signals and slots (which is another particular way of implementing the observer pattern using a preprocessor approach).

Now I have started and application on which I've used you delegate implementation to have the same mechanism. Obviously some questions come to mind when comparing to the Qt signal/slot system. Do you handle the delegate connection time manually or you have an automated system for disconnecting registered listeners when the listener itself is destroyed? Without such an automated system ensuring that things are unregistered from all the delegates they were connected before their destruction can be complex. For example lets thing a transform component in an entity that has a TransformChanged signal and several objects register into it. If one of the listener objects is detsroyed that leaves us with a dangling pointer on the delegate side, so we must ensure the listener deregisters itself before its destruction.

Ive taken a look at boost::signal interface and i dont like the fact that if you want something to be a 'managed' listener (understanding 'managed' as a listener that deregisters itself on destruction) you need to inherit from the trackable class. Inmy case by default initially i would want every listener to inherit from trackable, dirtying my application inheritance tree with this trackable inheritance troughout the code. I wonder if there is a simpler solution.

Reply

Stefan Reinalter on May 3, 2013 at 12:12 pm

Well, thanks for the kind feedback! I'm glad you enjoy reading my blog.

As you probably noticed by reading my blog, I'm not a big fan of certain full-blown doing-stuff-behind-your-back automatisms, but rather prefer to be explicit most of the time. The same is

true for delegates and events.
On the engine side, I use events only to decouple modules from each other, and only in places where the engine is responsible for setting up events and their listeners. One such example would be loading all kinds of different resources from a resource package. The engine sets up all listeners, starts loading the package, and after loading has finished, all listeners and the event go out of scope. This means that registering/unregistering is still fully controlled by the package loader, but subsystems still don't have to know anything about each other.

For entity components I prefer to give the components some kind of reference to other components if needed, and have them call functions directly. This makes it so much easier to see what is going on just by looking at the code, without having to track down where some event listener registered to an event N frames ago. In my experience, delegates/events tend to introduce all kinds of hard-to-find bugs if they're overused, and generally make it much harder to reason about program flow.

So in essence, I don't think I have an answer to your question. As I said, I tend to stay away from those "just register here and you'll be notified automatically!"-kind of things, they can lead to messy and hard-to-understand code.

↪ Reply

---

eduwushu on May 4, 2013 at 6:33 pm

Hey! Thanks for your reply!! 🙂 . Ive been thinking about it and, at the end is not an event a way of keeping a reference to objects that you can call? I suppose it is nothing more than a callback system. It is kind of the same thing with the exception that maybe in the delegate scheme you can register from outside the notifier object without it knowing anything about the objects that have registered. If you have for example different things that must respond to changes in a transform, it is not more convenient to have a delegate for such notification than having to store explicit pointers to each one and calling them explicitely from the code? I've seen other implementations that instead of using delegates they use message passing, but is kind of the same thing and for my taste slightly less efficient as those approaches normally rely on message broadcasting into all components of an entity.

Stefan Reinalter on May 4, 2013 at 7:27 pm

Yes, I can see what your point is. But I try to stay away from such "generic" things – if something needs to respond to changes in a transform, I want to know **who** that is, and try to be as explicit as possible.

A good example for this is reading/writing transformations in a MeshComponent. The renderer subsystem needs to render a bunch of meshes at different positions in the world, which might also be changed by e.g. a RigidBodyComponent.
So as long as a RigidBodyComponent only writes to a *Transform*, and a MeshComponent reads from a *Transform*, both don't have to know anything about each other, and both can explicitly write/read the transformation, without having to rely on callbacks/delegates/events.
Of course, the question arises who the owner of the Transform is, and how it can be efficiently updated in bulk in a multi-threaded, cache-friendly manner. That will be the topic of another DOD-related blog post :).

I agree that there are cases where it's just much, much more convenient to use an event and be done with it, but those should be the exception, not the norm.

Junyu on August 24, 2013 at 1:48 pm

Thanks a lot for your contribution. it's very helpful to me.

I have a question. When I try to modify the Delegate::Bind method with the following:

```
template <class C>
void Bind(C* instance, void (C::*Function)(int))
{
m_stub.first = instance;
m_stub.second = &ClassMethodStub<C, Function>;
}
```

visual c++ complaint C2440….

Reply

Stefan Reinalter on August 24, 2013 at 4:56 pm

Well, you cannot change the implementation like this.
ClassMethodStub needs Function to be a compile-time argument, but in your case, it's a runtime argument.

What it is that you're trying to do?

Reply

Junyu on August 24, 2013 at 7:18 pm

Thanks for your reply. I just want to reduce template arguments. XD

Reply

Stefan Reinalter on August 24, 2013 at 11:58 pm

Which is possible, but then you have to use an entirely different approach. One that would work is to use an abstract interface (possibly a nested class inside of Delegate) which has one virtual function for calling the actual, underlying free function or member function.
Then you'd need two implementations: one that is able to use a pointer-to-function, and one that uses a pointer-to-member-function. Furthermore, the delegate class could have two constructors: again, one for free functions, the other for member functions. The constructor then instantiates the right implementation, and hands over the argument to the implementation. All that Delegate::Invoke() then needs to do is call implementation->Invoke().

I hope that was clear :).

Reply

Pingback: Research Assignment – Event Systems | A study in Programming

Lin Wang on January 28, 2014 at 3:37 am

Great article! I'm just in the middle of making my own simple engine. It's really helpful. But I have a question, you mentioned using custom build to output the preprocesssed code to a file and included that file instead. I tried it, the problem is that the output code doesn't preserve the format(no line break), all code is all in one line, which is still not enough to actually inspect into the code because the compiler would only point to one line if anything goes wrong in that line.

How would that help for debugging the code? Thanks.

Reply

Garnold on January 28, 2014 at 10:23 am

I've used a python tool cog (http://www.python.org/about/success/cog/) to generate my delegate include file only once. If I want to change it, I just do so in a comment, run cog on the file and enjoy the changes in all implementations.

Reply

Lin Wang on January 29, 2014 at 6:38 am

Thanks, I checked it out, I will definitely try that!

Stefan Reinalter on January 28, 2014 at 4:55 pm

Which compiler do you use?

Reply

Lin Wang on January 29, 2014 at 6:38 am

It's Visual Studio 2010 student version

Stefan Reinalter on January 29, 2014 at 11:30 am

I also use VS 2010, so this should work. This is part of the command-line I use:

```
cl.exe /P /EP /C %(FullPath) /I"..\..\src"
```

You're likely missing the /C switch. Check here for a full list of compiler command-line options: http://msdn.microsoft.com/en-us/library/fwkeyyhe(v=vs.100).aspx

Nathan P. on April 17, 2014 at 6:52 pm

Hi, I just came across this article because I had to implement a similar system at work (not allowed to use boost, otherwise it would've been a good enough solution for my needs), and found this article to be very useful. I have some suggested "improvements" for the delegate.

1. I found it useful to add == and != operators so that delegates could be compared. I based my event system off of storing a std::list of delegates, found it useful to compare them for easy unregistration.

2. It would also be nice to create delegates that are bound upon construction. Is this possible?

Additionally, as much as I tried to follow the template specialization example of the delegate you provided, I just couldn't get it to work. Unfortunately work has tied me to VS2005, so maybe it doesn't support the required syntax. Instead I had to do "template class Delegate {…}; followed by "template class Delegate {…};" for my one-argument and zero-argument implementations, respectively. I only needed 0 or 1 arguments, so I didn't bother with the preprocessor magic.

Reply

Stefan Reinalter on April 17, 2014 at 7:25 pm

1. I found it useful to add == and != operators so that delegates could be compared. I based my event system off of storing a std::list of delegates, found it useful to compare them for easy unregistration.

True, my final implementation also has that.

2. It would also be nice to create delegates that are bound upon construction. Is this possible?

Unfortunately not, because there is no way to pass non-deducible template arguments to the constructor in C++.
You can however create a static function that does that for you, e.g. my implementation offers a Make() function that returns a fully-constructed and bound delegate.

Additionally, as much as I tried to follow the template specialization example of the delegate you provided, I just couldn't get it to work. Unfortunately work has tied me to VS2005, so maybe it doesn't support the required syntax.

I'm not sure if VS2005 fully understands partial template specializations, which is what we use here.

Reply

Shanker Sharma on March 23, 2019 at 8:25 pm

Hi Stefan,

I love your article and I have started to follow your blog posts:)

Can you please share the final implementation?

I am curious to see how you handled delegate comparisons.

Thanks,
Shanker

Stefan Reinalter on March 26, 2019 at 5:38 pm

Both pre-C++11 and C++11 implementations are available on the supplementary page of Game Engine Gems 3.

Pingback: Хорошие Книги » Шаблоны игрового программирования. Наблюдатель

Pingback: Generic Delegates: C++ Implementation | Anas Awad's T-Blog

Julien Kirsch on June 16, 2016 at 9:22 pm

Thanks for the post, helped me to create my own Event System.
However, how would you change your delegate class to make use of the RAII pattern?
I am using a static factory method for now.

And for those who are asking, why he's not using 'std::function'. I did a measurement. This implementation is faster by a factor of 32% percent compared to std::function.

Reply

Stefan Reinalter on June 20, 2016 at 10:21 am

Yes, just use a static function. I added a static Make() to the class for that purpose.

Reply

Pingback: Game Pattern – Observer – Life is a journey!

Pingback: Simple event system in C++ - Tutorial Guruji

Pingback: Как добавить делегаты в C++ - IT блог: новости it, программирование

RSS

RSS - Posts
RSS - Comments

Contact

Molecular Matters
Facebook

Recent Posts

Live++ 1.0.0 released!
C++ programming tips
Deleting .pdb files locked by Visual Studio
Job System 2.0: Lock-Free Work Stealing – Part 5: Dependencies
Getting the type of a template argument as string – without RTTI

Recent Comments

S Adlan on Deleting .pdb files locked by…

Stefan Reinalter on Job System 2.0: Lock-Free Work…

Ivan on Job System 2.0: Lock-Free Work…

Howard L on Baking signals into textures

Как добавить делегат… on Generic, type-safe delegates a…

Categories

Asset pipeline (3)
C++ (31)
Core (32)
Editor (1)
Graphics (17)
Input (2)
Math (1)
Uncategorized (13)

Archive

March 2018 (1)
September 2017 (1)
May 2017 (1)
April 2016 (1)
December 2015 (1)
November 2015 (1)
September 2015 (2)

August 2015 (4)
February 2015 (1)
December 2014 (3)
November 2014 (2)
August 2014 (1)
May 2014 (1)
February 2014 (1)
January 2014 (1)
October 2013 (1)
August 2013 (2)
July 2013 (1)
May 2013 (3)
March 2013 (1)
February 2013 (1)
January 2013 (1)
December 2012 (1)
November 2012 (1)
October 2012 (2)
September 2012 (1)
August 2012 (2)
July 2012 (1)
June 2012 (1)
May 2012 (2)
April 2012 (3)
March 2012 (1)
January 2012 (1)
December 2011 (2)
November 2011 (4)
October 2011 (2)
September 2011 (6)
August 2011 (4)
July 2011 (6)
June 2011 (6)

View Full Site