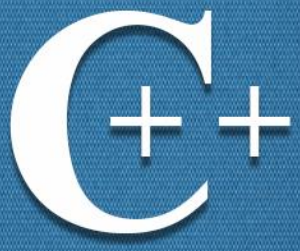

Delegate

vdksoft@gmail.com

Table of Contents

Introduction

Overview

Design Goals

Tutorial

Performance

Public API

Introduction

A delegate is a class that represents references to functions with a particular parameter list and return type. When a delegate is instantiated, it can be associated with any function with a compatible signature and return type. This function can be invoked later through the delegate instance. In short, a delegate is a form of type-safe function pointer that specifies a function to call and optionally an object to call this function on.

Delegates are widely used in situations where some function needs to be passed as an argument to some other function - callback mechanism. Of cause, in C++ function pointers could be used for this purpose. However, this approach causes some problems. The biggest one of them is the fact that in Object Oriented World non-static member function (method) pointers are completely useless without class instances they could be called on. Another problem is that member function pointers of different classes have different types even if method signatures and return types are absolutely the same. It may be not a bad thing itself, but makes it inconvenient to pass and store those function pointers as callback functions.

Unlike some other languages that already have a built-in solid and heavy weight mechanism for callbacks, C++ as usually gives us a light-weight, flexible and low-level building blocks to implement such a mechanism in the way we wish. So, function / method pointers and object pointers can be used as low-level building blocks to build higher level generalized abstraction that would encapsulate all details and keep type-safe function pointer regardless of whether this function is free function, member function or function object.

Overview

`vdk::delegate` is an implementation of delegate concept for C++. It is a general-purpose type-safe function pointer wrapper. Instances of **`vdk::delegate`** can store, copy, and invoke pointers to any callable target such as free (global) function, static member function of some class (static method), member function of some class (method), function object (functor), or lambda. Any of those that matches the delegate type – has a compatible signature and return type, can be assigned to the delegate and invoked later.

To assign some function / method / functor to **`vdk::delegate`** instance user passes function pointer / object pointer and method pointer / functor pointer to the delegate's constructor. To invoke the delegate "**`operator()`**" is used. If target function assigned to the delegate has an argument list, all arguments must be passed into the **`operator()`**.

Any delegate can be in one of two states: either it is associated with some target function or it is empty. In latter case it is equal to null pointer. Attempting to invoke a null delegate results in undefined behavior, just like attempting to dereference regular null pointer. Moreover, if an exception occurs in target function during a delegate invocation, and that exception is not caught within the target function that was called, the search for an exception catch clause continues in the function that called the delegate, as if that function had directly called the function to which that delegate referred.

`vdk::delegate` satisfies the requirements of Copy-constructible and Copy-assignable, so it is possible to store instances of this class in Standard Library containers. It is also possible to compare delegates for equality. Two delegates are considered to be equal if they point to the same function / object and method / functor. **`vdk::delegate`** is quite fast and does not use any dynamic memory allocation at all. On all main platforms it's size is either equal or less than the size of **`std::function`**.

In order to use this library, compiler must be compatible with at least C++11 Standard. **`vdk::delegate`** relies on Standard C++ only, and does not use any hacks, non-standard extensions or third party libraries.

Design Goals

The following list shows the main design goals which were set and achieved in this library.

1) Delegate must be as simple to use as possible, and must have clean and intuitive syntax;
2) Delegate must behave as generalized function pointer, without any unnecessary functionality;
3) Creation, copying, assigning, and invoking delegate instances must be as fast as possible;
4) Delegate must not use any dynamic memory allocation;
5) Delegate must rely on Standard C++ only without any non-standard extensions or third party libraries;
6) Delegate instances must be comparable to identify whether they point to the same function;
7) Delegate must be copy-constructible and copy-assignable to be used in Standard containers;
8) Delegate must be able to work with methods with qualifiers: **const**, **volatile**, **const volatile**.

Tutorial

Since the best way to explore new functionality is to look at some piece of code, let's have a look at several very simple examples below.

Suppose we have a free function, some class with three methods, and a functor.

```
struct SomeClass
{
	static int static_method()
	{
		std::cout << "SomeClass: int static_method()" << std::endl;
		return 0;
	}

	void method_1(int i)
	{
		std::cout << "SomeClass: void method_1(int " << i << ")" << std::endl;
	}

	void method_2(int i) const
	{
		std::cout << "SomeClass: void method_2(int " << i << ")const" << std::endl;
	}
};

void free_function(int i)
{
	std::cout << "void free_function(int " << i << ")" << std::endl;
}

struct Functor
{
	double operator()(int i1, int i2)
	{
		std::cout << "Functor: double operator()(int " << i1 << ", int " << i2 << ")\n";
		return 10.5;
	}
};
```

The following declares an empty delegate **d1** that takes one argument of type **int** and returns **void**.

```
vdk::delegate<void(int)> d1;
```

Instead of creating the empty delegate we could assign some function in delegate's constructor.

```
vdk::delegate<void(int)> d1(&free_function);
```

Now we can use **d1** to execute the target function with 10 as parameter according to function signature. To invoke this delegate just call **operator()** with expected argument type:

```
d1(10);
```

We can also assign new function / method / functor to the existing delegate. In this case previous function pointer will be lost. This time let's create an instance of **SomeClass** and assign its method to our delegate. We can also define type alias, just for convenience, like this:

```
using MyDelegate = vdk::delegate<void(int)>;

SomeClass object;

d1 = MyDelegate(&object, &SomeClass::method_1);
d1(10);
```

It does not matter whether assigned method has **const** / **volatile** qualifier, so **method_2()** could be assigned to the delegate as well:

```
d1 = MyDelegate(&object, &SomeClass::method_2);
d1(10);
```

Let's create new delegate with different type and assign **static_function()** of **SomeClass** to this delegate. Note, that **static_function()** does not take any parameters, so you can declare and invoke the delegate for this function like this:

```
vdk::delegate<int()> d2(&SomeClass::static_method); // Equivalent to vdk::delegate<int(void)>
d2();
```

Now let's create function object **Functor** that takes two arguments of type **int** and returns **double**. Then, we can assign this functor to a delegate. To do this, we need to pass the functor's address to the delegate's constructor.

```
Functor functor;

vdk::delegate<double(int, int)> d3(&functor);
d3(10, 20);
```

And, of cause, we need to look at our friends – lambdas. Let's declare a delegate for lambda that takes two arguments **std::string** and **double** and returns **bool**.

```
auto lambda = [](std::string s, double d)->bool
{
	std::cout << "Hi! I am lambda!" << std::endl;
	std::cout << "My arguments: " << s << ", " << d << std::endl;
	return true;
};

vdk::delegate<bool(std::string, double)> d4(&lambda);
bool result = d4("String", 10.5);
```

Just like regular pointer can be equal to null pointer, **vdk::delegate** can be equal to null pointer as well. We can assign any delegate to null pointer to make it empty (to clear it).

```
d1 = nullptr;
```

Of cause, we can check whether particular delegate has an associated function (is not equal to null). It can be done using **bool operator** or explicit comparison with null pointer, like this:

```
if (d1)
{
	std::cout << "delegate is NOT empty" << std::endl;
}
```

```
if (d1 == nullptr)
{
	std::cout << "delegate is empty" << std::endl;
}

if (d1 != nullptr)
{
	std::cout << "delegate is NOT empty" << std::endl;
}
```

It is possible to compare two delegates to find out whether they refer to the same function. **Note:** comparison of two delegates makes sense only if they have the same type. If they have different types they are a priori different and refer to different functions.

```
vdk::delegate<void(int)> d5(&object, &SomeClass::method_1);
vdk::delegate<void(int)> d6(&object, &SomeClass::method_2);
vdk::delegate<void(int)> d7(&object, &SomeClass::method_1);

if (d5 == d6)
{
	std::cout << "d5 and d6 point to the same method and the same object" << std::endl;
}

if (d5 == d7)
{
	std::cout << "d5 and d7 point to the same method and the same object" << std::endl;
}
```

We can also copy-construct new delegate from an existing one and we can assign one delegate to another:

```
vdk::delegate<void(int)> d8(d7);
d5 = d8;
```

Performance

Performance of **vdk::delegate** was measured on Windows (Visual Studio 2015 CE) and Linux (GCC 4.8.5) platforms in so called "release" builds (with optimizations, like –O3). However, to get an impression about how fast some piece of code is it should be compared to something else. It is pretty much silly to compare delegates with direct function or method calls, since if direct function calls had the same capabilities that delegates have, there would not be much point in delegates at all. So, a delegate should be treated as a generalized function pointer. As any pointer it will be eventually dereferenced (to invoke associated function) and this pointer dereference operation itself introduces the overhead.

So, at this point it should be obvious that any delegate has at least one dereference operation for function invocation, and this overhead is unavoidable. It is inherently bound to delegate concept itself. However, **vdk::delegate** implementation has overhead of two pointer dereference operations. Where does the second operation come from?

Well, it may not be obvious, but it is needed internally in the delegate class to provide type erasure. Any stored pointers regardless of whether they are function or method pointers, must not affect the delegate type itself. They must be stored in some generalized way. To achieve this some type erasure mechanism must be used. The mechanism implemented here does not use virtual functions, but it has its overhead (equal to pointer dereference) anyway.

Summarizing, performance of **vdk::delegate** is almost identical to two pointer dereference operations. Such overhead is not that much even for performance critical applications, so it should be acceptable in majority of cases.

Public API

vdk::delegate<ResT(ArgTs...)>	
`delegate() noexcept;`	Creates an empty delegate.
`explicit delegate(std::nullptr_t) noexcept;`	Creates an empty delegate.
`explicit delegate(Function * function) noexcept;`	Creates a delegate with a function as target.
`delegate(Class * object, Method * method) noexcept;`	Creates a delegate with an object and a method as target.
`explicit delegate(Class * functor) noexcept;`	Creates a delegate with a function object as target.
`delegate(const delegate & other) noexcept;`	Creates new delegate as a copy of an existing delegate.
`~delegate() noexcept;`	Destroys delegate instance.
`delegate & operator=(const delegate & other) noexcept;`	Assigns a delegate to the same target as another delegate.
`delegate & operator=(std::nullptr_t) noexcept;`	Assigns a delegate to null pointer (clears delegate).
`ResT operator(ArgTs ... arguments) const;`	Invokes target function associated with the delegate, passes arguments and returns value. **Note:** if delegate is equal to null pointer this call results in undefined behavior. If target function throws an exception during its execution, that exception is propagated to the delegate invoker.
`bool operator==(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if delegates are equal, i.e. point to the same function / the same functor / the same object and the same method.
`bool operator=!(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if delegates are not equal.
`bool operator==(std::nullptr_t) const noexcept;`	Compares the delegate with a null pointer (checks whether the delegate is empty).
`bool operator=!(std::nullptr_t) const noexcept;`	Compares the delegate with a null pointer. (checks whether the delegate is not empty).
`bool operator<(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if **`*this`** is less than the **`other`**.
`bool operator>(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if **`*this`** is greater than the **`other`**.
`bool operator<=(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if **`*this`** is less or equal than the **`other`**.
`bool operator>=(const delegate & other) const noexcept;`	Compares two delegates. Returns **`true`** if **`*this`** is greater or equal than the **`other`**.
`explicit operator bool() const noexcept;`	Checks whether the delegate is not empty. Returns **`true`** if delegate is not empty.

Non-member functions	
`bool operator==(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if delegates are equal.
`bool operator!=(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if delegates are not equal.
`bool operator<(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if **`lhs`** is less than **`rhs`**.
`bool operator<=(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if **`lhs`** is less or equal than **`rhs`**.

`bool operator>(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if **`lhs`** is greater than **`rhs`**.
`bool operator>=(const delegate & lhs, const delegate & rhs) noexcept;`	Compares two delegates. Returns **`true`** if **`lhs`** is greater or equal than **`rhs`**.
`bool operator==(const delegate & lhs, std::nullptr_t) noexcept;`	Compares the delegate with a null pointer (checks whether the delegate is empty).
`bool operator==(std::nullptr_t, const delegate & rhs) noexcept;`	Compares the delegate with a null pointer (checks whether the delegate is empty).
`bool operator!=(const delegate & lhs, std::nullptr_t) noexcept;`	Compares the delegate with a null pointer (checks whether the delegate is not empty).
`bool operator!=(std::nullptr_t, const delegate & rhs) noexcept;`	Compares the delegate with a null pointer (checks whether the delegate is not empty).

I hope this library will be useful to you. If you have any comments, questions, ideas, suggestions, or propositions, please feel free to contact me at: **vdksoft@gmail.com**. All feedbacks will be much appreciated.