# Variadic delegates

At some point I came to need of writing a subsystem for my framework that creates graphics effects. One of the traits of many render effects is the execution of operations’ set, changing the pipeline state before the shader part comes into action and after it. In other words we need kind of operator brackets – Begin()/End() functions – inside which effects’ algorithm will take the action. I didn’t find made-up solutions and wrote my own implementation.

It was needed to save function and method delegates side by side in one container. The signature of a delegated entity should be arbitrary. There must be the capability to bind delegated methods and functions with arguments before the actual call and thus – the capability to call right away the entire delegates’ collection with tied in advance arguments. The args should be passed and saved via perfect forwarding, in order not to make redundant copies. While making the delegate, user should not write superfluous code, the notation must be in terms with common sense, no signatures and data types in template parameters. That’s how the delegates supposed to be used:

// Constructing delegates' collection

DelegatesCollection dc;

// Adding delegates for functions f1, f2 and for methods meth1, meth2

dc.add(&f1);

dc.add(&obj1, &meth1);

dc.add(&obj2, &meth2);

dc.add(&f2);

// Calling f1 through its delegate and passing in arguments

dc[0](arg1, arg2, …, argN);

// Binding arguments to meth1's delegate

dc[1].bind\_args(arg1, arg2, …, argN);

// Calling meth1 with previously bound arguments

dc[1].call\_with\_bound\_args();

// Binding arguments to meth2 and f2

dc[2].bind\_args(arg1, arg2, …, argN);

dc[3].bind\_args(arg1, arg2, …, argN);

// Batch calling entire delegates' collection, i.e. f1, meth1, meth2 and f2, with previously bound arguments

dc.batch\_call\_with\_bound\_args();

All in all everything that had been put-up was implemented.

Features:

1) Capability of storing function and method delegates together in one container

2) Ability to batch call the entire delegates’ collection using bound args

3) Delegates are stored in container std::vector

4) Arguments and result data types are deduced automatically from function signature – there’s no need to specify them as template parameters

## Delegates design

Delegates are designed so that next tasks are solved:

1. To store method delegates (MD) and function delegates (FD), both entities must be described with one interface.

2. In the output we should receive Delegate class comprising the implementation of the notion delegate.

3. The user should have the possibility to create delegates’ collection. This will require DelegatesSystem class, which will be the container for Delegate instances.

4. DelegatesSystem must provide the interface for batch call of delegated functions and methods with bound arguments, i.e. for sequential launch of all delegates’ collection elements.

### Making the interface for MD and FD:

Since in general case we’ll delegate methods and functions with arbitrary parameters count, the interface being made must be a template. For MD creation pointers to object and its method must be present, for FD – pointer to function. These pointers will be passed through template params.

Class Delegate must be independent of concrete data types, since it is a generic abstraction of the delegate notion, thus it cannot be a template.

The question arises, in which way then we have to describe MD and FD with one interface and associate them with Delegate class?

The only way of doing it is to create an interface general for MD and FD and to store its instances, initialized with the appropriate pointers to implementations, inside Delegate.

The next question is how to get to template interface for MD and FD would have described these entities simultaneously?

Fortunately there is such thing as partial template specialization, which we’ll use for solving this task.

Basically MD and FD are the same thing, they differ only in one component. Therefore, the interface describing them will have two specializations.

# Implementation

Generic interface:

class IDelegateData abstract { *needed pure virtual methods are here* };

Generic template:

template<class...Args> class DelegateData : public IDelegateData {};

Since methods are distinct from functions in that former are called on a concrete object of a class, thus specialization of the generic template for MD will additionally have parameter for class of object to which the method belongs: class O.

So, we’ll get the following two specializations of generic template interface:

### For methods:

template<class R, class O, class... Args>

class DelegateData<R, O, R(Args...)> : public IDelegateData

{

public:

typedef R(O::\*M)(Args...);

DelegateData(O\* pObj, M pMethod) : m\_pObj(pObj), m\_pMethod(pMethod) {}

private:

O\* m\_pObj;

M m\_pMethod;

};

### For functions:

template<class R, class... Args>

class DelegateData<R, R(\*)(Args...)> : public IDelegateData

{

public:

typedef R(\*F)(Args...);

DelegateData(F pF) : m\_pF(pF) {}

private:

F m\_pF;

};

## Delegate

Storing instances of DelegateData inside Delegate:

class Delegate

{

public:

Delegate() = default;

// For methods

template<class R, class O, class...Args>

explicit Delegate(O\* pObj, R(O::\*M)(Args...))

{

bind(pObj, M);

}

template<class R, class O, class...Args>

void bind(O\* pObj, R(O::\*M)(Args...))

{

m\_data = new DelegateData<R, O, R(Args...)>(pObj, M);

}

// For functions

template<class R, class...Args>

explicit Delegate(R(\*F)(Args...))

{

bind(F);

}

template<class R, class...Args>

void bind(R(\*F)(Args...))

{

m\_data = new DelegateData<R, R(\*)(Args...)>(F);

}

private:

IDelegateData\* m\_data;

};

Field Delegate::m\_data will be filled with the specialization of DelegateData for MD or FD accordingly to which Delegate ctor is called.

## DelegatesSystem

Let’s consider DelegatesSystem class, which will store delegates’ collection in the form of Delegate objects:

class DelegatesSystem

{

private:

vector<Delegate> m\_delegates;

};

It’s left to “teach” DelegatesSystem to add delegates Delegate into the collection and to gain access to them by index.

The easiest way of adding delegates to the collection is to simply list the arguments needed for Delegate ctor in the addition method. For this we’ll require variadic templates, rvalue references and new sequential containers’ method emplace\_back(...), taking arguments pack and adding new element into the collection via constructing the object in place (in place construction), calling that ctor of its class which corresponds to the arguments passed.

class DelegatesSystem

{

public:

template<class...Args>

void add(Args&&... delegateCtorArgs)

{

m\_delegates.emplace\_back(std::forward<Args>(delegateCtorArgs)...);

}

Delegate& operator[](uint idx)

{

return delegates[idx];

}

private:

vector<Delegate> m\_delegates;

};

Now we can add delegates by calling any ctor of a Delegate class and all this is via one method.

## Arguments

We are close to our goal. The only thing left is to provide for the possibility of calling delegates with an arbitrary number of arguments. Thus you have to start with making changes in the interface, which is empty for the moment. Let’s add to IDelegateData pure virtual method for calling delegates:

class IDelegateData abstract { public: virtual void call(void\*) abstract; };

We’ll receive arguments inside the method DelegateData::call, passing them there being placed in the instance of Arguments class, from where we’ll extract them, casting void\* to Arguments<...>\*.

To the details. Arguments will be stored inside Arguments in tuple, since it’s a container, allowing to store arbitrary number of different type elements:

template<class... Args>

class Arguments

{

public:

Arguments(Args&&... args) : m\_args(std::forward\_as\_tuple(std::forward<Args>(args)...)) {}

public:

std::tuple<Args&&...> m\_args;

};

Everywhere while passing the arguments we use perfect forwarding in order to draw move semantics for rvalues and for lvalues to be passed by reference.

We have the following call chain leading to call of the method/function being delegated:

Delegate::operator() -> DelegateData::call

We write overload of operator Delegate::operator(), using variadic templates, to make it capable of receiving any number of arguments:

class Delegate

{

public:

*previous declarations*

template<class...Args>

void operator()(Args&&... args)

{

m\_data->call(new Arguments<Args...>(std::forward<Args>(args)...));

}

private:

IDelegateData\* m\_data;

};

Here we create instance of Arguments, store args arguments pack in it and pass it to the method call() of a polymorphic field m\_data, which triggers call() of a corresponding specialization of DelegateData, where arguments are *somehow* extracted and passed for invocation of a delegated function/method:

template<class R, class O, class... Args>

class DelegateData<R, O, R(Args...)> : public IDelegateData

{

public:

*previous declarations*

void call(void\* pArgs) override

{

*extraction of arguments from* pArgs *and passing them to the delegated function/method;*

*i.e. in the high-level sense the following will occur:*

(m\_pObj->\*m\_pMethod)(extraction of arguments pack from pArgs);

}

private:

O\* m\_pObj;

M m\_pMethod;

};

For DelegateData specialization implementing FD conception everything looks similarly.

## Extraction of arguments from tuple and passing them to delegate

Now arises the task of extracting arguments from tuple and passing them to method or function. At the moment there is a standard way of extracting tuple’s elements one at a time, for what we have template function std::get<idx>(tuple) described in header <tuple>, which takes index of required element as its template parameter idx. What we have is parameter pack Args (defined in specializations of DelegateData), which contains data types of arguments of delegated function/method, thus we know how to cast from void\*. We’d like to solve given task as follows (we’ll use example of MD and without perfect forwarding):

(m\_pObj->\*m\_pMethod)(std::get<what\_goes\_here?>(static\_cast<Arguments<Args...>\*>(pArgs)->args)...);

I.e. we need parameter pack Args for casting pArgs to the appropriate pointer to Arguments. Call of a function get(), should be done on account of indices pack expansion, what would allow to extract arguments stored in tuple as one list and to pass them to the delegated method/function. Therefore now we have to find a way of generating indices list corresponding to the number of params in Args pack. There is such a way, it’s underlain by metaprogramming and template recursion [IndicesTrick]. Here’s how indices creation mechanism looks like:

template<int... Idcs> class Indices{};

template<int N, int... Idcs> struct IndicesBuilder : IndicesBuilder<N - 1, N - 1, Idcs...> {};

template<int... Idcs>

struct IndicesBuilder<0, Idcs...>

{

typedef Indices<Idcs...> indices;

};

For the sake of speeding things up I’ll give a brief high-level explanation of this technique’s operational principles for those, who isn’t familiar with metaprogramming and template recursion [Metaprog]. IndicesBuilder template should be considered as a function f(N), which takes integral argument, denoting the number of indices, and returns the pack of these indices. To calculate the number of indices we’ll harness new function sizeof...(parameter\_pack), applying it to Args: sizeof...(Args).

Indices passing we’ll mediate with yet another one method where we’ll make call of the delegated method:

template<int...Idcs>

void invoker(Indices<Idcs...>, void\* pArgs)

{

(m\_pObj->\*m\_pMethod)(std::get<Idcs>(static\_cast<Arguments<Args...>\*>(pArgs)->args)...);

}

After adding perfect forwarding and simplifying the expression:

template<int...Idcs>

void invoker(Indices<Idcs...>, void\* pArgs)

{

auto pArguments = static\_cast<Arguments<Args...>\*>(pArgs);

(m\_pObj->\*m\_pMethod)(std::get<Idcs>(pArguments->m\_args)...);

}

The final implementation of the overridden method call we’ll look the following way:

void call(void\* pArgs) override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), pArgs);

}

The specialization of DelegateData template for MD entirely:

template<class R, class O, class... Args>

class DelegateData<R, O, R(Args...)> : public IDelegateData

{

public:

typedef R(O::\*M)(Args...);

DelegateData(O\* pObj, M pMethod) : m\_pObj(pObj), m\_pMethod(pMethod) {}

void call(void\* pArgs) override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), pArgs);

}

template<int...Idcs>

void invoker(Indices<Idcs...>, void\* pArgs)

{

auto pArguments = static\_cast<Arguments<Args...>\*>(pArgs);

(m\_pObj->\*m\_pMethod)(std::get<Idcs>(pArguments->m\_args)...);

}

private:

O\* m\_pObj;

M m\_pMethod;

};

The specialization for FD will look similarly.

So, the task of storing arbitrary number of function and method delegates with any given signature and making it possible to call them later is solved. It’s left to add arguments binding and batch call of the delegates.

## Arguments binding and batch call of the delegates

For making delegates’ batch call possible we’ll have to bind arguments to them. Respectively, each delegate should be able to store its argument set. And by the time of delegates’ batch call, arguments tied with them will be passed to the corresponding methods/functions.

### IDelegateData:

Let’s add interfaces of binding arguments to the delegate - bind\_args and of calling the delegate with bound arguments - call\_with\_bound\_args, and field void\* m\_pBound\_args, where the very arguments will be stored:

class IDelegateData abstract

{

public:

virtual void call(void\*) abstract;

virtual void call\_with\_bound\_args() abstract;

virtual void bind\_args(void\*) abstract;

protected:

void\* m\_pBound\_args;

};

### DelegateData:

For storing arguments let’s add in every specialization of DelegateData ctor overload, and override bind\_args and call\_with\_bound\_args:

// Data for methods

template<class R, class O, class... Args>

class DelegateData<R, O, R(Args...)> : public IDelegateData

{

public:

typedef R(O::\*M)(Args...);

DelegateData(O\* pObj, M pMethod) : m\_pObj(pObj), m\_pMethod(pMethod) {}

void call(void\* pArgs) override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), pArgs);

}

template<int...Idcs>

void invoker(Indices<Idcs...>, void\* pArgs)

{

auto pArguments = static\_cast<Arguments<Args...>\*>(pArgs);

(m\_pObj->\*m\_pMethod)(std::get<Idcs>(pArguments->m\_args)...);

}

public:

DelegateData(O\* pObj, M pMethod, Args&&... argsToBind) : m\_pObj(pObj), m\_pMethod(pMethod)

{

bind\_args(new Arguments<Args&&...>(std::forward<Args>(argsToBind)...));

}

virtual void bind\_args(void\* argsToBind) override

{

if (argsToBind != m\_pBound\_args)

{

delete m\_pBound\_args;

m\_pBound\_args = argsToBind;

}

}

void call\_with\_bound\_args() override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), m\_pBound\_args);

}

private:

O\* m\_pObj;

M m\_pMethod;

};

// Data for functions

template<class R, class... Args>

class DelegateData<R, R(\*)(Args...)> : public IDelegateData

{

public:

typedef R(\*F)(Args...);

DelegateData(F pF) : m\_pF(pF) {}

void call(void\* pArgs) override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), pArgs);

}

template<int...Idcs>

void invoker(Indices<Idcs...>, void\* pArgs)

{

auto pArguments = static\_cast<Arguments<Args...>\*>(pArgs);

m\_pF(std::get<Idcs>(pArguments->m\_args)...);

}

public:

DelegateData(F pF, Args&&... argsToBind) : m\_pF(pF)

{

bind\_args(new Arguments<Args&&...>(std::forward<Args>(argsToBind)...));

}

virtual void bind\_args(void\* argsToBind) override

{

if (argsToBind != m\_pBound\_args)

{

delete m\_pBound\_args;

m\_pBound\_args = argsToBind;

}

}

void call\_with\_bound\_args() override

{

invoker(typename IndicesBuilder<sizeof...(Args)>::indices(), m\_pBound\_args);

}

private:

F m\_pF;

};

### Delegate:

To the Delegate we’ll add overloads of ctors and methods bind, template bind\_args and call\_with\_bound\_args, which will serve merely as wrappers over interfaces of IDelegateData:

class Delegate

{

public:

...

template<class R, class O, class...Args, class...ArgsToBind>

explicit Delegate(O\* m\_pObj, R(O::\*M)(Args...), ArgsToBind&&... argsToBind)

{

bind(m\_pObj, M, std::forward<ArgsToBind>(argsToBind)...);

}

template<class R, class...Args, class...ArgsToBind>

explicit Delegate(R(\*F)(Args...), ArgsToBind&&... argsToBind)

{

bind(F, std::forward<ArgsToBind>(argsToBind)...);

}

template<class R, class O, class...Args, class...ArgsToBind>

void bind(O\* pObj, R(O::\*M)(Args...), ArgsToBind&&... argsToBind)

{

m\_data = new DelegateData<R, O, R(Args...)>(pObj, M, std::forward<ArgsToBind>(argsToBind)...);

}

template<class R, class...Args, class...ArgsToBind>

void bind(R(\*F)(Args...), ArgsToBind&&... args)

{

m\_data = new DelegateData<R, R(\*)(Args...)>(F, std::forward<ArgsToBind>(args)...);

}

template<class... Args>

void bind\_args(Args&&... args)

{

m\_data->bind\_args(new Arguments<Args...>(std::forward<Args>(args)...));

}

void call\_with\_bound\_args()

{

m\_data->call\_with\_bound\_args();

}

...

};

### DelegatesSystem:

To the DelegatesSystem we’ll add launch for a batch call of entire delegates’ collection:

class DelegatesSystem

{

public:

...

void launch()

{

for (auto& d : m\_delegates)

d.call\_with\_bound\_args();

}

private:

vector<Delegate> m\_delegates;

};

References:

Metaprog: Learn C++ Metaprogramming Concepts - <http://metacpp.codeplex.com/>

IndicesTrick: Variadic templates в C++0x <http://habrahabr.ru/post/101430/>, Indices (The indices trick) - <http://loungecpp.wikidot.com/tips-and-tricks:indices>