# Variadic delegates

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

В своё время потребовалось написать подсистему создания графических эффектов. Одной из особенностей многих рендер эффектов является потребность в выполнении последовательности операций, изменяющих состояние графического конвейера до отработки шейдерной части и после неё. Т.е. нужны операторные скобки - функции Begin() и End() – внутри которых будет отрабатывать алгоритм эффекта. Ну в общем, не нашёл я готовых решений, и написал свою реализацию.

Нужен был способ сохранять делегаты функций и методов бок о бок в одном контейнере. Сигнатура делегируемой сущности должна быть какой угодно. Должна быть возможность связывать делегируемые методы и функции с аргументами до момента вызова и соответственно – возможность вызывать сразу всю коллекцию делегатов с заранее привязанными к ним аргументами. Для передачи и сохраения аргументов использовать идеальное перенаправление, чтобы нигде не делать лишних копий. При создании делегата пользователь не должен писать лишнего кода, запись должна соответствовать здравому смыслу, никаких сигнатур и типов данных в параметрах шаблонов. Предполагалось следующее использование делегатов:

It was needed to save function and method delegates side by side in one container. The signature of a delegated entity should be arbitratry. There must be the capability to bind delegated methods and functions with arguments before the actual call and thus – the capability to call the entire delegates’ collection at once with arguments tied in advance. 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 I saw the delegates usage:

DelegatesCollection dc;

dc.add(&f1);

dc.add(&obj1, &meth1);

dc.add(&obj2, &meth2);

dc.add(&f2);

dc[0](arg1, arg2, …, argN);

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

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

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

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

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

В итоге всё задуманное осуществилось.

Особенности:

- возможность хранения в одном контейнере делегатов функций и методов

- возможность пакетного вызова всей коллекции делегатов, используя связанные аргументы

- делегаты хранятся в контейнере std::vector

- типы данных аргументов и возвращаемого результата автоматически выводятся из сигнатуры функции – нет необходимости указывать их в качестве параметров шаблона

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

Features:

- capability of storing function and method delegates together in one container

- can batch call the entire delegates collection using bound args

- delegates are stored in container std::vector

## Delegates design:

Initial task:

1. Сохранять делегаты методов (ДМ) и функций (ДФ), обе сущности должны описываться одним интерфейсом. To store method delegates (MD) and function delegates (FD), both entities must be described with one interface.

2. На выходе должен получиться класс Delegate, представляющий собой реализацию понятия делегат. In the output we should receive Delegate class comprising the implementation of the notion delegate.

3. Пользователь должен иметь возможность создать коллекцию делегатов. Для этого потребуется класс DelegatesSystem, который будет являться контейнером для экземпляров класса Delegate. 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 должен предоставлять интерфейс пакетного вызова делегируемых функций и методов с привязанными аргументами, т.е. последовательного запуска всех элементов коллекции делегатов. DelegatesSystem must provide the interface of batch call of delegated functions and methods with bound arguments, i.e. for sequential launch of all delegates’ collection elements.

### Создание интерфейса, описывающего ДМ и ДФ Making of 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.

Т.к. в общем случае мы будем делегировать методы и функции с произвольным количеством параметров, то создаваемый интерфейс должен быть шаблонным. Для создания ДМ нужно иметь указатели на объект и его метод, для ДФ – указатель на функцию. Данные указатели будут передаваться через параметры шаблона.

Сам класс Delegate должен быть независим от конкртеных типов данных, т.к. является обобщённой абстракцией понятия делегат, следовательно, он не может быть шаблонным.

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.

Возникает вопрос, каким образом тогда описать ДМ и ДФ одним интерфейсом и ассоциировать их с классом Delegate?

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

Единственный способ сделать это – создать общий для ДМ и ДФ интерфейс и сохранять его экземпляры, инициализированные соответствующими указателями, внутри Delegate.

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.

Следующий вопрос - как добиться того, чтобы шаблонный интерфейс для ДМ и ДФ, одновременно описывал эти разные сущности?

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

К счастью есть такое явление, как частичная специализация шаблона, которой мы воспользуемся для решения данной задачи.

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

По сути ДМ и ДФ это одно и то же, отличаются они лишь одним компонентом. Следовательно, описывающий их интерфейс будет иметь две специализации.

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 here* };

Generic template:

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

Since methdos are distinct from functions in that former are called on concrete object of a class, 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 following two specializations of the generic template interface:

### 1 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;

};

### 2 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, relatively 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 in the addition method needed for Delegate ctor. 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 passed arguments.

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 able to receive 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 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 *и передача их делегируемым функции/методу;*

*т.е. в высокоуровневом смысле произойдёт следующее:*

(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

Теперь возникает задача извлечения аргументов из кортежа и передачи их в метод или функцию. На данный момент существует стандартный способ извлекать элементы из кортежа по одному, для чего в модуле <tuple> описана шаблонная функция get<idx>(tuple), которая в качестве параметра шаблона idx принимает индекс требуемого элемента. У нас же есть пакет параметров Args (заданный в специализациях DelegateData), который содержит типы данных аргументов делегируемых функции/метода, т.о. мы знаем как преобразовать void\*. Мы хотели бы решить поставленную задачу следующим образом (рассмотрим на примере ДМ и без идеального перенаправления):

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

То есть пакет параметров Args нам нужен для преобразования pArgs к соответствующему указателю на Arguments, а также для пакетного вызова функции get(), что позволило бы извлечь сохранённые в кортеже аргументы одним списком и подставить их в делегируемый метод/функцию. Но возникает вопрос, а что передать в качестве параметра-индекса в get()?А передать туда мы можем только пакет индексов аргументов, хранящихся в кортеже. Т.е. теперь нам нужно найти способ формировать список индексов, в соответствии с количеством параметров в пакете Args. Такой способ есть, в его основе лежат метапрограммирование и шаблонная рекурсия [IndicesTrick]. Вот так выглядит механизм создания индексов:

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;

};

Для быстроты дела дам краткое высокоуровневое объяснение принципов работы данного способа тем, кто не знаком с метапрограммированием и шаблонной рекурсией [Metaprog]. Шаблон IndicesBuilder нужно рассматривать как функцию f(N), которая принимает целочисленный параметр, указывающий количество индексов, и возвращает массив этих индексов. Для вычисления количества индексов, мы воспользуемся новой функцией sizeof...(пакет\_параметров), применив её к параметру Args: sizeof...(Args).

Передачу индексов опосредуем ещё одним методом и в нём же осуществим вызов делегируемого метода:

template<int...Idcs>

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

{

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

}

Добавим идеальное перенаправление аргументов и упростим выражение:

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)...);

}

Конечная реализация переопределяемого метода call будет выглядеть следующим образом:

void call(void\* pArgs) override

{

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

}

Специализация шаблона DelegateData для ДМ полностью:

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;

};

Аналогично будет выглядеть и специализация для ДФ.

Итак, задача хранения произвольного количества делегатов функций и методов с произвольной сигнатурой и возможности последующего их вызова решена. Осталось добавить привязку аргументов и пакетный вызов.

## Привязка аргументов и пакетный вызов делегатов

Для пакетного вызова делегатов нам потребуется привязывать к делегатам аргументы. Соответственно, каждый делегат должен уметь хранить свой набор аргументов. У нас должна быть возможность связать делегат с аргументами в любой момент времени. А на момент пакетного вызова делегатов, связанные с ними аргументы будут передаваться соответствующим методам/функциям.

### IDelegateData:

Добавим интерфейсы - привязки аргументов к делегату bind\_args и вызова делегата с привязанными аргументами call\_with\_bound\_args, и поле void\* m\_pBound\_args, где будут храниться сами аргументы:

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:

Для сохранения аргументов добавим в каждую специализацию DelegateData перегрузку конструктора, и переопределим bind\_args и call\_with\_bound\_args:

// Данные для методов

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:

template<class...ArgsToBind>

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

{

bind\_args(new Arguments<ArgsToBind&&...>(std::forward<ArgsToBind>(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;

};

// Данные для функций

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:

template<class...ArgsToBind>

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

{

bind\_args(new Arguments<ArgsToBind&&...>(std::forward<ArgsToBind>(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:

В Delegate добавим перегрузки конструкторов и методов bind, шаблонный bind\_args и call\_with\_bound\_args, которые будут просто обёртками над интерфейсами 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:

В DelegatesSystem добавим метод launch для пакетного вызова всей коллекции делегатов:

class DelegatesSystem

{

public:

...

void launch()

{

for (auto& d : m\_delegates)

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

}

private:

vector<Delegate> m\_delegates;

};

Ссылки:

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>