# Вариативные делегаты

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 некие операторные скобки - функции Begin() и End() – внутри которых будет отрабатывать алгоритм эффекта. Я не нашёл 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:

// Создаём коллекцию делегатов

DelegatesCollection dc;

// Добавляем делегаты для функций f1, f2 и методов meth1, meth2

dc.add(&f1);

dc.add(&obj1, &meth1);

dc.add(&obj2, &meth2);

dc.add(&f2);

// Вызываем f1 через её делегат, передавая аргументы

dc[0](arg1, arg2, …, argN);

// Привязывем аргументы к делегату meth1

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

// Вызываем meth1 с ранее привязанными аргументами

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

// Привязываем аргументы к meth2 и f2

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

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

// Пакетный вызов всей коллекции делегатов, т.е. f1, meth1, meth2 и f2, с привязанными до этого аргументами

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

В итоге всё задуманное было реализовано.

Особенности:

1) Возможность хранения в одном контейнере делегатов функций и методов

2) Возможность пакетного вызова всей коллекции делегатов, используя связанные аргументы

3) Делегаты хранятся в контейнере vector

4) Типы данных аргументов и возвращаемого результата автоматически выводятся из сигнатуры функции – нет необходимости указывать их в качестве параметров шаблона

## Устройство делегатов

Устройство делегатов соответствует следующим задачам:

1) Сохранять делегаты методов (ДМ) и функций (ДФ), обе сущности должны описываться одним интерфейсом.

2) На выходе должен получиться класс Delegate, представляющий собой реализацию понятия делегат.

3) Пользователь должен иметь возможность создать коллекцию делегатов. Для этого потребуется класс DelegatesSystem, который будет являться контейнером для экземпляров класса Delegate.

4) DelegatesSystem должен предоставлять интерфейс пакетного вызова делегируемых функций и методов с привязанными аргументами, т.е. последовательного запуска всех элементов коллекции делегатов.

### Создание интерфейса, описывающего ДМ и ДФ:

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.

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

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

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

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

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

# Реализация

Общий интерфейс:

class IDelegateData abstract { *необходимые чисто виртуальные методы здесь* };

Общий шаблон:

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

Т.к. отличием методов от функций является то, что первые вызываются на конкретном объекте класса, специализация общего шаблона для ДМ будет дополнительно иметь параметр для класса объекта, которому принадлежит метод: class O.

Итак, мы получим следующие две специализации общего шаблонного интейрфейса:

### 1 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) {}

private:

O\* m\_pObj;

M m\_pMethod;

};

### 2 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) {}

private:

F m\_pF;

};

## Delegate

Сохранение экземпляров DelegateData в Delegate:

class Delegate

{

public:

Delegate() = default;

// Для методов

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);

}

// Для функций

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;

};

Поле Delegate::m\_data будет заполнено специализацией DelegateData для ДМ или ДФ, в зависимости от того какой конструктор Delegate отработает.

## DelegatesSystem

Рассмотрим класс DelegatesSystem, который будет хранить коллекцию делегатов в виде объектов Delegate:

class DelegatesSystem

{

private:

vector<Delegate> m\_delegates;

};

Осталось «научить» DelegatesSystem добавлять делегаты Delegate в коллекцию и получать к ним доступ по индексу.

Максимально лёгким способом добавления делегатов в коллекцию будет простое перечисление в методе добавления аргументов, требуемых конструктору Delegate. Для этого нам потребуются вариативные шаблоны, правозначные ссылки и новый метод последовательных контейнеров emplace\_back(...), принимающий пакет параметров и добавляющий новый элемент в последовательность путём конструирования объекта на месте (in place construction), вызывая тот конструконструктор его класса, который соответствует переданным аргументам.

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;

};

Теперь можно добавлять делегаты, вызывая любой конструктор класса Delegate, и всё это при помощи одного метода.

## Arguments

Мы уже близки к цели. Всё что осталось, это обеспечить возможность вызова делегатов с произвольным числом аргументов, следовательно, начинать надо с внесения изменений в интерфейс, который на данный момент у нас пустой. Добавим в класс IDelegateData чисто виртуальный метод для вызова делегатов:

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

Будем принимать аргументы внутри метода DelegateData::call, передавая их туда помещёнными в экземпляр класса Arguments, откуда мы будем их извлекать, преобразуя void\* к Arguments<...>\*.

Перейдём к деталям. Аргументы будем сохранять внутри Arguments в кортеже, т.к. это контейнер, позволяющий хранить произвольное количество разнотипных элементов:

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;

};

Везде при передаче аргументов используем идеальное перенаправление (perfect forwarding) для задействования семантики перемещения во время передачи правых значений и передачи левых значений по ссылке.

Мы имеем следующую цепочку вызовов, приводящую к вызову делегируемого метода:

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

Пишем перегрузку оператора Delegate::operator(), используя вариативные шаблоны, для того чтобы он мог принимать любое число параметров:

class Delegate

{

public:

*предыдущие объявления*

template<class...Args>

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

{

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

}

private:

IDelegateData\* m\_data;

};

Здесь мы создаём экземпляр Arguments, сохраняем в нём пакет параметров args и передаём его в метод call() полиморфного поля m\_data, что приводит к вызову call() у соответствующей специализации DelegateData, где аргументы *определённым образом* извлекаются и передаются для вызова делегируемой функции/методу:

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

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

{

public:

*предыдущие объявления*

void call(void\* pArgs) override

{

*извлечение аргументов из* pArgs *и передача их делегируемым функции/методу;*

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

(m\_pObj->\*m\_pMethod)(излечение пакета аргументов из pArgs);

}

private:

O\* m\_pObj;

M m\_pMethod;

};

Для специализации DelegateData, реализующей концепцию ДФ всё выглядит аналогично.

## Извлечение аргументов из кортежа и передача их делегату

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 std::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. Вызов функции get() должен произойти за счёт раскрытия пакета индексов, что позволило бы извлечь сохранённые в кортеже аргументы одним списком и подставить их в делегируемый метод/функцию. 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 аргумент, указывающий количество индексов, и возвращает пакет 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 ним аргументы. Соответственно, каждый делегат должен уметь хранить свой набор аргументов. А во время 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:

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;

};

// 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:

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:

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>