# Вариативные делегаты

Ссылки:

- дать ссылку на codeplex – по метапрограммированию и шаблонной рекурсии

Предыстория возникновения идеи:

- занимаюсь компьютерной графикой, использую для функций Begin() и End(), играющих роль, своего рода, операторных скобок

- поскольку все изученные мной варианты реализаций делегатов не в полной мере удовлетворяли моим задачам, решил написать ВГД самостоятельно

Особенности:

- типы данных аргументов и возвращаемого результата автоматически выводятся из сигнатуры ф-ии – нет необходимости указывать их в качестве парам-в шаблона – чистый и аккуратный код

- возможность пакетного хранения разнородных делегатов в произвольном порядке

- возможность пакетного вызова делегатов, используя «связанные» аргументы – аргументы, адреса которых заранее сохранены внутри делегата

- хранение произвольного числа разнородных делегатов в динамическом массиве vector – возможность динамического добавления/удаления делегатов

## Архитектура и устройство делегатов:

Исходная задача:

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

2. На выходе должен получиться класс Delegate, представляющий собой реализацию понятия гетерогенный делегат. Следовательно интерфейс класса Delegate должен быть таким, чтобы его пользователь имел возможность делегировать ДМ и ДФ.

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

4. При добавлении делегата в коллекцию должна присутствовать возможность назначить ему имя.

5. Пользователь DelegateSystem должен иметь возможность видоизменять коллекцию – добавлять, удалять, заменять, переименовывать её элементы.

6. DelegateSystem должен предоставлять интерфейс пакетного вызова делегируемых методов, т.е. последовательного запуска всех элементов коллекции делегатов.

1) Создание интерфейса, описывающего ДМ и ДФ

Т.к. в общем случае мы будем делегировать методы и функции с произвольным количеством параметров, то создаваемый интерфейс также должен иметь возможность принимать произвольное количество параметров. Следовательно, такой интерефейс должен быть шаблонным. Для создания ДМ нужно иметь у-ли на объект и его метод, для ДФ – у-ль на ф-ю. Данные у-ли и 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.

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 у-ми, внутри Delegate.

Очевидно, что для того, чтобы для хранения каких-либо данных внутри нешаблонного класса, тип поля также не может быть шаблонным, т.к. пришлось бы указывать параметры шаблона во время объявления такого поля.

Тогда можно использовать полиморфизм, объявив такое поле как указатель на абстрактный класс, ялвяющийся базовым для шаблонного класса, описывающего ДМ и ДФ.

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\* ptrObj, M \_method) : pObj(ptrObj), method(\_method) {}

private:

O\* pObj;

M method;

};

### 2 Для функций:

template<class R, class... Args>

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

{

public:

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

DelegateData(F ptrF) : pF(ptrF) {}

private:

F pF;

};

## Delegate

Сохранение экземпляров DelegateData в Delegate:

class Delegate

{

public:

// Для методов

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

explicit Delegate(std::string \_name, O\* pObj, R(O::\*M)(Args...)) : name(\_name) { bind(pObj, M); }

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

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

{

idata = new DelegateData<R, O, R(Args...)>(pObj, M);

}

// Для функций

template<class R, class... Args>

explicit Delegate(std::string \_name, R(\*pF)(Args...)) : name(\_name) { bind(pF); }

template<class R, class...Args>

void bind(R(\*F)(Args...))

{

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

}

private:

IDelegateData\* idata;

std::string name;

};

Поле Delegate::idata будет заполнено специализацией DelegateData для ДМ или ДФ, в зависимости от того какой ктор Delegate отработает.

## DelegatesSystem

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

class DelegatesSystem

{

private:

vector<Delegate> 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 ктору 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 конструктор его класса, который соответствует переданным аргументам.

class DelegatesSystem

{

public:

template<class...Args>

void add(Args&&... delegateCtorArgs)

{

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

}

Delegate& operator[](uint idx)

{

return delegates[idx];

}

private:

vector<Delegate> delegates;

};

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

## Arguments

Мы уже близки к цели. Всё что осталось, это обезпечить возможность вызова делегатов с произвольным числом параметров, 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:

std::tuple<Args&&...> args;

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

{

idata->call(new Arguments<Args...>(std::forward<Args>(args)...));

}

private:

IDelegateData\* idata;

};

Здесь мы создаём экземпляр Arguments, сохраняем в нём пакет параметров args и передаём его в метод call() полиморфного поля idata, 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\* p\_args) override

{

*извлечение аргументов из p\_args и передача их делегируемым функции/методу;*

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

(pObj->\*method)(излечение\_пакета\_аргументов\_из p\_args);

}

private:

O\* pObj;

M method;

};

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 одновременной их передачи 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):

(pObj->\*method)(std::get<что\_подставить\_сюда?>(static\_cast<Arguments<Args...>\*>(p\_args)->args)...);

То есть пакет параметров Args нам нужен для приведения p\_args 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 уже давно придуман, в его основе лежат метапрограммирование и шаблонная рекурсия дать ссылку на codeplex – по метапрограммированию и шаблонной рекурсии . Я нашёл его здесь дать ссылку на stackoverflow. 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 разъяснение принципов работы indices trick тем, кто не знаком с метапрограммированием и шаблонной рекурсией. 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\* p\_args)

{

(pObj->\*method)(std::get<Idcs>(static\_cast<Arguments<Args...>\*>(p\_args)->args)...);

}

С учётом идеального перенаправления аргументов:

template<int...Idcs>

void invoker(Indices<Idcs...> idcs, IArguments\* p\_args)

{

(pObj->\*method)(std::forward<Args>(std::get<Idcs>(static\_cast<Arguments<Args...>\*>(p\_args)->args))...);

}

В упрощённом виде:

template<int...Idcs>

void invoker(Indices<Idcs...> idcs, IArguments\* p\_args)

{

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

(pObj->\*method)(std::forward<Args>(std::get<Idcs>(pArguments->args))...);

}

Конечная реализация переопределяемого метода call() будет выглядеть следующим образом:

void call(void\* p\_args) override

{

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

}

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\* ptrObj, M \_method) : pObj(ptrObj), method(\_method) {}

void call(void\* p\_args) override

{

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

}

template<int...Idcs>

void invoker(Indices<Idcs...> idcs, IArguments\* p\_args)

{

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

(pObj->\*method)(std::forward<Args>(std::get<Idcs>(pArguments->args))...);

}

private:

O\* pObj;

M method;

};

Аналогично будет выглядеть и специализация для ДФ.

Итак, задача хранения произвольного количества делегатов методов и функций с произвольной сигнатурой и возможности последующего их вызова решена.

----------

#include <iostream>

#include <vector>

using namespace std;

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;

};

void main()

{

}

----------

## 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 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():

class IDelegateData abstract

{

public:

virtual void call(void\*) abstract;

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

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

};

### DelegateData:

Для сохранения аргументов добавим в каждую специализацию DelegateData поле void\* m\_bound\_args, перегрузку ктора, 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\* ptrObj, M \_method) : pObj(ptrObj), method(\_method) {}

template<class...Args>

void operator()(Args... args)

{

(pObj->\*method)(args...);

}

void call(void\* p\_args) override

{

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

}

template<int...Idcs>

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

{

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

(pObj->\*method)(std::get<Idcs>(pArguments->args)...);

}

public:

template<class...ArgsToBind>

DelegateData(O\* ptrObj, M \_method, ArgsToBind&&... argsToBind) : pObj(ptrObj), method(\_method)

{

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

}

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

{

if (argsToBind != m\_bound\_args)

{

delete m\_bound\_args;

m\_bound\_args = argsToBind;

}

}

void call\_with\_bound\_args() override

{

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

}

private:

O\* pObj;

M method;

void\* m\_bound\_args;

};

// Данные для функций

template<class R, class... Args>

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

{

public:

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

DelegateData(F ptrF) : pF(ptrF) {}

template<class...Args>

void operator()(Args... args)

{

pF(args...);

}

void call(void\* p\_args) override

{

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

}

template<int...Idcs>

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

{

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

pF(std::get<Idcs>(pArguments->args)...);

}

public:

template<class...ArgsToBind>

DelegateData(F ptrF, ArgsToBind&&... argsToBind) : pF(ptrF)

{

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

}

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

{

if (argsToBind != m\_bound\_args)

{

delete m\_bound\_args;

m\_bound\_args = argsToBind;

}

}

void call\_with\_bound\_args() override

{

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

}

private:

F pF;

void\* m\_bound\_args;

};

### Delegate:

В Delegate добавим перегрузки ктора и метода 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(std::string \_name, O\* pObj, R(O::\*M)(Args...), ArgsToBind&&... argsToBind) : name(\_name)

{

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

}

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

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

{

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

}

template<class... Args>

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

{

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

}

void call\_with\_bound\_args()

{

idata->call\_with\_bound\_args();

}

};

### DelegatesSystem:

В DelegatesSystem добавим метод для пакетного вызова всей коллекции делегатов:

class DelegatesSystem

{

public:

...

void launch()

{

for (auto& d : delegates)

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

}

private:

vector<Delegate> delegates;

};