Часть 1. Причёсываем код. Машины состояний

Очень часто необходимо, чтобы объект вёл себя определённым образом в зависимости от разных условий – состояний. К примеру, это может быть всплывающее окно, которое может иметь три состояния – появление, стандартная отрисовка, скрытие или же игровой персонаж, который может спокойно стоять, бегать или стрелять. Для определения подобного рода действий может использоваться код

///////////////////////////////

enum class State

{

Wait,

Dance,

Joke,

Move,

Run,

Attack,

};

void Update(float dt)

{

State state = GetCurrentState();

switch (state)

{

case State::Wait:

// animation idle

break;

case State::Move:

// animation move

// calculate new position

break;

case State::Attack:

// animation attack

// shoot the sheriff

break;

}

// очевидно изменяем состояние

state = State::Attack;

}

///////////////////////////////

Но может и техника, которая была для этого создана – машина состояний.

///////////////////////////////

CharacherFSM char\_fsm;

void Update(float dt)

{

char\_fsm.OnUpdate(dt);

…

char\_fsm.ProcessEvent(walk\_away(std::make\_pair(1, 1)));  
}

///////////////////////////////

Машина состояний уже не раз была описана и реализована разными людьми, но либо мне не хватало функционала, либо вместе с одной машиной нужно тащить целую [библиотеку](http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML/). Поэтому было решено залезть под капот и написать свою реализацию. О приключениях при создании будет рассказано в данной статье.

Часть 1. Теория, постановка задачи, имплементация.

Теоретическое определение машины состояний можно почитать на [Вики](https://en.wikipedia.org/wiki/Finite-state_machine), но нам – программистам – необходимо понять какой интерфейс нужен и как будут использовать сие творение. Данный шаблон проектирование планируется использоваться в системах реального времени, например, в играх. И входной/выходной набор символов, который необходим для полного соответствия математической абстракции будет упущен. Итак, что мы имеем.

1. Конечный набор состояний.
2. Начальное состояние.
3. Таблица переходов.
4. Конечное состояние.

И набор ограничений и пожеланий:

1. В машине может быть только одно активное состояние
2. Количество состояний и переходов должно быть масштабируемо.
3. Максимально простой интерфейс. Желаемый код:

/////////////////

StateMachine char\_fsm;

char\_fsm.AddState(Walk());

char\_fsm.AddState(Dance());

char\_fsm.SetTransitions({\_row<Wait, Dance, idle\_action\_completed>()});

…

char\_fsm.Start<FirstState>();

…

char\_fsm.OnUpdate(.1f);

…

char\_fsm.ProcessEvent(run\_away(position(0, 1)));

...

/////////////////

В главном классе машины –*StateMachine* – хранятся все состояния, одно из которых – текущее. В функции *Update* находится код, который переключает состояния. Есть два возможных подхода в переключении состояний:

1. Переход по триггеру: в каждом вызове Update пробегать по таблице переходов и, если триггер активирован, совершать переключение.
2. Переход по событию: извне машины или в одном из состояний передавать событие, а в Update просматривать произошедшее. Если пришло какое-либо событие, и оно совпадает с одним из переходов – переключаемся.

Коренное отличие – кто будет переключателем. В первом варианте переключать состояния будет машина, соответственно логика переключения будет в одном из имплементаций абстрактного *Transition*. Во втором варианте, события приходят извне и передавать события может кто угодно. Оба подхода имеют свои плюсы и минусы и применяются к разным задачам с переменным успехом. Выбор зависит от личных предпочтений. Мой выбор пал на второй вариант. Подобная модель также реализована в [boost::meta state machine](http://www.boost.org/doc/libs/1_57_0/libs/msm/doc/HTML/ch03s02.html#d0e358). Первый подход описан в AI for games (ISBN-13: 978-0123747310).

Первым делом создаём базовые классы события и состояния

///////////////////////////////////////////

struct Event

{

// do not forget about virtual destructor

virtual ~Event() {}

// method for coping events

virtual std::unique\_ptr<Event> Copy() const { return nullptr; }

};

class BaseState

{

public:

virtual ~BaseState() {}

virtual void OnEnter(const Event&) {}

virtual void OnExit() {}

virtual void OnUpdate(float dt) {}

};

///////////////////////////////////////////

Для того, чтобы не нагружать пользователя класса ненужной имплементацией методов, все методы *BaseState* не являются чистыми виртуальными. Часто достаточно имплементировать только один или два, например, обновления и входа. В *Event* объявлен только один виртуальный деструктор, чтобы создалась виртуальная таблица и мы могли использовать магию полиморфизма и кастов.

Набор членов класса машины состояний по сути отражает требования:

/////////////////////////////////////////////

class StateMachine

{

private:

std::vector<std::unique\_ptr<BaseState>> m\_states;

BaseState\* mp\_current;

BaseState\* mp\_next;

std::vector<Transition> m\_transitions;

...

};

/////////////////////////////////////////////

Что до переходов, то это простая структура с тремя полями, которые являются хэш-коду класса-состояния, из которого будет переход, по какому событию переход, в какое состояние переходить.

/////////////////////////////////////////////

struct Transition

{

size\_t m\_hash\_from;

size\_t m\_hash\_to;

size\_t m\_hash\_event;

};

/////////////////////////////////////////////

Теперь нужно выполнить условие старта – с какого-то состояния нужно начать. Есть два возможных сценария: старт с каким-то событием (возможно, понадобится передать дополнительные данные), старт с нуля. Для старта с определённого типа используется шаблон и поиск в векторе по хэш-коду состояния.

/////////////////////////////////////////////

class StateMachine

{

...

private:

BaseState\* FindState(size\_t i\_hash)

{

auto it = std::find\_if(m\_states.begin(), m\_states.end(), [i\_hash](std::unique\_ptr<BaseState>& ip\_state)

{

return i\_hash == typeid(\*ip\_state).hash\_code();

});

if (it != m\_states.end())

return it->get();

return nullptr;

}

...

public:

template<typename State>

void Start()

{

mp\_next = FindState(typeid(State).hash\_code());

if (mp\_next)

mp\_cashed\_event.reset(new Event());

m\_terminate = mp\_next == nullptr;

}

template<typename State>

void Start(const Event& i\_evt)

{

mp\_next = FindState(typeid(State).hash\_code());

if (mp\_next)

mp\_cashed\_event = std::move(i\_evt.Copy());

m\_terminate = mp\_next == nullptr;

}

};

/////////////////////////////////////////////

Для создания переходов воспользуемся шаблонной функцией

/////////////////////////////////////////////

template <typename StateFrom, typename StateTo, typename Event>

Transition \_row()

{

return{ typeid(StateFrom).hash\_code(), typeid(StateTo).hash\_code(), typeid(Event).hash\_code() };

}

class StateMachine

{

...

public:

void AddTransition(Transition i\_trans)

{

m\_transitions.push\_back(i\_trans);

}

void SetTransitions(std::initializer\_list<Transition> transitions)

{

std::copy(transitions.begin(), transitions.end(), std::back\_inserter(m\_transitions));

}

void AddState(std::unique\_ptr<BaseState> ip\_state)

{

m\_states.emplace\_back(std::move(ip\_state));

}

};

/////////////////////////////////////////////

Вот мы и добрались до главного – обновление и **переключение** состояний! Но сначала о жизни простого состояния:

1. Когда выполняется переключение состояния, то необходимо выполнить вход – OnEnter
2. Далее делаем однообразные, или не очень, действия – OnUpdate
3. Затираем следы и хвосты – OnExit

Естественно, перед входом в новое состояние необходимо выйти из текущего, если оно есть. Но сразу вопрос – какое ожидаемое поведение при событии, которое будет посылаться из OnEnter состояния? По нашей логике – выйти из текущего (то есть того, что послало событие) и зайти в следующее. Проблема в том, что текущее состояние может быть ещё не до конца инициализировано и падение или неожиданное поведение может настичь пользователя в самый неподходящий момент. Конечно, можно жёстко следить за входом/выходом своих состояний, но проще, если подобная ситуация вообще не возникнет. Делаем дополнительное ограничение – переход возможен только в *StateMachine::OnUpdate*, а в ProcessEvent только кэшируем событие и указатель на следующее состояние.

/////////////////////////////////////////////

class StateMachine

{

private:

void CheckNextStateTransition()

{

if (mp\_next == nullptr && !m\_terminate)

return;

if (mp\_current != nullptr)

mp\_current->OnExit();

mp\_current = mp\_next;

if (mp\_cashed\_event)

mp\_current->OnEnter(\*mp\_cashed\_event);

else

{

Event base\_event;

mp\_current->OnEnter(base\_event);

}

mp\_next = nullptr;

mp\_cashed\_event.reset();

}

...

public:

void ProcessEvent(std::unique\_ptr<Event> ip\_event)

{

if (mp\_current == nullptr)

return;

const size\_t cur\_hash = typeid(\*mp\_current).hash\_code();

const size\_t ev\_hash = typeid(\*ip\_event).hash\_code();

for (auto& transition : m\_transitions)

{

if (cur\_hash == transition.m\_hash\_from && ev\_hash == transition.m\_hash\_event)

{

auto p\_next = FindState(transition.m\_hash\_to);

if (p\_next)

{

mp\_next = p\_next;

mp\_cashed\_event = std::move(ip\_event);

}

break;

}

}

}

void OnUpdate(float dt)

{

CheckNextStateTransition();

if (mp\_current)

mp\_current->OnUpdate(dt);

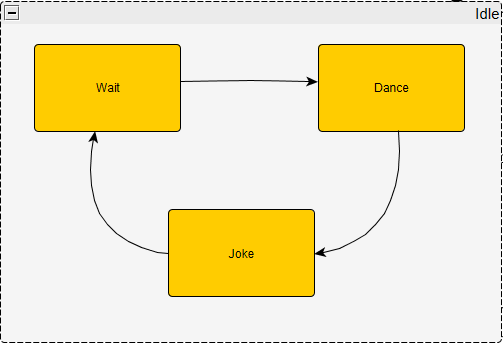
}

...

};

/////////////////////////////////////////////

Что ж, теперь можем попробовать создать тестовую машину состояний



В конструкторе каждого состояния передаём ссылку на родительскую машину, чтобы можно было отсылать события из самого состояния.

/////////////////////////////////////////////

struct idle\_action\_completed : public Event {};

struct Dance : public BaseState

{

StateMachine& m\_parent\_fsm;

Dance(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Dance] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

struct Joke : public BaseState

{

StateMachine& m\_parent\_fsm;

Joke(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Joke] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

struct Wait : public BaseState

{

StateMachine& m\_parent\_fsm;

Wait(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Wait] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(std::make\_unique<idle\_action\_completed>());

}

};

/////////////////////////////////////////////

Так как мы не привязаны ни к каким типам, то для создания машины будет отвечать функция, которая будет конструировать машину. Так как в состояниях используется ссылка на родительскую машину, то возвращать построенную машину по значению нельзя – поменяется адрес, поэтому передаём её как выходной параметр в функцию.

/////////////////////////////////////////////

void IdleFsm(StateMachine& o\_fsm)

{

o\_fsm.AddState(std::make\_unique<Wait>(o\_fsm));

o\_fsm.AddState(std::make\_unique<Dance>(o\_fsm));

o\_fsm.AddState(std::make\_unique<Joke>(o\_fsm));

o\_fsm.SetTransitions({

\_row<Wait, Dance, idle\_action\_completed>(),

\_row<Dance, Joke, idle\_action\_completed>(),

\_row<Joke, Wait, idle\_action\_completed>(),

});

o\_fsm.Start<Wait>();

}

void TestIdle()

{

StateMachine idle\_fsm;

IdleFsm(idle\_fsm);

idle\_fsm.Start<Wait>();

// or we can start with event

// idle\_fsm.Start<Wait>(idle\_action\_completed());

for (int i = 0; i < 10; ++i)

idle\_fsm.OnUpdate(.1f);

}

/////////////////////////////////////////////

***И что написать в выводах?***

Где добавить про компоновочные состояния – машина состояний в машине состояний? Если здесь, то получится слишком нагруженная статья. Возможно, в третьей части, где будет сравнение проводиться – мол, добавочные классы (они достаточно мелкие), но хотелось бы сравнение оставить сравнением, хотя придётся ещё буст туда притягивать (*или его притянуть во второй части, где будет шаблонная реализация*)

Часть 2. А как же шаблоны? / Больше контроля! Шаблонная реализация машины состояний. / *Машина состояний. На тёмной стороне С++*.

В первой части мы сделали машину состояний в классическом ООП-стиле. И есть очень неприятный недостаток в коде – никакого контроля за типами и, вследствие этого, в метод OnEnter может прийти любой класс-наследник Event. Допустим, переход в состояние может быть по нескольким событиям. Тогда код обработки будет выглядеть

///////////////////////////////////////////////

void OnEnter(const Event& ev)

{

if (const auto ptr = dynamic\_cast<idle\_action\_completed\*>(&ev))

…

else if (const auto ptr = dynamic\_cast<dance\_action\_completed \*>(&ev))

…

}

///////////////////////////////////////////////

Эти касты будут преследовать повсюду. Что ж, давайте посетим тёмную сторону С++ – шаблоны.

Постановка задачи такая же, как и в первой части, но с дополнением: контроль за типами подразумевает, что для каждого события будет свой метод и вызываться будет только он. Дополнительным условием хотелось бы ещё проверять наличие нужных методов в классе.

В ООП реализации были шаблонные классы для базового состояния и события. Событие может быть любым типом, так что для него базовый класс не нужен. В состоянии же поменяется только метод *OnEnter*:

///////////////////////////////////////////////

template <typename OnUdateParam = float>

class BaseState

{

public:

virtual ~BaseState() {}

// if you want to override OnEnter functions than you should define concrete functions

// with specific events that your state will handle

template <typename EventType>

void OnEnter(const EventType&) {}

// Is called for first state from SetStates

void OnEnter(void) {}

virtual void OnExit() {}

virtual void OnUpdate(OnUdateParam i\_elapsed\_time) {}

};

///////////////////////////////////////////////

Обычно в играх используется обновления с типом *float*, но в некоторых движках используется и *long*, поэтому в шаблоне добавим возможность управлять какой тип будет присылаться в метод *OnUpdate* (хотя это уже мои придирки и совсем не обязательно). Также не всегда будет нужно пользователю переопределять методы для всех типов событий либо вообще создавать оные, поэтому шаблонная реализация *OnEnter* будет подменять эти вызовы. Дополнительно нужен пустой *OnEnter(void)* для состояния, которое будет стоять первым в машине – переход в него не осуществляется, а оно устанавливается напрямую.

Теперь определимся с классом самой машины. На этапе компиляции мы уже знаем количество состояний, поэтому можем спокойно использовать старый и добрый массив, а вместо указателей на состояния использовать индексы в этом массиве:

///////////////////////////////////////////////

template <

size\_t StatesCount,

typename TransitionTable,

typename FirstStateType,

typename OnUpdateParam = float,

typename BaseStateType = BaseState<OnUpdateParam>,

class PtrType = std::unique\_ptr<BaseStateType>

>

class StateMachine

{

...

private:

PtrType m\_states[\_StatesCount];

size\_t m\_states\_hashes[\_StatesCount];

TransitionTable m\_transitions;

size\_t m\_current;

size\_t m\_next;

size\_t m\_prev;

...

};

///////////////////////////////////////////////

Вкратце о шаблонных параметрах:

1. StatesCount – количество состояний в машине
2. TransitionTable – то, в чём будут искаться переходы состояний. Теоретически, можно было бы выцеплять количество разных состояний из таблицы переходов, но это высший уровень чернокнижия.
3. FirstStateType – во время создания, состояние с данным типом выставится первым.
4. OnUpdateParam – то же, что и для базового состояния
5. BaseStateType – мало ли, захочется человеку свой собственный класс для работы с состояниями, а у нас такое можно.
6. PtrType – даёт возможность делать «владеющую» и «не владеющую» состояниями машину: если передавать сырой указатель – BaseStateType\*, – то объект после удаления машины останется жить дальше.

Ещё немножко методов интерфейса для машины: создания машины и состояний, обработка события, старт с состояния какого-либо типа (помимо установки первого состояния в аргументах шаблона, старт понадобится для других интересных целей, которые будут описаны ниже), форсированный переход в другое состояния без вызова обновления у нового состояния и метод обновления.

///////////////////////////////////////////////

class StateMachine

{

...

public:

StateMachine()

: m\_current(NullState)

, m\_next(NullNextState)

, m\_prev(NullState)

, m\_next\_executor(INVALID\_EXECUTOR\_INDEX)

{}

template <typename... Ptrs>

StateMachine(Ptrs... i\_states)

: StateMachine{}

{

SetStates(std::move(i\_states)...);

}

virtual ~StateMachine() {}

template <typename... Ptrs>

void SetStates(Ptrs... i\_states)

{

static\_assert(sizeof...(i\_states) == \_StatesCount, "Size of arguments must be same as size of states");

PtrType states[] = { std::move(i\_states)... };

for (size\_t i = 0; i < \_StatesCount; ++i)

{

m\_states[i] = std::move(states[i]);

m\_states\_hashes[i] = typeid(\*m\_states[i]).hash\_code();

}

m\_current = NullState;

m\_next = NullNextState;

m\_prev = NullState;

SetNext<\_FirstState>();

}

...

template <typename EventType>

void ProcessEvent(const EventType& i\_evt)

{

TransitionGetterResult result(0, INVALID\_EXECUTOR\_INDEX);

m\_transitions.GetNextState<EventType, \_ThisMachine>(result, i\_evt, \*this);

if (result.second != INVALID\_EXECUTOR\_INDEX)

SetNext(result);

}

template <typename State>

void Start()

{

const size\_t next\_state = typeid(State).hash\_code();

SetNext(std::make\_pair(

next\_state,

CacheVisit<State>()

));

}

template <typename State, typename Event>

void Start(const Event& i\_event)

{

static size\_t type\_to = typeid(State).hash\_code();

static void(State::\*enter\_func)(const Event&) = &State::OnEnter;

SetNext(std::make\_pair(

type\_to,

CacheVisit<State, Event>(i\_event)

));

}

void ForceChange()

{

ChangeStateIfNeeded();

}

void OnUpdate(OnUpdateParam i\_param)

{

ChangeStateIfNeeded();

if (m\_current != NullState)

m\_states[m\_current]->OnUpdate(i\_param);

}

...

};

///////////////////////////////////////////////

Теперь самая интересная часть – поиск нового состояния в таблице переходов, которая создаётся на этапе компиляции! И вкусная печенька от повара – контроль за тем, чтобы у классов были нужные перегруженные методы *OnEnter*.

Есть два варианта сделать карту переходов – через композицию и таблицей. Мы рассмотрим оба. Различие в скорости работы и в удобстве пользования: таблица удобней, но немного медленней и сложнее в реализации.

Структура, отвечающая за один рядок переходов из одного состояния (*StateFrom*) в другое (*StateTo*) по определённому событию (*TargetEventType*).

///////////////////////////////////////////////

typedef size\_t NextStateType;

typedef std::pair<NextStateType, int> TransitionGetterResult;

template <typename StateFrom, typename StateTo, typename TargetEventType>

struct Transition

{

using \_ThisType = Transition<StateFrom, StateTo, TargetEventType>;

template <typename EventType, typename StateMachine>

void GetNextState(TransitionGetterResult& o\_result, const EventType& i\_event, const StateMachine& i\_fsm) const

{

static size\_t caller\_type = typeid(\_ThisType).hash\_code();

RowChecker<StateFrom, StateTo, TargetEventType, EventType>::GetNextState(o\_result, i\_event, i\_fsm, caller\_type);

}

};

///////////////////////////////////////////////

*TransitionGetterResult* является парой, где в первом элементе хранится хэш типа, в который нужно перейти, а во втором – номер закешированного вызова *State::OnEnter* – об этом в подробностях ниже. Если переход нас не интересует, то в *pair.first* будет хранится хеш самого *Transition*.

Но это один переход, а как составить таблицу? Плох переход из которого нельзя вызвать другой переход!

///////////////////////////////////////////////

template <typename FirstTransition, typename SecondTransition>

struct CompoundTransition

{

FirstTransition first;

SecondTransition second;

template <typename EventType, typename StateMachine>

void GetNextState(TransitionGetterResult& o\_result, const EventType& i\_event, const StateMachine& i\_fsm)

{

static const size\_t first\_type = typeid(first).hash\_code();

static const size\_t second\_type = typeid(second).hash\_code();

static const size\_t this\_type = typeid(\*this).hash\_code();

first.GetNextState<EventType, StateMachine>(o\_result, i\_event, i\_fsm);

if (o\_result.first != first\_type)

return;

second.GetNextState<EventType, StateMachine>(o\_result, i\_event, i\_fsm);

if (o\_result.first != second\_type)

return;

o\_result.first = this\_type;

o\_result.second = StateMachine::INVALID\_EXECUTOR\_INDEX;

}

};

///////////////////////////////////////////////

CompoundTransition может состоять как из простых, так и из составных переходов, из чего и получается таблица. Чтобы не нагромождать имена определены помощники.

///////////////////////////////////////////////

template <typename FirstTrans, typename SecondTrans>

using \_comp\_tr = CompoundTransition<FirstTrans, SecondTrans>;

template <typename FrState, typename ToState, typename Event>

using \_tr =Transition<FrState, ToState, Event>;

///////////////////////////////////////////////

Два-три перехода смотрятся неплохо, но уже на четыре перехода глядеть неприятно (а ведь в реальности их будет больше):

///////////////////////////////////////////////

using TransitionTable = \_comp\_tr<

\_tr<MyState1, MyState2, Event22>,

\_comp\_tr<Transition<MyState2, MyState3, Event11>,

\_comp\_tr<Transition<MyState3, MyState4, Event11>, \_tr<MyState4, MyState1, Event11>> >

>;

///////////////////////////////////////////////

Добавление новых переходов становится с каждым новым всё более сложным и муторным, а ведь в бустовской машине можно удобно определять таблицей – да начнутся поиски!

Спойлер

--

На четырёх переходах *CompoundTrnasition* показал результат на 20% лучше, а на больше переходов делать его становилось уж очень неудобно.

--

Таблица переходов основывается на вариадических шаблонах и возможности вызова всех из них последовательно и все вызовы будут прописаны во время компиляции. Минус, естественно, что прервать вызовы, даже если нашли нужный переход, нельзя.

///////////////////////////////////////////////

template <typename StateFrom, typename StateTo, typename TargetEventType>

struct TransitionRow

{

using \_ThisType = TransitionRow<StateFrom, StateTo, TargetEventType>;

template <typename EventType, typename StateMachine>

static void GetNextState(TransitionGetterResult& o\_result, const EventType& i\_event, const StateMachine& i\_fsm)

{

static size\_t caller\_type = typeid(\_ThisType).hash\_code();

RowChecker<StateFrom, StateTo, TargetEventType, EventType>::GetNextState(o\_result, i\_event, i\_fsm, caller\_type);

}

};

template <typename... Transitions>

struct TransitionsTable

{

template <typename EventType, typename StateMachine, typename row>

bool CheckTransition(TransitionGetterResult& o\_result, const EventType& i\_event, const StateMachine& i\_fsm)

{

if (o\_result.second != StateMachine::INVALID\_EXECUTOR\_INDEX)

return false;

row::GetNextState<EventType, StateMachine>(o\_result, i\_event, i\_fsm);

return true;

}

template <typename EventType, typename StateMachine>

void GetNextState(TransitionGetterResult& o\_result, const EventType& i\_event, const StateMachine& i\_fsm)

{

static size\_t this\_index = typeid(\*this).hash\_code();

o\_result.first = this\_index;

bool tr\_results[] = { CheckTransition<EventType, StateMachine, Transitions>(o\_result, i\_event, i\_fsm)... };

}

};

//////////////////////////////////////////////

Теперь пришла пора посмотреть под капот *RowChecker*-a. Есть два варианта вызова – переход нам интересен и переход нам не интересен. Первый вариант происходит, когда текущий тип события и нужный тип события совпадают, второй вариант, соответственно, с любым другим типом. На помощь приходит *std::enable\_if*.

//////////////////////////////////////////////

template <typename StateFrom, typename StateTo, typename TargetEventType, typename Event, typename Enable = void>

struct RowChecker {};

template <typename StateFrom, typename StateTo, typename TargetEventType, typename Event>

struct RowChecker <StateFrom, StateTo, TargetEventType, Event, typename std::enable\_if<std::is\_same<TargetEventType, Event>::value>::type>

{

template <typename StateMachine>

static void GetNextState(TransitionGetterResult& o\_result, const TargetEventType& i\_event, const StateMachine& i\_fsm, const size\_t i\_caller\_type)

{

static size\_t type\_to = typeid(StateTo).hash\_code();

static size\_t type\_from = typeid(StateFrom).hash\_code();

static void(StateTo::\*enter\_func)(const TargetEventType&) = &StateTo::OnEnter;

if (i\_fsm.IsStateCurrent(type\_from))

{

o\_result.first = type\_to;

o\_result.second = i\_fsm.CacheVisit<StateTo, TargetEventType>(i\_event);

}

}

};

template <typename StateFrom, typename StateTo, typename TargetEventType, typename Event>

struct RowChecker <StateFrom, StateTo, TargetEventType, Event, typename std::enable\_if<!std::is\_same<TargetEventType, Event>::value>::type>

{

template <typename StateMachine>

static void GetNextState(TransitionGetterResult& o\_result, const Event& i\_event, const StateMachine& i\_fsm, const size\_t i\_caller\_type)

{

}

};

//////////////////////////////////////////////

В случае неинтересного перехода нам ничего делать не нужно, в случае нужного – устанавливаем соответствующие значения в результат:

1. Кеш типа состояния, в которое нужно перейти
2. Номер закешированного вызова (точнее говоря посетителя, который зайдёт в нужный метод)

Таблица переходов уже выглядит куда симпатичнее:

//////////////////////////////////////////////

using TestTr = TransitionsTable<

\_row<MyState1, MyState2, Event22>,

\_row<MyState2, MyState3, Event11>,

\_row<MyState3, MyState4, Event11>,

\_row<MyState4, MyState1, Event11>

>;

//////////////////////////////////////////////

Теперь вернёмся к машине состояний и работы в ней – установка следующего состояния, смена состояния и кеширование вызова.

//////////////////////////////////////////////

template < ...>

class StateMachine

{

constexpr static size\_t NullState = \_StatesCount;

constexpr static size\_t NullNextState = \_StatesCount + 1;

private:

mutable details::Visitors m\_visitors;

int m\_next\_executor;

void ChangeStateIfNeeded()

{

if (m\_next == NullNextState)

return;

if (m\_current != NullState)

m\_states[m\_current]->OnExit();

m\_prev = m\_current;

m\_current = m\_next;

if (m\_current != NullState && m\_next\_executor != INVALID\_EXECUTOR\_INDEX)

{

m\_visitors[m\_next\_executor]->Execute();

m\_next\_executor = INVALID\_EXECUTOR\_INDEX;

}

m\_next = NullNextState;

}

template <typename State>

void SetNext()

{

const size\_t next\_state = typeid(State).hash\_code();

SetNext(std::make\_pair(

next\_state,

CacheVisit<State>()

));

}

void SetNext(TransitionGetterResult i\_result)

{

m\_next = NullNextState;

m\_next\_executor = INVALID\_EXECUTOR\_INDEX;

const size\_t next\_state = i\_result.first;

for (size\_t i = 0; i < \_StatesCount; ++i)

{

const auto& state = m\_states[i];

if (next\_state == m\_states\_hashes[i])

{

m\_next = i;

m\_next\_executor = i\_result.second;

break;

}

}

}

};

//////////////////////////////////////////////

В SetNext всего лишь ищется нужный тип и выставляется индекс следующего состояния, если он не NullNextState, то нужно совершить переход. Сам переход – смена индексов *m\_current, m\_prev* и *m\_next*. Куда интереснее как сделать вызов нужного OnEnter-a. Для этого использован шаблон проектирования «посетитель». Создаётся набор таких посетителей для каждого класса-события. Так как они шаблонные, то знают какой метод вызвать и через них же на этапе компиляции проверяется, что этот метод присутствует в классе (при чём именно в том, в котором он нужен – к которому идёт переход по этому событию). Детали класса под спойлером.

Спойлер:

//////////////////////////////////////////////

struct Visitor

{

size\_t m\_state\_hash;

size\_t m\_ev\_hash;

Visitor(size\_t i\_state, size\_t i\_ev\_hash)

: m\_state\_hash(i\_state)

, m\_ev\_hash(i\_ev\_hash)

{}

inline bool Compare(size\_t i\_state, size\_t i\_event) const

{

return m\_state\_hash == i\_state && m\_ev\_hash == i\_event;

}

virtual void Execute() = 0;

};

template <typename State, typename Ev>

struct VisitorImpl : Visitor

{

State& m\_state;

Ev m\_cached\_ev;

VisitorImpl(size\_t i\_state\_hash, size\_t i\_ev\_hash, State& i\_state, const Ev& i\_event)

: Visitor(i\_state\_hash, i\_ev\_hash)

, m\_state(i\_state)

, m\_cached\_ev(i\_event)

{}

virtual void Execute() override

{

m\_state.OnEnter(m\_cached\_ev);

}

};

template <typename State>

struct VisitorVoidImpl : Visitor

{

State& m\_state;

VisitorVoidImpl(size\_t i\_state\_hash, State& i\_state)

: Visitor(i\_state\_hash, 0)

, m\_state(i\_state)

{}

virtual void Execute() override

{

m\_state.OnEnter();

}

};

//////////////////////////////////////////////

Обратите внимание на использование конструкций

/////////////////////////////////////////////

statuc const size\_t type\_hash = typeid(StateFrom).hash\_code();

////////////////////////////////////////////

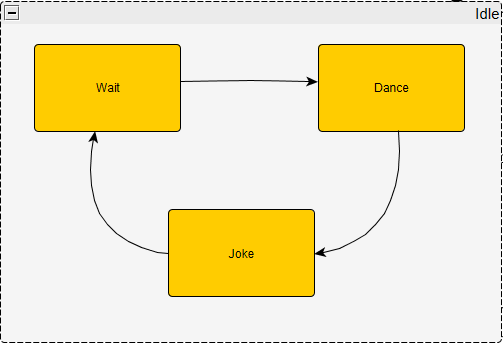
В профайлере typeid съедал большую часть времени, а так как С++ нам гарантирует две вещи:

1. *hash\_code* только одного и того же типа будут равны
2. Для каждого типа будет сгенерирован свой метод

То можно использовать данную статическую переменную для оптимизации работы, чтобы не искать при каждом вызове RTTI.

Теперь сконструируем такую же машину, как и в первой части.

Спойлер:



//////////////////////////////////////////////

struct idle\_action\_completed {};

struct Wait;

struct Dance : public BaseState<>

{

std::function<void(const idle\_action\_completed&)> m\_process\_func;

template <typename StateMachine>

Dance(StateMachine& i\_fsm)

{

m\_process\_func = BindProcessEventFunction<idle\_action\_completed, StateMachine>(i\_fsm);

}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Dance] OnUpdate" << std::endl;

m\_process\_func(idle\_action\_completed());

}

};

template <typename StateMachine>

struct Joke : public BaseState<>

{

StateMachine& m\_parent\_fsm;

Joke(StateMachine& i\_fsm) : m\_parent\_fsm(i\_fsm) {}

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Joke] OnUpdate" << std::endl;

m\_parent\_fsm.ProcessEvent(idle\_action\_completed());

}

void OnEnter(const stop& i\_evt)

{

std::cout << "\t[Joke] Position achieved - hah" << std::endl;

}

void OnEnter(const idle\_action\_completed& i\_evt)

{

}

};

struct Idle;

using Tr = TransitionsTable<

\_row<Wait, Dance, idle\_action\_completed>,

\_row<Dance, Joke<Idle>, idle\_action\_completed>,

\_row<Joke<Idle>, Wait, idle\_action\_completed>

>;

struct Idle : public StateMachine<3, Tr, Wait>

{

Idle();

void OnEnter()

{ }

void OnEnter(const position\_achieved&)

{ }

void OnEnter(const stop&)

{ }

void OnEnter(const stop\_attacking&)

{ }

};

struct Wait : public BaseState<>

{

Idle& m\_fsm;

Wait(Idle& i\_fsm)

: m\_fsm(i\_fsm)

{ }

virtual void OnUpdate(float dt) override

{

std::cout << "\t[Wait] OnUpdate" << std::endl;

m\_fsm.ProcessEvent(idle\_action\_completed());

}

};

Idle::Idle()

{

SetStates(

std::make\_unique<Wait>(\*this),

std::make\_unique<Dance>(\*this),

std::make\_unique<Joke<Idle>>(\*this)

);

}

void TestIdle()

{

Idle idle\_fsm;

// or we can start with event

// idle\_fsm.Start<Wait>(idle\_action\_completed());

for (int i = 0; i < 10; ++i)

idle\_fsm.OnUpdate(.1f);

}

//////////////////////////////////////////////

Первое же неудобство – нельзя передать указатель на машину состояний, так как её тип зависит от типов состояний и получается коллизия. Данный вопрос можно решить несколькими способами:

1. Сначала сделать объявление машины состояний – class Idle; - а в самих состояниях принимать в конструкторе. Минус в том, что нужно правильно расположить определение классов. Используется с состоянием *Wait*.
2. Использовать шаблонный конструктор, который будет принимать ссылку на любую машину состояний и создавать *std::function* для вызова *ProcessEvent* – основное, что требуется делать с родительской машиной – состояние *Dance*.

Для удобства определён помощник в этом деле

//////////////////////////////////////////////////

template <typename Event, typename StateMachine>

std::function<void(const Event&)> BindProcessEventFunction(StateMachine& i\_fsm)

{

static void(StateMachine::\*process\_func)(const Event&) = &StateMachine::ProcessEvent;

return std::bind(process\_func, std::ref(i\_fsm), std::placeholders::\_1);

}

//////////////////////////////////////////////////

1. Сделать шаблонное состояние, параметром которого будет тип машины состояний – *Joke*.

В реальном проекте удобнее всего было использовать первый метод, так как поощряет правильный разнос классов по файлам и поддерживает структуру расположения классов в коде, хотя и встречаются кое-где и остальные два.

Второе неудобство – нельзя сделать функцию, которая вернёт объект машины состояний, а конструировать нужно всё в конструкторе. Спорное неудобство, но это создаёт дополнительные связи. Плюс нужно наследоваться от базового класса StateMachine, хотя нужен только конструктор.

Код проверки почти такой же как и в случае ООП машины. Единственное отличие – мы уже знаем какое состояние будет первым и вызов метода Start можно опустить.

**Ещё нужно описать CompoundState и StateMachineWrapper.**

**Выводы:**

Получили машину, в которой:

* существует проверка типов на этапе компиляции
* для каждого типа события свой метод
* работает она быстрее (примерно раз в 10, но описано это будет в отдельной части, где будет ещё сравнение с boost::msm)

Начальные требования мы выполнили, но и минусы тоже присутствуют (неудобства в работе с машиной из состояний).

Часть 3. Сравнение. Тёмная и светлая сторона С++.