C++ **Mach**ine **O**bjects (MACHO)

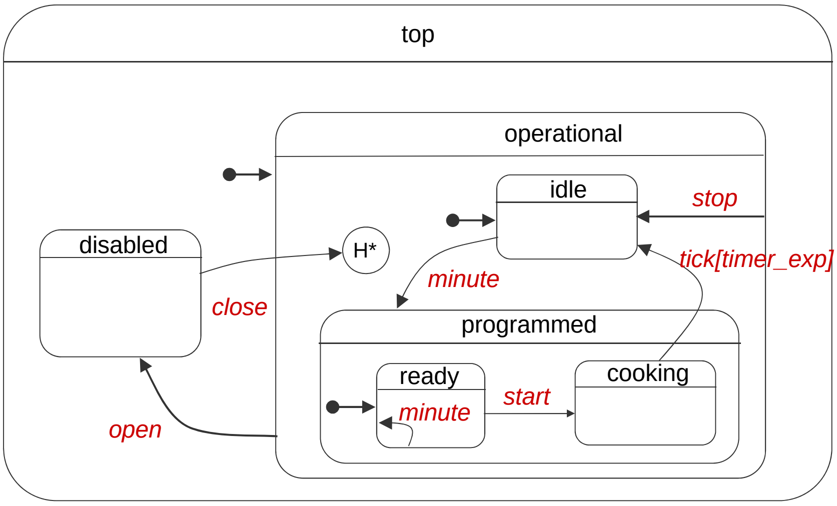
Macho

The *Machine Objects* class library allows the creation of state machines based on the "State" design pattern in C++.

It extends the pattern with the option to create **hierarchical state machines**, entry and exit actions, state histories and state variables.

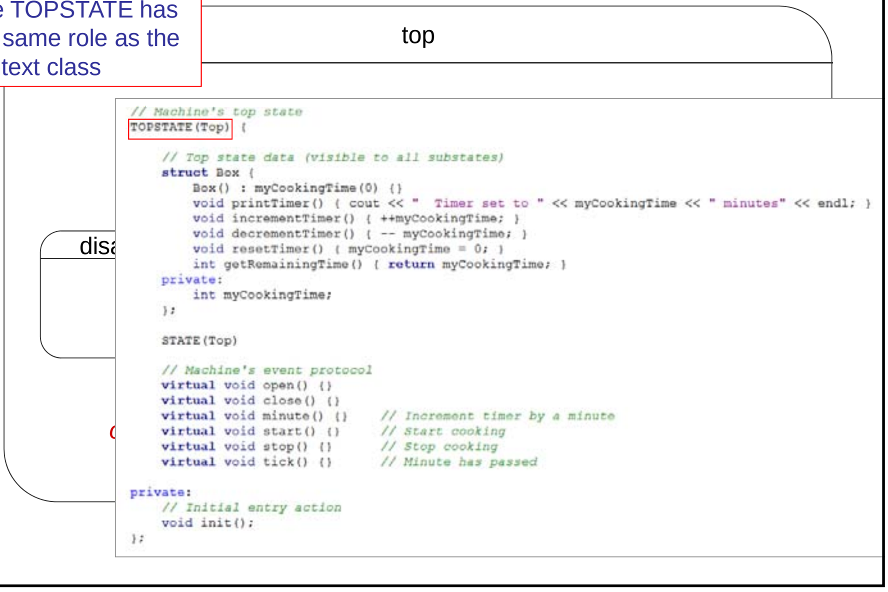
Freely available at
http://ehiti.de/machine_objects/

A Macho Hello World



A Macho Hello World

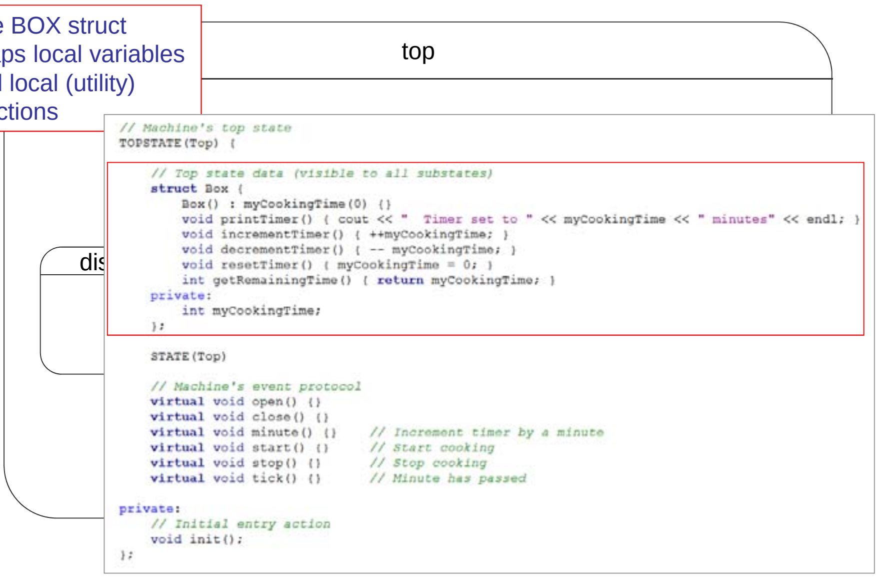
The TOPSTATE has the same role as the context class



```
// Machine's top state
TOPSTATE(Top) {

    // Top state data (visible to all substates)
    struct Box {
        Box() : myCookingTime(0) {}
        void printTimer() { cout << "  Timer set to " << myCookingTime << " minutes" << endl; }
        void incrementTimer() { ++myCookingTime; }
        void decrementTimer() { -- myCookingTime; }
        void resetTimer() { myCookingTime = 0; }
        int getRemainingTime() { return myCookingTime; }
    private:
        int myCookingTime;
    };

    STATE(Top)

    // Machine's event protocol
    virtual void open() {}
    virtual void close() {}
    virtual void minute() {}    // Increment timer by a minute
    virtual void start() {}     // Start cooking
    virtual void stop() {}      // Stop cooking
    virtual void tick() {}      // Minute has passed

private:
    // Initial entry action
    void init();
};
```

A Macho Hello World

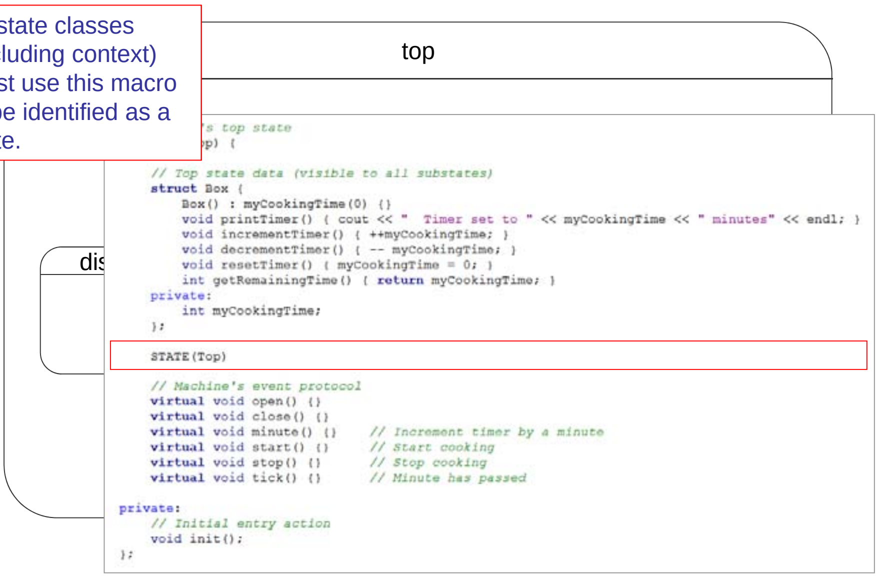
The BOX struct wraps local variables and local (utility) functions



```
// Machine's top state
TOPSTATE(Top) {

    // Top state data (visible to all substates)
    struct Box {
        Box() : myCookingTime(0) {}
        void printTimer() { cout << "  Timer set to " << myCookingTime << " minutes" << endl; }
        void incrementTimer() { ++myCookingTime; }
        void decrementTimer() { -- myCookingTime; }
        void resetTimer() { myCookingTime = 0; }
        int getRemainingTime() { return myCookingTime; }
    private:
        int myCookingTime;
    };

    STATE(Top)

    // Machine's event protocol
    virtual void open() {}
    virtual void close() {}
    virtual void minute() {}    // Increment timer by a minute
    virtual void start() {}     // Start cooking
    virtual void stop() {}      // Stop cooking
    virtual void tick() {}      // Minute has passed

private:
    // Initial entry action
    void init();
};
```

A Macho Hello World

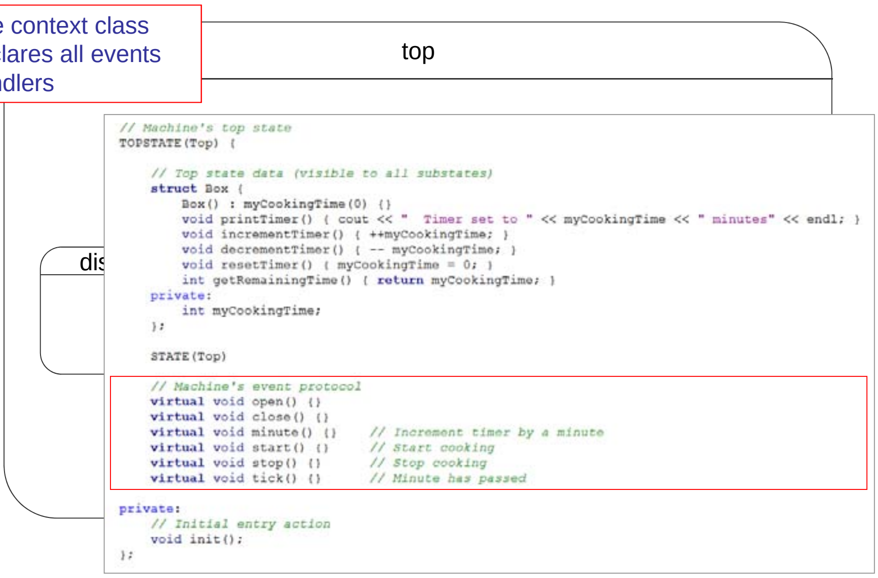
All state classes (including context) must use this macro to be identified as a state.



```
's top state
p) {

    // Top state data (visible to all substates)
    struct Box {
        Box() : myCookingTime(0) {}
        void printTimer() { cout << "  Timer set to " << myCookingTime << " minutes" << endl; }
        void incrementTimer() { ++myCookingTime; }
        void decrementTimer() { -- myCookingTime; }
        void resetTimer() { myCookingTime = 0; }
        int getRemainingTime() { return myCookingTime; }
    private:
        int myCookingTime;
    };

    STATE(Top)

    // Machine's event protocol
    virtual void open() {}
    virtual void close() {}
    virtual void minute() {}    // Increment timer by a minute
    virtual void start() {}     // Start cooking
    virtual void stop() {}      // Stop cooking
    virtual void tick() {}      // Minute has passed

private:
    // Initial entry action
    void init();
};
```

A Macho Hello World

The context class declares all events handlers

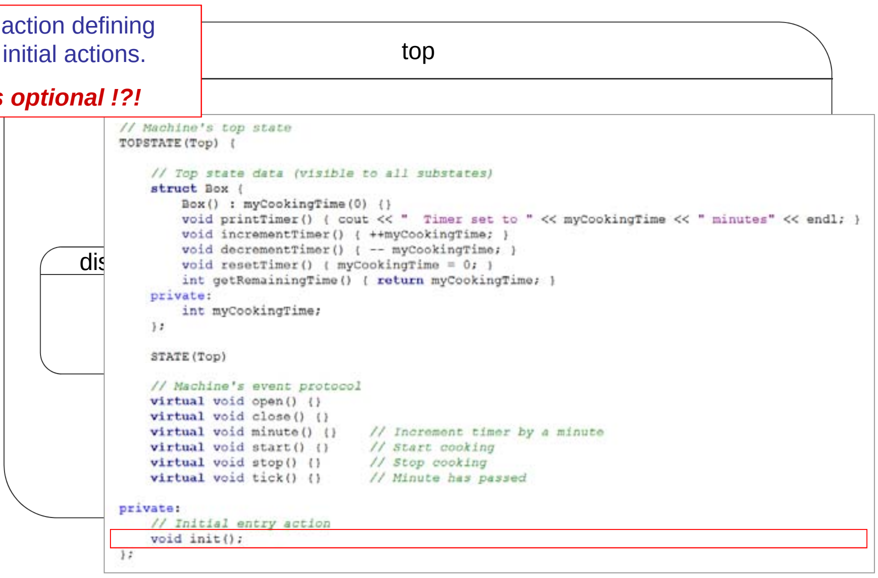


```
// Machine's top state
TOPSTATE(Top) {

    // Top state data (visible to all substates)
    struct Box {
        Box() : myCookingTime(0) {}
        void printTimer() { cout << "  Timer set to " << myCookingTime << " minutes" << endl; }
        void incrementTimer() { ++myCookingTime; }
        void decrementTimer() { -- myCookingTime; }
        void resetTimer() { myCookingTime = 0; }
        int getRemainingTime() { return myCookingTime; }
    private:
        int myCookingTime;
    };

    STATE(Top)

    // Machine's event protocol
    virtual void open() {}
    virtual void close() {}
    virtual void minute() {}    // Increment timer by a minute
    virtual void start() {}     // Start cooking
    virtual void stop() {}      // Stop cooking
    virtual void tick() {}      // Minute has passed

private:
    // Initial entry action
    void init();
};
```

A Macho Hello World

Init action defining the initial actions.

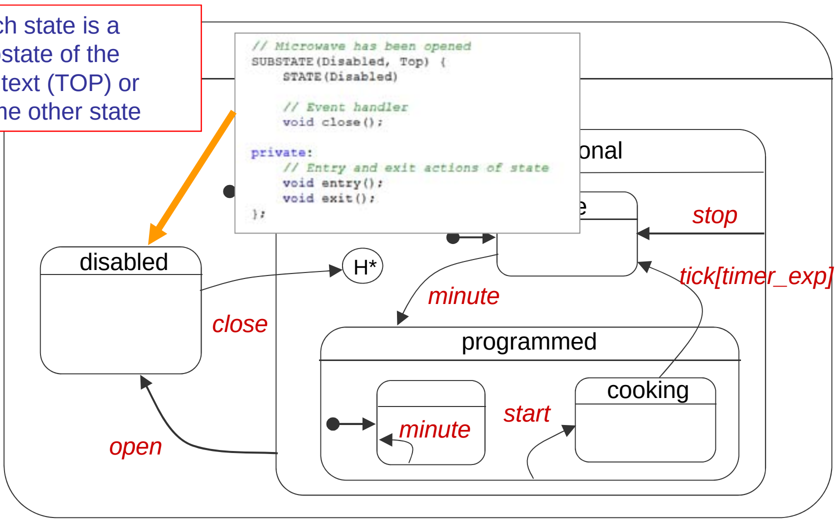
It is optional !?!



```
// Machine's top state
TOPSTATE(Top) {

    // Top state data (visible to all substates)
    struct Box {
        Box() : myCookingTime(0) {}
        void printTimer() { cout << "  Timer set to " << myCookingTime << " minutes" << endl; }
        void incrementTimer() { ++myCookingTime; }
        void decrementTimer() { -- myCookingTime; }
        void resetTimer() { myCookingTime = 0; }
        int getRemainingTime() { return myCookingTime; }
    private:
        int myCookingTime;
    };

    STATE(Top)

    // Machine's event protocol
    virtual void open() {}
    virtual void close() {}
    virtual void minute() {}    // Increment timer by a minute
    virtual void start() {}     // Start cooking
    virtual void stop() {}      // Stop cooking
    virtual void tick() {}      // Minute has passed

private:
    // Initial entry action
    void init();
};
```

A Macho Hello World

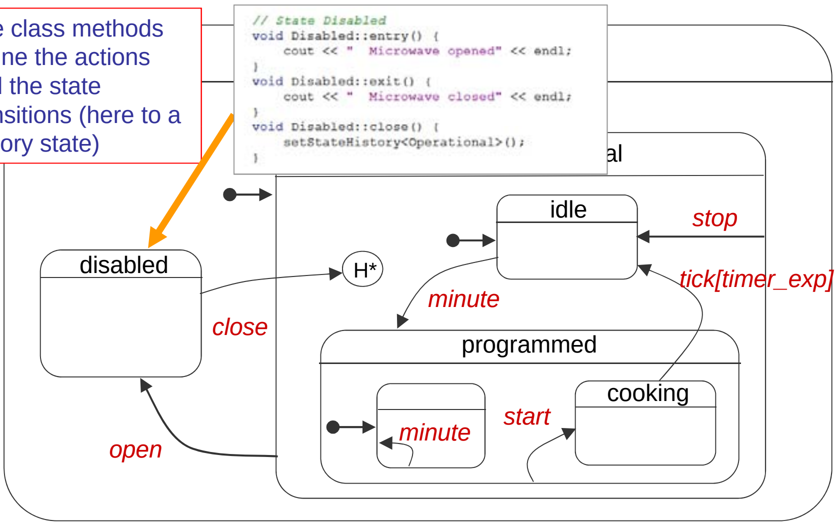
Each state is a substate of the context (TOP) or some other state



```
// Microwave has been opened
SUBSTATE(Disabled, Top) {
    STATE(Disabled)

    // Event handler
    void close();

private:
    // Entry and exit actions of state
    void entry();
    void exit();
};
```

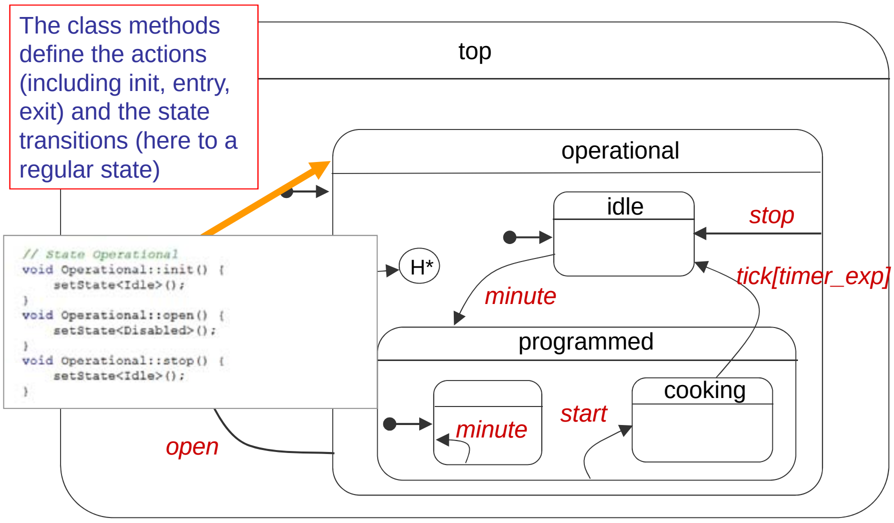
A Macho Hello World

The class methods define the actions and the state transitions (here to a history state)



```
// State Disabled
void Disabled::entry() {
    cout << "  Microwave opened" << endl;
}
void Disabled::exit() {
    cout << "  Microwave closed" << endl;
}
void Disabled::close() {
    setStateHistory<Operational>();
}
```

A Macho Hello World



The class methods define the actions (including init, entry, exit) and the state transitions (here to a regular state)

```
// State Operational
void Operational::init() {
    setState<Idle>();
}
void Operational::open() {
    setState<Disabled>();
}
void Operational::stop() {
    setState<Idle>();
}
```

What is needed

The class library as such does not need to be installed. Just include the header file Macho.hpp and add the file Macho.cpp to your make file or project definition.

Prerequisite is a C++ compiler with support for templates.

```
# GCC
g++ -o microwave Microwave.cpp Macho.cpp
# MSVC7
cl /EHsc Microwave.cpp Macho.cpp
```

Representing states

- The starting point is the "State" design pattern.
- The essence of the pattern is to represent states by classes. State transitions are performed by instantiating objects of these classes.
- *In contrast to the pattern discussed in the previous lesson, where all state classes are instantiatied statically as part of the context class.*
- From this perspective the constructors and destructors of state classes can take the role of entry and exit actions.
- Object attributes represent state variables.
- Events are dispatched by calling methods on state objects which implement the guards, actions and transitions of states.

Representing hierarchy

- Substates must be able to take over the event handling logic of superstates, redefining it where necessary. There exists a mechanism in C++ allowing redefinition of behaviour on the level of methods: polymorphism through class inheritance.

Representing hierarchy

- However, modeling the substate/superstate relation with class inheritance is problematic:
  - the use of constructors and destructors as entry and exit actions is not possible anymore,
  - neither is keeping state variables in objects.
- The reason is that base classes are constituent parts of deriving classes, meaning object construction or destruction will trigger all involved class constructors or destructors and initialize or destroy all data members.
- This may be against the semantics of entry/exit actions and state variables, where a transition between sibling substates should not trigger superstate entry/exit actions nor destroy superstate state variables.

Representing hierarchy

- The solution is to use explicit methods for state entry and exit, being called in the correct sequence on state transitions.
  - The framework takes care of calling them in the right order
- State variables are kept in separate state specific data structures which have a life cycle consistent with the hierarchy of states.

State Definition

- Macho machines are embedded within a single Top state
- The top state's interface defines the machine's event protocol: only the public virtual methods of the top state can be event handlers.
- The top state is defined by the macro **TOPSTATE**

```
TOPSTATE(Top) {
    ...
};
```

- The top state has a typedef alias **TOP** that is available to all states of the machine.
- All states are in reality substates (of top state or some other states)!

State Definition

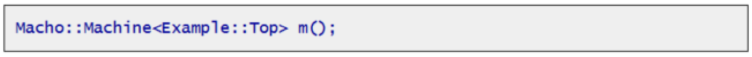
- Regular states and superstates are defined using the SUBSTATE macro

```
SUBSTATE(Super, Top) {
    ...
};
```

- The macro parameters are the substate's name and the name of its superstate.
- A typedef alias SUPER will point to the superstate within the substate class.

Machine creation

- A state machine is created by Instantiating a machine object

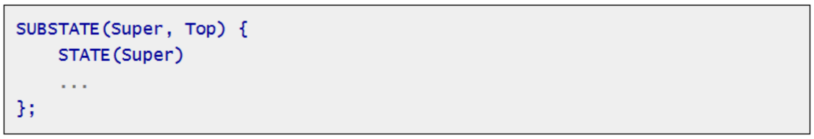


```
Macho::Machine<Example::Top> m();
```

- Machine is a template class in the MACHO namespace. Its template parameter is the top state of the machine.
- The top state is immediately entered and initialized upon machine creation

State Definition

- The macro STATE(state_name) **must** be invoked in every state body:

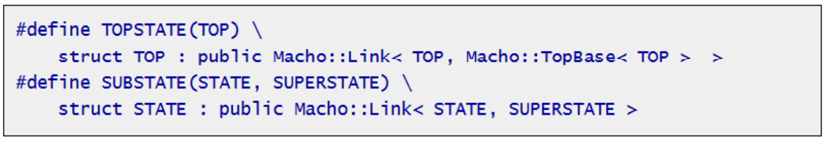


```
SUBSTATE(Super, Top) {
    STATE(Super)
    ...
};
```

- The macro parameter is the state name. The purpose of the macro is to automatically generate a few definitions (a constructor for instance).
- Every state class must be instantiable (including the top state). This means states must not have pure virtual methods!

Macro internals

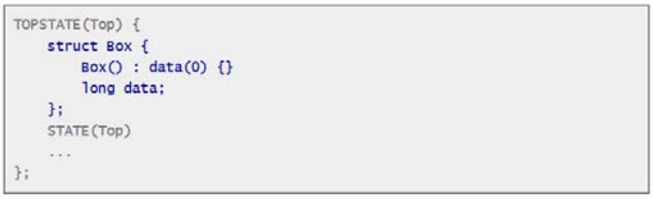
- Macros TOPSTATE, SUBSTATE, STATE



```
#define TOPSTATE(TOP) \
    struct TOP : public Macho::Link< TOP, Macho::TopBase< TOP >  >
#define SUBSTATE(STATE, SUPERSTATE) \
    struct STATE : public Macho::Link< STATE, SUPERSTATE >
```

State Variables

- State variables store information maintained in the scope of the associated state and accessible to any of its substates.
- Top state variables are accessible to all states of a machine.

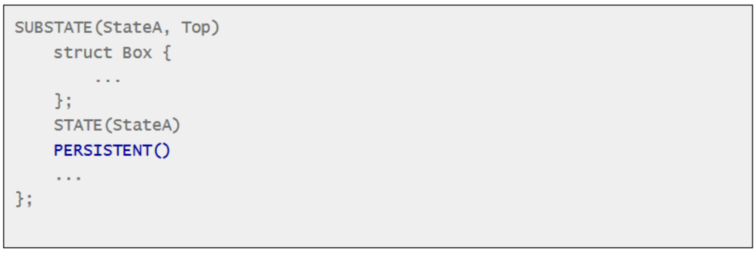


```
TOPSTATE(Top) {
    struct Box {
        Box() : data(0) {}
        long data;
    };
    STATE(Top)
    ...
};
```

- State variables are contained in a data type named Box (the name Box **<u>is mandatory</u>**) nested into the state class.
- The box definition **<u>must</u>** be **before** the use of the STATE macro.
- The box type must be default constructable (has a default constructor).
  - Apart from this the box type can be any C++ type. (even a typedef)

States with persistent data

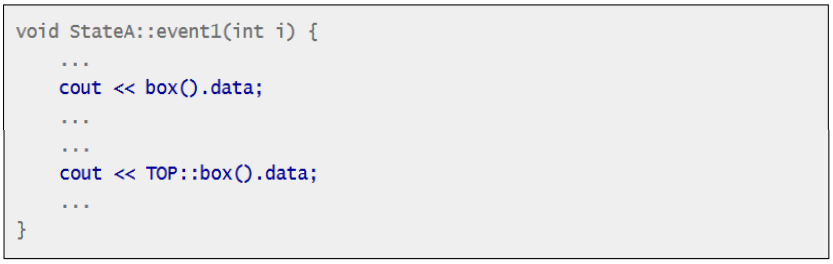
- A box is created before its state is entered (before the call to entry) and by default destroyed after the state is left (after the call to exit). By marking a state class with the PERSISTENT macro, you can override this default behaviour and have boxes survive state transitions:
- Persistent boxes are created once at first entry of their state and exist for as long as the state machine instance itself exists.



```
SUBSTATE(StateA, Top)
    struct Box {
        ...
    };
    STATE(StateA)
    PERSISTENT()
    ...
};
```

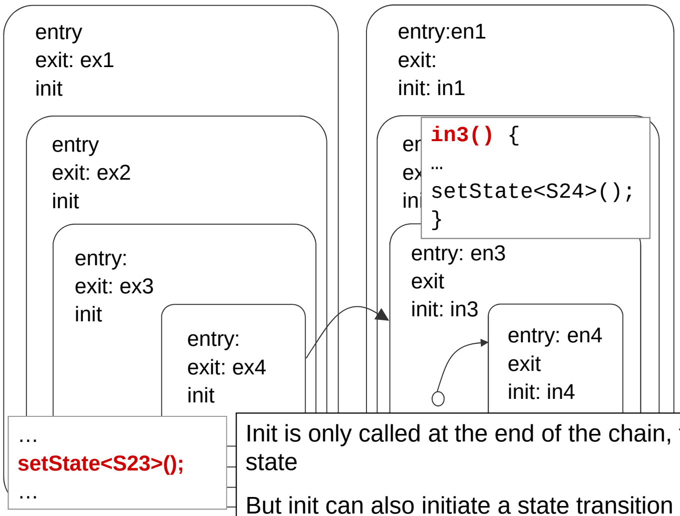
Accessing the state data

- A state's box is accessed by calling the method box, which returns a reference to the state's box object:
- Superstate boxes are available by qualifying the box method with the superstate's name:



```
void StateA::event1(int i) {
    ...
    cout << box().data;
    ...
    ...
    cout << TOP::box().data;
    ...
}
```

Entry, exit, init



entry
exit: ex1
init

entry
exit: ex2
init

entry:
exit: ex3
init

entry:
exit: ex4
init

…
setState<S23>();
…

entry:en1
exit:
init: in1

er
ex
in

```
in3() {
…
setState<S24>();
}
```

entry: en3
exit
init: in3

entry: en4
exit
init: in4

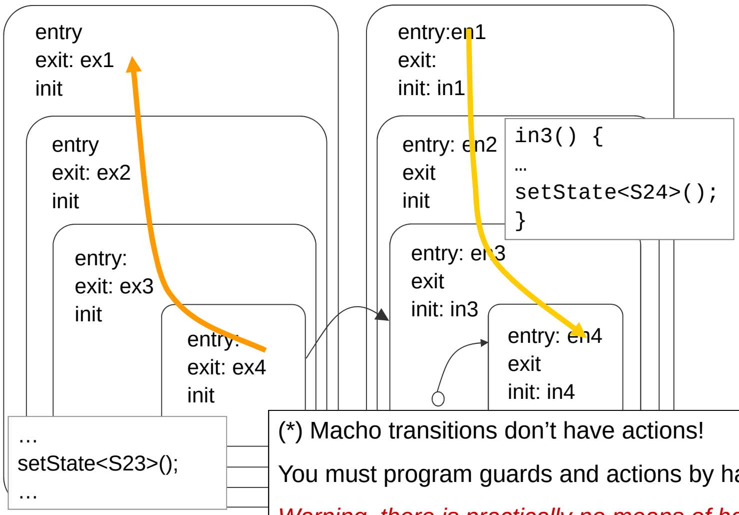
Sequence of calls:

```
ex4();
ex3();
ex2();
ex1();
(*)
en1();
en2();
en3();
in3();
en4();
in4();
```

Init is only called at the end of the chain, for the destination state

But init can also initiate a state transition to a substate (entry and exit cannot!), triggering another entry and another init.

Entry, exit, init



entry
exit: ex1
init

entry
exit: ex2
init

entry:
exit: ex3
init

entry:
exit: ex4
init

…
setState<S23>();
…

entry:en1
exit:
init: in1

entry: en2
exit
init

```
in3() {
…
setState<S24>();
}
```

entry: en3
exit
init: in3

entry: en4
exit
init: in4

Sequence of calls:

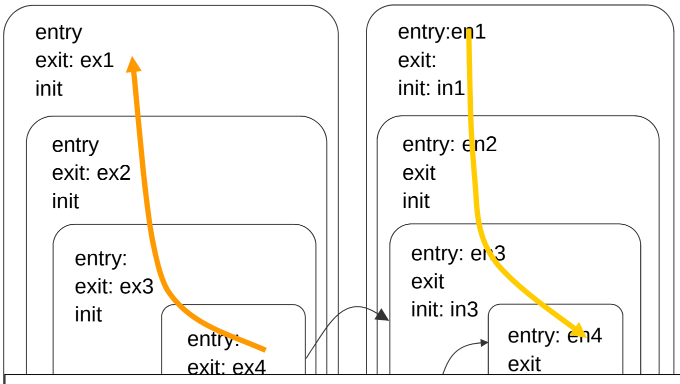
```
ex4();
ex3();
ex2();
ex1();
(*)
en1();
en2();
en3();
in3();
en4();
in4();
```

(*) Macho transitions don't have actions!

You must program guards and actions by hand

Warning, there is practically no means of having the action between exit and entry

Entry, exit, init: calling order



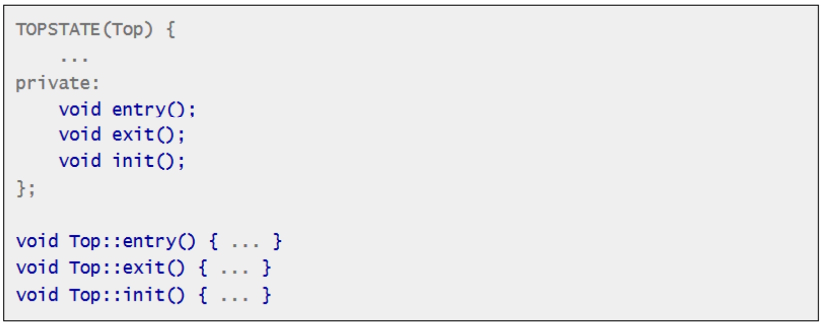
Sequence of calls:

```
ex4();
ex3();
ex2();
ex1();
(*)
en1();
en2();
en3();
in3();
en4();
in4();
```

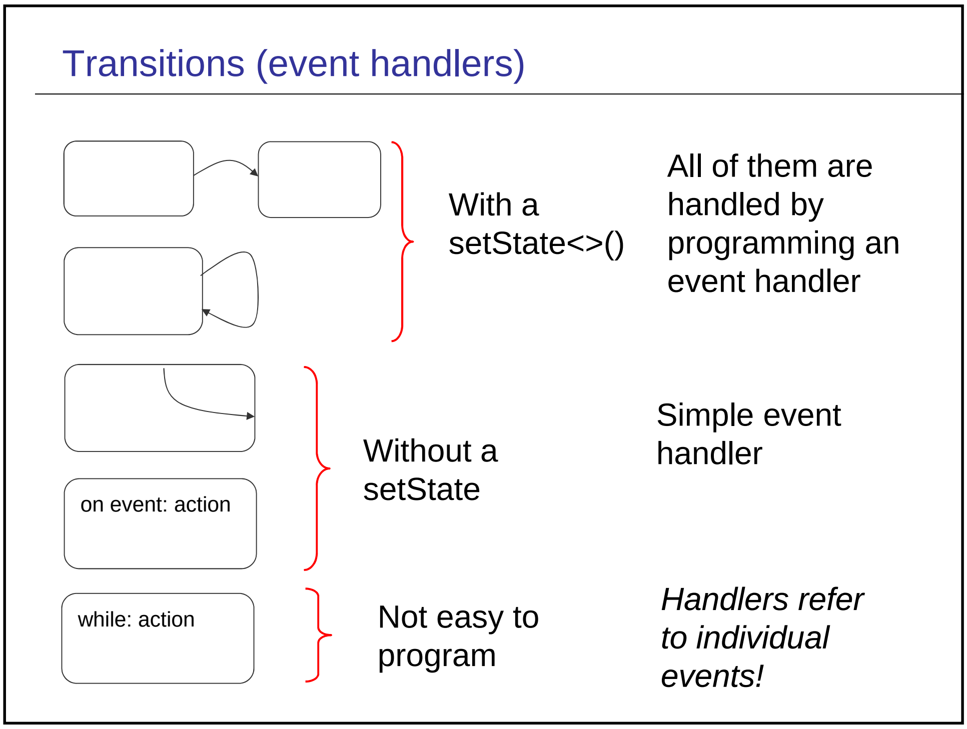
The methods entry and exit of a state are called upon transitioning into or out of it.

- First the exit action of the current state is called and then those of its superstates (in bottom up order),
- Then entry actions of superstates of the new state are called, top down from the first superstate that is not also a superstate of the previous state.

Entry, exit, init: use



```
TOPSTATE(Top) {
    ...
private:
    void entry();
    void exit();
    void init();
};

void Top::entry() { ... }
void Top::exit() { ... }
void Top::init() { ... }
```



Transitions (event handlers)

With a setState<>()

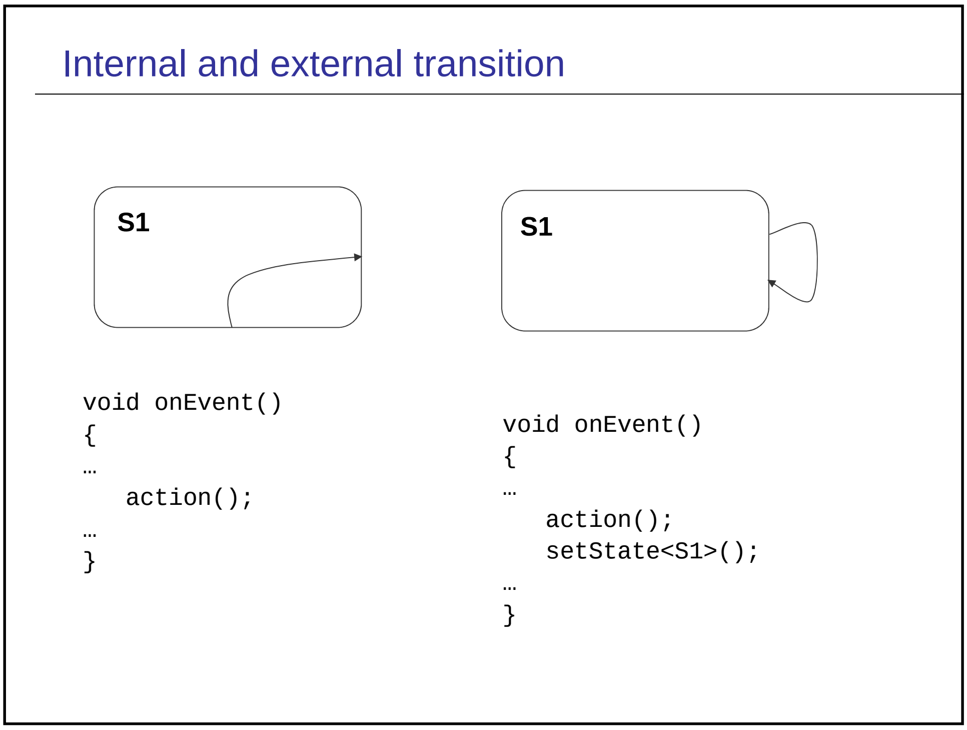
All of them are handled by programming an event handler

Without a setState

Simple event handler

Not easy to program

Handlers refer to individual events!



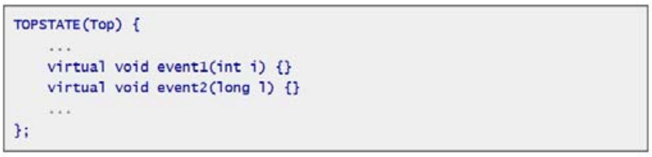
Internal and external transition

```
void onEvent()
{
…
   action();
…
}
```

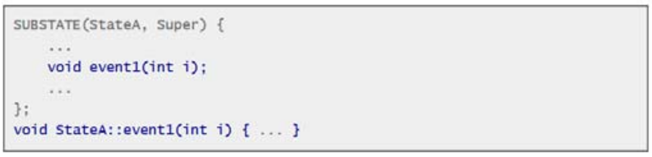
```
void onEvent()
{
…
   action();
   setState<S1>();
…
}
```

Event handlers

- The set of events handled by a state machine is simply defined defined by its top state public interface
  - Events are called like their even handler methods
  - No name conventions enforced (but a reasonable choice to call these functions event_XXX())
- Event handlers are simple C++ methods and may have arbitrary parameters and a return value



```
TOPSTATE(Top) {
    ...
    virtual void event1(int i) {}
    virtual void event2(long l) {}
    ...
};
```



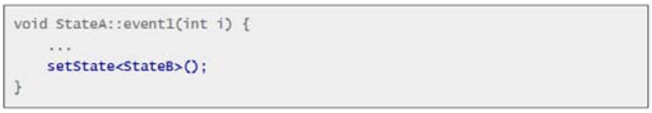
```
SUBSTATE(StateA, Super) {
    ...
    void event1(int i);
    ...
};
void StateA::event1(int i) { ... }
```

Event handlers

- The top state event handlers define the default behaviour for the whole state machine.
- If there is no meaningful implementation for an event handler at top level, the handler could either
  - be implemented empty: no reaction to event is then default.
  - signal some error (for example with assert(false)): not handling the event will be a runtime error.

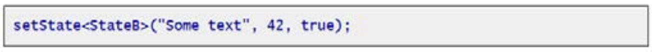
State transitions

- State transitions are made by calling the method setState
- The template parameter to setState is the new state.
- The transition takes place AFTER the control flow leaves the event handler. Functions can be executed after calling setState.
- It is not allowed to call setState multiple times with different states in a single event handler run

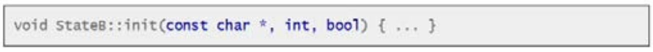


```
void StateA::event1(int i) {
    ...
    setState<StateB>();
}
```

- setState can take up to six parameters of arbitrary type:
- The arguments provided to setState are used to invoke an init method of the target state with matching signature:



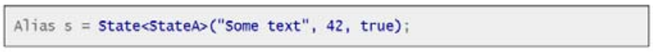
```
setState<StateB>("Some text", 42, true);
```



```
void StateB::init(const char *, int, bool) { ... }
```

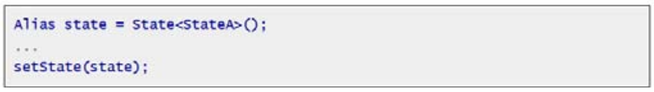
State alias objects

- When a state is entered, an object is instantiated for it.
- There is another option for creating a state object, through an object of Alias type
  - Possibly with initialization arguments



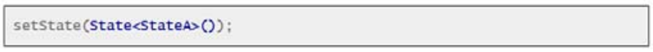
```
Alias s = State<StateA>("Some text", 42, true);
```

- Alias objects can be stored, reused and passed as parameters for a state transition

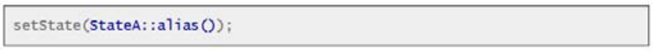


```
Alias state = State<StateA>();
...
setState(state);
```

- … or instantiated at transition time



```
setState(State<StateA>());
```

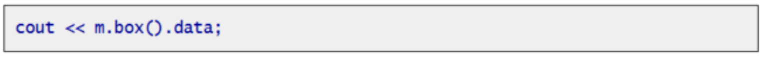


```
setState(StateA::alias());
```

Reflection (static queries)

- The FSM instance can be obtained by using the method machine().

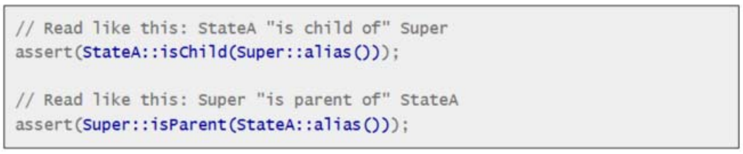
- The top state box can be obtained by calling the box method of Machine



```
cout << m.box().data;
```

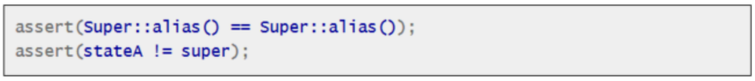
Reflection (static queries)

- The structure of the FSM (state relationships) can be detected at runtime.
- For example, it is possible to check if one state is parent/child of another



```
// Read like this: StateA "is child of" Super
assert(StateA::isChild(Super::alias()));

// Read like this: Super "is parent of" StateA
assert(Super::isParent(StateA::alias()));
```

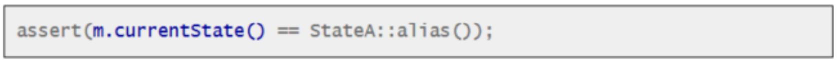
- Or to check for state equality



```
assert(Super::alias() == Super::alias());
assert(stateA != super);
```

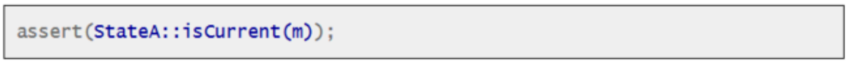
Dynamic queries

- An alias to the current state of a given machine can be obtained by calling the method currentState



```
assert(m.currentState() == StateA::alias());
```

- The current state of a machine can be checked (against an alias state) by

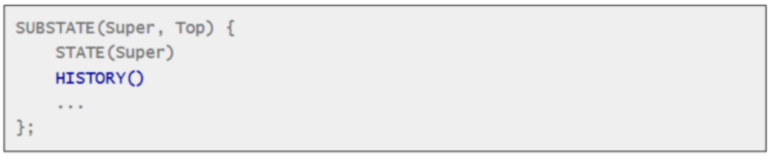


```
assert(StateA::isCurrent(m));
```

- The method isCurrent returns true if the given machine object is in the specified state or any of its substates at that moment.
- `StateA::alias() == m.currentState()` checks if the machine is exactly in StateA.

History states

- MACHO handles both History and Deephistory
- History for a state is enabled by invoking the macro HISTORY or DEEPHISTORY in the state definition:

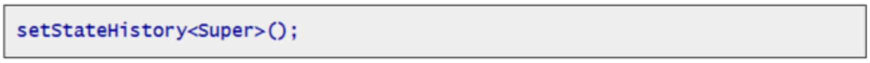


```
SUBSTATE(Super, Top) {
    STATE(Super)
    HISTORY()
    ...
};
```

- Remember that all the variables in the Box will be destroyed and recreated unless PERSISTENT is used

History states

- The history of a state can be the target of a state transition:

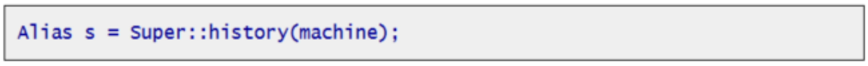


```
setStateHistory<Super>();
```

- Entry actions of all involved states will be invoked, with a final call to the parameterless init method of the actual history state. If no history information is available (because the state has not been entered yet or history was not enabled), Super itself is the target of the transition.

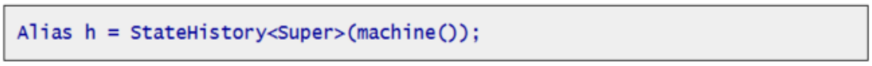
Inspecting history

- The history at some given point can be inspected by setting up an Alias state to the current history. If there is no history, the Super State itself is returned



```
Alias s = Super::history(machine);
```

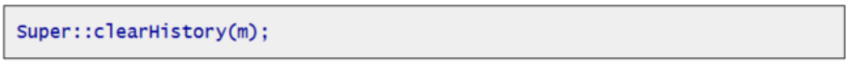
- A special Alias can be used to get an updated snapshot (as if it were a pointer) to the history of a given state.



```
Alias h = StateHistory<Super>(machine());
```

Clearing history

- A state's history for a particular machine instance can be cleared by calling the state's static method clearHistory with the machine object as argument:

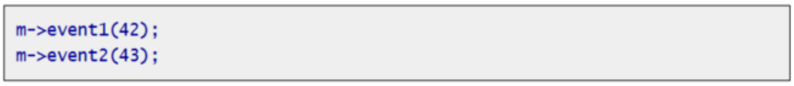


```
Super::clearHistory(m);
```

- This statement resets history information for Super inside machine m, without affecting substate history however (use clearHistoryDeep for this).

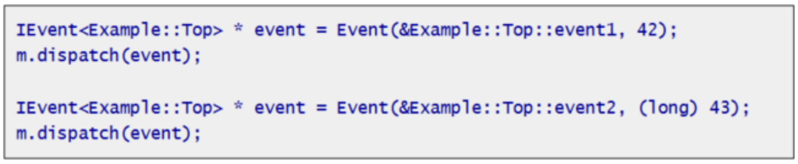
Event Dispatch

- The simplest way to dispatch events (synchronously) to a state machine is by calling event handlers through the machine object's arrow operator, a technique commonly found with C++ smart pointers:



```
m->event1(42);
m->event2(43);
```

- The other option is to call the dispacth method of the machine and pass an IEvent object to it



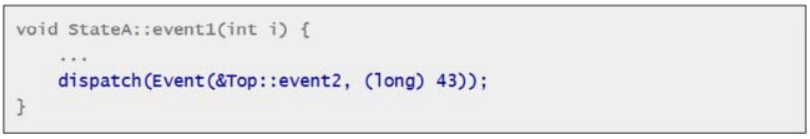
```
IEvent<Example::Top> * event = Event(&Example::Top::event1, 42);
m.dispatch(event);

IEvent<Example::Top> * event = Event(&Example::Top::event2, (long) 43);
m.dispatch(event);
```

Event Dispatch

- The Event function takes as arguments a pointer-to-member to an event handler and all arguments needed to invoke that event handler. Of course the event parameters must be consistent with the event handler's signature.
- The result of an Event call is a pointer to an object with the interface IEvent<T> created on the heap (with T being the top state of the state machine to dispatch to). This pointer can then be queued for later asynchronous dispatching:

Event Dispatch

- Event objects can also be dispatched inside an event handler:



```
void StateA::event1(int i) {
    ...
    dispatch(Event(&Top::event2, (long) 43));
}
```

- The event object is dispatched after the control flow has left the event handler, and after a possible state transition has been performed.
  - Run-to-completion
  - Event managed in the destination state
- Only one event can be dispatched inside an event handler
- It is not possible to dispatch events in entry, exit or init

Event Dispatch

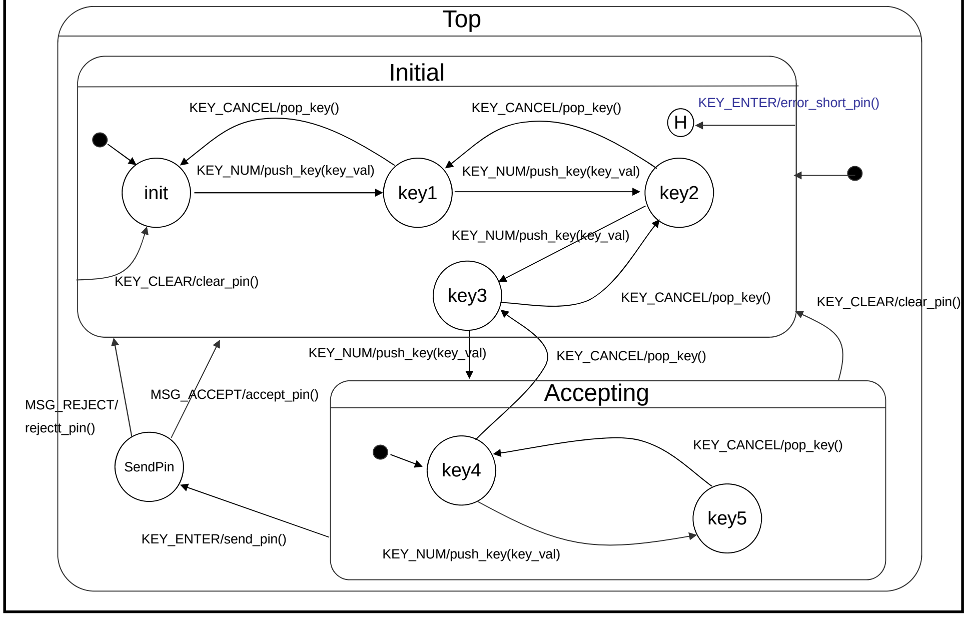
- The following are the same …

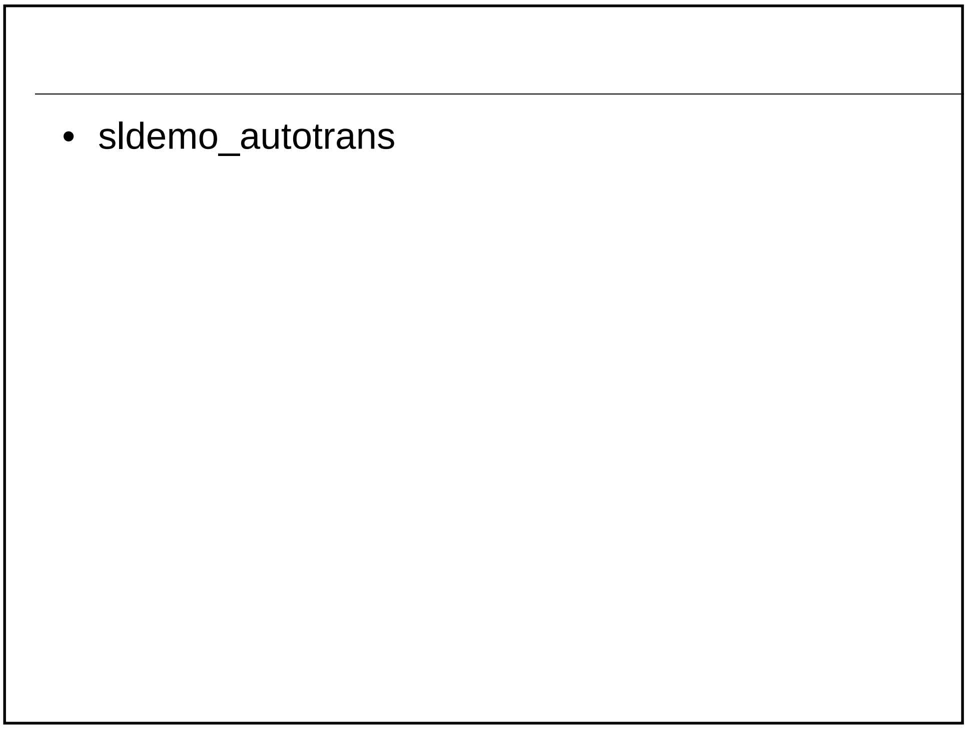
```
void StateA::event1(int i) {
    setState<StateB>();
    dispatch(Event(&Top::event1, i));
}
```

=

```
void StateA::event1(int i) {
    dispatch(Event(&Top::event1, i));
    setState<StateB>();
}
```

Example: Revised ATM keypad





- sldemo_autotrans