Parsing and State Machines

This section daily deals with state machines and the use of visitors as events.

WE'LL COVER THE FOLLOWING

- Parsing a Config File
- State Machines

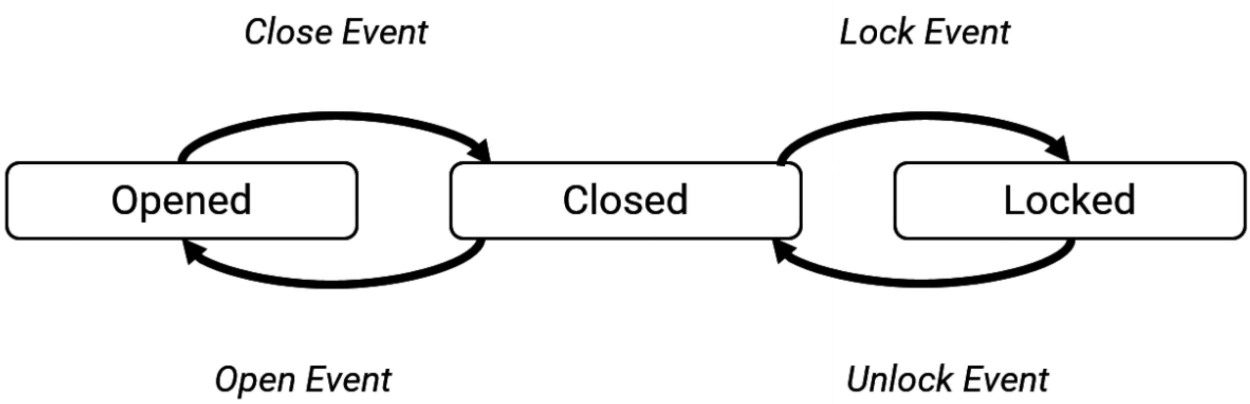
Parsing a Config File

The idea comes from the previous example of a command line. In the case of a configuration file, we usually work with pairs of `<Name, Value>`. Where `Value` might be a different type: `string`, `int`, `array`, `bool`, `float`, etc.

For such a use case, even `void*` could be used to hold such an unknown type. However, this pattern is extremely error-prone. We could improve the design by using `std::variant` if we know all the possible types, or leverage `std::any`.

State Machines

How about modelling a state machine? For example, a door's state:



We can use different types of states and the use visitors as events:

```
#include <iostream>
using namespace std;


struct DoorState
{
    struct DoorOpened {};
    struct DoorClosed {};
    struct DoorLocked {};

    using State = std::variant<DoorOpened, DoorClosed, DoorLocked>;

    void open()
    {
        m_state = std::visit(OpenEvent{}, m_state);
    }
    void close()
    {
        m_state = std::visit(CloseEvent{}, m_state);
    }
    void lock()
    {
        m_state = std::visit(LockEvent{}, m_state);
    }
    void unlock()
    {
        m_state = std::visit(UnlockEvent{}, m_state);
    }
    State m_state;
};
```

And here are the events:

```
#include <iostream>
using namespace std;

struct OpenEvent
{
    State operator()(const DoorOpened&) {
        return DoorOpened();
    }
    State operator()(const DoorClosed&) {
        return DoorOpened();
    }
    // cannot open locked doors
    State operator()(const DoorLocked&) {
        return DoorLocked();
    }
};

struct CloseEvent
{
    State operator()(const DoorOpened&) {
        return DoorClosed();
    }
    State operator()(const DoorClosed&) {
        return DoorClosed();
    }
    State operator()(const DoorLocked&) {
```

```
        return DoorLocked();
    }
};

struct LockEvent
{
    // cannot lock opened doors
    State operator()(const DoorOpened&) {
        return DoorOpened();
    }
    State operator()(const DoorClosed&) {
        return DoorLocked();
    }
    State operator()(const DoorLocked&) {
        return DoorLocked();
    }
};

struct UnlockEvent
{
    // cannot unlock opened doors
    State operator()(const DoorOpened&) {
        return DoorOpened();
    }
    State operator()(const DoorClosed&) {
        return DoorClosed();
    }
    // unlock
    State operator()(const DoorLocked&) {
        return DoorClosed();
    }
};
```

We can now create `Door` object and switch between states:

```
DoorState state;
assert(std::holds_alternative<DoorState::DoorOpened>(state.m_state));
state.lock();
assert(std::holds_alternative<DoorState::DoorOpened>(state.m_state));
```

Code in action:

```
#include <iostream>
#include <variant>
#include <cassert>

struct DoorState {
    struct DoorOpened {};
    struct DoorClosed {};
    struct DoorLocked {};

    using State = std::variant<DoorOpened, DoorClosed, DoorLocked>;

    void open() {
        m_state = std::visit(OpenEvent{}, m_state);
    }
```

```
    void close() {
        m_state = std::visit(CloseEvent{}, m_state);
    }

    void lock() {
        m_state = std::visit(LockEvent{}, m_state);
    }

    void unlock() {
        m_state = std::visit(UnlockEvent{}, m_state);
    }

    struct OpenEvent {
        State operator()(const DoorOpened&){ return DoorOpened(); }
        State operator()(const DoorClosed&){ return DoorOpened(); }
        // cannot open locked doors
        State operator()(const DoorLocked&){ return DoorLocked(); }
    };

    struct CloseEvent {
        State operator()(const DoorOpened&){ return DoorClosed(); }
        State operator()(const DoorClosed&){ return DoorClosed(); }
        State operator()(const DoorLocked&){ return DoorLocked(); }
    };

    struct LockEvent {
        // cannot lock opened doors
        State operator()(const DoorOpened&){ return DoorOpened(); }
        State operator()(const DoorClosed&){ return DoorLocked(); }
        State operator()(const DoorLocked&){ return DoorLocked(); }
    };

    struct UnlockEvent {
        // cannot unlock opened doors
        State operator()(const DoorOpened&){ return DoorOpened(); }
        State operator()(const DoorClosed&){ return DoorClosed(); }
        // unlock
        State operator()(const DoorLocked&){ return DoorClosed(); }
    };

    State m_state;
};

int main() {
    DoorState s;
    assert(std::holds_alternative<DoorState::DoorOpened>(s.m_state));
    s.lock();
    assert(std::holds_alternative<DoorState::DoorOpened>(s.m_state));
    s.close();
    assert(std::holds_alternative<DoorState::DoorClosed>(s.m_state));
    s.lock();
    assert(std::holds_alternative<DoorState::DoorLocked>(s.m_state));
    s.open();
    assert(std::holds_alternative<DoorState::DoorLocked>(s.m_state));

    return 0;
}
```

> You can read more about state machines and implementation of a simple space game in the following blog post: A std::variant-Based State Machine by Example.

The next lessons discusses the concept of using `std::vector` and `std::variant` in Polymorphism.