C++17 `std::variant`

for type-safe state machines

Nick Sarten

October 26, 2017

Outline

- Introduction to `std::variant`
- Basic `std::variant` usage
- Practical `std::variant` example
- Comparison with (`boost::variant`)
- Comparison with `std::any`
- `std::variant` based state machines
- Comparison with other methods:
  - Performance
  - Implementation size
  - Safety

Why C++17?

Why `std::variant`?

What is `std::variant`?

- Tagged union
- Discriminated union
- Type-safe union
- Sum type

"Like `union`, but safer"

Using `std::variant`

Requires

- GCC/libstdc++ 7.0+ `-std=c++17`
- Clang/libc++ 4.0+ `-std=c++1z`
- MSVC 15 (2017) `\std:c++17` or `\std:latest`

Alternatives:

- `boost::variant` C++98 (boost 1.31)
- `mapbox::variant` C++11
- `mpark::variant` C++11

std::variant usage

```
std::variant<int, float, std::string> stuff;
stuff = "wat"; // stuff contains string
assert(std::holds_alternative<std::string>(stuff));

stuff = 2.1f; //stuff contains float
float fuff = std::get<float>(stuff); // throws on error
fuff = std::get<1>(stuff); // get by index, throw on error

stuff = 12; // stuff contains int
int* iuff = std::get_if<int>(&stuff); // nullptr on error
iuff = std::get_if<0>(&stuff); // get by index, nullptr on error
```

Value-or-error return types

```
using Error = std::string;
template<typename T>
using Result = std::variant<T, Error>;

Result<int> up_to_100(int x) {
    if (x < 100) {
        return ++x;
    } else {
        return "x must be less than 100";
    }
}
```

Value-or-error return types

```
int main() {
    int count = 0;
    while (count < 200) {
        Result<int> next = up_to_100(count);
        if (std::holds_alternative<int>(next)) {
            count = std::get<int>(next);
        } else {
            std::cout << "ERROR: " << std::get<Error>(next) << std::endl;
            break;
        }
    }
}
```

`std::variant` alternative types

Can go in `std::variant`

- value types (e.g. `int`)
- POD structs
- class types
- pointers
- unions

Can't go in `std::variant`

- C arrays (use `std::array`)
- `void` (use `std::monostate`)
- Reference types (use pointers or `std::reference_wrapper`)

- Types can be cv-qualified
- Copy or move semantics will only be implemented if all alternatives implement them

std::monostate

`std::variant` doesn't have a *usable* null state by default.

```
struct monostate { };
```

Useful for:

- Implementing a `null` variant state
- Making a `std::variant` default constructible

valueless_by_exception

Triggered by:

- An exception is thrown during move or copy assignment
- An exception is thrown during construction by emplace

Result:

- `variant::valueless_by_exception` returns `true`
- `variant::index` returns `std::variant_npos`
- `std::get` and `std::visit` throw `std::bad_variant_access`

Visitation API

```cpp
struct ExampleVisitor {
    void operator()(int x) {
        std::cout << x + 1 << std::endl;
    }

    void operator()(const std::string& x) {
        std::cout << "hello, " << x << std::endl;
    }
};

int main() {
    std::variant<int, std::string> var("Pacific++");
    std::visit(ExampleVisitor(), var);
    var = 4;
    std::visit(ExampleVisitor(), var);
}
```

output

```
hello, Pacific++
5
```

Visitation with generic lambdas

```cpp
std::variant<int, std::string, float> var("Pacific++");
auto visitor = [](const auto& x) {
    using T = std::decay_t<decltype(x)>;
    if constexpr (std::is_same_v<T, int>) {
        std::cout << x + 1 << std::endl;
    } else if constexpr (std::is_same_v<T, std::string>) {
        std::cout << "hello, " << x << std::endl;
    } else {
        std::cout << "unhandled variant type" << std::endl;
    }
};
std::visit(visitor, var);
var = 4;
std::visit(visitor, var);
var = 2.0f;
std::visit(visitor, var);
```

std::visit applied to value-or-error return types

```
int count = 0;
while (count < 200) {
    Result<int> next = up_to_100(count);
    if (std::visit([&](auto& v) {
        using V = std::decay_t<decltype(v)>;
        if constexpr (std::is_same_v<V, int>) {
            count = v;
            return false;
        } else if constexpr (std::is_same_v<V, Error>) {
            std::cout << "ERROR: " << v << std::endl;
            return true;
        }
    }, next)) {
        break;
    }
}
```

Compared to `boost::variant`

`std::variant` is based on `boost::variant` with a few differences:

- `boost::variant` *may* allocate memory on assignment.
- `boost::variant` has `recursive_variant` and associated visitor.
- `std::monostate` (`boost::blank`)

Compared to std::any

std::any

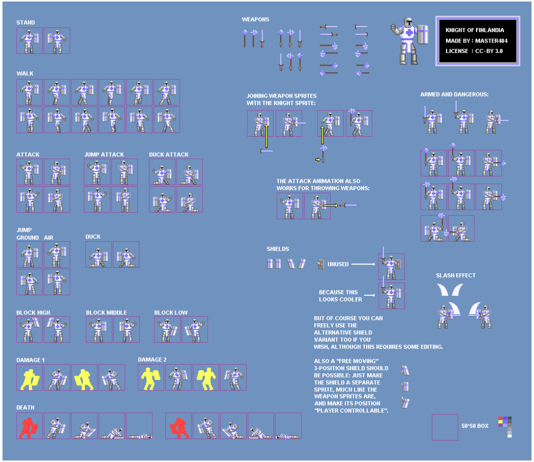
- Allowed to dynamically allocate additional memory.
- Contained type(s) don't form part of the type signature.
- Contained type(s) don't have to be named in advance.
- Contained type(s) must be copy-constructible.
- Has an implicit empty state.
- May have a small-value optimization.

```
std::any stuff; // stuff is empty
stuff = 12; // stuff contains int
int* iuff = std::any_cast<int>(&stuff); // nullptr on error
stuff = 2.0f; //stuff contains float
float fuff = std::any_cast<float>(stuff); // throws on error
```

Applications

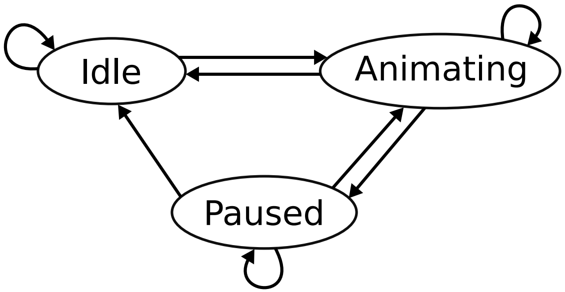
- Multi-type variables, containers, parameters or return types
  - Heterogeneous containers
  - Value-or-error return types (Result type)
  - Optional values (use `std::optional`)
- Finite State Machines

Finite State Machine: Animation Engine



`https://opengameart.org/content/knight-of-finlandia`

FSM Example: Animation Engine



States

- Idle: draw the first frame of the animation
- Animating: draw frame `n` of the animation, incrementing
- Paused: draw frame `n` of the animation

Inputs

- `Play()`
- `Pause()`
- `Stop()`

Naive FSM

```
class Animation {
public:
    enum class State {
        Animating,
        Paused,
        Idle
    };
//...snip
    void stop() {
        counter_ = 0;
        state_ = State::Idle;
    }
//..snip
    const std::vector<Frame> frames_;
    uint32_t counter_;
    State state_;
};
```

Advantages

- Well known pattern.
- Easy to optimize.
- State can be exposed directly.

Disadvantages

- Difficult to separate mechanics and data for each state.
- Potentially larger memory footprint vs. other techniques.

FSM with polymorphism

```
class AnimationState {
public:
    AnimationState(const std::vector<FrameType>& frames)
        : frames_(frames) {}
    virtual std::unique_ptr<AnimationState> update() = 0;
    virtual std::unique_ptr<AnimationState> play() const = 0;
    virtual std::unique_ptr<AnimationState> stop() const = 0;
    virtual std::unique_ptr<AnimationState> pause() const = 0;
    virtual const FrameType& current_frame() const = 0;
protected:
    const std::vector<FrameType>& frames_;
};
```

FSM with polymorphism

```
//...snip
class StateIdle : public AnimationState { /*...snip*/ };
class StatePaused : public AnimationState { /*...snip*/ };
class StateAnimating : public AnimationState {
//...snip
    virtual std::unique_ptr<AnimationState> stop() const override {
        return std::make_unique<StateIdle>(this->frames_);
    }
//...snip
private:
    uint32_t counter_;
};
//...snip
```

FSM with polymorphism

```
class Animation {
public:
//...snip
    void stop() {
        update_state(state_->stop());
    }

    void update_state(std::unique_ptr<AnimationState>&& new_state) {
        if (new_state) {
            state_ = std::move(new_state);
        }
    }
//...snip
private:
    class StateIdle : public AnimationState { /*...snip*/ };
    class StatePaused : public AnimationState { /*...snip*/ };
    class StateAnimating : public AnimationState { /*...snip*/ };
//...snip
    const std::vector<Frame> frames_;
    std::unique_ptr<AnimationState> state_;
};
```

Advantages

- Cleaner separation of state mechanics and data.

Disadvantages

- Dynamic allocation.
- Virtual method calls.
- Verbose.

FSM with std::variant

```
class Animation {
public:
    struct StateIdle {};
    struct StatePaused {
        uint32_t counter_;
    };
    struct StateAnimating {
        uint32_t counter_;
    };

    using State = std::variant<StateIdle, StatePaused, StateAnimating>;
//...snip
    void stop() {
        std::visit([&](const auto& state) {
            using T = std::decay_t<decltype(state)>;
            if constexpr (std::is_same_v<T, StateAnimating> || std::is_same_v<T, StatePaused>) {
                state_ = StateIdle{};
            }
        }, state_);
    }
//...snip
    std::vector<Frame> frames_;
    State state_;
};
```

Type safety in transitions

```cpp
std::visit([&](const auto& state) {
    using T = std::decay_t<decltype(state)>;
    if constexpr (std::is_same_v<T, StateIdle>) {
        state_ = StateAnimating{0};
    } else if constexpr (std::is_same_v<T, StatePaused>) {
        state_ = StateAnimating{state.counter_};
    }
}, state_);
```

```cpp
struct VisitorPlay {
    typedef State result_type;

    State operator()(const StateIdle&) const {
        return StateAnimating(0);
    }

    State operator()(const StatePaused& state) const {
        return StateAnimating(state.counter_);
    }

    State operator()(const StateAnimating& state) const {
        return state;
    }
};
```

Transition return type safety

```
void play() {
    std::visit([&](auto new_state){
        using T = std::decay_t<decltype(new_state)>;
        if constexpr (!std::is_same_v<T, std::monostate>) {
            state_ = new_state;
        }
    }, event_play());
}

std::variant<std::monostate, StateAnimating> event_play() const {
    return std::visit([&](const auto& state) -> std::variant<std::monostate, StateAnimating> {
        using T = std::decay_t<decltype(state)>;
        if constexpr (std::is_same_v<T, StateIdle>) {
            return StateAnimating{};
        } else if constexpr (std::is_same_v<T, StatePaused>) {
            return StateAnimating{};
        }
        return std::monostate{};
    }, state_);
}
```

FSM with `std::variant`

Advantages

- Separation of state mechanics and data.
- Low overhead - zero extra heap allocation.
- Increased type safety.

Disadvantages

- Level of abstraction/complexity.
- Longer compile times.
- Requires C++17

FSM with boost::variant and C++03

```
class Animation {
public:
    struct StateIdle {};
    struct StatePaused {
        StatePaused(size_t counter) : counter_(counter) {}
        size_t counter_;
    };
    struct StateAnimating {
        StateAnimating(size_t counter) : counter_(counter) {}
        size_t counter_;
    };

    typedef boost::variant<StateIdle, StatePaused, StateAnimating> State;

    struct VisitorCurrentFrame {
        typedef const Frame& result_type;

        VisitorCurrentFrame(const std::vector<Frame>& frames) : frames_(frames) {}
        const Frame& operator()(const StateIdle&) const {
            return frames_[0];
        }
        //...snip
        const std::vector<Frame>& frames_;
    };
//...snip
    std::vector<Frame> frames_;
    State state_;
};
```

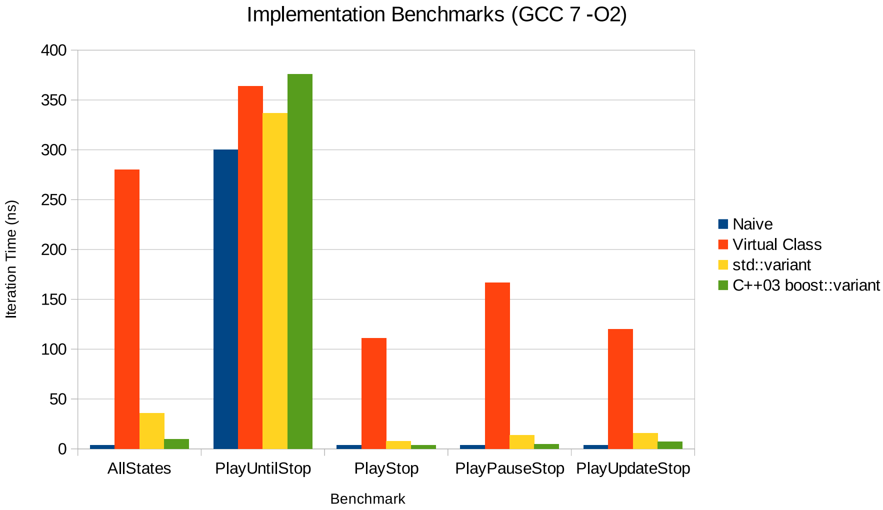
Disadvantages compared to std::variant

- Slower
- More verbose

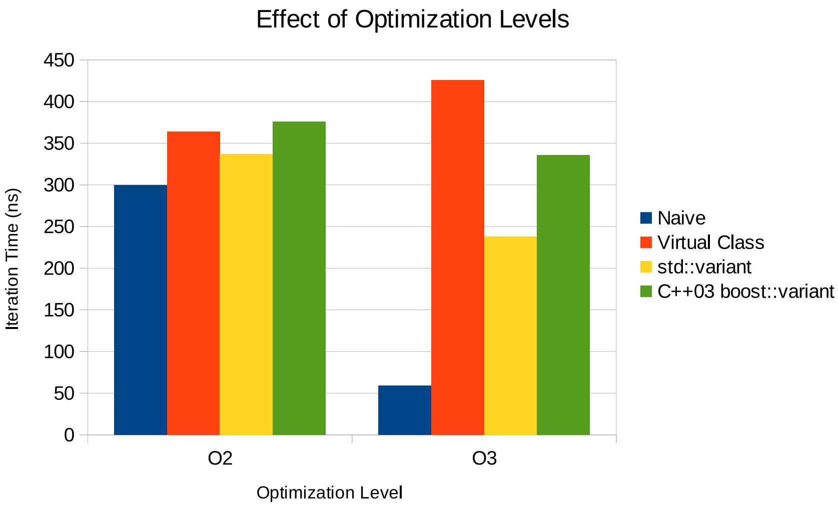
Still better in every way than using virtual classes.

Lies, damn lies and benchmarks

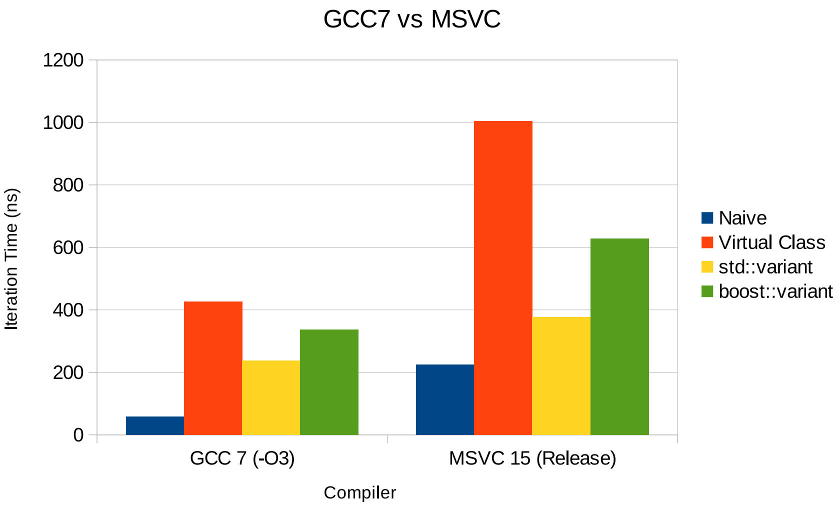
Performance Measurements (−O2)



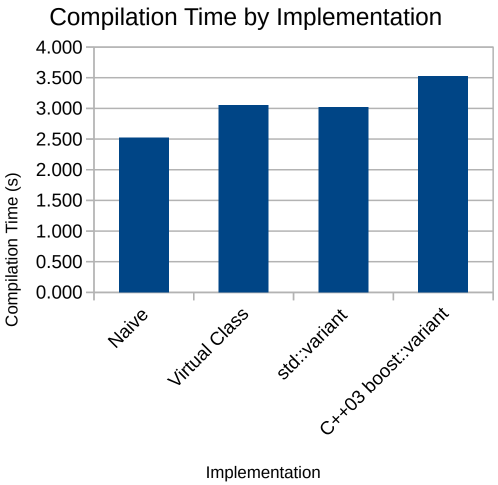
Effect of Optimization Levels

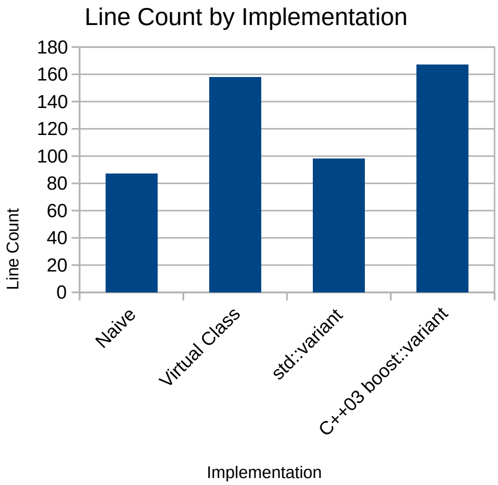


GCC vs. MSVC


GCC7 vs MSVC
1200
1000
800
600
400
200
0
Iteration Time (ns)
GCC 7 (-O3)
MSVC 15 (Release)
Compiler
Naive
Virtual Class
std::variant
boost::variant

Compilation Time and Code Size





Performance conclusions

- Take these results with a grain of salt
- Naive approach won't be displaced in high-performance applications
- Anyone using a polymorphism-based state machine implementation would benefit from moving to a `variant`-based approach
- `boost::variant` is a viable alternative for anyone stuck on an old compiler

Conclusions about `std::variant` for FSMs

- Naive implementation for raw performance

`std::variant` offers

- Good performance
- Separation of state properties
- Type safety

Should you use `std::variant` for FSMs?
Test for yourself.

References

Type-Safe Unions in C++ and Rust
https://genbattle.bitbucket.io/blog/2016/10/07/Type-Safe-Unions-in-C-and-Rust/

Variant: a type-safe union for C++17 (v8)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r3.html

Boost Variant
http://www.boost.org/doc/libs/1_64_0/doc/html/variant.html

Ben Deane: Using Types Effectively
https://www.youtube.com/watch?v=ojZbFIQSdl8

David Sankel: Variants Past, Present and Future
https://www.youtube.com/watch?v=k3O4EKX4z1c

Vittorio Romero: Implementing 'variant' visitation using lambdas
https://www.youtube.com/watch?v=3KyW5Ve3LtI

Thanks for listening!
Code: `https://goo.gl/uYvBuL`

Twitter: @NickSarten

Thanks to Pacific++ for giving me the opportunity to speak to all of you!

https://fearlesscoder.blogspot.co.nz/2017/02/the-c17-lands.html