`std::variant`

and the power of

pattern matching

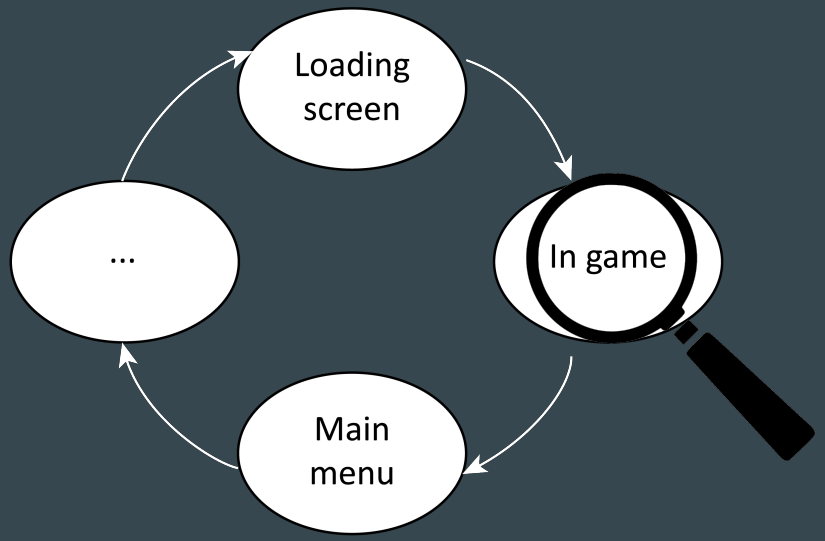
About Me

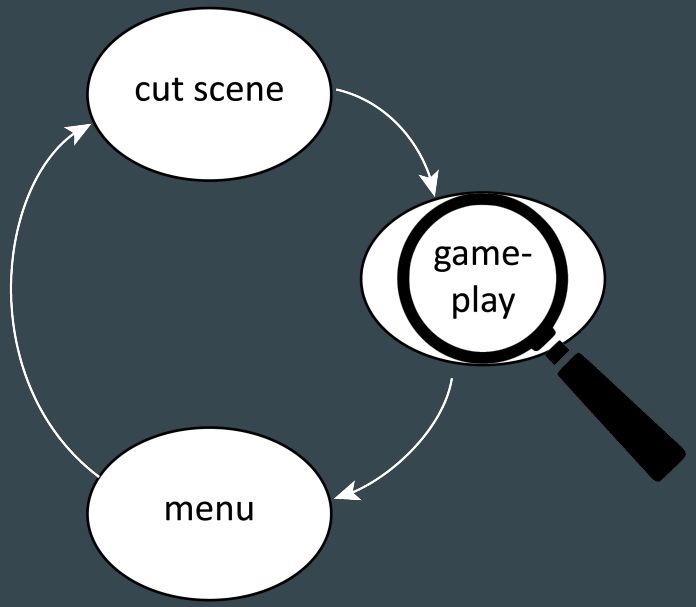
Tech Principal at Ableton

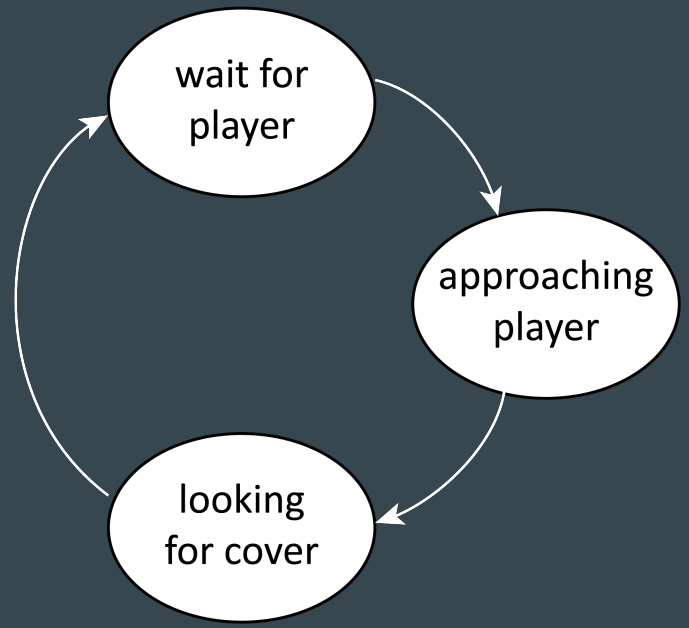
Making music hardware & software

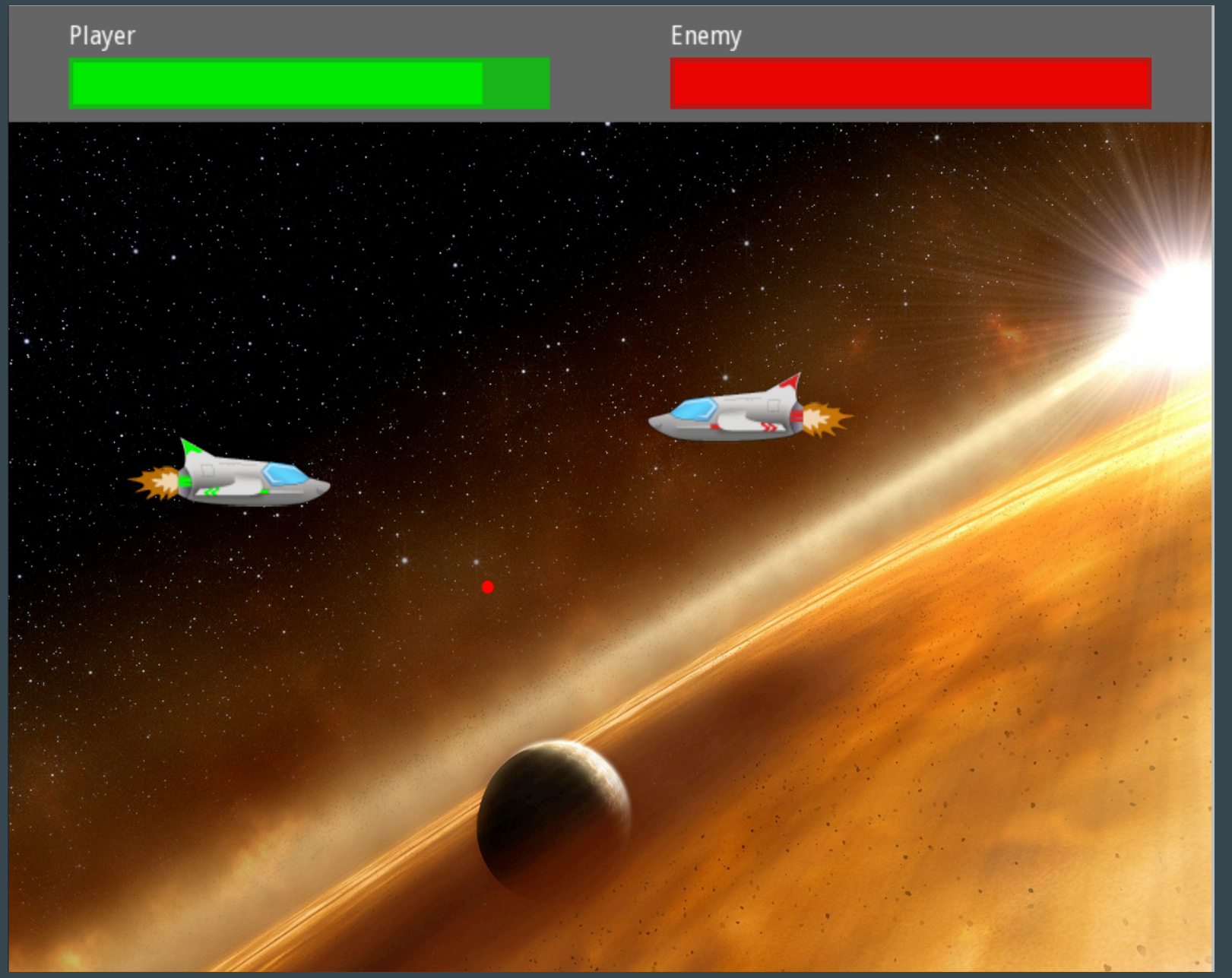
Into: Games, guitar playing, vintage computers

Motivation

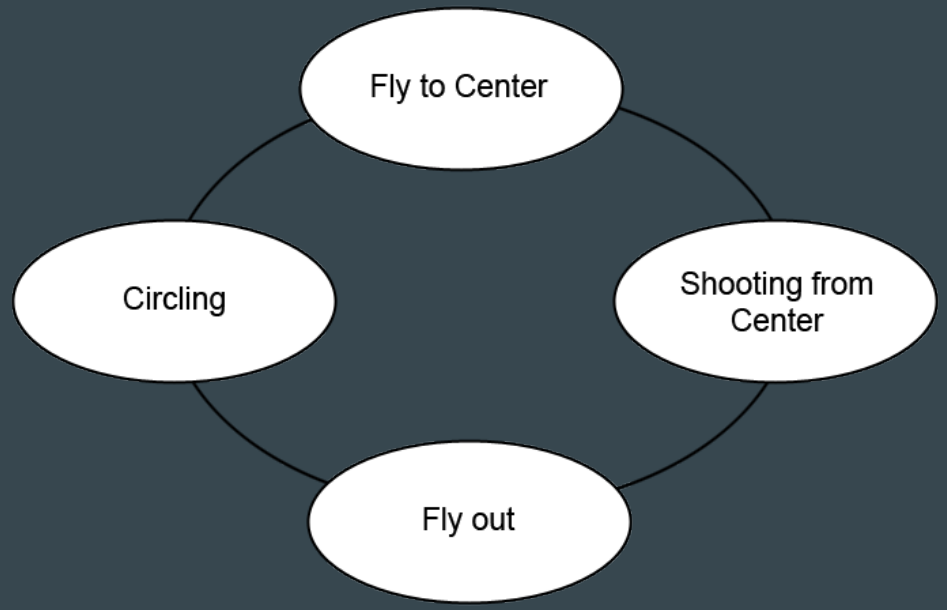

Loading screen
In game
Main menu
...


cut scene
game-play
menu


wait for player
approaching player
looking for cover


Player
Enemy

Space Game: Opponent state machine



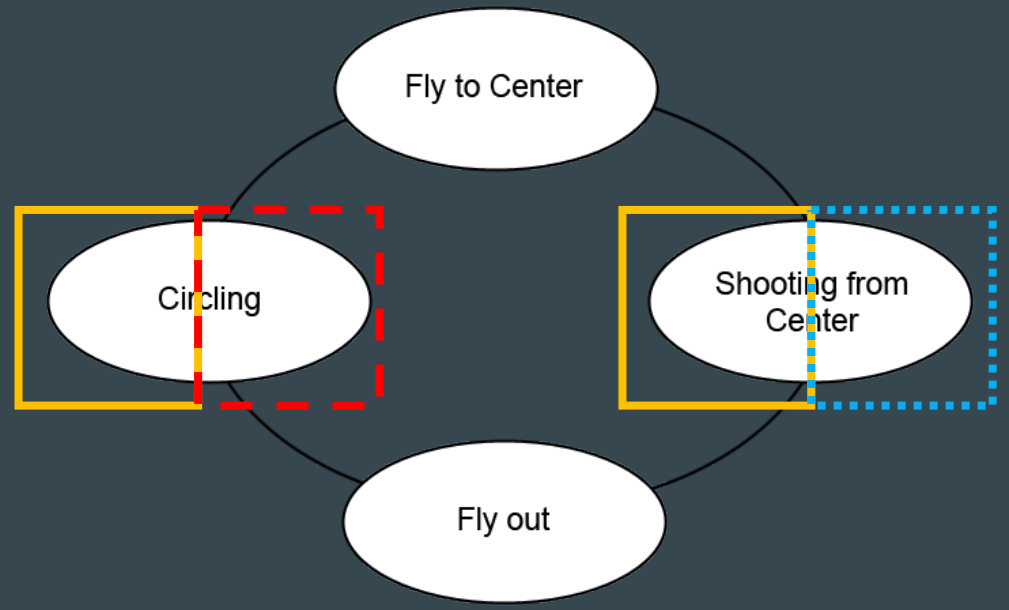
Traditional: **enum** + state variables

```
struct Enemy {
  enum class State {
    Circling,
    FlyToCenter,
    ShootingFromCenter,
    FlyOut
  };

  State state;
  double timeSinceLastShot;
  double timeSpentInCenter;
  int nextCirclePosIndex;
};
```

```
switch (state) {
  case S::Circling:
    // ...
    break;


  case S::FlyToCenter:
    // ...
    break;


  case S::ShootingFromCenter:
    // ...
    break;


  // ...
}
```

Traditional: **enum** + state variables

```
struct Enemy {
  enum class State {
    Circling,
    FlyToCenter,
    ShootingFromCenter,
    FlyOut
  };

  State state;
  double timeSinceLastShot;
  double timeSpentInCenter;
  int nextCirclePosIndex;
};
```



Wanted: Modeling states explicitly

Definition: Sum Type

"a datatype which can be *one of several types of things*"

"Only one of the types can be in use at any one time"

en.wikipedia.org/wiki/Algebraic_data_type
en.wikipedia.org/wiki/Sum_type

Tagged Union

```
struct TaggedUnion
{
  enum class Tag
  {
    Int,
    Float,
    // ...
  };

  union Value
  {
    int i;
    float f;
    // ...
  };

  Tag tag;
  Value value;
};
```

```
switch (v.tag)
{
  case TaggedUnion::Tag::Int:
    // Do something with v.value.i
    break;

  case TaggedUnion::Tag::Float:
    // Do something with v.value.f
    break;

  // etc.
}
```

std::variant

Variant in C++ 17

```
using Value = std::variant<int, float, std::string>;


auto v = Value{100};
auto v2 = Value{2.5f};

v2 = "Hello”;


std::vector<Value> values{1, 2.5f, "Hello"};
```

Getting at the current value

```
variant<int, float> v{2.5f};

if (holds_alternative<int>(v))
{
  auto i = get<int>(v);
  // do something with i
}
else if (holds_alternative<float>(v))
{
  // etc.
}
```

```
variant<int, float> v{2.5f};

if (auto pi = get_if<int>(&v))
{
  auto i = *pi;
  // do something with i
}
else if (auto pf = get_if<float>(&v))
{
  // etc.
}
```

no exhaustiveness check

Using `std::visit()`

```
struct Visitor
{
  void operator()(int i) {
    // Do something with i
  }

  void operator()(float f) {
    // Do something with f
  }

  // etc.
};
```

```
variant<int, float> var{2.5f};

visit(Visitor{}, var);
```

Pattern Matching

Definition

„Pattern matching is a mechanism for checking a **value against** a **pattern**. A successful match can also **deconstruct** a value into its **constituent parts**. It is a **more powerful** version of the **switch** statement [...] and it can likewise be used in place of a series of if/else statements.”

https://docs.scala-lang.org/tour/pattern-matching.html

Matching values

```
if (n == 5) {
  // ...
} else if (n >= 6 && n <= 10) {
  // ...
} else if (n < 0) {
  // ...
} else {
  // ...
}
```

```
match (n) {
  5           => // ...,
  [6..10]     => // ...,
  n if n < 0 => // n ...,
  _           => // ...
}
```

Native pattern matching: Haskell (1)

```
int factorial(int n) {
  if (n == 0) { return 1; }

  return n * factorial(n - 1);
}
```

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Sum types & pattern matching

```
variant<Type1, Type2, Type3> variantValue;

match (variantValue) {
  Type1 t1 => ...,
  Type2 t2 => ...,
  Type3 t3 => ...,
}
```

Native pattern matching: Haskell (2)

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)

countNodes :: Tree a -> Int
countNodes (Leaf a) = 0
countNodes (Node left right) = 1 + countNodes left + countNodes right
```

Native pattern matching: Rust

```
enum Message {
  Quit,
  ChangeColor(i32, i32, i32),
  Move { x: i32, y: i32 },
  Write(String),
}

fn process_message(msg: Message) {
  match msg {
    Message::Quit => quit(),
    Message::ChangeColor(r, g, b) => change_color(r, g, b),
    Message::Move { x: x, y: y } => move_cursor(x, y),
    Message::Write(s) => println!("{}", s),
  };
}
```

Wanted: Pattern matching a C++ variant

```
variant<int, float, ...> var{2.5f};

match(var) {
  int i => {
    // Do something with i
  },

  float f => {
    // Do something with f
  },

  // etc.
}
```

`overloaded` helper

```
template <typename... Ts>
struct overloaded : Ts...
{
  using Ts::operator()...;
};


template <typename... Ts>
overloaded(Ts...) -> overloaded<Ts...>;
```

```
variant<int, float> myVar{2.5f};

visit(overloaded{
  [](int i) {},

  [](float f) {}

  }, myVar);
```

Code from cppreference.com (`std::visit` page)

Proposal to add `std::overload`: P0051

Writing a `match()` function

```
template<typename Variant, typename... Handlers
auto match(Variant&& v, Handlers&&... handlers)
{
  return std::visit(
    overloaded{std::forward<Handlers>(handlers)...},
    std::forward<Variant>(v));
}
```

Using the `match()` function

```
variant<int, float, ...> var{2.5f};

match(var,
  [](int i) {
    // Do something with i
  },

  [](float f) {
    // Do something with f
  },

  // etc.
);
```

```
variant<int, float, ...> var{2.5f};

match(var) {
  int i => {
    // Do something with i
  },

  float f => {
    // Do something with f
  },

  // etc.
}
```

C++ 11 & 14 support

Variant libraries

- Boost.Variant (many differences to std version)
- github.com/eggs-cpp/variant (close to std)
- github.com/mpark/variant (based on libc++ std::variant impl)
- Abseil Variant (equivalent to std)
- …

Match function

github.com/mapbox/variant → include/mapbox/variant_visitor.hpp

godbolt.org/z/rgaEpu

Branch: master ▾ variant / include / mapbox / variant_visitor.hpp Find file Copy path

ricardocosme Use forwarding reference in make_visitor and visitor 9f991da on 27 Jun 2017

2 contributors

44 lines (33 sloc) | 946 Bytes Raw Blame History

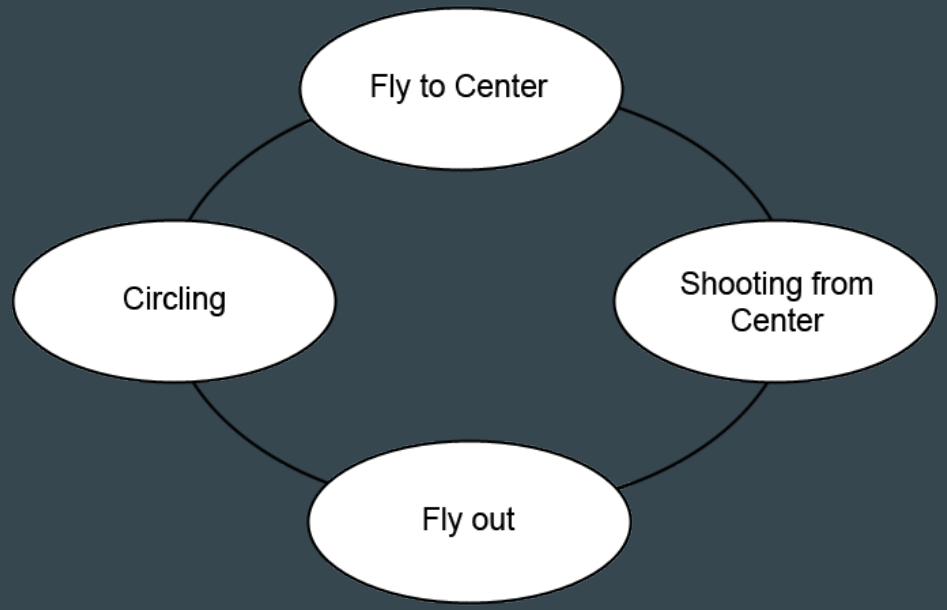
```
#ifndef MAPBOX_UTIL_VARIANT_VISITOR_HPP
#define MAPBOX_UTIL_VARIANT_VISITOR_HPP

#include <utility>

namespace mapbox {
namespace util {

template <typename... Fns>
struct visitor;

template <typename Fn>
struct visitor<Fn> : Fn
{
    using Fn::operator();

    template<typename T>
    visitor(T&& fn) : Fn(std::forward<T>(fn)) {}
};

template <typename Fn, typename... Fns>
struct visitor<Fn, Fns...> : Fn, visitor<Fns...>
{
    using Fn::operator();
    using visitor<Fns...>::operator();

    template<typename T, typename... Ts>
    visitor(T&& fn, Ts&&... fns)
        : Fn(std::forward<T>(fn))
        , visitor<Fns...>(std::forward<Ts>(fns)...) {}
};

template <typename... Fns>
visitor<typename std::decay<Fns>::type...> make_visitor(Fns&&... fns)
{
    return visitor<typename std::decay<Fns>::type...>
        (std::forward<Fns>(fns)...);
}

} // namespace util
} // namespace mapbox

#endif // MAPBOX_UTIL_VARIANT_VISITOR_HPP
```

Examples

1. State Machines
2. Event Handling
3. More examples (brief)

1: State Machines

Reminder: Space game state machine



```
struct Circling {
  double mTimeSinceLastShot = ENEMY_SHOT_DELAY;
  int mNextCirclePosIndex = 0;
};

struct FlyToCenter { };

struct ShootingFromCenter {
  double mTimeSinceLastShot = ENEMY_SHOT_DELAY;
  double mTimeSpentInCenter = 0;
};

struct FlyOut {
  int mTargetCornerIndex;
};

using State = std::variant<
  Circling,
  FlyToCenter,
  ShootingFromCenter,
  FlyOut>;
```

Implementing state-specific logic

```
class Enemy {
  // ...
private:
  State mState;
};


void Enemy::update() {
  match(mState,
    [](Circling& state) { /* ... */ },

    [](const FlyToCenter&) { /* ... */ },

    [](ShootingFromCenter& state) { /* ... */ },

    [](const FlyOut& state) { /* ... */ }
  );
}
```

State transitions: pitfall (1)

```
[=](const FlyToCenter& state)
{
  if (reachedCurrentTarget)
  {
    mState = ShootingFromCenter{};
  }
},
```

State transitions: pitfall (2)

```
[=](const FlyToCenter& state)
{
  if (reachedCurrentTarget)
  {
    mState = ShootingFromCenter{};
  }

  if (state.mFoo ...) {
  }
},
```

⚠️ undefined behavior!!

```
using MaybeNextState = std::optional<State>;

auto maybeNextState = match(mState,
  // ...

  [=](const FlyToCenter&) -> MaybeNextState {
    if (reachedCurrentTarget) {
      return State{ShootingFromCenter{}};
    }

    return std::nullopt;
  },

  // ...
);

if (maybeNextState) {
  mState = *maybeNextState;
}
```

Evaluation

Comparison

Version	*Compile time*	*Binary size*	*Binary (stripped)*	*Performance*
Pattern matching	~ 2.39 s	45,364 B	35,288 B	~ 9 ns / iteration
State enum	~ 2.19 s	43,972 B	35,288 B	~ 6 ns / iteration
Difference	+ 200 ms	+ 1392 B	0 B	+ 3 ns

Core i5, 2 x 2.8 GHz (w/ Hyperthreading)

clang 7.0.0, release build (-O3)
2 translation units, compiled sequentially
measured using `time` utility

Pro/Con

Pro

- Guaranteed invariants
- Exhaustiveness checking
- Straightforward translation of conceptual model

Con

- Needs adherence to “safe transition” pattern
- No formalized separation between per-state behavior and state transitions

Recommendation

Use when:

- Your logic can be organized as a state machine
- Number of states and transitions is not too large

2: Event handling

Event handling

MouseClickEvent

`x, y`

`button`

KeyPressEvent

`key code`

`isRepeat`

WindowResizedEvent

`width, height`

ClientConnected

`...`

ConnectionInterrupted

`...`

Traditional approaches

Event ID plus untyped payload (Win32 API)

Tagged union (X11, SDL, SFML, ...)

⚠ type safety

Handler function per event type (Qt, GTK, Cocoa)

boiler-plate code

Demo

```cpp
enum class MouseButton {
  Left,
  Middle,
  Right,
  Unknown
};

struct MouseMoved {
  int x = 0;
  int y = 0;
};

struct MouseButtonDown { MouseButton button; };

struct MouseButtonUp { MouseButton button; };

struct WindowResized {
  int newWidth = 0;
  int newHeight = 0;
};


using Event = std::variant<
  MouseMoved,
  MouseButtonDown,
  MouseButtonUp,
  WindowResized>;
```

```
match(event,
  [this](const MouseMoved& mouseMove) {
    if (mShouldPrintMouseMoves) {
      std::cout << "mouse moved to " << mouseMove.x << ", " << mouseMove.y << "\n";
    }
  },

  [this](const MouseButtonDown& buttonDown) {
    std::cout << "mouse button down (" << buttonDown.button << ")\n";

    if (buttonDown.button == MouseButton::Right) {
      mShouldPrintMouseMoves = !mShouldPrintMouseMoves;
      std::cout << "\nprinting mouse moves: " << std::boolalpha << mShouldPrintMouseMoves << "\n\n";
    }
  },

  [](const MouseButtonUp& buttonUp) {
    std::cout << "mouse button up (" << buttonUp.button << ")\n";
  },

  [](const WindowResized& resized) {
    std::cout << "window resized to " << resized.newWidth << ", " << resized.newHeight << "\n";
  });
```

Evaluation

```
match(event,
  [](const MouseMoved& m) {
    // ...
  },


  [](const MouseButtonDown& b) {
    // ...
  },


  [](const MouseButtonUp& b) {
    // ...
  },


  [](const WindowResized& r) {
    // ...
  });
```

```
switch (event.type) {
  case sf::Event::MouseMoved: {
      const auto& m = event.mouseMove;
      // ...
      } break;

  case sf::Event::MouseButtonPressed: {
      const auto& b = event.mouseButton;
      // ...
      } break;

  case sf::Event::MouseButtonReleased: {
      const auto& b = event.mouseButton;
      // ...
      } break;

  case sf::Event::Resized: {
      const auto& r = event.size;
      // ...
      } break;

  default:
    break;
}
```

Comparison

Version	*Compile time*	*Binary size*	*Binary (stripped)*	*Performance*
Pattern matching	~ 1.57 s	22,048 B	19,500 B	29 ns / iteration
Tagged union	~ 1.37 s	19,652 B	19,124 B	28 ns / iteration
Difference	+ 200 ms	+ 2,396 B	+ 376 B	+ 1 ns

Core i5, 2 x 2.8 GHz (w/ Hyperthreading)

clang 7.0.0, release build (-O3)
2 translation units, compiled sequentially
measured using `time` utility

Pro/Con

Pro

- Type safety
- Exhaustiveness checking
- Conciseness

Con

- Needs adaptation layer for existing libraries

Recommendation

Use for:

- Building your own platform abstraction library
- Internal event handling

More examples

- Function return values

```
using IntersectResult = std::variant<Point, NoIntersection, Containment>;

IntersectResult testIntersection(const Ray& ray, const Plane& plane);
```

- Scripting language OpCode dispatcher? → see GitHub repo

- ???

C++ Pattern Matching: Future

Beyond `std::variant`

https://github.com/mpark/patterns

```
#include <mpark/patterns.hpp>


void fizzbuzz() {
  using namespace mpark::patterns;
  for (int i = 1; i <= 100; ++i) {
    match(i % 3, i % 5)(
        pattern(0, 0) = []  { std::printf("fizzbuzz\n"); },
        pattern(0, _) = []  { std::printf("fizz\n"); },
        pattern(_, 0) = []  { std::printf("buzz\n"); },
        pattern(_, _) = [i] { std::printf("%d\n", i); });
  }
}
```

Native pattern matching proposal

Pattern matching: P1308

Language variants: P0095

Figure 1: Switching an enum.

before

```
enum color { red, yellow, green, blue };
```

```
const Vec3 opengl_color = [&c] {
  switch(c) {
    case red:
      return Vec3(1.0, 0.0, 0.0);
      break;
    case yellow:
      return Vec3(1.0, 1.0, 0.0);
      break;
    case green:
      return Vec3(0.0, 1.0, 0.0);
      break;
    case blue:
      return Vec3(0.0, 0.0, 1.0);
      break;
    default:
      std::abort();
  }();
```

after

```
const Vec3 opengl_color =
  inspect(c) {
    red    => Vec3(1.0, 0.0, 0.0)
    yellow => Vec3(1.0, 1.0, 0.0)
    green  => Vec3(0.0, 1.0, 0.0)
    blue   => Vec3(0.0, 0.0, 1.0)
  };
```

Figure 2: `struct` inspection

```
struct player {
  std::string name;
  int hitpoints;
  int lives;
};
```

before

```
void takeDamage(player &p) {
  if(p.hitpoints == 0 && p.lives == 0)
    gameOver();
  else if(p.hitpoints == 0) {
    p.hitpoints = 10;
    p.lives--;
  }
  else if(p.hitpoints <= 3) {
    p.hitpoints--;
    messageAlmostDead();
  }
  else {
    p.hitpoints--;
  }
}
```

after

```
void takeDamage(player &p) {
  inspect(p) {
    [hitpoints:   0, lives:0]   => gameOver();
    [hitpoints:hp@0, lives:l]   => hp=10, l--;
    [hitpoints:hp] if (hp <= 3) => { hp--; messageAlmostDead(); }
    [hitpoints:hp] => hp--;
  }
}
```

Figure 1. Declaration of a command data structure.

before

```
struct set_score {
  std::size_t value;
};

struct fire_missile {};

struct fire_laser {
  unsigned intensity;
};

struct rotate {
  double amount;
};

struct command {
  std::variant<
    set_score,
    fire_missile,
    fire_laser,
    rotate > value;
};
```

after

```
lvariant command {
  std::size_t set_score;
  std::monostate fire_missile;
  unsigned fire_laser;
  double rotate;
};
```

Summary and conclusion

Pattern matching today

- Limited, but already useful
- Many good use cases

- Possible impact on compile time, binary size & performance
- Harder to use for non-experts

Native pattern matching

- Easier to use
- Better performance (compile & run time)
- More powerful! (destructuring, matching values etc.)

github.com/lethal-guitar/VariantTalk

We are hiring!

Look for „Software Engineer, Live“ at

meetingcpp.com/jobs

or

www.ableton.com/jobs