Simple, Extensible Pattern Matching in C++14

John R. Bandela, MD

SEPTEMBER 22, 2015

Goals

- Brief introduction to pattern matching
- Survey of existing solutions
- Design and implement the library
  - Some useful building blocks
  - Core
  - Examples and implementation
- QA Time

What is pattern matching

- Instead of explicitly extracting (parts of) values and testing them (in an if-else ladder), you specify what the data should look like, and actions to take if it does.
- Kinda like regular expressions for arbitrary values

Q: How well does C++14 support pattern matching?

A: Very well. It is probably in the top tier of support among widely used languages.

At compile time

Here is what it supports at compile time

- Function overloading
- Partial ordering of template functions
- Template specialization
- Partial template specializations

Here is what it supports at run-time

```
switch (i) {
case 1: std::cout << "one"; break;
case 2: std::cout << "two"; break;
case 3: std::cout << "three"; break;
default: std::cout << "unknown";
}
```

Haskell

```haskell
first :: (a, b, c) -> a
first (x, _, _) = x

second :: (a, b, c) -> b
second (_, y, _) = y

third :: (a, b, c) -> c
third (_, _, z) = z


describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

From http://learnyouahaskell.com/syntax-in-functions

Rust

```
let x = 1;
match x {
  1 | 2 => println!("one or two"),
  3 => println!("three"),
 _ => println!("anything"),
}
```

From https://doc.rust-lang.org/nightly/book/patterns.html

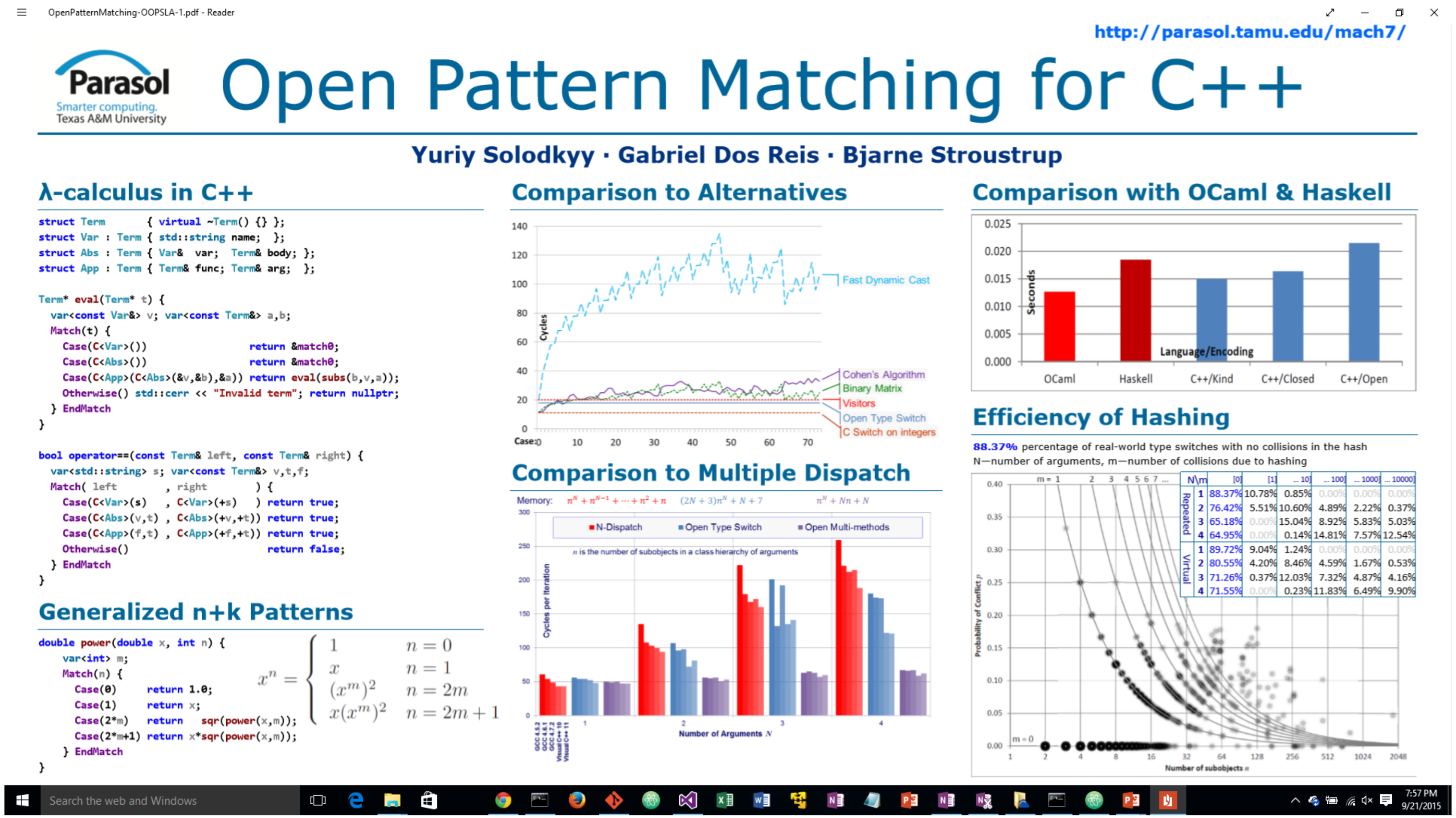
Rust

```
let tup = (1,0);

match tup{
  (0,0) => println!("both zero"),
  (x,0) => println!("{} and zero",x),
  (0,y) => println!("zero and {}",y),
  _     => println!("did not match")

}
```

Rust

```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),
     };
}
```

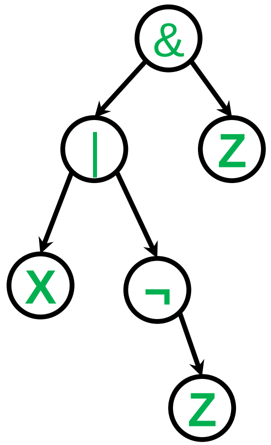
https://github.com/solodon4/Mach7



Motivating Example

Grammar

```
// Abstract syntax of boolean expressions
BoolExp ::= VarExp | ValExp | NotExp | AndExp | OrExp | '(' BoolExp ')'
VarExp  ::= 'A' | 'B' | ... | 'Z'
ValExp  ::= 'true' | 'false'
NotExp  ::= 'not' BoolExp
AndExp  ::= BoolExp 'and' BoolExp
OrExp   ::= BoolExp 'or'  BoolExp
```



Working with *Mach7*

- Declare your variants

```
struct BoolExp          { virtual ~BoolExp() {} };
struct VarExp : BoolExp { std::string name;  };
struct ValExp : BoolExp { bool        value; };
struct NotExp : BoolExp { BoolExp* e; };
struct AndExp : BoolExp { BoolExp* e1; BoolExp* e2; };
struct OrExp  : BoolExp { BoolExp* e1; BoolExp* e2; };
```

Note to myself:

- Non-intrusive!
- Respects member access

- Define bindings (mapping of members to pattern-matching positions)

```
namespace mch { ///< Mach7 library namespace
    template <> struct bindings<VarExp> { Members(VarExp::name); };
    template <> struct bindings<ValExp> { Members(ValExp::value); };
    template <> struct bindings<NotExp> { Members(NotExp::e); };
    template <> struct bindings<AndExp> { Members(AndExp::e1, AndExp::e2); };
    template <> struct bindings<OrExp>  { Members( OrExp::e1,  OrExp::e2); };
}
```

- Pick the patterns you'd like to use

```
using mch::C; using mch::var; using mch::_;
```

Example: eval

```
bool eval(Context& ctx, const BoolExp* exp)
{
    var<std::string> name; var<bool> value; var<const BoolExp*> e1, e2;

    Match(exp)
    {
        Case(C<VarExp>(name)  ) return ctx[name];
        Case(C<ValExp>(value)) return value;
        Case(C<NotExp>(e1)    ) return!eval(ctx, e1);
        Case(C<AndExp>(e1,e2)) return eval(ctx, e1) && eval(ctx, e2);
        Case(C<OrExp >(e1,e2)) return eval(ctx, e1) || eval(ctx, e2);
    }
    EndMatch
}
```

Note to self:

- Patterns in the LHS, values in the RHS
- No control inversion!
    - Direct access to arguments
    - Direct return from the function

Mach7 vs simple_match

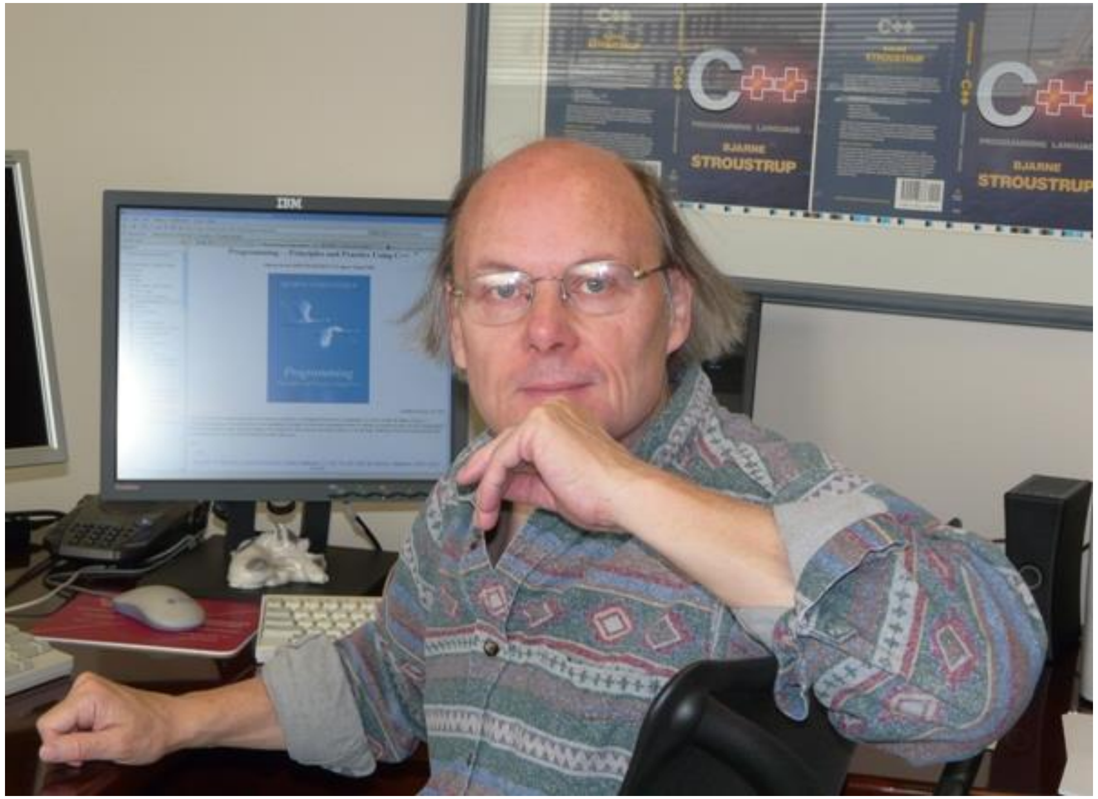
Mach7

- Very smart people
- Works with older compilers
- Uses macros
- Staging for possible language features
- Focus on performance

simple_match

- Yours truly
- C++14 only
- No macros
- Maybe boost?
- Focus on clarity/simplicity

“C++11 feels like a new language”



Lines of code

```
johnb@WIN-5K7QN7M62G5 ~/Repos/simple_match/include (master)
$ find -type f -exec wc -l {} \;
    53 ./simple_match/boost/any.hpp
    30 ./simple_match/boost/optional.hpp
    90 ./simple_match/boost/variant.hpp
   368 ./simple_match/implementation/some_none.hpp
   403 ./simple_match/simple_match.hpp
    35 ./simple_match/utility.hpp
```

simple_match

```
struct VarExp;
struct NotExp;
struct AndExp;
struct OrExp;

using BoolExp = boost::variant<boost::recursive_wrapper<VarExp>, bool,
boost::recursive_wrapper<NotExp>, boost::recursive_wrapper<AndExp>,
boost::recursive_wrapper<OrExp>>;

struct VarExp : std::tuple<std::string> { using tuple::tuple; };
struct NotExp : std::tuple<BoolExp> { using tuple::tuple; };
struct AndExp : std::tuple<BoolExp,BoolExp> {using tuple::tuple; };
struct OrExp : std::tuple<BoolExp,BoolExp> {using tuple::tuple; };
```

simple_match

```
bool eval(const Context& ctx, const BoolExp& exp)
{
  using namespace simple_match;
  using namespace simple_match::placeholders;

  return match(exp,
     some<VarExp>(ds(_x)), [&](auto& x) {
       auto iter = ctx.find(x);
       return iter == ctx.end() ? false : iter->second;
     },
     some<bool>(), [](auto& x) {return x;},
     some<NotExp>(ds(_x)), [&](auto& x){return !eval(ctx,x);},
     some<AndExp>(ds(_x, _y)), [&](auto& x, auto& y){return eval(ctx,x) && eval(ctx,y);},
     some<OrExp>(ds(_x, _y)), [&](auto& x, auto& y){return eval(ctx,x) || eval(ctx,y);}
  );

}
```

simple_match

```
bool eval(const Context& ctx, const BoolExp& exp)
{
  using namespace simple_match;
  using namespace simple_match::placeholders;

  return match(exp,
     some<VarExp>(ds(_x)), [&](auto& x) {
       auto iter = ctx.find(x);
       return iter == ctx.end() ? false : iter->second;
     },
     some<bool>(), [](auto& x) {return x;},
     //some<NotExp>(ds(_x)), [&](auto& x){return !eval(ctx,x);},
     some<AndExp>(ds(_x, _y)), [&](auto& x, auto& y){return eval(ctx,x) && eval(ctx,y);},
     some<OrExp>(ds(_x, _y)), [&](auto& x, auto& y){return eval(ctx,x) || eval(ctx,y);}
  );

}
```

Visual C++ 2015 Error Message

```
1>------ Build started: Project: TestMatch, Configuration: Debug x64 ------
1>  test.cpp
1>c:\users\johnb\repos\simple_match\include\simple_match\implementation\some_none.hpp(321
): error C2338: This type is not in the match
1>
c:\users\johnb\repos\simple_match\include\simple_match\implementation\some_none.hpp(327):
note: see reference to class template instantiation
'simple_match::detail::not_in_match_asserter<false,First>' being compiled
1>          with
1>          [
1>              First=NotExp
1>          ]
```

Design and Implementation of the Library

Building Blocks

C++14 Language/Library features

- Function return type deductions
- Generic lambdas
- index_sequence

Function Return type deductions

```
template<class C>
typename C::const_iterator get_middle_iterator(const C& c){
    return c.cbegin() + (c.size() / 2);
}

template<class C>
auto get_middle_iterator14(const C& c) {
    return c.cbegin() + (c.size() / 2);
}

int main(){
    std::vector<int> v{1,2,3};
    std::cout << *get_middle_iterator(v) << "\n";
    std::cout << *get_middle_iterator14(v) << "\n";
}
```

Generic lambdas

```
int main(){
    std::vector<std::vector<int>> v{ {1,2},{2,3},{2}};
    std::sort(v.begin(), v.end(), [](auto& a, auto& b) {
      return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
    });
}
```

index_sequence

```
template<std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

```
using sequence = std::make_index_sequence<5>::type;
static_assert(std::is_same<sequence, std::index_sequence<0,1,2,3,4>>::value,"Not same");
```

`std::tuple`

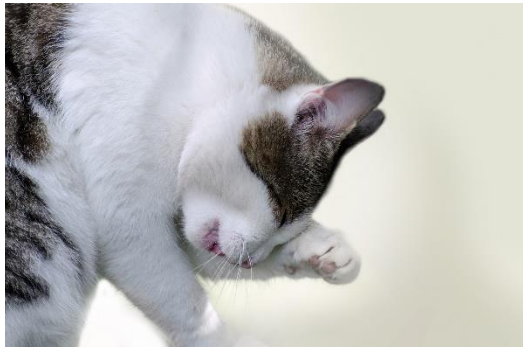
Tuple

```
template< class... Types >
class tuple;

std::tuple<double, char, std::string> get_student(int id)
{
    if (id == 0) return std::make_tuple(3.8, 'A', "Lisa Simpson");
    if (id == 1) return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
    if (id == 2) return std::make_tuple(1.7, 'D', "Ralph Wiggum");
    throw std::invalid_argument("id");
}
int main()
{
    auto student0 = get_student(0);
    std::cout << "ID: 0, "
        << "GPA: " << std::get<0>(student0) << ", "
        << "grade: " << std::get<1>(student0) << ", "
        << "name: " << std::get<2>(student0) << '\n'
}
```

http://en.cppreference.com/w/cpp/utility/tuple

What is this?

std::tuple<



,

,

>

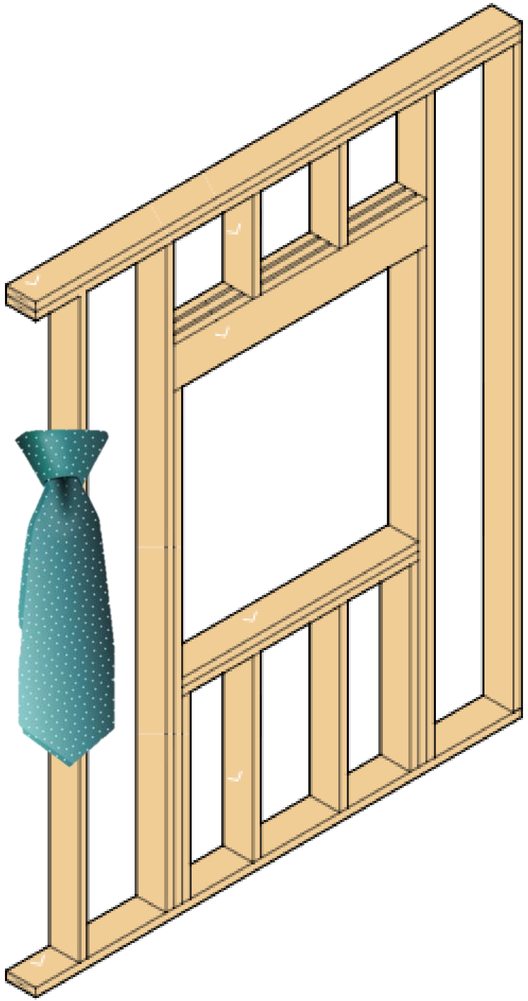
tuple_cat

```
template <class... Tuples>
constexpr std::tuple<CTypes...>
tuple_cat(Tuples&&... args);
```

```
int main()
{
    std::tuple<int, std::string, float> t1(10, "Test", 3.14);
    auto t2 = std::tuple_cat(t1, std::make_pair("Foo", "bar"), t1);
}
```

http://en.cppreference.com/w/cpp/utility/tuple/tuple_cat

What is this



std::tie

```
template< class... Types >
constexpr tuple<Types&...> tie( Types&... args );
```

```
struct S {
    int n;
    std::string s;
    float d;
    bool operator<(const S& rhs) const
    {
        // compares n to rhs.n,
        // then s to rhs.s,
        // then d to rhs.d
        return std::tie(n, s, d) < std::tie(rhs.n, rhs.s, rhs.d);
    }
};
```

http://en.cppreference.com/w/cpp/utility/tuple/tie

apply

```
template <class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);
```

apply implementation (N3915)

```
template <typename F, typename Tuple, size_t... I>
decltype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
  return forward<F>(f)(get<I>(forward<Tuple>(t))...);
}
template <typename F, typename Tuple>
decltype(auto) apply(F&& f, Tuple&& t) {
  using Indices =  make_index_sequence<tuple_size<decay_t<Tuple>>::value>;
  return apply_impl(forward<F>(f), forward<Tuple>(t), Indices{});
}
```

Design and Implementation of the Library

Core

match

```
template<class T, class A1, class F1>
auto match(T&& t, A1&& a, F1&& f) {
  if (match_check(std::forward<T>(t), std::forward<A1>(a))) {
    return detail::apply(f, match_get(std::forward<T>(t), std::forward<A1>(a)));
  }
  else {
    throw std::logic_error("No match");
  }
}

template<class T, class A1, class F1, class A2, class F2, class... Args>
auto match(T&& t, A1&& a, F1&& f, A2&& a2, F2&& f2, Args&&... args) {
  if (match_check(t, a)) {
    return detail::apply(f, match_get(std::forward<T>(t), std::forward<A1>(a)));
  }
  else {
  return match(t, std::forward<A2>(a2), std::forward<F2>(f2),
    std::forward<Args>(args)...);
  }
}
```

match_check/match_get

```
template<class T, class U>
bool match_check(T&& t, U&& u) {
  using namespace customization;
  using m = matcher<std::decay_t<T>, std::decay_t<U>>;
  return m::check(std::forward<T>(t), std::forward<U>(u));
}


template<class T, class U>
auto match_get(T&& t, U&& u) {
  using namespace customization;
  using m = matcher<std::decay_t<T>, std::decay_t<U>>;
  return m::get(std::forward<T>(t), std::forward<U>(u));
}
```

matcher

```
namespace customization {
template<class T, class U>
  struct matcher;

}
```

Design and Implementation of the Library

Extending the core – simple

Extending the Core

- Currently does not do anything useful
- Make it useful by specializing simple_match::customization::matcher
- Also this is the customization hook for user-defined extensions

Matching same type

```
int x = get_int();
match(x,
  1, []() {std::cout << "The answer is one\n"; },
  2, []() {std::cout << "The answer is two\n"; },
  3, []() {std::cout << "The answer is three\n"; }
);
```

Matching the same type

```
// Match same type
template<class T>
struct matcher<T, T> {
    static bool check(const T& t, const T& v) {
      return t == v;
    }
    static auto get(const T&, const T&) {
      return std::tie();
    }

};
```

Matching string literals (const char*)

```
std::string s = get_string();

match(s,
  "Zero", []() {std::cout << “0\n"; },
  "One",  []() {std::cout << “1\n"; },
  "Two",  []() {std::cout << “2\n"; }
);
```

Matching string literals (const char*)

```
template<class T>
struct matcher<T, const char*> {
  static bool check(const T& t, const char* str) {
    return t == str;
  }
  static auto get(const T&, const T&) {
    return std::tie();
  }
};
```

Matching otherwise (equivalent of default)

```
using namespace simple_match;
using namespace simple_match::placeholders;

int x = get_int();

match(x,
  1, []() {std::cout << "The answer is one\n"; },
  2, []() {std::cout << "The answer is two\n"; },
  _, []() {std::cout << "Did not match\n"; }
);
```

Matching otherwise (equivalent of default)

```
struct otherwise_t {};

namespace placeholders {
  const otherwise_t otherwise{};
  const otherwise_t _{};
}
```

Matching otherwise (equivalent of default)

```
template<class Type>
struct matcher<Type, otherwise_t> {
  template<class T>
  static bool check(T&&, otherwise_t) {
    return true;
  }
  template<class T>
  static auto get(T&&, otherwise_t) {
    return std::tie();
  }

};
```

Match and capture anything

```
using namespace simple_match;
using namespace simple_match::placeholders;

int x = get_int();

match(x,
  1,  []() {std::cout << "The answer is one\n"; },
  2,  []() {std::cout << "The answer is two\n"; },
  _x, [](auto x) {std::cout << x << " did not match\n"; }

);
```

Match and capture anything

```
template<class F>
struct matcher_predicate {
  F f_;
};

template<class F>
matcher_predicate<F> make_matcher_predicate(F&& f) {
  return matcher_predicate<F>{std::forward<F>(f)};
}
```

Match and capture anything

```
template<class Type, class F>
struct matcher<Type, matcher_predicate<F>> {
    template<class T, class U>
    static bool check(T&& t, U&& u) {
        return u.f_(std::forward<T>(t));
    }
    template<class T, class U>
    static auto get(T&& t, U&&) {
        return std::tie(std::forward<T>(t));
    }

};
```

Match and capture anything

```
namespace placeholders {
    const auto _x = make_matcher_predicate(
        [](auto&&) {return true; }
    );
    const auto _y = make_matcher_predicate(
        [](auto&&) {return true; }
    );
    const auto _z = make_matcher_predicate(
        [](auto&&) {return true; }
    );
}
```

Relational operators

```
using namespace simple_match;
using namespace simple_match::placeholders;

int x = 0;

match(x,
    1,          [](){std::cout << "The answer is one\n"; },
    2,          [](){std::cout << "The answer is two\n"; },
    _x < 10,    [](auto&& a) {std::cout << a << " is less than 10\n"; },
    10 < _x < 20, [](auto&& a) {std::cout << a << " is between 10 and 20 exclusive\n"; },
    _,          [](){std::cout << "Did not match\n"; }
);
```

Relational operators

```
namespace placeholders {
    template<class F, class T>
    auto operator<(const matcher_predicate<F>& m, const T& t) {
        return make_matcher_predicate(
            [m, &t](const auto& x) {return m.f_(x) && x < t; }
        );
    }
    template<class F, class T>
    auto operator<(const T& t, const matcher_predicate<F>& m) {
        return make_matcher_predicate(
            [m, &t](const auto& x) {return m.f_(x) && t < x; }
        );
    }
}
```

Design and Implementation of the Library

Extending the core – compound

Recall this example from Rust

```
let tup = (1,0);

match tup{
  (0,0) => println!("both zero"),
  (x,0) => println!("{} and zero",x),
  (0,y) => println!("zero and {}",y),
  _     => println!("did not match")

}
```

Destructuring

- Break down a larger structure into its components
- Can do further pattern matching on each of the components
- Will base this on tuples

FizzBuzz

- Described by Reginald Brathwaite and popularized by Jeff Attwood
- Print numbers from 1 to 100
  - For multiples of 3 print Fizz instead of number
  - For multiples of 5 print Buzz instead of number
  - For multiples of both print FizzBuzz instead of number

FizzBuzz

```
using namespace simple_match;
using namespace simple_match::placeholders;
for (int i = 1; i <= 100; ++i) {
  match(std::make_tuple(i%3,i%5),
    ds(0, 0 ), []() {std::cout << "FizzBuzz\n";},
    ds(0, _),  []() {std::cout << "Fizz\n";},
    ds(_, 0),  []() {std::cout << "Buzz\n";},
    _,        [i]() {std::cout << i << "\n";}
  );
}
```

Tuple

```
template<class... A>
const std::tuple<A...>& simple_match_get_tuple(const std::tuple<A...>& t) {
    return t;
}
template<class... A>
std::tuple<A...>& simple_match_get_tuple(std::tuple<A...>& t) {
    return t;
}
template<class Type>
struct tuple_adapter {
    template<size_t I, class T>
    static decltype(auto) get(T&& t) {
        using namespace simple_match::customization;
        return std::get<I>(simple_match_get_tuple(std::forward<T>(t)));
    }
};
```

Tuple – customization::matcher

```
template<class Type, class... Args>
struct matcher<Type, std::tuple<Args...>> {
    using tu = tuple_adapter<Type>;
    enum { tuple_len = sizeof... (Args)-1 };
```

Tuple – matcher continued

```
template<size_t pos, size_t last>
    struct helper {
        template<class T, class A>
        static bool check(T&& t, A&& a) {
            return match_check(tu::template get<pos>(std::forward<T>(t)),
                std::get<pos>(std::forward<A>(a)))
                && helper<pos + 1, last>::check(std::forward<T>(t), std::forward<A>(a));

        }

        template<class T, class A>
        static auto get(T&& t, A&& a) {
            return std::tuple_cat(match_get(tu::template get<pos>(std::forward<T>(t)),
                std::get<pos>(std::forward<A>(a))),
                helper<pos + 1, last>::get(std::forward<T>(t), std::forward<A>(a)));

        }
    };
```

Tuple – matcher continued

```
template<size_t pos>
   struct helper<pos, pos> {
       template<class T, class A>
       static bool check(T&& t, A&& a) {
           return match_check(tu::template get<pos>(std::forward<T>(t)),
             std::get<pos>(std::forward<A>(a)));

       }
       template<class T, class A>
       static auto get(T&& t, A&& a) {
           return match_get(tu::template get<pos>(std::forward<T>(t)),
             std::get<pos>(std::forward<A>(a)));

       }
   };
```

Tuple – matcher continued

```
    template<class T, class A>
    static bool check(T&& t, A&& a) {
        return helper<0, tuple_len - 1>::check(std::forward<T>(t), std::forward<A>(a));

    }
    template<class T, class A>
    static auto get(T&& t, A&& a) {
        return helper<0, tuple_len - 1>::get(std::forward<T>(t), std::forward<A>(a));

    }

};
```

Destructure

```
namespace detail {
    // differentiate between matcher <T,T> and matcher <T,std::tuple<T...>
    struct tuple_ignorer {};
}

// destructure a tuple or other adapted structure
template<class... A>
auto ds(A&& ... a) {
    return std::make_tuple(std::forward<A>(a)..., detail::tuple_ignorer{});
}
```

Adapting a class/struct to be destructured

```
struct point {
    int x;
    int y;
    point(int x_, int y_) :x(x_), y(y_) {}
};
```

Adapting a class/struct to be destructured

```cpp
using namespace simple_match;
using namespace simple_match::placeholders;

auto m = [](auto&& v) {
    match(v,
        ds(1, 2), []() {std::cout << "one,two\n"; },
        ds(_x, _y), [](int x, int y) {std::cout << x << " " << y << "\n"; },
    );
};


auto tup_12 = std::tuple<int, int>> {1, 2};
auto point_12 = point{ 1, 2 };

m(tup_12);
m(point_12);
```

Adapting a class/struct to be destructured

```
struct point {
    int x;
    int y;
    point(int x_, int y_) :x(x_), y(y_) {}
};


auto simple_match_get_tuple(const point& p) {
    return std::tie(p.x, p.y);
}
```

Design and Implementation of the Library

Pointers

Error handling with optional or Maybe

- Used in functional programming a lot
- A function that can fail for some inputs returns an optional type that either has Nothing or the value it is returning
- You can pattern match to determine which

Optional/Maybe

```haskell
divSafe :: Integral a => a -> a -> Maybe a
divSafe a 0 = Nothing
divSafe a b = Just (a `div` b)

let a = divSafe 4 0
case a of
  Just a -> print a
  Nothing -> print "We divided by zero"
```

Pointers as an optional type

- We can treat pointers as an optional type
- A nullptr means Nothing
- For a valid pointer we extract the value
- We can do further pattern matching on the extracted value

Pointer Example

```
std::unique_ptr<int> nothing;
auto five = std::make_unique<int>(5);
auto ten = std::make_unique<int>(10);
auto twelve = std::make_unique<int>(12);

auto m = [](auto&& v) {
match(v,
  some(5),                [](){std::cout << "five\n"; },
  some(11 <= _x <= 20), [](int x) {std::cout << x << " is on the range [11,20] \n"; },
  some(),                 [](int x) {std::cout << x << "\n"; },
  none(),                 []() {std::cout << "Nothing\n"; }
);
};
m(nothing.get());
m(five.get());
m(ten.get());
m(twelve.get());
```

matcher

```
template<class Type, class Class, class Matcher>
struct matcher<Type, detail::some_t<Class, Matcher>> {
    template<class T, class U>
    static bool check(T&& t, U&& u) {
        return u.check(std::forward<T>(t));
    }
    template<class T, class U>
    static auto get(T&& t, U&& u) {
        return u.get(std::forward<T>(t));
    }

};
```

matcher

```
template<class Type>
struct matcher<Type, detail::none_t> {
    template<class T, class U>
    static bool check(T&& t, U&& u) {
        return u.check(std::forward<T>(t));
    }
    template<class T, class U>
    static auto get(T&& t, U&& u) {
        return u.get(std::forward<T>(t));
    }

};
```

some_t

```
template<class Class, class Matcher>
struct some_t;
```

some_t

```
template<>
struct some_t<void, void> {
    template<class T>
    bool check(T&& t) {
        auto ptr =
customization::pointer_getter<std::decay_t<T>>::get_pointer_no_cast(std::forward<T>(t));
        if (!ptr) return false;
        return true;
    }
    template<class T>
    auto get(T&& t) {
        auto ptr =
customization::pointer_getter<std::decay_t<T>>::get_pointer_no_cast(std::forward<T>(t));
        return std::tie(*ptr);
    }
};
```

some_t

```
struct some_t<void, Matcher> {
    Matcher m_;
    template<class T>
    bool check(T&& t) {
        auto ptr =
customization::pointer_getter<std::decay_t<T>>::get_pointer_no_cast(std::forward<T>(t));
        if (!ptr) return false;
        return match_check(*ptr, m_);
    }
    template<class T>
    auto get(T&& t) {
        auto ptr =
customization::pointer_getter<std::decay_t<T>>::get_pointer_no_cast(std::forward<T>(t));
        return match_get(*ptr, m_);
    }
};
```

none_t

```
struct none_t{
  template<class T>
  bool check(T&& t) {
   // If you get an error here, this means that none() is not supported
    // Example is boost::variant which has a never empty guarantee
    return customization::pointer_getter<std::decay_t<T>>::is_null(std::forward<T>(t));
  }

  template<class T>
  auto get(T&& t) {
    return std::tie();
  }
};
```

pointer_getter

```
namespace customization {
  template<class Type>
  struct pointer_getter<Type*> {
    static auto get_pointer_no_cast(Type* t) {
      return t;
    }
    static auto is_null(Type* t) {
      return !t;
    }
};
```

Functions for use with match

```
inline detail::none_t none() { return detail::none_t{}; }

inline detail::some_t<void, void> some() { return detail::some_t<void, void>{}; }

template<class Matcher>
detail::some_t<void, Matcher> some(Matcher&& m) {
  return detail::some_t<void, Matcher> { std::forward<Matcher>(m) };
}
```

unique_ptr

```
std::unique_ptr<int> nothing;
auto five = std::make_unique<int>(5);
auto ten = std::make_unique<int>(10);
auto twelve = std::make_unique<int>(12);

auto m = [](auto&& v) {
match(v,
  some(5),              []() {std::cout << "five\n"; },
  some(11 <= _x <= 20), [](int x) {std::cout << x << " is on the range [11,20] \n"; },
  some(),               [](int x) {std::cout << x << "\n"; },
  none(),               []() {std::cout << "Nothing\n"; }
);
};
m(nothing);
m(five);
m(ten);
m(twelve);
```

pointer_getter – unique_ptr

```
namespace customization {
  template<class Type, class D>
  struct pointer_getter<std::unique_ptr<Type, D>> {
    template<class T>
    static auto get_pointer_no_cast(T&& t) {
      return t.get();
    }
    template<class T>
    static auto is_null(T&& t) {
      return !t;
    }
  };

}
```

More fun with some

```
struct holder { virtual ~holder() {} };

template<class T>
struct holder_t:holder {
    T value_;
    holder_t(T v) :value_{ std::move(v) } {}
};

template<class T>
auto simple_match_get_tuple(const holder_t<T>& h) {
    return std::tie(t.value_);
}

template<class T>
std::unique_ptr<holder> make_holder(T&& t) {
    return std::make_unique<holder_t<std::decay_t<T>>>(std::forward<T>(t));
}
```

More fun with some

```
using namespace simple_match;
using namespace simple_match::placeholders;
auto m = [](auto&& v) {
  match(v,
      some<holder_t<int>>(ds(5)),  [ ]() {std::cout << "Got five\n";},
      some<holder_t<int>>(ds(_x)), [](auto x) {std::cout << "Got int " << x << "\n";},
      some(), [](auto& x) {std::cout << "Got some other type of holder\n";},
      none(), [](){std::cout << "Got nullptr\n";}
  );
};
auto five = make_holder(5);
auto ten = make_holder(10);
auto pi = make_holder(3.14);
std::unique_ptr<holder> nothing;
m(five);
m(ten);
m(pi);
m(nothing);
```

some_t

```
template<class Class, class Matcher>
struct some_t{
    Matcher m_;
    template<class T>
    bool check(T&& t) {
        auto ptr = customization::pointer_getter<std::decay_t<T>>::template
get_pointer<Class>(std::forward<T>(t));
        if (!ptr) return false;
        return match_check(*ptr, m_);
    }
    template<class T>
    auto get(T&& t) {
        auto ptr = customization::pointer_getter<std::decay_t<T>>::template
get_pointer<Class>(std::forward<T>(t));
        return match_get(*ptr, m_);
    }
};
```

some_t

```
template<class Class>
struct some_t<Class, void> {
    template<class T>
    bool check(T&& t) {
        auto ptr = customization::pointer_getter<std::decay_t<T>>::template
get_pointer<Class>(std::forward<T>(t));
        if (!ptr) return false;
        return true;
    }
    template<class T>
    auto get(T&& t) {
        auto ptr = customization::pointer_getter<std::decay_t<T>>::template
get_pointer<Class>(std::forward<T>(t));
        return std::tie(*ptr);
    }
};
```

pointer_getter

```
namespace customization {
  template<class Type>
  struct pointer_getter<Type*> {
    template<class To>
    static auto get_pointer(Type* t) {
      return dynamic_cast<utils::cv_helper<decltype(t),To>>(t);
    }
    static auto get_pointer_no_cast(Type* t) {
      return t;
    }
    static auto is_null(Type* t) {
      return !t;
    }
};
```

Functions for use with match

```
inline detail::none_t none() { return detail::none_t{}; }

inline detail::some_t<void, void> some() { return detail::some_t<void, void>{}; }

template<class Matcher>
detail::some_t<void, Matcher> some(Matcher&& m) {
  return detail::some_t<void, Matcher> { std::forward<Matcher>(m) };
}

template<class Class, class Matcher>
detail::some_t<Class, Matcher> some(Matcher&& m) {
    return detail::some_t<Class, Matcher> { std::forward<Matcher>(m) };
}
template<class Class>
detail::some_t<Class, void> some() {
    return detail::some_t<Class, void>{ };
}
```

Design and Implementation of the Library

Integrating with Boost

boost::optional

```
boost::optional<int> safe_div(int num ,int denom) {
    using namespace simple_match;
    using namespace simple_match::placeholders;
    return match(std::tie(num, denom),
        ds(_, 0),  []() {return boost::optional<int>{}; },
        ds(_x,_y), [](int x, int y) {return boost::optional<int>{x/y}; }
     );
}
```

boost::optional

```
using namespace simple_match;
using namespace simple_match::placeholders;
auto m = [](auto&& v) {
    return match(v,
       some(), [](auto x) {std::cout << "the safe_div answer is " << x << "\n";},
       none(), []() {std::cout << "Tried to divide by 0 in safe_div\n";}
    );
};

m(safe_div(4, 2));
m(safe_div(4, 0));
```

boost::optional

```
namespace customization {
  template<class Type>
  struct pointer_getter<boost::optional<Type>> {
    template<class T>
      static auto get_pointer_no_cast(T&& t) {
        return t.get_ptr();
      }
      template<class T>
      static auto is_null(T&& t) {
        return !t;
      }
   };
}
```

holder_t is a poor man's version of …

```
struct holder { virtual ~holder() {} };
template<class T>
struct holder_t:holder {
    T value_;
    holder_t(T v) :value_{ std::move(v) } {}
};

template<class T>
auto simple_match_get_tuple(const holder_t<T>& h) {
    return std::tie(t.value_);
}

template<class T>
std::unique_ptr<holder> make_holder(T&& t) {
    return std::make_unique<holder_t<std::decay_t<T>>>(std::forward<T>(t));
}
```

boost::any

- Able to hold any type
- Allows you to query/extract exact type which was previously stored

boost::any

```
using namespace simple_match;
using namespace simple_match::placeholders;
auto m = [](auto&& v) {
   match(v,
     some<int>(5), []() {std::cout << "Got five\n";},
     some<int>(), [](auto x) {std::cout << "Got int " << x << "\n";},
     none(), []() {std::cout << "Got nullptr\n";},
     _, []() {std::cout << "Got some other type of any\n";}
   );
};
auto five = boost::any{5};
auto ten = boost::any{10};
auto pi = boost::any{3.14};
boost::any nothing;
m(five);
m(ten);
m(pi);
m(nothing);
```

boost::any

```
namespace customization {
  template<>
  struct pointer_getter<boost::any> {
    template<class To, class T>
    static auto get_pointer(T&& t) {
      return boost::any_cast<To>(&t);
    }
    template<class T>
    static auto is_null(T&& t) {
      return t.empty();
    }
  };
}
```

What is this?

boost::variant (“Weary Ant”)

- Is a type-safe union
- Allows you to store 1 of a set of types, and get back what you stored
- Can be recursive
- Common example JSON – Null, Number, String, Object, Array

Algebraic Data Types

- Composite types consisting of
  - Sum type – union or variant
  - Product type – struct or tuple
- Use pattern matching to break down an ADT

Rust example

```
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),
     };
}
```

boost::variant example

```
struct add;
struct sub;
struct neg;
struct mul;

using math_variant2_t = boost::variant<boost::recursive_wrapper<add>,
boost::recursive_wrapper<sub>, boost::recursive_wrapper<neg>,
boost::recursive_wrapper<mul>,int >;

struct add :std::tuple<math_variant2_t, math_variant2_t> { using tuple::tuple; };
struct sub :std::tuple<math_variant2_t, math_variant2_t> { using tuple::tuple; };
struct mul :std::tuple<math_variant2_t, math_variant2_t> { using tuple::tuple; };
struct neg :std::tuple<math_variant2_t> { using tuple::tuple; };
```

boost::variant example

```
int eval(const expression& e) {
    using namespace simple_match;
    using namespace simple_match::placeholders;
    return match(e,
        some<add>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) + eval(y); },
        some<sub>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) - eval(y); },
        some<mul>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) * eval(y); },
        some<neg>(ds(_x)),     [](auto&& x) {return -eval(x); },
        some<int>(),            [](auto x) {return x; }
    );
}
```

boost::variant

```
namespace customization {
    template<class... A>
    struct pointer_getter < boost::variant<A...> > {
        template<class To, class T>
        static auto get_pointer(T&& t) {
            return boost::get<To>(&t);
        }
    };
}
```

Multi-methods

```
struct Base { virtual ~Base() {} };
struct Paper:Base {};
struct Rock:Base {};
struct Scissors:Base {};
```

Multi-methods

```
void paper_rock_scissors(const Base* b1, const Base* b2) {
  using namespace simple_match;
  using namespace simple_match::placeholders;

  match(std::tie(b1, b2),
    ds(some<Paper>(), some<Paper>()), [](auto&, auto&){
       std::cout << "Tie with both Paper\n";
    },
    ds(some<Paper>(), some<Rock>()), [](auto&, auto&){
       std::cout << "Winner 1 - Paper covers Rock\n";
    },
    …
  );
}
```

Design and Implementation of the Library

Exhaustive patterns

Non-Exhaustive patterns

```
int eval(const expression& e) {
    using namespace simple_match;
    using namespace simple_match::placeholders;
    return match(e,
        some<add>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) + eval(y); },
        some<sub>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) - eval(y); },
        //some<mul>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) * eval(y); },
        some<neg>(ds(_x)), [](auto&& x) {return -eval(x); },
        some<int>(), [](auto x) {return x; }
    );
}
```

Non-Exhaustive patterns

- Making sure that the pattern match covers all possibilities
- Haskell throws a run-time error
- Rust enforces exhaustive patterns and compile time

Non-Exhaustive patterns

```
int eval(const expression& e) {
    using namespace simple_match;
    using namespace simple_match::placeholders;
    return match(e,
        some<add>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) + eval(y); },
        some<sub>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) - eval(y); },
        //some<mul>(ds(_x, _y)), [](auto&& x, auto&& y) {return eval(x) * eval(y); },
        some<neg>(ds(_x)), [](auto&& x) {return -eval(x); },
        some<int>(), [](auto x) {return x; }
    );
}
```

Visual C++ 2015

```
1>c:\users\johnb\repos\simple_match\include\simple_match\implemen
tation\some_none.hpp(320): error C2338: This type is not in the
match
1>
c:\users\johnb\repos\simple_match\include\simple_match\implementa
tion\some_none.hpp(326): note: see reference to class template
instantiation
'simple_match::detail::not_in_match_asserter<false,First>' being
compiled
1>          with
1>          [
1>              First=mul
1>          ]
```

G++ 5.1

```
C:\Users\johnb\Repos\simple_match\test>g++ --std=c++14 test.cpp
In file included from
../include/simple_match/simple_match.hpp:390:0,
                 from test.cpp:5:
../include/simple_match/implementation/some_none.hpp: In
instantiation of 'struct
simple_match::detail::not_in_match_asserter<false, mul>':
```

How does that work

Non-Exhaustive pattern checking

```
template<class T, class... Args>
auto match(T&& t, Args&&... a) {
    using atypes = typename detail::arg_types<Args...>::type;
    using ec = typename customization::exhaustiveness_checker<std::decay_t<T>>::type;
    using checker = typename ec::template type<atypes>;
    static_assert(checker::value, "Not all types are tested for in match");
    return detail::match_helper(std::forward<T>(t), std::forward<Args>(a)...);
}
```

Non-Exhaustive pattern checking

```
template<class... A>
struct arg_types {};

template<class A, class F>
struct arg_types<A, F> {
    using type = std::tuple<std::decay_t<A>>;
};

template<class A1, class F1, class A2, class F2, class... Args>
struct arg_types<A1, F1, A2, F2, Args...>{
    using type = cat_tuple_t<std::tuple<std::decay_t<A1>, std::decay_t<A2>>,
typename arg_types<Args...>::type>;
};

template<>
struct arg_types<>{
    using type = std::tuple<>;
};
```

Non-Exhaustive pattern checking

```
namespace detail{
    template<class ArgTypes, class... RequiredTypes>
    struct some_exhaustiveness_helper<false, ArgTypes, RequiredTypes... > {
        using some_classes = typename get_some_classes<ArgTypes>::type;
        using a = all_in<some_classes, RequiredTypes... > ;
        static const bool value = a::value;
    };
}

template<class... Types>
struct some_exhaustiveness {
    template<class ArgTypes>
    struct type {
        static const bool v = detail::has_otherwise<ArgTypes>::value;
        using seh = detail::some_exhaustiveness_helper<v, ArgTypes, Types...>;
        static const bool value = seh::value;
    };
};
```

Non-Exhaustive pattern checking

```
template<class... T>
struct some_exhaustiveness_variant_generator<std::tuple<T...>> {
    using type = some_exhaustiveness<T...>;
};

namespace customization {
    template<class... T>
    struct exhaustiveness_checker<boost::variant<T...>> {
        using vtypes = typename
detail::extract_variant_types<boost::variant<T...>>::type;
        using type = typename
detail::some_exhaustiveness_variant_generator<vtypes>::type;
    };

}
```

https://github.com/jbandela/simple_match

Questions?