Pattern Matching

Document #:	D1260R1
Date:	2018-10-09
Project:	Programming Language C++ Evolution
Reply-to:	Michael Park <mcypark@gmail.com>

Contents

1 Revision History

- R0 — Initial Draft:
- R1
  - Removed the binding (@) pattern
  - Foo

2 Introduction

As algebraic data types gain better support in C++ with facilities such as `tuple` and `variant`, the importance of mechanisms to interact with them have increased. While mechanisms such as `apply` and `visit` have been added, their usage is quite complex and limited even for simple cases. Pattern matching is a widely adopted mechanism across many programming languages to interact with algebraic data types that can help greatly simplify C++. Examples of programming languages include text-based languages such as SNOBOL back in the 1960s, functional languages such as Haskell and OCaml, and "mainstream" languages such as Scala, Swift, and Rust.

Inspired by Bjarne's pattern matching presentation from Urbana-Champaign 2015 and P0095 [5] — which proposed pattern matching and language-level variant simultaneously — this paper explores a possible direction for pattern matching only, and does not address language-level variant design. This is in correspondence with a straw poll from Kona 2015, which encouraged exploration of a full solution for pattern matching. SF: 16, WF: 6, N: 5, WA: 1, SA: 0.

3 Motivation and Scope

Virtually every program involves branching on some predicates applied to a value and conditionally binding names to some of its components for use in subsequent logic. Today, C++ provides two types of selection statements: the `if` statement and the `switch` statement.

Since `switch` statements can only operate on a *single* integral value and `if` statements operate on an *arbitrarily* complex boolean expression, there is a significant gap between the two constructs even in inspection of the "vocabulary types" provided by the standard library.

In C++17, structured binding declarations [9] introduced the ability to concisely bind names to components of `tuple`-like values. The proposed direction of this paper aims to naturally extend this notion by performing **structured inspection** prior to forming the **structured bindings** with a third selection statement: the `inspect` statement. The goal of the `inspect` statement is to bridge the gap between `switch` and `if` statements with a **declarative**, **structured**, **cohesive**, and **composable** mechanism.

4 Before/After Comparisons

4.1 Matching Integrals

Before	After

```cpp
switch (x) {
  case 0: std::cout << "got zero"; break;
  case 1: std::cout << "got one"; break;
  default: std::cout << "don't care";
}
```

```cpp
inspect (x) {
  0: std::cout << "got zero";
  1: std::cout << "got one";
  _: std::cout << "don't care";
}
```

4.2 Matching Strings

Before	After

```cpp
if (s == "foo") {
  std::cout << "got foo";
} else if (s == "bar") {
  std::cout << "got bar";
} else {
  std::cout << "don't care";
}
```

```cpp
inspect (s) {
  "foo": std::cout << "got foo";
  "bar": std::cout << "got bar";
  _: std::cout << "don't care";
}
```

4.3 Matching Tuples

Before	After

```cpp
auto&& [x, y] = p;
if (x == 0 && y == 0) {
  std::cout << "on origin";
} else if (x == 0) {
  std::cout << "on y-axis";
} else if (y == 0) {
  std::cout << "on x-axis";
} else {
  std::cout << x << ',' << y;
}
```

```cpp
inspect (p) {
  [0, 0]: std::cout << "on origin";
  [0, y]: std::cout << "on y-axis";
  [x, 0]: std::cout << "on x-axis";
  [x, y]: std::cout << x << ',' << y;
}
```

4.4 Matching Variants

Before	After
| ```cpp
struct visitor {
  void operator()(int i) const {
    os << "got int: " << i;
  }
  void operator()(float f) const {
    os << "got float: " << f;
  }
  std::ostream& os;
};
std::visit(visitor{strm}, v);
``` | ```cpp
inspect (v) {
  <int> i: strm << "got int: " << i;
  <float> f: strm << "got float: " << f;
}
``` |

4.5 Matching Polymorphic Types

```cpp
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

Before	After
| ```cpp
virtual int Shape::get_area() const = 0;


int Circle::get_area() const override {
  return 3.14 * radius * radius;
}
int Rectangle::get_area() const override {
  return width * height;
}
``` | ```cpp
int get_area(const Shape& shape) {
  inspect (shape) {
    (as<Circle> ? [r]): return 3.14 * r * r;
    (as<Rectangle> ? [w, h]): return w * h;
  }
}
``` |

4.6 Evaluating Expressions

```cpp
struct Expr;
struct Neg { std::shared_ptr<Expr> expr; };
struct Add { std::shared_ptr<Expr> lhs, rhs; };
struct Mul { std::shared_ptr<Expr> lhs, rhs; };
struct Expr : std::variant<int, Neg, Add, Mul> { using variant::variant; };

namespace std {
  template <>
  struct variant_size<Expr> : variant_size<Expr::variant> {};
```

```
  template <std::size_t I>
  struct variant_alternative<I, Expr> : variant_alternative<I, Expr::variant> {};
}
```

Before	After

```
int eval(const Expr& expr) {
  struct visitor {
    int operator()(int i) const {
      return i;
    }
    int operator()(const Neg& n) const {
      return -eval(*n.expr);
    }
    int operator()(const Add& a) const {
      return eval(*a.lhs) + eval(*a.rhs);
    }
    int operator()(const Mul& m) const {
      return eval(*m.lhs) * eval(*m.rhs);
    }
  };
  return std::visit(visitor{}, expr);
}
```

```
int eval(const Expr& expr) {
  inspect (expr) {
    <int> i: return i;
    <Neg> [e]: return -eval(*e);
    <Add> [l, r]: return eval(*l) + eval(*r);
    <Mul> [l, r]: return eval(*l) * eval(*r);
  }
}
```

5 Design Overview

5.1 Basic Syntax

```
inspect constexpr_opt ( init-statement_opt condition ) {
    pattern guard_opt : statement
    pattern guard_opt : statement
    ...
}

guard:
    if ( expression )
```

5.2 Basic Model

Within the parentheses, the `inspect` statement is equivalent to `switch` and `if` statements except that no conversion nor promotion takes place in evaluating the value of its condition.

When the `inspect` statement is executed, its condition is evaluated and matched in order (first match semantics) against each pattern. If a pattern successfully matches the value of the condition and the boolean expression in the guard evaluates to `true` (or if there is no guard at all), control is passed to the statement following the matched pattern label. If the guard expression evaluates to `false`, control flows to the subsequent pattern. If no pattern matches, none of the statements are executed.

5.3   Types of Patterns

5.3.1   Primary Patterns

5.3.1.1   Wildcard Pattern

The wildcard pattern has the form:

    _

and matches any value `v`.

```
int v = /* ... */;

inspect (v) {
    _: std::cout << "ignored";
//  ^ wildcard pattern
}
```

[*Note:* Even though `_` is a valid identifier, it does not introduce a name.]

5.3.1.2   Identifier Pattern

The identifier pattern has the form:

    identifier

and matches any value `v`. The introduced name behaves as an lvalue referring to `v`, and is in scope from its point of declaration until the end of the statement following the pattern label.

```
int v = /* ... */;

inspect (v) {
    x: std::cout << x;
//  ^ identifier pattern
}
```

[*Note:* If the identifier pattern is used as a top-level pattern, it has the same syntax as a `goto` label.]

5.3.1.3   Constant Pattern

The constant pattern has the form:

    constant expression

and matches value `v` if a call to member `c.match(v)` or else a non-member ADL-only `match(c, v)` is contextually convertible to `bool` and evaluates to `true` where `c` is the *constant expression*.

The following is the default behavior of `match(x, y)`.

```
template <typename T, typename U>
constexpr auto match(T&& lhs, U&& rhs)
        -> decltype(std::forward<T>(lhs) == std::forward<U>(rhs)) {
    return std::forward<T>(lhs) == std::forward<U>(rhs);
}
```

```cpp
int v = /* ... */;

inspect (v) {
    0: std::cout << "got zero";
    1: std::cout << "got one";
//  ^ constant pattern
}
```

[*Note:* (`id`) is needed to disambiguate with the identifier pattern.]

```cpp
static constexpr int zero = 0, one = 1;
int v = /* ... */;

inspect (v) {
    (zero): std::cout << "got zero";
    (one): std::cout << "got one";
//  ^^^^^ constant pattern
}
```

5.3.2 Compound Patterns

5.3.2.1 Structured Binding Pattern

The structured binding pattern has the form:

$$[\ pattern_0,\ pattern_1,\ \ldots,\ pattern_N\ ]$$

and matches value v if each $pattern_i$ matches the ith component of v. The components of v are given by the structured binding declaration: `auto&& [__e`$_0$`, __e`$_1$`, ..., __e`$_N$`] = v;` where each `__e`$_i$ are unique exposition-only identifiers.

```cpp
std::pair<int, int> p = /* ... */;

inspect (p) {
    [0, 0]: std::cout << "on origin";
    [0, y]: std::cout << "on y-axis";
//      ^ identifier pattern
    [x, 0]: std::cout << "on x-axis";
//      ^ constant pattern
    [x, y]: std::cout << x << ',' << y;
//  ^^^^^^ structured binding pattern
}
```

5.3.2.2 Alternative Pattern

The alternative pattern has the form:

    `<` `auto` `>` *pattern*
    `<` *concept* `>` *pattern*
    `<` *type* `>` *pattern*
    `<` *constant expression* `>` *pattern*

Let `v` be the value being matched and `V` be `std::remove_cvref_t<decltype(v)>`.
Let `Alt` be the entity inside the angle brackets.

If `std::variant_size_v<V>` is well-formed and evaluates to an integral, the alternative pattern matches `v` if `Alt` is compatible with the current index of `v` and *pattern* matches the active alternative of `v`.

Let `I` be the current index of `v` given by a member `v.index()` or else a non-member ADL-only `index(v)`. The active alternative of `v` is given by `std::variant_alternative_t<I, V>&` initialized by a member `v.get<I>()` or else a non-member ADL-only `get<I>(v)`.

`Alt` is compatible with `I` if one of the following four cases is true:

- `Alt` is `auto`
- `Alt` is a *concept* and `std::variant_alternative_t<I, V>` satisfies the *concept*.
- `Alt` is a *type* and `std::is_same_v<Alt, std::variant_alternative_t<I, V>>` is `true`
- `Alt` is a *constant expression* that can be used in a `switch` and is the same value as `I`.

Before	After

```
std::visit([&](auto&& x) {
  strm << "got auto: " << x;
}, v);
```

```
inspect (v) {
   <auto> x: strm << "got auto: " << x;
}
```

```
std::visit([&](auto&& x) {
  using X = std::remove_cvref_t<decltype(x)>;
  if constexpr (C1<X>()) {
    strm << "got C1: " << x;
  } else if constexpr (C2<X>()) {
    strm << "got C2: " << x;
  }
}, v);
```

```
inspect (v) {
   <C1> c1: strm << "got C1: " << c1;
   <C2> c2: strm << "got C2: " << c2;
}
```

```
std::visit([&](auto&& x) {
  using X = std::remove_cvref_t<decltype(x)>;
  if constexpr (std::is_same_v<int, X>) {
    strm << "got int: " << x;
  } else if constexpr (
      std::is_same_v<float, X>) {
    strm << "got float: " << x;
  }
}, v);
```

```
inspect (v) {
   <int> i: strm << "got int: " << i;
   <float> f: strm << "got float: " << f;
}
```

```
std::variant<int, int> v = /* ... */;

std::visit([&](int x) {
  strm << "got int: " << x;
}, v);
```

```
std::variant<int, int> v = /* ... */;

inspect (v) {
   <int> x: strm << "got int: " << x;
}
```

Before	After
| ```cpp
std::variant<int, int> v = /* ... */;

std::visit([&](auto&& x) {
  switch (v.index()) {
    case 0: {
      strm << "got first: " << x;
      break;
    }
    case 1: {
      strm << "got second: " << x;
      break;
    }
  }
}, v);
``` | ```cpp
std::variant<int, int> v = /* ... */;

inspect (v) {
  <0> x: strm << "got first: " << x;
  <1> x: strm << "got second: " << x;
}
``` |

5.3.2.3 Extractor Pattern

The extractor pattern has the form:

(*constant expression* ? *pattern*)

Let `e` be the result of a call to member `c.extract(v)` or else a non-member ADL-only `extract(c, v)` where `c` is the *constant expression*.

The extractor pattern matches value `v` if `e` is contextually convertible to `bool` and evaluates to `true` and *pattern* matches `*e`.

```cpp
struct {
    std::optional<std::array<std::string_view, 2>> extract(std::string_view sv) const;
} email;

struct {
    std::optional<std::array<std::string_view, 3>> extract(std::string_view sv) const;
} phone_number;

inspect (s) {
    (email ? [address, domain]): std::cout << "got an email";
    (phone_number ? ["415", _, _]): std::cout << "got a San Francisco phone number";
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ extractor pattern
}
```

5.3.2.4 As Pattern

The as pattern is a special instance of the extractor pattern, and behaves as:

```cpp
template <typename Derived>
struct As {
    template <typename Base>
    auto* extract(Base& base) const {
```

9

```
        static_assert(std::is_polymophic_v<Base>);
        static_assert(std::is_convertible_v<Derived*, Base*>,
                      "cross-casts are not allowed.");
        using R = /* `Derived` with the same _cv_-qualification as `Base` */;
        return dynamic_cast<R*>(&base);
    }
};

template <typename Derived>
inline constexpr As<Derived> as;
```

While this is a possible library implementation, it will likely benefit from being implemented as a compiler intrinsic for optimization opportunities.

N3449 [8] describes techniques involving vtable pointer caching and hash conflict minimization that are implemented in the Mach7 [6] library, but also mentions further opportunities available for a compiler solution.

Given the following definition of a `Shape` class hierarchy:

```
struct Shape { virtual ~Shape() = default; };

struct Circle : Shape { int radius; };
struct Rectangle : Shape { int width, height; };
```

Before	After
``virtual int Shape::get_area() const = 0;

int Circle::get_area() const override {
  return 3.14 * radius * radius;
}

int Rectangle::get_area() const override {
  return width * height;
}``` | ```int get_area(const Shape& shape) {
  inspect (shape) {
    (as<Circle> ? [r]): return 3.14 * r * r;
    (as<Rectangle> ? [w, h]): return w * h;
//   ^^^^^^^^^^^^^^^^^^^^^^^^^ as pattern
  }
}``` |

5.4   Pattern Guard

The pattern guard has the form:

> if (*expression*)

Let `e` be the result of *expression* contextually converted to `bool`. If `e` is `true`, control is passed to the corresponding statement. Otherwise, control flows to the subsequent pattern.

The pattern guard allows to perform complex tests that cannot be performed within the *pattern*. For example, performing tests across multiple bindings:

```
inspect (p) {
    [x, y] if (test(x, y)): std::cout << x << ',' << y << " passed";
```

```
//         ^^^^^^^^^^^^^ pattern guard
}
```

This also diminishes the desire for fall-through semantics within the statements, an unpopular feature even in `switch` statements. For the reified semantics of the pattern guard, consider the following snippet:

```
switch (x) {
    case c1: if (cond1) { stmt1; break; } [[fallthrough]]
    case c2: if (cond2) { stmt2; break; } [[fallthrough]]
}
```

5.5 `inspect constexpr`

Note that every *pattern* is able to determine whether it matches value `v` as a boolean expression in isolation.

Let `MATCHES` be the condition for which a *pattern* matches a value `v`. Ignoring any potential optimization opportunities, we're able to perform the following transformation:

inspect	if
```inspect (v) { pattern1 if (cond1): stmt1 pattern2: stmt2 // ... }```	```if (MATCHES(pattern1. v) && cond1) stmt1 else if (MATCHES(pattern2, v)) stmt2 // ...```

`inspect constexpr` is then formulated by applying `constexpr` to every `if` branch.

inspect constexpr	if constexpr
```inspect constexpr (v) { pattern1 if (cond1): stmt1 pattern2: stmt2 // ... }```	```if constexpr (MATCHES(pattern1, v) && cond1) stmt1 else if constexpr (MATCHES(pattern2, v)) stmt2 // ...```

5.6 Exhaustiveness and Usefulness Checking

`inspect` can be declared `[[strict]]` for implementation-defined exhaustiveness and usefulness checking.

Exhaustiveness means that all values of the type of the value being matched is handled by at least one of the cases. For example, having a `_:` case makes any `inspect` statement exhaustive.

Usefulness means that every case handles at least one value of the type of the value being matched. For example, any case that comes after a `_:` case would be useless.

Warnings for pattern matching [2] discusses and outlines an algorithm for exhaustiveness and usefulness for OCaml, and is the algorithm used by Rust.

6 Proposed Wording

The following is the beginning of an attempt at a syntactic structure.

Add to **§8.4 [stmt.select]** of . . .

1   Selection statements choose one of several flows of control.

> *selection-statement:*
>   if constexpr$_{opt}$ (*init-statement$_{opt}$ condition*) *statement*
>   if constexpr$_{opt}$ (*init-statement$_{opt}$ condition*) *statement* else *statement*
>   switch (*init-statement$_{opt}$ condition*) *statement*
>   inspect constexpr$_{opt}$ (*init-statement$_{opt}$ condition*) { *inspect-case-seq* }
>
> *inspect-case-seq:*
>   *inspect-case*
>   *inspect-case-seq inspect-case*
>
> *inspect-case:*
>   *attribute-specifier-seq$_{opt}$ inspect-pattern inspect-guard$_{opt}$* : *statement*
>
> *inspect-pattern:*
>   *wildcard-pattern*
>   *identifier-pattern*
>   *constant-pattern*
>   *structured-binding-pattern*
>   *alternative-pattern*
>   *binding-pattern*
>   *extractor-pattern*
>
> *inspect-guard:*
>   if (*expression*)

7 Design Decisions

7.1 Extending Structured Bindings Declaration

The design is intended to be consistent and to naturally extend the notions introduced by structured bindings. That is, The subobjects are **referred** to rather than being assigned into new variables.

7.2 `inspect` rather than `switch`

This proposal introduces a new `inspect` statement rather than trying to extend the `switch` statement. P0095R0 [4] had proposed extending `switch` and received feedback to "leave `switch` alone" in Kona 2015.

The following are some of the reasons considered:

- `switch` allows the `case` labels to appear **anywhere**, which hinders the goal of pattern matching in providing **structured** inspection.

- The fall-through semantics of `switch` generally results in `break` being attached to every case, and is known to be error-prone.
- `switch` is purposely restricted to integrals for **guaranteed** efficiency. The primary goal of pattern matching in this paper is expressiveness while being at least as efficient as the naively hand-written code.

7.3 First Match rather than Best Match

The proposed matching algorithm has first match semantics. The choice of first match is mainly due to complexity. Our overload resolution rules for function declarations are extremely complex and is often a mystery.

Best match via overload resolution for function declarations are absolutely necessary due to the non-local and unordered nature of declarations. That is, function declarations live in different files and get pulled in via mechanisms such as `#include` and `using` declarations, and there is no defined order of declarations like Haskell does, for example. If function dispatching depended on the order of `#include` and/or `using` declarations being pulled in from hundreds of files, it would be a complete disaster.

Pattern matching on the other hand do not have this problem because the construct is local and ordered in nature. That is, all of the candidate patterns appear locally within `inspect (x) { /* ... */ }` which cannot span across multiple files, and appear in a specified order. Note that this is consistent with `try`/`catch` for the same reasons: locality and order.

Consider also the amount of limitations we face in overload resolution due to the opacity of user-defined types. `T*` is related to `unique_ptr<T>` as it is to `vector<T>` as far as the type system is concerned. This limitation will likely be even bigger in a pattern matching context with the amount of customization points available for user-defined behavior.

7.4 Statement rather than Expression

This paper diverges from P0095 [5] in that it proposes to add `inspect` as a statement only rather than trying to double as a statement and an expression. The main reason here is that the semantic differences between the statement and expression forms are not trivial.

- In the situation where none of the cases match, the statement form simply skips over the entire statement à la `switch`, whereas the expression form throws an exception since it is required to yield a value.
- Resulting type of the statement form of `inspect` within an "immediately-invoked-lambda" is required to be explicitly specified, or is determined by the first `return` statement. In contrast, the expression form will probably need to use `std::common_type_t<Ts...>` where `Ts...` are types of N expressions to be consistent with the ternary operator.

While an expression form of `inspect` would be useful, the author believes that it can and should be introduced later, with different enough syntax such as `x inspect { p1 => e1, p2 => e2 }`. The proposed syntax of the `inspect` statement in this paper consistent with every other statement in C++ today.

7.5 Language rather than Library

There are three popular pattern matching libraries for C++ today: Mach7 [6], MPark.Patterns [3], and simple_match [1].

While the libraries have been useful for gaining experience with implementation and cleaner interfaces, the issue of introducing identifiers, syntactic overhead of the patterns, and the reduced optimization opportunities justify support as a language feature from a usability standpoint.

8   Runtime Performance

The following are few of the optimizations that are worth noting.

8.0.1   Structured Binding Pattern

Structured binding patterns can be optimized by performing `switch` over the columns with the duplicates removed, rather than the naive approach of performing a comparison per element. This removes unnecessary duplicate comparisons that would be performed otherwise. This would likely require some wording around "comparison elision" in order to enable such optimizations.

8.0.2   Alternative Pattern

The sequence of alternative patterns can be executed in a `switch`.

8.0.3   Open Class Hierarchy

N3449 [8] describes techniques involving vtable pointer caching and hash conflict minimization that are implemented in the Mach7 [6] library, but also mentions further opportunities available for a compiler solution.

9   Examples

// TODO

10   Future Work

10.1   Language Support for Variant

The design of this proposal also accounts for a potential language support for variant. It achieves this by keeping the alternative pattern flexible for new extensions via **<** *new_entity* **>** *pattern*.

Consider an extension to `union` that allows it to be tagged by an integral, and has proper lifetime management such that the active alternative need not be destroyed manually.

```
// `: type` specifies the type of the underlying tag value.
union U : int { char small[32]; std::vector<char> big; };
```

We could then allow **<** *qualified-id* **>** that refers to a `union` alternative to support pattern matching.

```
U u = /* ... */;

inspect (u) {
  <U::small> s: std::cout << s;
  <U::big> b: std::cout << b;
}
```

The main point is that whatever entity is introduced as the discriminator, the presented form of alternative pattern should be extendable to support it.

10.2   Patterns in range-based `for` loop

```
for (auto&& [0, y] : points) {
  // only operate on points on the y-axis.
}
```

Structured binding declaration is allowed in range-based `for` loop:

```
for (auto&& [x, y] : points) { /* ... */ }
```

The `[x, y]` part can also be a pattern of an `inspect` statement rather than a structured binding declaration.

Before	After
```for (auto&& p : points) { auto&& [x, y] = p; // ... }```	```for (auto&& p : points) { inspect (p) { [x, y]: // ... } }```

With this model, allowing patterns directly in range-based `for` loop becomes natural.

Code	Expanded
```for (auto&& [0, y] : points) { // only points on the y-axis. }```	```for (auto&& p : points) { inspect (p) { [0, y]: // ... } // falls through if no match }```

10.3   Note on Ranges

The benefit of pattern matching for ranges is unclear. While it's possible to come up with a ranges pattern, e.g., `{x, y, z}` to match against a fixed-size range, it's not clear whether there is a worthwhile benefit.

The typical pattern found in functional languages of matching a range on head and tail doesn't seem to be all that common or useful in C++ since ranges are generally handled via loops rather than recursion.

Ranges likely will be best served by the range adaptors / algorithms, but further investigation is needed.

11   Acknowledgements

References

[1] John Bandela. Simple, Extensible C++ Pattern Matching Library. Retrieved from https://github.com/jbandela/simple_match

[2] Luc Maranget. Warnings for pattern matching. Retrieved from http://moscova.inria.fr/~maranget/papers/warn/index.html

[3] Michael Park. Pattern Matching in C++. *MPark.Patterns*. Retrieved from https://github.com/mpark/patterns

[4] David Sankel. 2015. Pattern Matching and Language Variants. *P0095R0*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0095r0.html

[5] David Sankel. 2016. Pattern Matching and Language Variants. *P0095R1*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0095r1.html

[6] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Mach7: Pattern Matching for C++. *Mach7*. Retrieved from https://github.com/solodon4/Mach7

[7] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open Pattern Matching for C++. Retrieved from http://www.stroustrup.com/OpenPatternMatching.pdf

[8] Bjarne Stroustrup. 2012. Open and Efficient Type Switch for C++. *N3449*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3449.pdf

[9] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. 2016. Structured bindings. *P0144*. Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0144r2.pdf