A New Approach For Compiling C++

Document #:	D1371R4
Date:	2023-09-19
Project:	Programming Language C++
Audience:	SG15

Reply-to:	HassanSajjad <hassan.sajjad069@gmail.com>

Contents

1 Abstract

This paper specifies the API for the interaction between the build system and the compiler where the compiler is available as a shared library. This results in better compilation speed.

2 Changes

2.1 R0 (Initial)

3 Introduction

A very brief high level view of your proposal, understandable by C++ committee members who are not necessarily experts in whatever domain you are addressing.

Today, almost all compilation happens by user or build system invoking the compiler executable. This paper proposes for the availability of the compiler as a shared library. Both the compiler executable and the compiler shared library can co-exist. The build tool can then interact with the shared library with the API specified in this paper. Compared to the current approach, this will result in faster compilation speed, close to 25 - 40 % in some cases.

4 Motivation and Scope

Why is this important? What kinds of problems does it address? What is the intended user community? What level of programmers (novice, experienced, expert) is it intended to support? What existing practice is it based on? How widespread is its use? How long has it been in use? Is there a reference implementation and test suite available for inspection?

Most operating systems today support dynamic loading of the shared library. By making compiler a shared library, we can achieve faster build speeds.

4.1 Why is this faster

i) API allows the build system to intercept the files read by the compiler. So, the compilation of multiple compilation units can use the same cached file.

ii) Module files need to be scanned to determine the dependencies of the file. Only after that the files could be compiled in order. In this approach, the scanning is not needed.

4.2 How much faster this could be?

Tests conducted imitating real world scenarios revealed that it could be `25 - 40 %` faster https://github.com/HassanSajjad-302/solution5.

Tests were performed with C++20 MSVC compiler on Windows 11 operating system on modern hardware at the time of writing. Repositories used include SFML and LLVM.

The highlights include.

i) Estimated scanning time percent of the total compilation time (scanning time + compilation time) per compilation unit for LLVM is `8.46%`.

ii) SFML was compiled with C++20 header units. Because compilation became faster, the scanning time took a larger proportion of the total time. Scanning took `25.72%` of the total compilation time. For few files scanning was actually slower than compilation.

iii) Estimated average scanning time for the project LLVM was `217.9ms`. For some files it was more than `400ms`.

iv) On average the LLVM source file includes `400` header files. `3598` unique header files are read from the disk while compiling `2854` source files. If LLVM is built with C++20 modules, there will be `2854 + 3598` process launches instead of `2854`. Few compilers use `two phase` model instead of `one phase`. Described here https://gitlab.kitware.com/cmake/cmake/-/issues/18355#note_1329192. In such a case, there will be `2854 + (2 * 3598)` process launches instead of `2854`. By avoiding these costs of process setup and file reads should result in `1 - 2 %` compilation speed-up in a clean build in the project size of LLVM.

5 Impact On the Standard

What other library components does does it depend on, and what depends on it? Is it a pure extension, or does it require changes to standard components? Can it be implemented using current C++ compilers and libraries, or does it require language or library features that are not part of C++ today?

6 Design Decisions

6.0.1

Why did you choose the specific design that you did? What alternatives did you consider, and what are the tradeoffs? What are the consequences of your choice, for users and implementers? What decisions are left up to implementers? If there are any similar libraries in use, how do their design decisions compare to yours?

6.1 What are the tradeoffs?

Memory consumption of such an approach could be higher than the current approach. That is because more `compilerState` and the cached files need to be kept in the memory.

However, this can be alleviated by the following:

Few build systems generally group files together in a `target`. A `target` can depend on one or more other `target`. Such build systems can alleviate the higher memory consumption by ordering the compilation of files of the dependency before the dependent target. This way only the `compileState` of the files of the dependency target need to be kept in the memory. `compilerState` of the files of the dependent target only come in the picture once the dependent target files are compiled. ifcfile of a `target` are kept in the memory until there is no file of any dependent target left to be compiled. At which point, this is cleared from the memory.

In case the similar file is being read by multiple compilations, the memory consumption could be a little less than the current approach. Because all such compilations can use one cached read instead of reading themselves.

6.2 Consequences of this choice for the current build systems

This proposal does not impact the current build systems in any way. Few build systems, however, might need non-trivial changes to support this. Plans for supporting this in the build system HMake are described here: https://lists.isocpp.org/sg15/att-2033/Analysis.pdf

7 Technical Specifications

The committee needs technical specifications to be able to fully evaluate your proposal. Eventually these technical specifications will have to be in the form of full text for the standard or technical report, often known as "Standardese", but for an initial proposal there are several possibilities:

Provide some limited technical documentation. This might be OK for a very simple proposal such as a single function, but for anything beyond that the committee will likely ask for more detail. Provide technical documentation that is complete enough to fully evaluate your proposal. This documentation can be in the proposal itself or you can provide a link to documentation available on the web. If the committee likes your proposal, they will ask for a revised proposal with formal standardese wording. The committee recognizes that writing the

formal ISO specification for a library component can be daunting and will make additional information and help available to get you started. Provide full "Standardese." A standard is a contract between implementers and users, to make it possible for users to write portable code with specified semantics. It says what implementers are permitted to do, what they are required to do, and what users can and can't count on. The "standardese" should match the general style of exposition of the standard, and the specific rules set out in the Specification Style Guidelines, but it does not have to match the exact margins or fonts or section numbering; those things will all be changed anyway if the proposal gets accepted into the working draft for the next C++ standard.

Instead of an executable, the compiler could be a shared-library. Or there could be a separate driver supporting the functionality described here. Compilation pause and resume capability needs to be built into the compiler. HMake loads this driver dll or the compiler dll. Now it calls the following symbols `newCompile` and `resumeCompile`.

```
struct String
{
    const char *ptr;
    unsigned long size;
};

// Those char pointers that are pointing to the path are platform dependent i.e. whcar_t* in-case of Wind
struct CompileOutput
{
    // if (!completed && !errorOccurred), then pointer to the compiler state to be preserved by the build
    // nullptr
    void *compilerState;

    // if (completed || errorOccurred), then compiler output and errorOutput, else nullptr
    String stdout;
    String stderr;

    // if (!completed && !errorOccurred), then the header-unit path the compiler is waiting on if any,
    // else if(completed && !errorOccurred), then the pointer to the returned ifcFile if any, else
    // nullptr
    String ifcFileOrHeaderUnitPath;

    // if (!completed && !errorOccurred), then the name of the module the compiler is waiting on if any,
    // else if(completed && !errorOccurred), then the pointer to the returned objectFile if any, else
    // nullptr
    String objectFileOrModuleName;

    // if completed == true, then the logicalName of exported module if any.
    String logicalName;

    // Following is the array size, next is the array of header-includes.
    unsigned long headerIncludesCount;
    String *headerIncludes;

    // true if compilation succeeded, false otherwise
    bool completed;

    // true if an error occurred
    bool errorOccurred;
};

CompileOutput newCompile(String compileCommand,
```

```
                          String (*getFileContents)(String filePath));
CompileOutput resumeCompile(void *compilerState, String ifcFile);
String getObjectFile(String ifcFile);
```

The compiler calls `newCompile` function passing it the compile-command for the module file. The compile command, however, does not include any dependencies. If the compiler sees an import of a module, it sets the `objectFileOrModuleName` String of the `CompileOutput` return value. Same is the case for header-units but as mentioned in declaration, ifcFileOrHeaderUnitPath is used. The build-system now will preserve the `compilerState` and will check if the required file is already built, or it needs to be built, or it is being built. Only after the file is available, the build-system will call `resumeCompile` function passing it the ifcFile. `resumeCompile` is called until file has no dependency not provided and the compilation succeeds. If only the ifcFile is returned and no objectFile on compilation completion, the build-system assumes that the compiler is using 2-phase model. In this case, on a different thread it will call `getObjectFile` to get objectFile. Distinction between the two models is discussed here: https://gitlab.kitware.com/cmake/cmake/-/issues/18355#note_1329192. The argument `getFileContents` is used by the compiler to get file contents of any file instead of reading itself. This means that a file does not get read twice for different compilations. As compilation completes, build-system will write ifc and object files to the memory as-well.

This solution does not have the drawbacks of the above solution with the added advantages that:

1) Files are directly compiled instead of being scanned first before compilation.
2) Build-system keeps the files in the cache. As build-systems spend a lot of time reading from disk, an often used technique to improve speed is to launch more threads than supported by hardware. But this might not be an effective solution anymore as building modules is not embarrassingly parallel.

8 Acknowledgements

9 References

The template above is based on the one in N3370 Call for Library Proposals, which also has some other tips for writing a good library proposal. Please note that the "Submission procedures" section in that document are outdated and should not be used. The start of this page describes the new procedures.

10 Motivation

One of the most contentious and difficult design problem to solve in introducing pattern matching has been the issue of dealing with identifiers. Specifically, whether an identifier in a pattern introduces a new name or refers to an existing name. This paper aims to address this concern by making names refer to existing names by default, and use `let` as the introducer for new names.

Another key goal that Herb Sutter aims for in [P2392R0] is the ability for pattern syntax to be used outside of `inspect`. More specifically, the goal is to be able to perform pattern matching easily in boolean-expecting contexts such as `if` and `requires`. This paper aims to address this concern with the *expr* `match` *pattern* expression.

11 History

[P1371R0] proposed `id` to introduce a new name, and for `^id` to refer to an existing name. [P1371R1] removed the `^` due to feedback about it being a Microsoft C++ extension for smart pointers, as well as its effect on code performing simple enumeration matching.

```
enum Color { Red, Green, Blue };

inspect (e) {
```

```
  ^Red   => // I'm just trying to
  ^Green => // match simple enums...
  ^Blue  => // What's with these carets?
}
```

[P1371R1] and [P1371R2] kept `id` introducing a new name, but replaced `^id` with `case id` to refer to an existing name and `let id` to explicitly introduce a new name.

One of the motivations for this was to make simple code such as enumeration matching familiar again:

```
enum Color { Red, Green, Blue };

inspect (e) {
  case Red   => // Huh, just looks like `switch` now.
  case Green => // ...
  case Blue  => // ...
}
```

The `case` and `let` applied recursively to identifiers to allow patterns such as `case [a, b]` to match against existing `a` and `b`, and `let [x, y]` to introduce new names `x` and `y`. This recursive behavior received positive feedback from EWG in Cologne 2019:

"We support the authors' direction of `let` and `case` modes applying to subpatterns.": SF: 7, F: 7, N: 5, A: 4, SA: 0

The problem with this approach appeared to be due to the overriding behavior of the `case` and `let`. For example, constructs such as `case [a, b, let x]` were allowed to match against existing `a` and `b` for the first two elements and bind `x` to the third.

In more complex examples however, it becomes more difficult to parse through which names are new and which refer to existing.

For example in `let [a, case [let b, c, d], [e]]`, `a`, `b`, `e` are new names, `c`, `d` refer to an existing names.

In response, [P1371R3] chose to keep `case` and drop `let`. Because the top-level already had `let` behavior in that it introduced new names, we already effectively had "recursive `let`". We really only needed to provide a way to specify an existing name. This led to `case` becoming non-recursive.

At this point, there were several push backs:

— "We shouldn't to bifurcate expressions like this."

Some expressions didn't need `case` (e.g. `0`), but some did (e.g. `id`). If `case expr` is pattern matching syntax for expressions, that would at least be consistent.

This paper aims to address this concern by not requiring `case` for expressions at all.

— "Declarations of new names should have an introducer like most other places."

The comment is in regard to code like this:

```
inspect (e) {
  x => ...
}
```

Some people found it surprising that that would introduce a new name, as most identifiers are introduced with an introducer. Notable exceptions being lambda captures and structured bindings. Even then, `[x, y]() {}` doesn't introduce `x` and `y` out of thin-air, it captures existing `x` and `y` at the same time. Structured bindings at least has the `auto` in front, like: `auto [x, y]` which doesn't apply to the `x` and `y` at all, but does indicate that we're in a declaration context.

This paper aims to address this concern by introducing a recursive `let` introducer for new names.

12 Comparison Tables

12.1 Matching Integrals

C++20

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

R3

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

R4

```
x match {
  0 => { std::cout << "got zero"; }
  1 => { std::cout << "got one"; }
  _ => { std::cout << "don't care"; }
};
```

P2392R0

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

12.2 Matching Strings

C++20

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

R3

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

R4

```
c match {
  "foo" => { std::cout << "got foo"; }
  "bar" => { std::cout << "got bar"; }
  _ => { std::cout << "don't care"; }
};
```

P2392

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

12.3 Matching Tuples

C++20

```
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;
}
```

R3

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

R4

```
p match {
  [0, 0] => {
    std::cout << "on origin";
  }
  [0, let y] => {
    std::cout << "on y-axis at " << y;
  }
  [let x, 0] => {
    std::cout << "on x-axis at " << x;
  }
  let [x, y] => {
    std::cout << x << ',' << y;
  }
};
```

P2392

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

12.4 Matching Variants

C++20

```
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);
```

R3

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

R4

```
v match {
  int: let i => {
    strm << "got int: " << i;
  }
  float: let f => {
    strm << "got float: " << f;
  }
};
```

P2392

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

12.5 Matching Polymorphic Types

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

C++20

```
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;
}
```

R3

```
int get_area(const Shape& shape) {
  return inspect (shape) {
    <Circle>    [r]    => 3.14 * r * r;
    <Rectangle> [w, h] => w * h;
  };
}
```

R4

```
int get_area(const Shape& shape) {
  return shape match {
    Circle:    let [r]    => 3.14 * r * r;
    Rectangle: let [w, h] => w * h;
  };
}
```

P2392

```
int get_area(const Shape& shape) {
  return inspect (shape) {
    [r]    as Circle    => 3.14 * r * r;
    [w, h] as Rectangle => w * h;
  };
}
```

12.6 Evaluating Expression Trees

```
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

```
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 {
      // Optimize multiplication by 0.
      if (int* i = std::get_if<int>(m.lhs.get()); i && *i == 0) {
        return 0;
      }
      if (int* i = std::get_if<int>(m.rhs.get()); i && *i == 0) {
        return 0;
      }
      return eval(*m.lhs) * eval(*m.rhs);
    }
  };
  return std::visit(visitor{}, expr);
}
```

R3

```
int eval(const Expr& expr) {
  return inspect (expr) {
    <int> i => i;
    <Neg> [(*!) e] => -eval(e);
    <Add> [(*!) l, (*!) r] => eval(l) + eval(r);
    // Optimize multiplication by 0.
    <Mul> [(*!) <int> 0, __] => 0;
    <Mul> [__, (*!) <int> 0] => 0;
    <Mul> [(*!) l, (*!) r] => eval(l) * eval(r);
  };
}
```

R4

```
int eval(const Expr& expr) {
  return expr match {
    int: let i => i;
    Neg: [*: let e] => -eval(e);
    Add: let [*: l, *: r] => eval(l) + eval(r);
    // Optimize multiplication by 0.
    Mul: ([*: int: 0, _] || [_, *: int: 0]) => 0;
    Mul: let [*: l, *: r] => eval(l) * eval(r);
    _ => 0;
  };
}
```

P2392

```
int eval(const Expr& expr) {
  return inspect (expr) {
    i as int => i;
    [*e] as Neg => -eval(e);
    [*l, *r] as Add => eval(l) + eval(r);
    [*l, *r] as Mul {
      // Optimize multiplication by 0.
      if (l as int == 0 || r as int == 0) => 0;
      is _ => eval(l) * eval(r);
    }
    is _ => 0; // default case is required
  };
}
```

12.7 Nested Structured Bindings

Before / After

```
auto const& [topLeft, unused] = getBoundaryRectangle();
auto const& [topBoundary, leftBoundary] = topLeft;
```

```
auto const& [[topBoundary, leftBoundary], unused] = getBoundaryRectangle();
```

12.8 Pattern Matching in Declarations

```
if (auto p = f(); p.first == 0) {  // C++20
  auto&& y = p.second;
  // ...
}

if (auto p match [0, let y] = f()) {  // R4
  // ...
}

if (auto [_, y] is [0, _] = f()) {  // P2392
  // ...
}
```

13 Design Overview

13.1 Basic Syntax

expression `match` *pattern guard*$_{opt}$

expression `match` *trailing-return-type*$_{opt}$ `{`
  pattern guard$_{opt}$ `=>` *statement*
  pattern guard$_{opt}$ `=>` `!`$_{opt}$ `{` *statement-seq* `}`
…
`}`

guard:
  `if` *expression*

13.2 Basic Model

`match` is an expression in all contexts. Depending on the enclosed statements it may either yield a `void` result or a value, the type of which will be statically deduced from the statements themselves or specified by a trailing return type. The deduction is analogous to that performed when determining the return type of a lambda expression. A pattern that passes control to a compound statement yields a `void` result. The return types of all patterns must match. If a trailing return type is provided, all patterns must result in an expression returning a type that is implicitly convertible to the trailing return type.

If `!` prefix is used before compound statement - the statement would not contribute to return type deduction for `match` expression. Such a statement is not expected to yield a value and should stop the execution either by returning from the enclosing function, throwing an exception or terminating the program. This allows users to express desired no-match behaviour or to act upon broken invariant, without affecting return type of the whole of `match` expression. If execution reaches end of the compound statement `std::terminate` is called.

When `match` is executed, the expression to the left 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), then the value of the resulting expression is yielded or control is passed to the compound statement, depending on whether the inspect yields a value. If the guard expression evaluates to `false`, control flows to the subsequent pattern.

If no pattern matches, none of the expressions or compound statements specified are executed. In that case if the `match` expression yields `void`, control is passed to the next statement. If the `match` expression does not yield `void`, `std::terminate` will be called.

13.3 Types of Patterns

13.3.1 Primary Patterns

13.3.1.1 Wildcard Pattern

The wildcard pattern has the form:

```
_
```

and matches any value `v`.

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

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

Refer to the discussion for choice of `_` in Wildcard Pattern.

13.3.1.2 Expression Pattern

The expression pattern has the form:

cast-expression

An expression pattern `e` matches value `v`

1. If `e(v)` is valid, `e` matches `v` if `e(v)` is contextually convertible to `bool` and evaluates to `true`.
2. Otherwise, if `e == v` is valid, `e` matches `v` if `e == v` is contextually convertible to `bool` and evaluates to `true`.

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

v match {
    0 => { std::cout << "got zero"; }
    1 => { std::cout << "got one"; }
//  ^ expression pattern
};
```

```
enum Color { Red, Green, Blue };
Color color = /* ... */;

color match {
    Red   => // ...
    Green => // ...
    Blue  => // ...
//  ^^^^^ expression pattern
};
```

13.3.2 Compound Patterns

13.3.2.1 Let Pattern

The let pattern has the form:

`let` *let-pattern*

The *let-pattern* allows all patterns and *identifier*, without any expressions.

identifier within *let-pattern* match any value and are bindings to the corresponding component as defined by its parent pattern. It is in scope from the pattern guard if one exists, otherwise from the start of the statement following the pattern label.

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

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

```
static constexpr int zero = 0, one = 1;
std::pair<int, std::pair<int, int>> p = /* ... */

inspect (p) {
    [zero, let [x, y]]: std::cout << "got zero" << ' ' << x << ' ' << y;
//              ^  ^ identifier pattern
    [one, let [x, y]]: std::cout << "got one" << ' ' << x << ' ' << y;
//        ^^^^^^^^^^  binding pattern
}
```

13.3.2.2 Structured Binding Pattern

The structured binding pattern has the following two forms:

[$pattern_0$, $pattern_1$, … , $pattern_N$]
[$designator_0$: $pattern_0$, $designator_1$: $pattern_1$, … , $designator_N$: $pattern_{N\sim}$]

The first form matches value v if each $pattern_i$ matches the i^{th} 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.

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

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

The second form matches value v if each $pattern_i$ matches the direct non-static data member of v named *identifier* from each $designator_i$. If an *identifier* from any $designator_i$ does not refer to a direct non-static data member of v, the program is ill-formed.

```
struct Player { std::string name; int hitpoints; int coins; };

void get_hint(const Player& p) {
    inspect (p) {
        [.hitpoints: 1] => { std::cout << "You're almost destroyed. Give up!\n"; }
        [.hitpoints: 10, .coins: 10] => { std::cout << "I need the hints from you!\n"; }
        [.coins: 10] => { std::cout << "Get more hitpoints!\n"; }
        [.hitpoints: 10] => { std::cout << "Get more ammo!\n"; }
```

```
        [.name: let n] => {
            if (n != "The Bruce Dickenson") {
                std::cout << "Get more hitpoints and ammo!\n";
            } else {
                std::cout << "More cowbell!\n";
            }
        }
    };
}
```

[*Note:* Unlike designated initializers, the order of the designators need not be the same as the declaration order of the members of the class. — *end note*]

13.3.2.3 Alternative Pattern

// TODO: Think through this further…

The alternative pattern has the following forms:

`auto` : *pattern*
concept : *pattern*
type : *pattern*
cast-expression : *pattern*

```
*: let foo
```

```
e match {
  std::integral => ...
  int => ...
  int: let i => ...
  std::integral: let i => ...
};
```

```
s match {
  ctre::match<"[a-z]+([0-9]+)"> => // ...

  ctre::match<"[a-z]+([0-9]+)">: let [m, n] => ...
};
```

`<` `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.

Case 1: `std::variant`-like

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;
  }
};
```

```
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);
```

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

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

Case 2: `std::any`-like

`<` *type* `>` *pattern*

If `Alt` is a *type* and there exists a valid non-member ADL-only `any_cast<Alt>(&v)`, let `p` be its result. The alternative pattern matches if `p` contextually converted to `bool` evaluates to `true`, and *pattern* matches `*p`.

Before

```
std::any a = 42;

if (int* i = any_cast<int>(&a)) {
  std::cout << "got int: " << *i;
} else if (float* f = any_cast<float>(&a)) {
  std::cout << "got float: " << *f;
}
```

After

```
std::any a = 42;

inspect (a) {
  <int> i => {
    std::cout << "got int: " << i;
  }
  <float> f => {
    std::cout << "got float: " << f;
  }
};
```

Case 3: Polymorphic Types

`<` *type* `>` *pattern*

If `Alt` is a *type* and `std::is_polymorphic_v<V>` is `true`, let `p` be `dynamic_cast<Alt'*>(&v)` where `Alt'` has the same *cv*-qualifications as `decltype(&v)`. The alternative pattern matches if `p` contextually converted to `bool` evaluates to `true`, and *pattern* matches `*p`.

While the **semantics** of the pattern is specified in terms of `dynamic_cast`, [N3449] describes techniques involving vtable pointer caching and hash conflict minimization that are implemented in the [Mach7] library, as well as mentions of further opportunities available for a compiler intrinsic.

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; };
```

C++20

```
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;
}
```

R3

```
int get_area(const Shape& shape) {
  return inspect (shape) {
    <Circle> [r] => 3.14 * r * r;
    <Rectangle> [w, h] => w * h;
  };
}
```

R4	P2392

`int get_area(const Shape& shape) {`
`  return shape match {`
`  Circle:  let [r] => 3.14 * r * r;`
`  Rectangle: let [w, h] => w * h;`
`  };`
`}`
	`int get_area(const Shape& shape) {`
	`  return inspect (shape) {`
	`  [r] as Circle => 3.14 * r * r;`
	`  [w, h] as Rectangle => w * h;`
	`  };`
	`}`

13.3.2.4 Parenthesized Pattern

The parenthesized pattern has the form:

(*pattern*)

and matches value `v` if *pattern* matches `v`.

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

inspect (v) {
    Point: ([let x, 0] | [0, let x]) => // ...
//          ^^^^^^^^^^^^^^^^^^^^^^^^ parenthesized pattern
};
```

13.3.2.5 Dereference Pattern

The dereference pattern has the form:

`*` : *pattern*

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

```
struct Node {
    int value;
    std::unique_ptr<Node> lhs, rhs;
};

void print_leftmost(const Node& node) {
    inspect (node) {
        [.value: v, .lhs: nullptr] => { std::cout << v << '\n'; }
        [.lhs: *: l] => { print_leftmost(l); }
//              ^^^^ dereference pattern
    };
}
```

[*Note:* Refer to Red-black Tree Rebalancing for a more complex example. — *end note*]

13.3.2.6 Extractor Pattern

The extractor pattern has the following two forms:

cast-expression : *pattern*

Let `c` be the *cast-expression* and `e` be the result of `c(v)`. The extractor pattern matches value `v` if `e` is contextually convertible to `bool`, evaluates to `true` and *pattern* matches `*e`.

```
std::optional<std::array<std::string_view, 2>> email(std::string_view sv);
std::optional<std::array<std::string_view, 3>> phone_number(std::string_view sv);

s match {
    email: let [address, domain] => { std::cout << "got an email"; }
    phone_number: ["415", _, _]  => { std::cout << "got a San Francisco phone number"; }
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ extractor pattern
};
```

13.4 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:

```
p match {
    let [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.

13.5 `match constexpr`

// TODO: check if this actually does what we want…

Every *pattern* is able to determine whether it matches value `v` as a boolean expression in isolation. Ignoring any potential optimization opportunities, we're able to perform the following transformation:

match	if

`v match {`
`  pattern1 if (cond1) => { stmt1 }`
`  pattern2 => { stmt2 }`
`  // ...`
`};`
	`if (v match pattern1 && cond1) stmt1`
	`else if (v match pattern2) stmt2`
	`// ...`

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

match constexpr	if constexpr

`v match constexpr {`
`  pattern1 if (cond1) => { stmt1 }`
`  pattern2 => { stmt2 }`
`  // ...`
`};`
	`if constexpr (v match pattern1 && cond1) stmt1`
	`else if constexpr (v match pattern2) stmt2`
	`// ...`

13.6 Exhaustiveness and Usefulness

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

13.7 Refutability

Patterns that cannot fail to match are said to be *irrefutable* in contrast to *refutable* patterns which can fail to match. For example, the identifier pattern is *irrefutable* whereas the expression pattern is *refutable*.

The distinction is useful in reasoning about which patterns should be allowed in which contexts. For example, the structured bindings declaration is conceptually a restricted form of pattern matching. With the introduction of expression pattern in this paper, some may question whether structured bindings declaration should be extended for examples such as `auto [0, x] = f();`.

This is ultimately a question of whether structured bindings declaration supports *refutable* patterns or if it is restricted to *irrefutable* patterns.

14 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* `)` *trailing-return-type*$_{opt}$ `{`

inspect-case-seq `}`]{.add}

inspect-case-seq:
inspect-statement-case-seq
inspect-expression-case-seq

inspect-statement-case-seq:
inspect-statement-case
inspect-statement-case-seq inspect-statement-case

inspect-expression-case-seq:
inspect-expression-case
inspect-expression-case-seq `,` *inspect-expression-case*

inspect-statement-case:
inspect-pattern inspect-guard$_{opt}$ `=>` *statement*

inspect-expression-case:
inspect-pattern inspect-guard$_{opt}$ `=>` *assignment-expression*

inspect-pattern:
alternative-pattern
case-pattern
dereference-pattern
expression-pattern
extractor-pattern
identifier-pattern
structured-binding-pattern
wildcard-pattern

inspect-guard:
`if (` *expression* `)`

Change **§9.1 [dcl.dcl]**

simple-declaration:
decl-specifier-seq init-declarator-list$_{opt}$ `;`
attribute-specifier-seq decl-specifier-seq init-declarator-list `;`
[*attribute-specifier-seq*$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ `[` *identifier-list* `]`

initializer `;`]{.rm} | *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ *structured-binding-pattern initializer* `;`

15 Design Decisions

15.1 Wildcard Pattern

This paper proposes that `_` be used as the wildcard pattern.

// TODO

15.2 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.

We propose any **irrefutable** pattern to be **allowed** in structured binding declaration, as it does not introduce any new behaviour. A separate paper will explore possibility of allowing **refutable** patterns to be used in declarations.

15.3 `inspect` rather than `switch`

This proposal introduces a new `inspect` statement rather than trying to extend the `switch` statement. [P0095R0] 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.

15.4 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. 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.

15.5 Unrestricted Side Effects

We considered the possibility of restricting side-effects within patterns. Specifically whether modifying the value currently being matched in the middle of evaluation should have defined behavior.

The consideration was due to potential optimization opportunities.

```
bool f(int &);  // defined in a different translation unit.
int x = 1;

inspect (x) {
  0 => { std::cout << 0; }
  1 if (f(x)) => { std::cout << 1; }
  2 => { std::cout << 2; }
};
```

If modifying the value currently being matched has undefined behavior, a compiler can assume that `f` (defined in a different translation unit) will not change the value of `x`. This means that the compiler can generate code that uses a jump table to determine which of the patterns match.

If on the other hand `f` may change the value of `x`, the compiler would be forced to generated code checks the patterns in sequence, since a subsequent pattern may match the updated value of `x`.

The following are **illustrations** of the two approaches written in C++:

Not allowed to modify

```
bool f(int &);
int x = 1;

switch (x) {
  case 0: std::cout << 0; break;
  case 1: if (f(x)) { std::cout << 1; } break;
  case 2: std::cout << 2; break;
}
```

Allowed to modify

```
bool f(int &);
int x = 1;

if (x == 0) std::cout << 0;
else if (x == 1 && f(x)) std::cout << 1;
else if (x == 2) std::cout << 2;
```

However, we consider this opportunity too niche. Suppose we have a slightly more complex case: `struct S { int x; };` and `bool operator==(const S&, const S&);`. Even if modifying the value being matched has undefined behavior, if the `operator==` is defined in a different translation unit, a compiler cannot do much more than generate code that checks the patterns in sequence anyway.

15.6 Language rather than Library

There are three popular pattern matching libraries for C++ today: [Mach7], [Patterns], and [SimpleMatch].

While the libraries have been useful for gaining experience with interfaces and implementation, 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.

15.7 Matchers and Extractors

Many languages provide a wide array of patterns through various syntactic forms. While this is a potential direction for C++, it would mean that every new type of matching requires new syntax to be added to the language. This would result in a narrow set of types being supported through limited customization points.

Matchers and extractors are supported in order to minimize the number of patterns with special syntax. The following are example matchers and extractors that commonly have special syntax in other languages.

Matchers / Extractors	Other Languages
`any_of{1, 2, 3}`	`1 | 2 | 3`
`within{1, 10}`	`1..10`

Matchers / Extractors	Other Languages
`(both!) [[x, 0], [0, y]]`	`[x, 0] & [0, y]`
`(at!) [p, [x, y]]`	`p @ [x, y]`

Each of the matchers and extractors can be found in the Examples section. The example extractors and matchers are not proposed for standardisation in this paper, and presented just for demonstration.

15.8 Expression vs Pattern Disambiguation

[P1371R0] had proposed a unary `^` as an "expression introducer". The main motivation was to leave the design space open for patterns that look like expressions. For example, many languages spell the alternation pattern with `|`, resulting in a pattern such as `1 | 2` which means "match `1` or `2`". However, to allow such a pattern a disambiguation mechanism would be required since `1 | 2` is already a valid expression today.

That paper also included what is called a dereference pattern with the syntax of `*` *pattern*. There was clear guidance from EWG to change the syntax of this pattern due to confusion with the existing dereference operator. As such, the design direction proposed in this paper is to allow expressions in patterns without an introducer, and to require that new patterns be syntactically unambiguous with an expression in general.

The following is a flow graph of decisions that need to be made:

15.9 Forbid `break` inside `inspect` expression

Since `inspect` is always an expression we decided to forbid using `break` keyword inside `inspect` expressions.

The problem lies with two possible use cases where `inspect` would be used.

```
for (const auto& el: some_vec) {
  // If-else-if chain
  if (el.type() == "NotInteresting") {
    break;
  } else if (el.type() == "SomeOther") {
    break;
  }

  // Switch statement
  switch (el.value()) {
    case 1: /* ... */
      break; // no fallthrough
    case 2: /* ... */
      break; // no fallthrough
    default:
      /* ... */
  }
}
```

In the example above, if we're replacing existing `switch` statement, `break` is used to terminate current statement sequence and jump to first statement after the `switch`. In particular it is required to prevent fallthrough.

If the code being replaced is a sequence of if-else branches, `break` there would indicate iteration stop for the enclosing loop.

Both use cases are interesting and valid for `inspect`, but resulting `break` behaviour differs. If we were to adopt one, the other use case would be prone to error. So for now we decided to forbid using `break` statements inside `inspect` expression branches.

Note, it is generally desirable to be able to yield from `inspect` expression branch early, but currently there is no syntax that would allow specifying yield value with `break` statement (i.e. `break 2;`). We think this behaviour is valuable, but not crucial for this proposal.

16 Runtime Performance

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

16.1 Structured Binding Patterns

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.

16.2 Alternative Patterns

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

16.3 Open Class Hierarchy

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

17 Examples

17.1 Predicate-based Discriminator

Short-string optimization using a **predicate** as a discriminator rather than an explicitly stored **value**. Adapted from Bjarne Stroustrup's pattern matching presentation at Urbana-Champaign 2014 [PatMatPres].

```
struct String {
  enum Storage { Local, Remote };

  int size;
  union {
    char local[32];
    struct { char *ptr; int unused_allocated_space; } remote;
  };

  // Predicate-based discriminator derived from `size`.
  Storage index() const { return size > sizeof(local) ? Remote : Local; }

  // Opt into Variant-Like protocol.
  template <Storage S>
  auto &&get() {
    if constexpr (S == Local) return local;
    else if constexpr (S == Remote) return remote;
  }

  char *data();
};

namespace std {
```

```
  // Opt into Variant-Like protocol.

  template <>
  struct variant_size<String> : std::integral_constant<std::size_t, 2> {};

  template <>
  struct variant_alternative<String::Local, String> {
    using type = decltype(String::local);
  };

  template <>
  struct variant_alternative<String::Remote, String> {
    using type = decltype(String::remote);
  };
}

char* String::data() {
  return inspect (*this) {
    <Local> l => l;
    <Remote> r => r.ptr;
  };
  // switch (index()) {
  //   case Local: {
  //     std::variant_alternative_t<Local, String>& l = get<Local>();
  //     return l;
  //   }
  //   case Remote: {
  //     std::variant_alternative_t<Remote, String>& r = get<Remote>();
  //     return r.ptr;
  //   }
  // }
}
```

17.2 "Closed" Class Hierarchy

A class hierarchy can effectively be closed with an `enum` that maintains the list of its members, and provide efficient dispatching by opting into the Variant-Like protocol.

A generalized mechanism of pattern is used extensively in LLVM; `llvm/Support/YAMLParser.h` [YAMLParser] is an example.

```
struct Shape { enum Kind { Circle, Rectangle } kind; };

struct Circle : Shape {
  Circle(int radius) : Shape{Shape::Kind::Circle}, radius(radius) {}

  int radius;
};

struct Rectangle : Shape {
  Rectangle(int width, int height)
    : Shape{Shape::Kind::Rectangle}, width(width), height(height) {}

  int width, height;
};
```

```
namespace std {
  template <>
  struct variant_size<Shape> : std::integral_constant<std::size_t, 2> {};

  template <>
  struct variant_alternative<Shape::Circle, Shape> { using type = Circle; };

  template <>
  struct variant_alternative<Shape::Rectangle, Shape> { using type = Rectangle; };
}

Shape::Kind index(const Shape& shape) { return shape.kind; }

template <Kind K>
auto&& get(const Shape& shape) {
  return static_cast<const std::variant_alternative_t<K, Shape>&>(shape);
}

int get_area(const Shape& shape) {
  return inspect (shape) {
    <Circle> c => 3.14 * c.radius * c.radius;
    <Rectangle> r => r.width * r.height;
  };
  // switch (index(shape)) {
  //   case Shape::Circle: {
  //     const std::variant_alternative_t<Shape::Circle, Shape>& c =
  //         get<Shape::Circle>(shape);
  //     return 3.14 * c.radius * c.radius;
  //   }
  //   case Shape::Rectangle: {
  //     const std::variant_alternative_t<Shape::Rectangle, Shape>& r =
  //         get<Shape::Rectangle>(shape);
  //     return r.width * r.height;
  //   }
  // }
}
```

17.3 Matcher: `any_of`

The logical-or pattern in other languages is typically spelled $pattern_0$ | $pattern_1$ | ... | $pattern_N$, and matches value `v` if any $pattern_i$ matches `v`.

This provides a restricted form (constant-only) of the logical-or pattern.

```
template <typename... Ts>
struct any_of : std::tuple<Ts...> {
  using tuple::tuple;

  template <typename U>
  bool match(const U& u) const {
    return std::apply([&](const auto&... xs) { return (... || xs == u); }, *this);
  }
};
```

```
int fib(int n) {
  return inspect (n) {
    x if (x < 0) => 0;
    any_of{1, 2} => n;  // 1 | 2
    x => fib(x - 1) + fib(x - 2);
  };
}
```

17.4 Matcher: `within`

The range pattern in other languages is typically spelled `first..last`, and matches `v` if `v` ∈ `[first, last]`.

```
struct within {
  int first, last;

  bool match(int n) const { return first <= n && n <= last; }
};
```

```
inspect (n) {
  within{1, 10} => {  // 1..10
    std::cout << n << " is in [1, 10].";
  }
  __ => {
    std::cout << n << " is not in [1, 10].";
  }
};
```

17.5 Extractor: `both`

The logical-and pattern in other languages is typically spelled *pattern*$_0$ `&` *pattern*$_1$ `& ... &` *pattern*$_N$, and matches `v` if all of *pattern*$_i$ matches `v`.

This extractor emulates binary logical-and with a `std::pair` where both elements are references to value `v`.

```
struct Both {
  template <typename U>
  std::pair<U&&, U&&> extract(U&& u) const {
    return {std::forward<U>(u), std::forward<U>(u)};
  }
};

inline constexpr Both both;
```

```
inspect (v) {
  (both!) [[x, 0], [0, y]] => // ...
};
```

17.6 Extractor: `at`

The binding pattern in other languages is typically spelled *identifier* `@` *pattern*, binds *identifier* to `v` and matches if *pattern* matches `v`. This is a special case of the logical-and pattern (*pattern*$_0$ `&` *pattern*$_1$) where *pattern*$_0$ is an *identifier*. That is, *identifier* `&` *pattern* has the same semantics as *identifier* `@` *pattern*, which means we get `at` for free from `both` above.

```
inline constexpr at = both;
```

```
inspect (v) {
  <Point> (at!) [p, [x, y]] => // ...
  // ...
};
```

17.7 Red-black Tree Rebalancing

Dereference patterns frequently come into play with complex patterns using recursive variant types. An example of such a problem is the rebalance operation for red-black trees. Using pattern matching this can be expressed succinctly and in a way that is easily verified visually as having the correct algorithm.

Given the following red-black tree definition:

```
enum Color { Red, Black };

template <typename T>
struct Node {
  void balance();

  Color color;
  std::shared_ptr<Node> lhs;
  T value;
  std::shared_ptr<Node> rhs;
};
```

The following is what we can write with pattern matching:

```
template <typename T>
void Node<T>::balance() {
  *this = inspect (*this) {
    // left-left case
    //
    //        (Black) z              (Red) y
    //         /     \              /      \
    //     (Red) y    d        (Black) x  (Black) z
    //     /      \       ->     /   \      /   \
    //  (Red) x    c             a     b    c     d
    //   /    \
    //  a      b
    [case Black, (*?) [case Red, (*?) [case Red, a, x, b], y, c], z, d]
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, (*?) [case Red, a, x, (*?) [case Red, b, y, c]], z, d]  // left-right case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, (*?) [case Red, b, y, c], z, d]]  // right-left case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    [case Black, a, x, (*?) [case Red, b, y, (*?) [case Red, c, z, d]]]  // right-right case
      => Node{Red, std::make_shared<Node>(Black, a, x, b),
                   y,
                   std::make_shared<Node>(Black, c, z, d)};
    self => self;  // do nothing
  };
}
```

The following is what we currently need to write:

```
template <typename T>
void Node<T>::balance() {
  if (color != Black) return;
  if (lhs && lhs->color == Red) {
    if (const auto& lhs_lhs = lhs->lhs; lhs_lhs && lhs_lhs->color == Red) {
      // left-left case
      //
      //        (Black) z               (Red) y
      //         /     \               /      \
      //     (Red) y    d         (Black) x  (Black) z
      //     /      \        ->     /   \      /   \
      //  (Red) x    c             a     b    c     d
      //   /    \
      //  a      b
      *this = Node{
          Red,
          std::make_shared<Node>(Black, lhs_lhs->lhs, lhs_lhs->value, lhs_lhs->rhs),
          lhs->value,
          std::make_shared<Node>(Black, lhs->rhs, value, rhs)};
      return;
    }
    if (const auto& lhs_rhs = lhs->rhs; lhs_rhs && lhs_rhs->color == Red) {
      *this = Node{  // left-right case
          Red,
          std::make_shared<Node>(Black, lhs->lhs, lhs->value, lhs_rhs->lhs),
          lhs_rhs->value,
          std::make_shared<Node>(Black, lhs_rhs->rhs, value, rhs)};
      return;
    }
  }
  if (rhs && rhs->color == Red) {
    if (const auto& rhs_lhs = rhs->lhs; rhs_lhs && rhs_lhs->color == Red) {
      *this = Node{  // right-left case
          Red,
          std::make_shared<Node>(Black, lhs, value, rhs_lhs->lhs),
          rhs_lhs->value,
          std::make_shared<Node>(Black, rhs_lhs->rhs, rhs->value, rhs->rhs)};
      return;
    }
    if (const auto& rhs_rhs = rhs->rhs; rhs_rhs && rhs_rhs->color == Red) {
      *this = Node{  // right-right case
          Red,
          std::make_shared<Node>(Black, lhs, value, rhs->lhs),
          rhs->value,
          std::make_shared<Node>(Black, rhs_rhs->lhs, rhs_rhs->value, rhs_rhs->rhs)};
      return;
    }
  }
}
```

18 Future Work

18.1 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.

18.2 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.

19 Acknowledgements

Thanks to all of the following:

— Yuriy Solodkyy, Gabriel Dos Reis, Bjarne Stroustrup for their prior work on [N3449], Open Pattern Matching for C++ [OpenPM], and the [Mach7] library.
— Pattern matching presentation by Bjarne Stroustrup at Urbana-Champaign 2014. [PatMatPres]
— Jeffrey Yasskin/JF Bastien for their work on [P1110R0].
— (In alphabetical order by last name) Dave Abrahams, John Bandela, Agustín Bergé, Ori Bernstein, Matt Calabrese, Alexander Chow, Louis Dionne, Michał Dominiak, Vicente Botet Escribá, Eric Fiselier, Bengt Gustafsson, Zach Laine, Jason Lucas, Barry Revzin, John Skaller, Bjarne Stroustrup, Tony Van Eerd, and everyone else who contributed to the discussions.

20 References

[Mach7] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Mach7: Pattern Matching for C++.

[N3449] B. Stroustrup, G. Dos Reis, Y. Solodkyy. 2012-09-23. Open and Efficient Type Switch for C++.
https://wg21.link/n3449

[OpenPM] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open Pattern Matching for C++.

[P0095R0] David Sankel. 2015-09-24. The case for a language based variant.
https://wg21.link/p0095r0

[P1110R0] Jeffrey Yasskin, JF Bastien. 2018-06-07. A placeholder with no name.
https://wg21.link/p1110r0

[P1371R0] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019-01-21. Pattern Matching.
https://wg21.link/p1371r0

[P1371R1] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019-06-17. Pattern Matching.
https://wg21.link/p1371r1

[P1371R2] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2020-01-13. Pattern Matching.
https://wg21.link/p1371r2

[P1371R3] Michael Park, Bruno Cardoso Lopes, Sergei Murzin, David Sankel, Dan Sarginson, Bjarne Stroustrup. 2020-09-15. Pattern Matching.
https://wg21.link/p1371r3

[P2392R0] Herb Sutter. 2021-06-14. Pattern matching using "is" and "as."
https://wg21.link/p2392r0

[PatMatPres] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. "Pattern Matching for C++" presentation at Urbana-Champaign 2014.

[Patterns] Michael Park. Pattern Matching in C++.

[SimpleMatch] John Bandela. Simple, Extensible C++ Pattern Matching Library.

[Warnings] Luc Maranget. Warnings for pattern matching.

[YAMLParser]
http://llvm.org/doxygen/YAMLParser_8h_source.html