P2688R2 / D2688R3: Pattern Matching

`match` Expression

Michael Park, 2024-10-10

Anatomy of `match`

High-Level Structure

No change since R1

```
expr match {
   pattern => expr-or-braced-init-list;
   pattern => break ;
   pattern => continue ;
   pattern => return expr-or-braced-init-list_opt ;
   pattern => co_return expr-or-braced-init-list_opt ;
   ...
}
```

```
expr match pattern
```

Anatomy of `match`

Specifying the return type

No change since R1

```
expr match -> return-type {
    pattern => expr-or-braced-init-list;
    pattern => break ;
    pattern => continue ;
    pattern => return expr-or-braced-init-list_opt ;
    pattern => co_return expr-or-braced-init-list_opt ;
    ...
}
```

Anatomy of `match`

Matching constexpr

No change since R1

```
expr match constexpr {
   pattern => expr-or-braced-init-list;
   pattern => break ;
   pattern => continue ;
   pattern => return expr-or-braced-init-list_opt ;
   pattern => co_return expr-or-braced-init-list_opt ;
   ...
}
```

Anatomy of `match`

Guards

New in R3: Parentheses are required.

```
expr match {                                          expr match pattern if (cond)
   pattern if (cond) => expr-or-braced-init-list;
   pattern if (cond) => break ;
   pattern if (cond) => continue ;
   pattern if (cond) => return expr-or-braced-init-list_opt ;
   pattern if (cond) => co_return expr-or-braced-init-list_opt ;
   ...
}
```

Anatomy of `match`

Matching multiple values

New in R3: Feature removed due to implementation concerns. Now relies on `std::tuple` facilities.

```
{ e1, e2, ... } match {
    pattern => expr-or-braced-init-list;
    pattern => break;
    pattern => continue;
    pattern => return expr-or-braced-init-list_opt;
    pattern => co_return expr-or-braced-init-list_opt;
    ...
}
```

```
{ e1, e2, ... } match pattern
```

Anatomy of `match`

Putting them together

```
expr match {
   pattern1 => expression1 ;
   pattern2 if (cond) => expression2 ;
   ...
}

{ e1, e2, ... } match {
   pattern1 => expression1 ;
   ...
}

expr match constexpr -> return-type {
   pattern1 => expression1 ;
   ...
}
```

```
bool b = expr match pattern;
f(expr match pattern if (cond));

{ e1, e2, ... } match pattern;

if (expr match pattern) {
   // names injected here
}

while (expr match pattern) {
   // names injected here
}
```

Overview of Patterns

Implementation Status

- **Wildcard Pattern** `_` ✅
  - Ignore values
- **Let Pattern** `let` *binding-pattern* ✅
  match-pattern `let` *binding-pattern* ❌
  - Introduce bindings
- **Parenthesized Pattern** `(` *pattern* `)` ✅
  - Grouping
  - Useful since not all patterns are delimited
- **Constant Pattern** *constant-expression* ✅
  - `enum` values
  - `constexpr` computed values
- **Optional Pattern** `?` *pattern* ✅
  - pointers
  - `std::unique_ptr`, `std::shared_ptr`
  - `std::optional`
- **Structured Bindings Pattern** `[` *pattern*... `]` ✅
  - arrays, `std::array`, `std::pair`, `std::tuple`
- **Alternative Pattern** *type-id*`:` *pattern* ✅
  type-constraint`:` *pattern* ❌
  - `std::variant`, `std::expected` ✅
  - `std::any`, `std::exception_ptr` ❌
  - polymorphic types ✅

Current Implementation

- Source: https://github.com/mpark/llvm-project/tree/p2688-pattern-matching
- Enable with `-fpattern-matching`
- Available on Compiler Explorer: godbolt.org
  - Under `x86-64 clang (pattern matching - P2688)`

- Thank you Bruno Cardoso Lopes, Dan Sarginson, and Steven Newell for their prior work on the implementation!

Matching Integrals

https://godbolt.org/z/hEzeWoeh5

```
constexpr auto f(int x) {
  return x match {
    0 => 101;
    1 => 202;
    _ => -1;
  };
}

static_assert(f(0) == 101);
static_assert(f(1) == 202);
static_assert(f(2) == -1);
```

Matching Strings

https://godbolt.org/z/v8qEo77de

```
constexpr auto f(const std::string_view sv) {
  return sv match {
    "foo" => 101;
    "bar" => 202;
    _ => -1;
  };
}

static_assert(f("foo") == 101);
static_assert(f("bar") == 202);
static_assert(f("baz") == -1);
```

Matching Tuples

https://godbolt.org/z/c4eaq1ceK

```
constexpr int zero = 0;

constexpr auto f(const std::pair<int, int>& p) {
  return p match {
    [zero, zero] => 0;
    [zero, let y] => y + 2;
    [let x, zero] => x + 4;
    let [x, y] => x * y;
  };
}

static_assert(f({0, 0}) == 0);
static_assert(f({0, 2}) == 4);
static_assert(f({2, 0}) == 6);
static_assert(f({2, 4}) == 8);
```

Matching Multiple Values

https://godbolt.org/z/aYWPnEb73

```
constexpr int zero = 0;

constexpr auto f(int a, int b) {
  return (a, b) match {
    [zero, zero] => 0;
    [zero, let y] => y + 2;
    [let x, zero] => x + 4;
    let [x, y] => x * y;
  };
}

static_assert(f(0, 0) == 0);
static_assert(f(0, 2) == 4);
static_assert(f(2, 0) == 6);
static_assert(f(2, 4) == 8);
```

Matching Variants

https://godbolt.org/z/fcKr3zcd3

```
constexpr int zero = 0;

constexpr auto f(const std::variant<int, float, double>& v) {
  return v match {
    int: zero => -1;
    int: let i => i;
    float: let f => int(f) + 2;
    double: let d => int(d) + 4;
  };
}

static_assert(f(0) == -1);
static_assert(f(1) == 1);
static_assert(f(2.f) == 4);
static_assert(f(3.0) == 7);
```

Matching Polymorphic Types

https://godbolt.org/z/K4TaPW3Mn

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

struct Circle : Shape {
  constexpr Circle(int r) : radius(r) {}
  int radius;
};

struct Rectangle : Shape {
  constexpr Rectangle(int w, int h) : width(w), height(h) {}
  int width, height;
};
```

Matching Polymorphic Types

https://godbolt.org/z/K4TaPW3Mn

```
constexpr auto f(const Shape& s) {
  return s match {
    Circle: let [r] => r;
    Rectangle: let [w, h] => w * h;
    _ => -1;
  };
}

static_assert(f(Circle(4)) == 4);
static_assert(f(Rectangle(2, 3)) == 6);
```

Matching Optionals

https://godbolt.org/z/K4TaPW3Mn

```
constexpr auto f(const std::optional<int>& opt) {
  return opt match {
    ? let i => i;
    _ => -1;
  };
}

static_assert(f(101) == 101);
static_assert(f(std::nullopt) == -1);
```

Matching Optionals

https://godbolt.org/z/Wac15W7ab

```
constexpr auto f(const std::optional<int>& opt) {
  if (opt match ? let i) {
    return i;
  }
  return -1;
}

static_assert(f(101) == 101);
static_assert(f(std::nullopt) == -1);
```

Matching Nested Structures

https://godbolt.org/z/d9sG69Gn9

```
struct Rgb { int r, g, b; };
struct Hsv { int h, s, v; };

using Color = variant<Rgb, Hsv>;

struct Quit {};
struct Move { int x, y; };
struct Write { string text; };
struct ChangeColor { Color color; };

using Command = variant<Quit, Move, Write, ChangeColor>;
```

Matching Nested Structures

https://godbolt.org/z/d9sG69Gn9

```
constexpr auto ProcessCommand(const Command& cmd) {
  return cmd match -> int {
    Quit: _ => 0;
    Move: let [x, y] => x + y;
    Write: let [text] => text.size();
    ChangeColor: [Rgb: let [r, g, b]] => r + g + b;
    ChangeColor: [Hsv: let [h, s, v]] => h * s * v;
  };
}

static_assert(ProcessCommand(Quit{}) == 0);
static_assert(ProcessCommand(Move{2, 3}) == 5);
static_assert(ProcessCommand(Write{"hello world"}) == 11);
static_assert(ProcessCommand(ChangeColor{Rgb{2, 3, 4}}) == 9);
static_assert(ProcessCommand(ChangeColor{Hsv{2, 3, 4}}) == 24);
```

Matching Expected

https://godbolt.org/z/Yq9Wvasbf

```
constexpr std::expected<int, std::string_view> div(int n, int d) {
  if (d == 0) return std::unexpected("division by 0");
  return n / d;
}
```

Matching Expected

https://godbolt.org/z/Yq9Wvasbf

```
namespace std {
  template <typename T, typename E>
  struct variant_size<expected<T, E>> {
    static constexpr size_t value = 2;
  };

  template <typename T, typename E>
  struct variant_alternative<0, expected<T, E>> { using type = T; };

  template <typename T, typename E>
  struct variant_alternative<1, expected<T, E>> { using type = E; };
}
```

Matching Expected

https://godbolt.org/z/Yq9Wvasbf

```
namespace std {
  template <size_t I, typename T, typename E>
  constexpr variant_alternative_t<I, expected<T, E>>&
  get(expected<T, E>& e) { /* … */ }
  // 4x

  template <typename T, typename E>
  constexpr size_t index(const expected<T, E>& e) {
    return e.has_value() ? 0 : 1;
  }
}
```

Matching Expected

https://godbolt.org/z/Yq9Wvasbf

```
constexpr auto f(int n, int d) {
  return div(n, d) match {
    int: let result => result;
    std::string_view: let sv => -1;
  }
}

static_assert(f(3, 0) == -1);
static_assert(f(6, 3) == 2);
```

Current Implementation

What's in it?

- **Lexing**
  - => added as a new token
  - `match`, `let`, and _ (underscore) added as context-sensitive keywords

- **Parsing**
  - Structural changes completed
    - `match` expressions, trailing return type, match cases, match guards, etc
  - Most patterns completed
    - Wildcard, Constant, Optional, Alternative, Structured Bindings, Parenthesized
    - *Missing*: "match-and-bind" (e.g. `[0, 1] let whole`)
    - *Missing*: Concept-based alternative pattern. *type-constraint*`:` *pattern*

Current Implementation

What's in it?

- **Semantic Analysis**
  - AST construction, Type deduction, Type checking
  - Inject bindings into enclosing control statement.
    - e.g. `if (expr match [0, let x]) { /* 'x' available here */ }`
  - *Missing*: Handling of templates

- **Code Gen**
  - Most of constant evaluation
  - *Missing*: Handling of jump-statements
  - *Missing*: `std::cast` protocol for type-based alternative pattern. *type-id*`:` *pattern*
  - *Missing*: Emitting actual code gen

Design Updates and Lessons from Implementation

Precedence of `match`

Between Pointer-to-member and Multiplication

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right →
2	`a++` `a--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`.` `->`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left ←
	`+a` `-a`	Unary plus and minus	
	`!` `~`	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` `new[]`	Dynamic memory allocation	
	`delete` `delete[]`	Dynamic memory deallocation	
4	`.*` `->*`	Pointer-to-member	Left-to-right →
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b` `a-b`	Addition and subtraction	
7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=` `>` `>=`	For relational operators < and ≤ and > and ≥ respectively	
10	`==` `!=`	For equality operators = and ≠ respectively	
11	`a&b`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional[note 2]	Right-to-left ←
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`=`	Direct assignment (provided by default for C++ classes)	
	`+=` `-=`	Compound assignment by sum and difference	
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder	
	`<<=` `>>=`	Compound assignment by bitwise left shift and right shift	
	`&=` `^=` `|=`	Compound assignment by bitwise AND, XOR, and OR	
17	`,`	Comma	Left-to-right →

`match` Expression

Precedence of `match`

Examples

Input	Parsed
`*a match { /* … */ }`	`(*a) match { /* … */ }`
`a.*b match { /* … */ }`	`(a.*b) match { /* … */ }`
`a * b match { /* … */ }`	`a * (b match { /* … */ })`
`a + b match { /* … */ }`	`a + (b match { /* … */ })`
`a << b match { /* … */ }`	`a << (b match { /* … */ })`
`a <=> b match { /* … */ }`	`a <=> (b match { /* … */ })`
`a < b match { /* … */ }`	`a < (b match { /* … */ })`
`a == b match { /* … */ }`	`a == (b match { /* … */ })`
`a & b match { /* … */ }`	`a & (b match { /* … */ })`
`a && b match { /* … */ }`	`a && (b match { /* … */ })`

Precedence of `match`

Examples

Input	Parsed
`*a match c`	`(*a) match c`
`a.*b match c`	`(a.*b) match c`
`a * b match c`	`a * (b match c)`
`a + b match c`	`a + (b match c)`
`a << b match c`	`a << (b match c)`
`a <=> b match c`	`a <=> (b match c)`
`a < b match c`	`a < (b match c)`
`a == b match c`	`a == (b match c)`
`a & b match c`	`a & (b match c)`
`a && b match c`	`a && (b match c)`

Require Parentheses of Match Guards

Match Test Expressions

```
std::pair p(1, 2);

bool b = p match let [x, y] if x == y;

// Given the previous precedence, would parse as:
bool b = (p match let [x, y] if x) == y;

// error: use of undefined variable 'y'

// Requiring the parentheses solves the problem:
bool b = p match let [x, y] if (x == y);
```

Require Parentheses of Match Guards

- Better error messages
- Easier to bring to parity with existing `if`
  - *pattern* `if` (*init-stmt; cond*)
  - *pattern* `if` (*auto var = expr*)

Require Parentheses of Match Guards

Also in Match Select Expressions

```
std::pair p(1, 2);

int i = p match {
  [0, 0] => 0;
  let [x, y] if (x == y) => 1;
  _ => 2;
};
```

Matching Multiple Values with Parentheses

```
void f(int a, int b) {
  int x = {a, b} match { // R2
    [0, 0] => 1;
    _ => 2;
  };
}
```

```
void f(int a, int b) {
  int x = (a, b) match { // Attempted
    [0, 0] => 1;
    _ => 2;
  };
}
```

Matching Multiple Values with Parentheses

Example from Tokyo 2024

```
void fizzbuzz() {
  for (int i = 1; i <= 100; ++i) {
    {i%3, i%5} match { // R2
      [0, 0] => std::print("fizzbuzz");
      [0, _] => std::print("fizz");
      [_, 0] => std::print("buzz");
      [_, _] => std::print("{}\n", i);
    };
  }
}

// Interpreted as a new block scope 😢
```

```
void fizzbuzz() {
  for (int i = 1; i <= 100; ++i) {
    (i%3, i%5) match { // Attempted
      [0, 0] => std::print("fizzbuzz");
      [0, _] => std::print("fizz");
      [_, 0] => std::print("buzz");
      [_, _] => std::print("{}\n", i);
    };
  }
}

// Works in block scope!
```

Matching Multiple Values with Parentheses

```
void f(int a, int b) {
  int x = ({a, b} match { // R2
    [0, 0] => 1;
    _ => 2;
  }) + 1;
}

// Not a language issue, but
// GCC and Clang at least interpret
// this as statement-expression
```

```
void f(int a, int b) {
  int x = ((a, b) match { // Attempted
    [0, 0] => 1;
    _ => 2;
  }) + 1;
}

// Okay
```

EWG Telecon was okay with the parentheses but implementation concerns were raised

Parsing Rules around _ (underscore) and let

```
expr match {
  _ => // wildcard pattern
       // NOT expression '_'.
  *_ => // dereference _
  _ + 1 => // error: expected '=>' after wildcard pattern
           // NOT expression '_ + 1'.
  let => // error: expected identifier or '[' after 'let'
         // NOT expression 'let'
  let x => // let pattern
  let [x] => // structured binding pattern
             // NOT expression 'let[x]', i.e. indexing into 'let' with 'x'
}
```