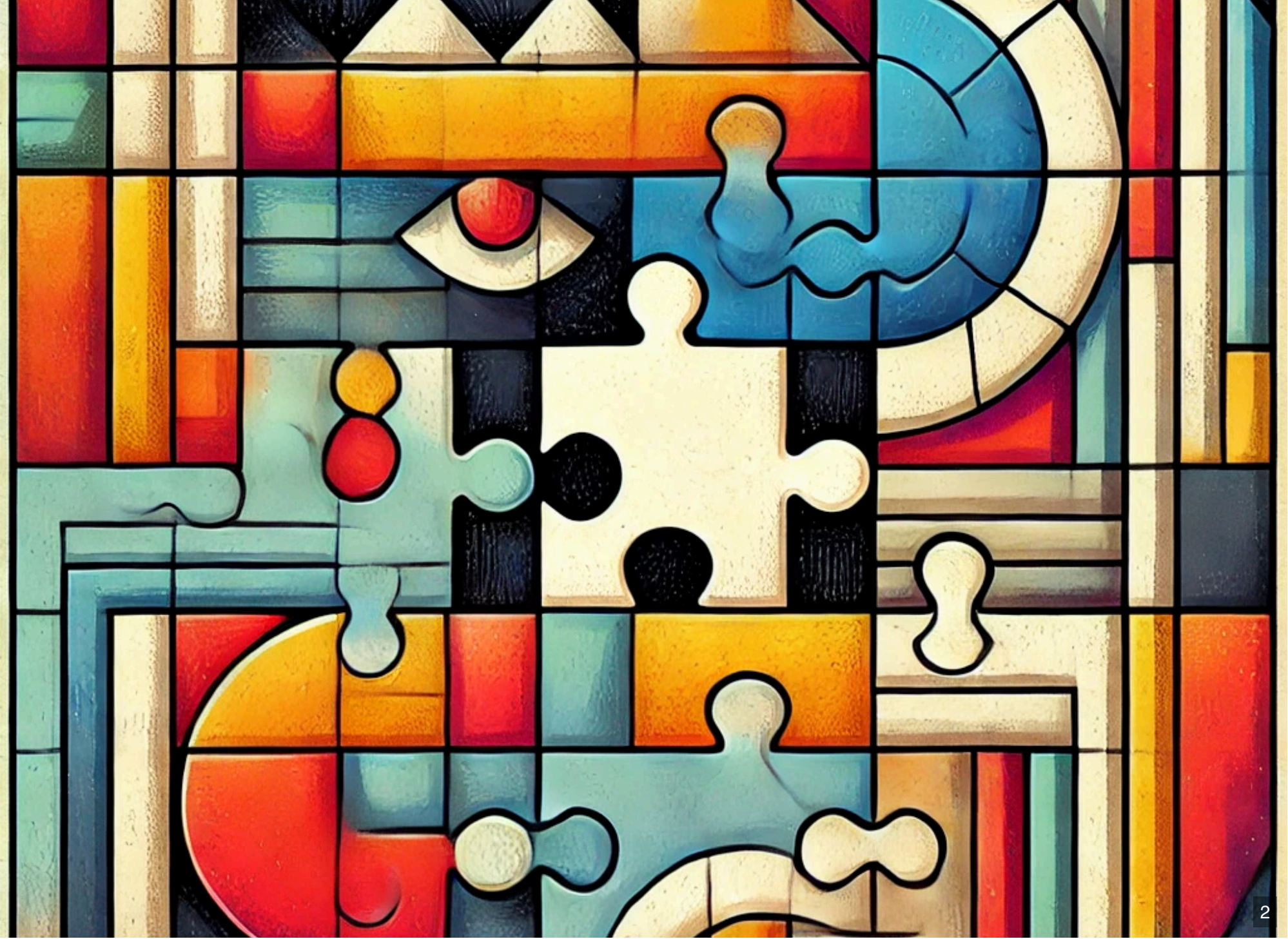
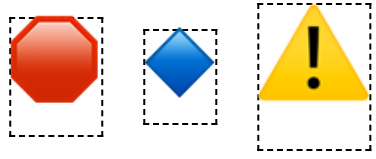
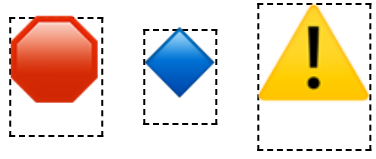
REUSABLE CODE, REUSABLE DATA STRUCTURES

Sebastian Theophil
stheophil@think-cell.com

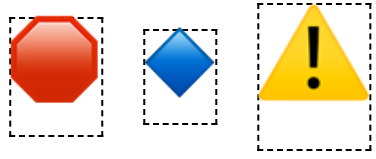


WHY THIS TALK?

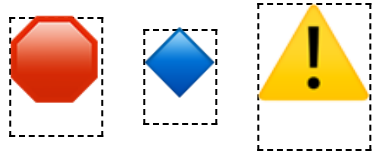




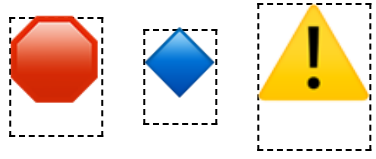
```
// https://github.com/think-cell/think-cell-library
void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```



```
// https://github.com/think-cell/think-cell-library
void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```



```
1  // https://github.com/think-cell/think-cell-library
2  void vertical_center(WidgetContainer const& c) {
3    auto rects = tc::make_vector(tc::transform(c,
4      [](widget const& w) { return w.bounding_box(); }
5    ));
6    // 🧙 Magic happens
7    tc::for_each(tc::zip(c, rects),
8      [&](widget& w, rect r) {
9        widget.place(r);
10     }
11   );
12 }
```



```
// https://github.com/think-cell/think-cell-library
void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```cpp
// https://github.com/think-cell/think-cell-library
void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```cpp
// https://github.com/think-cell/think-cell-library
void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```
1  // https://github.com/think-cell/think-cell-library
2  void vertical_center(WidgetContainer const& c) {
3    auto rects = tc::make_vector(tc::transform(c,
4      [](widget const& w) { return w.bounding_box(); }
5    ));
6    // 🧙 Magic happens
7    tc::for_each(tc::zip(c, rects),
8      [&](widget& w, rect r) {
9        widget.place(r);
10     }
11   );
12 }
13
14 void center_bitmaps() {
15   std::array<bitmap, 3> = {🌇, 🏙, 🌉};
16   // What now?
17 }
```

```
1 void vertical_center(WidgetContainer const& c) {
2   // 🧙 Magic happens
3 }
4
5 void center_bitmaps() {
6   std::array<bitmap, 3> = {🌇, 🏙, 🏔};
7   // What now?
8 }
```

```
1  struct widget {
2    rect bounding_box();
3    void place(rect r);
4  };
5
6  struct bitmap {
7    std::array<std::byte, N> data;
8    int width, height;
9  };
10
11 void vertical_center(WidgetContainer const& c) {
12   // 🧙 Magic happens
13 }
14
15 void center_bitmaps() {
16   std::array<bitmap, 3> = {🌇, 🏙, 🗻};
17   // What now?
18 }
```

```
 1 struct widget {
 2   rect bounding_box();
 3   void place(rect r);
 4 };
 5
 6 struct bitmap {
 7   std::array<std::byte, N> data;
 8   int width, height;
 9 };
10
11 void vertical_center(auto const& c) {
12   // 🧙 Magic happens
13 }
14
15 void center_bitmaps() {
16   std::array<bitmap, 3> = {🌇, 🏙, 🗻};
17   // What now?
18 }
```

```
1  struct widget {
2    rect bounding_box();
3    void place(rect r);
4  };
5
6  struct bitmap {
7    std::array<std::byte, N> data;
8    int width, height;
9  };
10
11 void vertical_center(auto const& c) {
12   // 🧙 Magic happens
13 }
14
15 void center_bitmaps() {
16   struct bitmap_widget : bitmap {
17     rect bounding_box() { ... }
18     void place(rect r) { ... }
19   };
20   std::array<bitmap_widget, 3> a = {🌇, 🏙, 🏔};
21   // What now?
22 }
```

```
1  struct widget {
2    rect bounding_box();
3    void place(rect r);
4  };
5
6  struct bitmap {
7    std::array<std::byte, N> data;
8    int width, height;
9  };
10
11 void vertical_center(auto const& c) {
12   // 🧙 Magic happens
13 }
14
15 void center_bitmaps() {
16   struct bitmap_widget : bitmap {
17     rect bounding_box() { ... }
18     void place(rect r) { ... }
19   };
20   std::array<bitmap_widget, 3> a = {🌇, 🏙, 🏔 };
21   vertical_center(a); // Boom! Code reused!
22 }
```

SUCCESS!

SUCCESS!

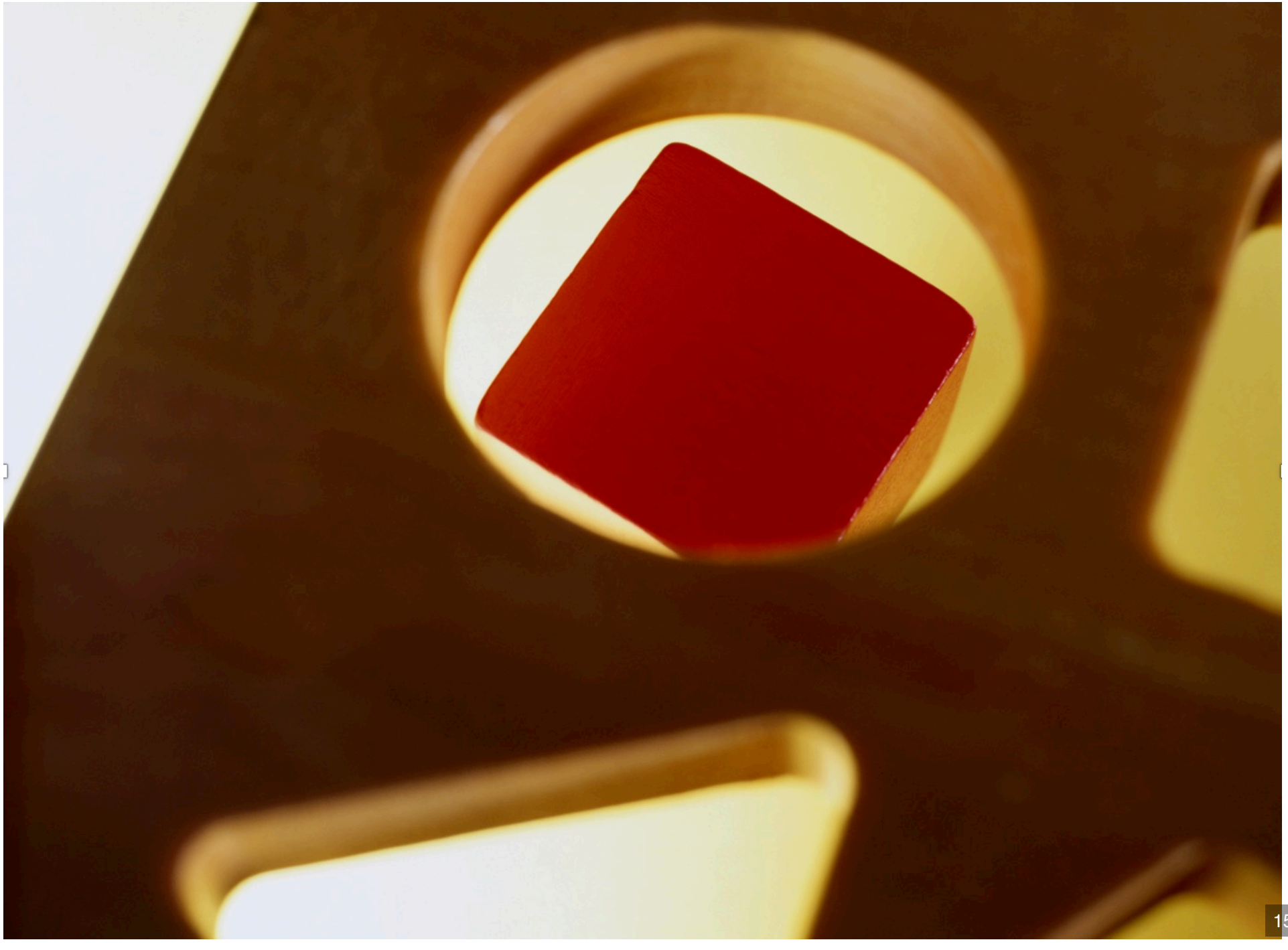
SUCCESS?

UNDERSTANDING CODE IS HARD,
WRITING CODE IS EASY

We love writing code!

We have tools!

Classes, inheritance, variant, optional, virtual functions!



HOW TO GENERALIZE

1. Understand **your** problems: What do you need?
2. Understand available algorithm: What does it do?
3. Describe both in terms of the problem, not C++ jargon!
4. Generalize

```cpp
struct bitmap {
  std::array<std::byte, N> data;
  int width, height;
};

void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```cpp
struct bitmap {
  std::array<std::byte, N> data;
  int width, height;
};

void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```cpp
struct bitmap {
  std::array<std::byte, N> data;
  int width, height;
};

void vertical_center(WidgetContainer const& c) {
  auto rects = tc::make_vector(tc::transform(c,
    [](widget const& w) { return w.bounding_box(); }
  ));
  // 🧙 Magic happens
  tc::for_each(tc::zip(c, rects),
    [&](widget& w, rect r) {
      widget.place(r);
    }
  );
}
```

```cpp
struct bitmap {
  std::array<std::byte, N> data;
  int width, height;
};

void vertical_center(ranges::forward_range auto rects)
  requires is_same_v<
    ranges::range_reference_t<decltype(rects)>,
    rect&
  >
 {
  // 🧙 Magic happens
 }
```

```cpp
struct bitmap {
  std::array<std::byte, N> data;
  int width, height;
};

void vertical_center(ranges::forward_range auto rects)
  requires is_same_v<
    ranges::range_reference_t<decltype(rects)>,
    rect&
  >
 {
  // 🧙 Magic happens
 }

void center_bitmaps() {
  std::array<bitmap, 3> a = {🌇, 🏙, 🏔};
  std::array<rect, 3> rects = { ... };
  vertical_center(rects); // Much better
}
```

1. `vertical_center` requires only minimal concepts
2. `vertical_center` is more generic, easier to use
3. Reduced dependencies

SHARED ALGORITHM, DIFFERENT DATA: GENERIC FUNCTION

- Define minimal required operations as concepts
- **Better**: *"requirements expressed in terms of fundamental and complete concepts."* (Stroustrup, P0557r1)
- Requires problem analysis!
- Don't generalize from single use case

CUSTOMIZING GENERIC FUNCTIONS

Client code → generic library code → client code

1. DISPATCH TO FUNCTION OVERLOADS

```
 1 void place(rect r, great_widget);
 2 void place(rect r, awesome_widget);
 3
 4 void place_items(rect bounds, forward_range auto rng) {
 5   auto const results = /* magic */;
 6   tc::for_each(
 7     tc::zip(results, rng),
 8     [](rect r, auto& item) { place(r, item); }
 9   );
10 }
11
12 void main() {
13   place_items(rect{}, std::vector<great_widget>());
14   place_items(rect{}, std::vector<awesome_widget>());
15 }
```

1. DISPATCH TO FUNCTION OVERLOADS

```
1  void place(rect r, great_widget);
2  void place(rect r, awesome_widget);
3
4  void place_items(rect bounds, forward_range auto rng) {
5    auto const results = /* magic */;
6    tc::for_each(
7      tc::zip(results, rng),
8      [](rect r, auto& item) { place(r, item); }
9    );
10 }
11
12 void main() {
13   place_items(rect{}, std::vector<great_widget>());
14   place_items(rect{}, std::vector<awesome_widget>());
15 }
```

1. DISPATCH TO FUNCTION OVERLOADS

```
1  void place(rect r, great_widget);
2  void place(rect r, awesome_widget);
3
4  void place_items(rect bounds, forward_range auto rng) {
5    auto const results = /* magic */;
6    tc::for_each(
7      tc::zip(results, rng),
8      [](rect r, auto& item) { place(r, item); }
9    );
10 }
11
12 void main() {
13   place_items(rect{}, std::vector<great_widget>());
14   place_items(rect{}, std::vector<awesome_widget>());
15 }
```

e.g. `std::size`, `std::begin`, `std::end`

2. OUTPUT FUNCTIONS

2. OUTPUT FUNCTIONS

```
1  template<
2    forward_range Rng,
3    invocable<rect, range_value_t<Rng>> F
4  >
5  void place_items(rect bounds, Rng const& rng, F func) {
6    auto const results = /* 🧙 magic */;
7    tc::for_each(
8      tc::zip(results, rng),
9      func
10   );
11 }
12
13 void main() {
14   place_items(
15     rect{},
16     std::vector<great_widget>(),
17     [](rect r, great_widget const& w) {}
18   );
19 }
```

2. OUTPUT FUNCTIONS

```
1  template<
2    forward_range Rng,
3    invocable<rect, range_value_t<Rng>> F
4  >
5  void place_items(rect bounds, Rng const& rng, F func) {
6    auto const results = /* 🧙 magic */;
7    tc::for_each(
8      tc::zip(results, rng),
9      func
10   );
11 }
12
13 void main() {
14   place_items(
15     rect{},
16     std::vector<great_widget>(),
17     [](rect r, great_widget const& w) {}
18   );
19 }
```

2. OUTPUT FUNCTORS

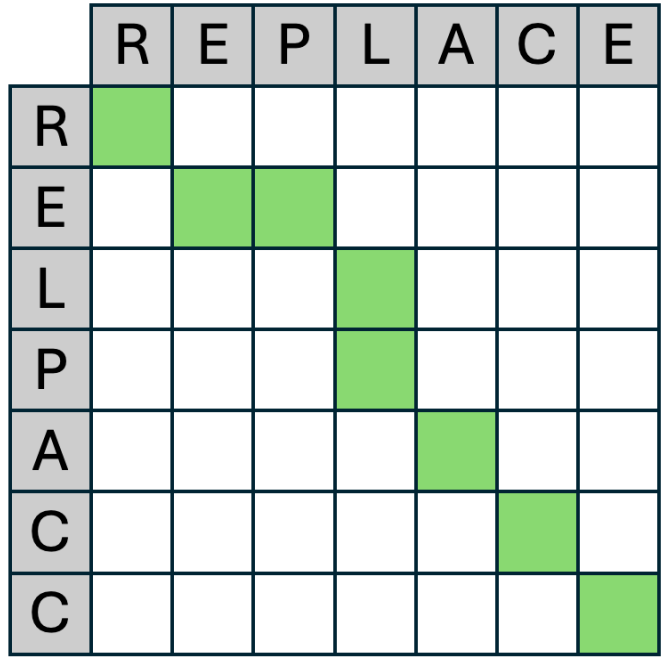
```
template<
  forward_range Rng,
  invocable<rect, range_value_t<Rng>> F
>
void place_items(rect bounds, Rng const& rng, F func) {
  auto it = std::begin(rng);
  /* 🧙 magic */;
  func(*it, rect{});
  ++it;
  /* 🧙 more magic */;
  func(*it, rect{});
}

void main() {
  place_items(
    rect{},
    std::vector<great_widget>(),
    [](rect r, great_widget const& w) {}
  );
}
```

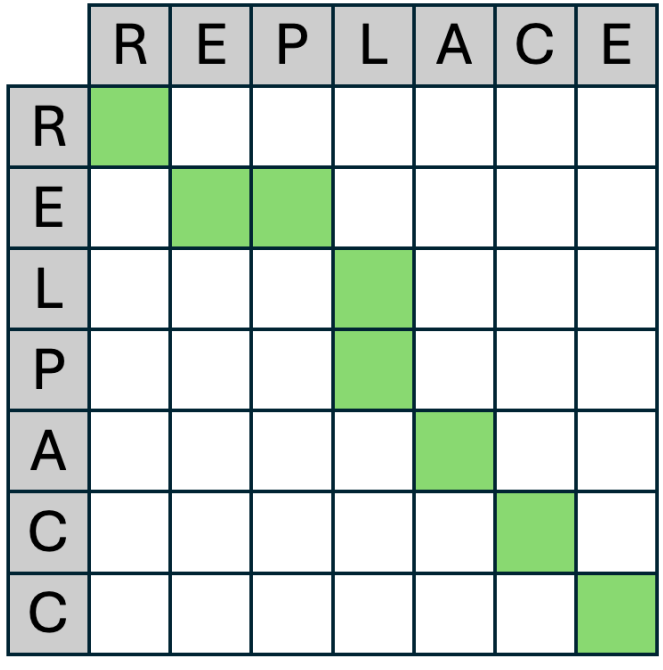
3. PASS FUNCTORS

```
template<
  forward_range Rng,
  predicate<range_value_t<Rng>> P,
  invocable<rect, range_value_t<Rng>> F
>
void place_items(rect bounds, Rng const& rng, P pred, F f)
  auto it = std::begin(rng);
  /* 🧙 magic */;
  if(pred(*it)) {
    f(*it, rect{});
  }
  ++it;
  /* 🧙 more magic */;
  if(pred(*it)) {
    f(*it, rect{});
  }
}
```

4. COMMAND PATTERN



4. COMMAND PATTERN




```
1 enum class match { skip_left, match, skip_right };
2 vector dynamic_programming(
3   forward_range auto first,
4   forward_range auto second
5 );
```

4. COMMAND PATTERN

```
1  enum class match { skip_left, match, skip_right };
2
3  struct SMatcher {
4    using cost = int;
5    constexpr static cost initial = 0;
6
7    cost accumulate_cost(cost prev, int i, int j, match m);
8    bool better(cost lhs, cost rhs) { return lhs < rhs; }
9  };
10
11 vector dynamic_programming(
12   forward_range auto first,
13   forward_range auto second,
14   auto matcher
15 );
```

4. COMMAND PATTERN

```
1  enum class match { skip_left, match, skip_right };
2
3  struct SMatcher {
4    using cost = int;
5    constexpr static cost initial = 0;
6
7    cost accumulate_cost(cost prev, int i, int j, match m);
8    bool better(cost lhs, cost rhs) { return lhs < rhs; }
9  };
10
11 vector dynamic_programming(
12   forward_range auto first,
13   forward_range auto second,
14   auto matcher
15 );
```

GENERIC FUNCTIONS

- Need clear requirements
- expressed in well-known concepts
- Reduce dependencies
- Flexible and customizable through customization points

```cpp
enum class match { skip_left, match, skip_right };

template<
  typename Derived, typename Cost, Cost c_costInitial
>
struct dynamic_programming {
private:
  vector<match> m_vecematch;

public:
  using const_iterator
    = typename vector<match>::const_reverse_iterator;
  using iterator = const_iterator;

  const_iterator begin() { return m_vecematch.rbegin(); }
  const_iterator end() { return m_vecematch.rend(); }

protected:
  void calculate(auto first, auto second);
};
```

```
enum class match { skip_left, match, skip_right };

template<
  typename Derived, typename Cost, Cost c_costInitial
>
struct dynamic_programming {
private:
  vector<match> m_vecematch;

public:
  using const_iterator
    = typename vector<match>::const_reverse_iterator;
  using iterator = const_iterator;

  const_iterator begin() { return m_vecematch.rbegin(); }
  const_iterator end() { return m_vecematch.rend(); }

protected:
  void calculate(auto first, auto second);
};
```

```
enum class match { skip_left, match, skip_right };

template<
  typename Derived, typename Cost, Cost c_costInitial
>
struct dynamic_programming {
private:
  vector<match> m_vecematch;

public:
  using const_iterator
    = typename vector<match>::const_reverse_iterator;
  using iterator = const_iterator;

  const_iterator begin() { return m_vecematch.rbegin(); }
  const_iterator end() { return m_vecematch.rend(); }

protected:
  void calculate(auto first, auto second);
};
```

```
enum class match { skip_left, match, skip_right };

template<
  typename Derived, typename Cost, Cost c_costInitial
>
struct dynamic_programming {
private:
  vector<match> m_vecematch;

public:
  using const_iterator
    = typename vector<match>::const_reverse_iterator;
  using iterator = const_iterator;

  const_iterator begin() { return m_vecematch.rbegin(); }
  const_iterator end() { return m_vecematch.rend(); }

protected:
  void calculate(auto first, auto second);
};
```

CURIOUSLY RECURRING TEMPLATE PATTERN

```
1  enum class match { skip_left, match, skip_right };
2
3  template<
4    typename Derived, typename Cost, Cost c_costInitial
5  >
6  struct dynamic_programming {
7    // ...
8  };
9
10 struct string_match
11   : dynamic_programming<string_match, int, 0>
12 {
13   bool better(int lhs, int rhs) { return lhs < rhs; }
14   int accumulate(int costPrev, int i, int j, match m) {
15     // ...
16   }
17 };
```

CURIOUSLY RECURRING TEMPLATE PATTERN

```
 1 enum class match { skip_left, match, skip_right };
 2
 3 template<
 4   typename Derived, typename Cost, Cost c_costInitial
 5 >
 6 struct dynamic_programming {
 7   // ...
 8 };
 9
10 struct string_match
11   : dynamic_programming<string_match, int, 0>
12 {
13   bool better(int lhs, int rhs) { return lhs < rhs; }
14   int accumulate(int costPrev, int i, int j, match m) {
15     // ...
16   }
17 };
```

SHARING CODE AND DATA: GENERIC CLASSES

SHARING CODE AND DATA: GENERIC CLASSES

- *Curiously Recurring Template Pattern (CRTP)*
- Compile-time polymorphism
- Generic interface, algorithms, data in templated base class
- Customization points in derived class

TEMPLATED BASE CLASS

TEMPLATED BASE CLASS

```
1  template<typename base>
2  redirected_process : base {
3    redirected_process& operator<<(std::string const& s)
4  };
5
6  struct async_base {
7    void write(std::string const& s);
8  };
9
10 struct sync_base {
11   void write(std::string const& s);
12 };
```

TEMPLATED BASE CLASS

```
template<typename base>
redirected_process : base {
  redirected_process& operator<<(std::string const& s)
};

struct async_base {
  void write(std::string const& s);
};

struct sync_base {
  void write(std::string const& s);
};
```

WHEN TO USE WHICH?

- Often a question of convenience
- Who owns data? Initialization order?
- What creates less friction, less casts?

MIXIN CLASS

MIXIN CLASS

```
1  struct widget_base {
2    void mouse_move(point const& pt);
3  };
4
5  struct button : widget_base {};
6  struct dropdown : widget_base {};
7
8  template<typename base>
9  struct beep : base {
10   void mouse_move(point const& pt) {
11     base::mouse_move(pt);
12     beep();
13   }
14 };
15
16 struct annoying_button : beep<button> {};
17 struct sane_button : button {};
18 struct annoying_dropdown : beep<dropdown> {};
```

MIXIN CLASS

```
1  struct widget_base {
2    void mouse_move(point const& pt);
3  };
4
5  struct button : widget_base {};
6  struct dropdown : widget_base {};
7
8  template<typename base>
9  struct beep : base {
10   void mouse_move(point const& pt) {
11     base::mouse_move(pt);
12     beep();
13   }
14 };
15
16 struct annoying_button : beep<button> {};
17 struct sane_button : button {};
18 struct annoying_dropdown : beep<dropdown> {};
```

MIXIN CLASS

```
 1 struct widget_base {
 2   void mouse_move(point const& pt);
 3 };
 4
 5 struct button : widget_base {};
 6 struct dropdown : widget_base {};
 7
 8 template<typename base>
 9 struct beep : base {
10   void mouse_move(point const& pt) {
11     base::mouse_move(pt);
12     beep();
13   }
14 };
15
16 struct annoying_button : beep<button> {};
17 struct sane_button : button {};
18 struct annoying_dropdown : beep<dropdown> {};
```

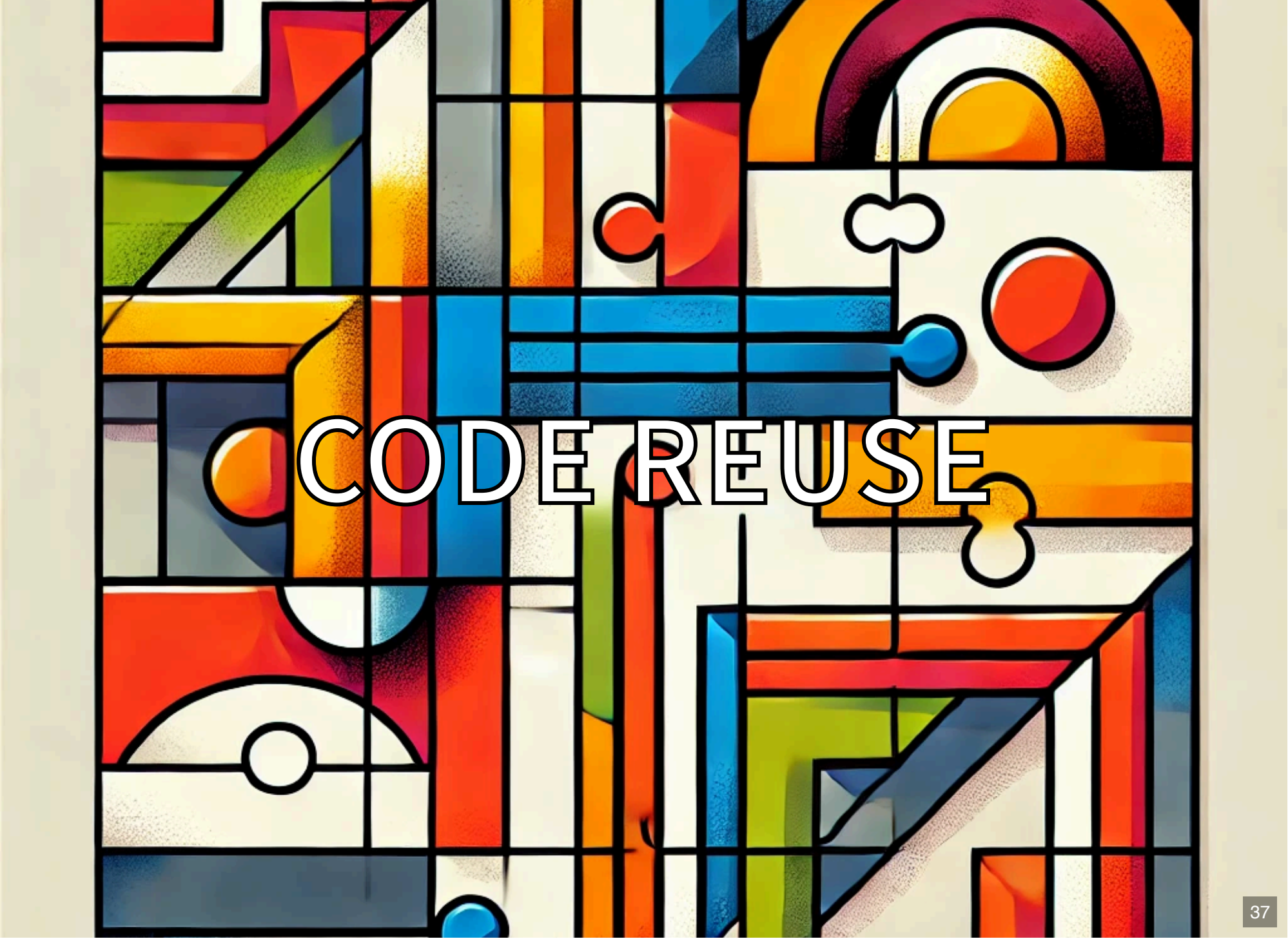
MIXIN CLASS

```
 1 struct widget_base {
 2   void mouse_move(point const& pt);
 3 };
 4
 5 struct button : widget_base {};
 6 struct dropdown : widget_base {};
 7
 8 template<typename base>
 9 struct beep : base {
10   void mouse_move(point const& pt) {
11     base::mouse_move(pt);
12     beep();
13   }
14 };
15
16 struct annoying_button : beep<button> {};
17 struct sane_button : button {};
18 struct annoying_dropdown : beep<dropdown> {};
```

What about inheritance and virtual functions?

What about inheritance and virtual functions?

What about `std::variant`?


CODE REUSE

CODE REUSE

- Implementation decision: DRY
- Reusing algorithm: Generic Function
- Reusing interface and data: Generic Classes

CODE REUSE

- Implementation decision: DRY
- Reusing algorithm: Generic Function
- Reusing interface and data: Generic Classes
- We reuse in unrelated contexts

CODE REUSE

- Implementation decision: DRY
- Reusing algorithm: Generic Function
- Reusing interface and data: Generic Classes
- We reuse in unrelated contexts
- Out of convenience, not design decision


RUNTIME
POLYMORPHISM

RUNTIME POLYMORPHISM

- Design decision!
- Strong coupling between types
- Share base classes and common interface

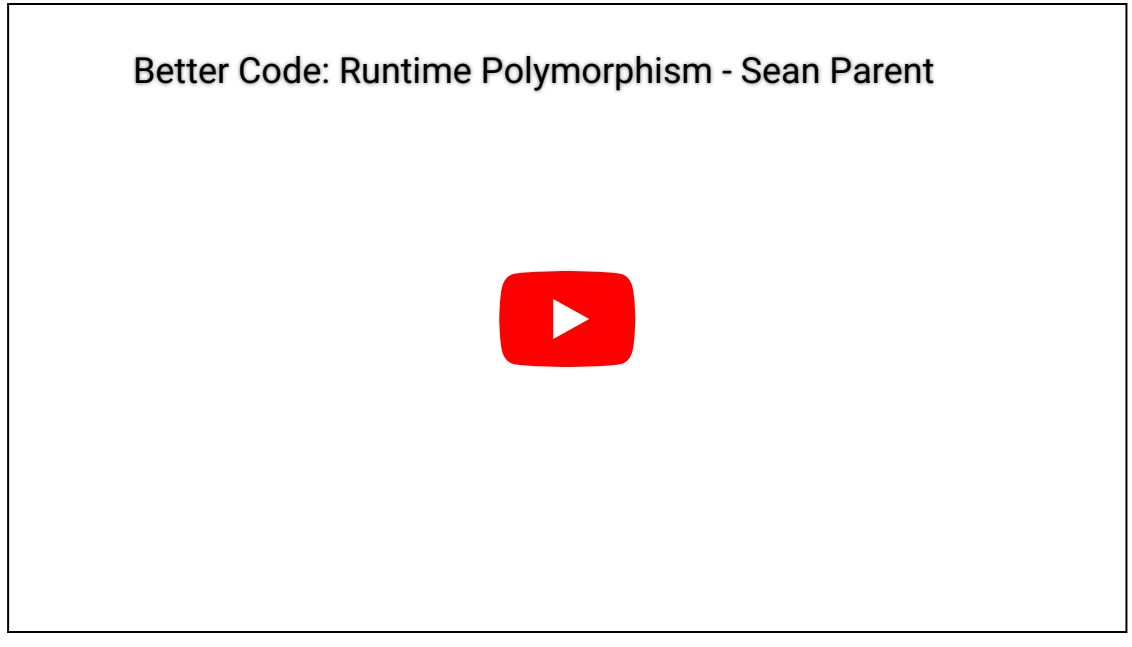
RUNTIME POLYMORPHISM

- Design decision!
- Strong coupling between types
- Share base classes and common interface
- Interface must have same meaning for all types

RUNTIME POLYMORPHISM

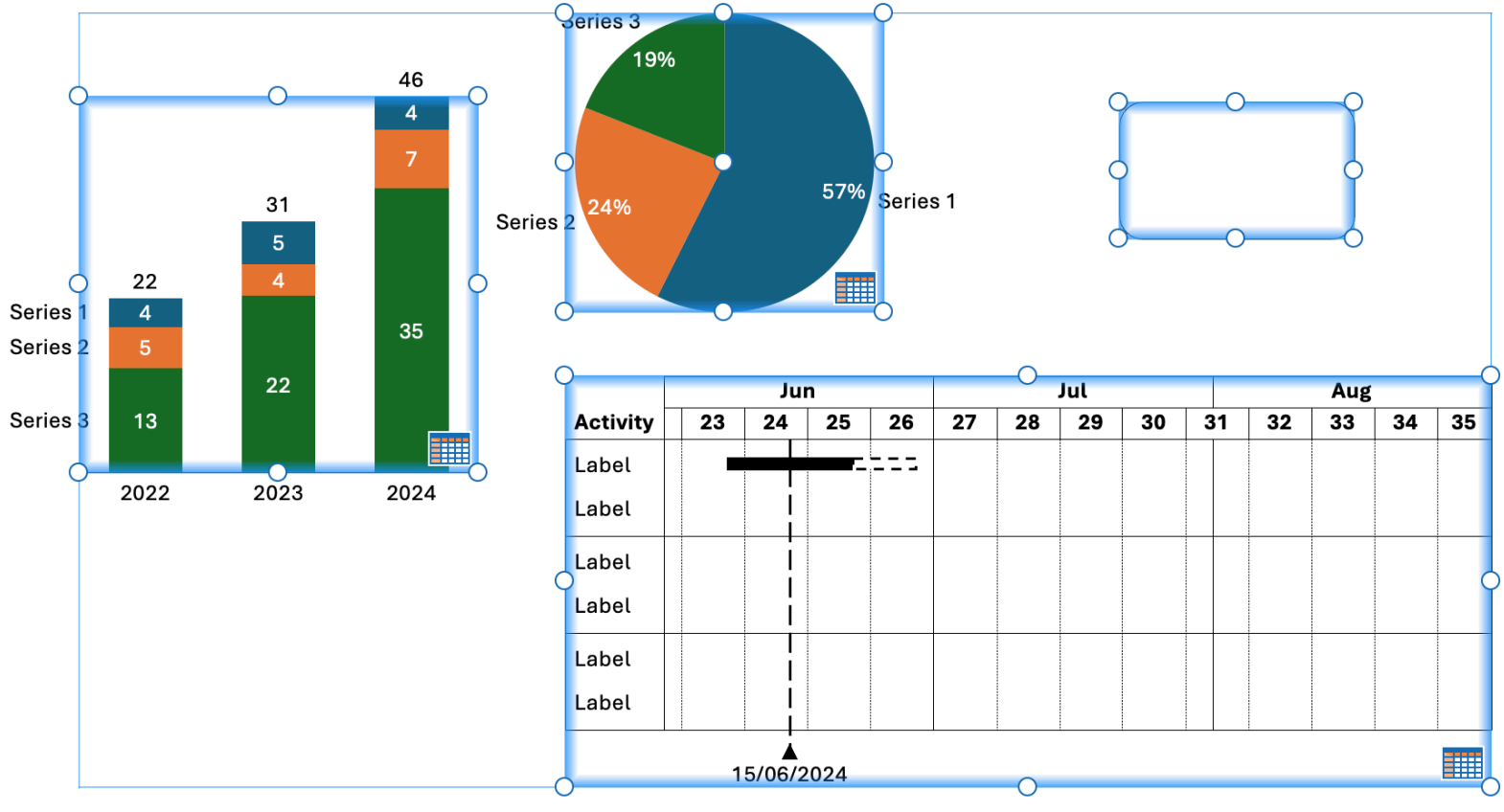
- Design decision!
- Strong coupling between types
- Share base classes and common interface
- Interface must have same meaning for all types
- Implementation must be correct for all types

RUNTIME POLYMORPHISM



"The requirement of a polymorphic type, by definition, comes from its use. There are no polymorphic types, only polymorphic uses of similar types."

RUNTIME POLYMORPHISM



RUNTIME POLYMORPHISM

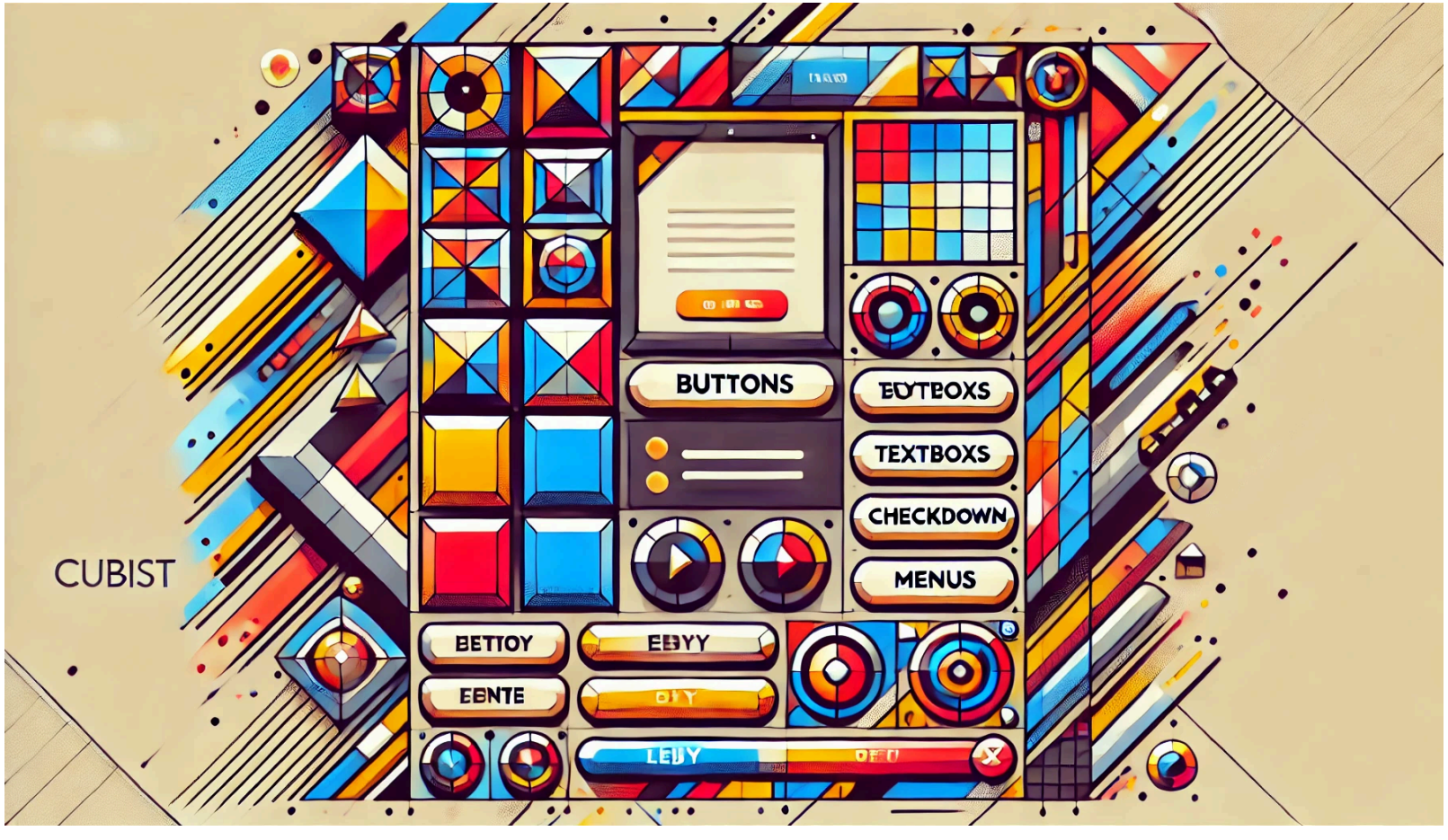
RUNTIME POLYMORPHISM

```
1 class object {
2   rect r;
3   bool dirty;
4
5   virtual void build_context_menu();
6   virtual void on_context_menu_click();
7
8   virtual void draw() = 0;
9 };
```

RUNTIME POLYMORPHISM

```
 1 class object {
 2   rect r;
 3   bool dirty;
 4
 5   virtual void build_context_menu();
 6   virtual void on_context_menu_click();
 7
 8   virtual void draw() = 0;
 9 };
10
11 class document {
12   vector<std::unique_ptr<object>> objects;
13
14   void select_all();
15   void group_all();
16 };
```

RUNTIME POLYMORPHISM



RUNTIME POLYMORPHISM

RUNTIME POLYMORPHISM

```
1 class dialog_widget {
2   virtual size min_size();
3   virtual void add_constraints(solver& s);
4   virtual void place(solver_result const& r);
5 };
6
7 class button : dialog_widget { /* ... */ };
8 class textbox : dialog_widget { /* ... */ };
```

RUNTIME POLYMORPHISM

```
 1 class dialog_widget {
 2   virtual size min_size();
 3   virtual void add_constraints(solver& s);
 4   virtual void place(solver_result const& r);
 5 };
 6
 7 class button : dialog_widget { /* ... */ };
 8 class textbox : dialog_widget { /* ... */ };
 9
10 class dialog {
11   void on_resize() {
12     // collect widget constraints
13     // solve
14     // place widgets
15   }
16 };
```

RUNTIME POLYMORPHISM

```
1 class my_dialog : dialog {
2   textbox txtbox;
3   button btn;
4
5   my_dialog()
6     : txtbox(*this, "Enter your text")
7     , btn(*this, [&]() { do_something(tb.text()); })
8   {}
9 };
```

RUNTIME POLYMORPHISM

```
1  class my_dialog : dialog {
2    textbox txtbox;
3    button btn;
4
5    my_dialog()
6      : txtbox(*this, "Enter your text")
7      , btn(*this, []() { do_something(tb.text()); })
8    {}
9
10   void for_each(std::function<dialog_widget&> f) {
11     f(textbox);
12     f(btn);
13   }
14 };
```

RUNTIME POLYMORPHISM

```
 1 class dialog_widget
 2 {
 3   dialog_widget(dialog& parent) {
 4   }
 5 };
 6
 7 class button : dialog_widget { /* ... */ };
 8 class textbox : dialog_widget { /* ... */ };
 9
10 class dialog {
11 };
```

RUNTIME POLYMORPHISM

```
 1 using bi = boost::intrusive;
 2 class dialog_widget
 3 : bi::list_base_hook<bi::link_mode<bi::auto_unlink>>
 4 {
 5   dialog_widget(dialog& parent) {
 6     parent.widgets.insert(this);
 7   }
 8 };
 9
10 class button : dialog_widget { /* ... */ };
11 class textbox : dialog_widget { /* ... */ };
12
13 class dialog {
14   bi::list<dialog_widget> widgets;
15 };
```

RUNTIME POLYMORPHISM

```
 1 using bi = boost::intrusive;
 2 class dialog_widget
 3 : bi::list_base_hook<bi::link_mode<bi::auto_unlink>>
 4 {
 5   dialog_widget(dialog& parent) {
 6     parent.widgets.insert(this);
 7   }
 8 };
 9
10 class button : dialog_widget { /* ... */ };
11 class textbox : dialog_widget { /* ... */ };
12
13 class dialog {
14   bi::list<dialog_widget> widgets;
15 };
```

RUNTIME POLYMORPHISM

```
1  using bi = boost::intrusive;
2  class dialog_widget
3  : bi::list_base_hook<bi::link_mode<bi::auto_unlink>>
4  {
5    dialog_widget(dialog& parent) {
6      parent.widgets.insert(this);
7    }
8  };
9
10 class button : dialog_widget { /* ... */ };
11 class textbox : dialog_widget { /* ... */ };
12
13 class dialog {
14   bi::list<dialog_widget> widgets;
15 };
```

RUNTIME POLYMORPHISM

Better Code: Runtime Polymorphism - Sean Parent

RUNTIME POLYMORPHISM

RUNTIME POLYMORPHISM

```
1  struct object_t {
2  };
3
4  using document_t = vector<object_t>;
5
6  void draw(document_t const& d) {
7    tc::for_each(d, [](auto const& o) {
8      draw(o);
9    });
10 }
11
12 void main() {
13   document_t d;
14   d.emplace_back(5);
15   d.emplace_back("Hello World");
16   draw(d);
17 }
```

RUNTIME POLYMORPHISM

```
 1 struct object_t {
 2 };
 3
 4 using document_t = vector<object_t>;
 5
 6 void draw(document_t const& d) {
 7   tc::for_each(d, [](auto const& o) {
 8     draw(o);
 9   });
10 }
11
12 void main() {
13   document_t d;
14   d.emplace_back(5);
15   d.emplace_back("Hello World");
16   draw(d);
17 }
```

RUNTIME POLYMORPHISM

```
 1 struct object_t {
 2 };
 3
 4 using document_t = vector<object_t>;
 5
 6 void draw(document_t const& d) {
 7   tc::for_each(d, [](auto const& o) {
 8     draw(o);
 9   });
10 }
11
12 void main() {
13   document_t d;
14   d.emplace_back(5);
15   d.emplace_back("Hello World");
16   draw(d);
17 }
```

RUNTIME POLYMORPHISM

```
1  struct object_t {
2    template<typename T>
3    object_t(T t) : self_(make_unique<model<T>>(move(t)) {}
4    // + move, copy & assignment
5
6    friend void draw(object_t const& o) { o.self_->draw_(); }
7
8  private:
9    struct concept_t {
10     virtual concept_t() = default;
11     virtual void draw_() = 0;
12   };
13
14   template<typename T>
15   struct model final : concept_t {
16     model(T t) : data_(move(t)) {}
17     void draw_() final {
18       draw(data_);
19     }
20     T data_;
21   };
22   unique_ptr<concept_t> self_;
```

```
23 };
```

RUNTIME POLYMORPHISM

```
1  struct object_t {
2    template<typename T>
3    object_t(T t) : self_(make_unique<model<T>>(move(t)) {}
4    // + move, copy & assignment
5
6    friend void draw(object_t const& o) { o.self_->draw_(); }
7
8  private:
9    struct concept_t {
10     virtual concept_t() = default;
11     virtual void draw_() = 0;
12   };
13
14   template<typename T>
15   struct model final : concept_t {
16     model(T t) : data_(move(t)) {}
17     void draw_() final {
18       draw(data_);
19     }
20     T data_;
21   };
22   unique_ptr<concept_t> self_;
23 };
```

RUNTIME POLYMORPHISM

```
1  struct object_t {
2    template<typename T>
3    object_t(T t) : self_(make_unique<model<T>>(move(t)) {}
4    // + move, copy & assignment
5
6    friend void draw(object_t const& o) { o.self_->draw_(); }
7
8  private:
9    struct concept_t {
10     virtual concept_t() = default;
11     virtual void draw_() = 0;
12   };
13
14   template<typename T>
15   struct model final : concept_t {
16     model(T t) : data_(move(t)) {}
17     void draw_() final {
18       draw(data_);
19     }
20     T data_;
21   };
22   unique_ptr<concept_t> self_;
23 };
```

RUNTIME POLYMORPHISM

```
1  struct object_t {
2    template<typename T>
3    object_t(T t) : self_(make_unique<model<T>>(move(t)) {}
4    // + move, copy & assignment
5
6    friend void draw(object_t const& o) { o.self_->draw_(); }
7
8  private:
9    struct concept_t {
10     virtual concept_t() = default;
11     virtual void draw_() = 0;
12   };
13
14   template<typename T>
15   struct model final : concept_t {
16     model(T t) : data_(move(t)) {}
17     void draw_() final {
18       draw(data_);
19     }
20     T data_;
21   };
22   unique_ptr<concept_t> self_;
23 };
```

RUNTIME POLYMORPHISM

```
 1 void draw(int i) {}
 2 void draw(std::string s) {}
 3
 4 using document_t = vector<object_t>;
 5
 6 void draw(document_t const& d) {
 7   tc::for_each(d, [](auto const& o) {
 8     draw(o);
 9   });
10 }
11
12 void main() {
13   document_t d;
14   d.emplace_back(5);
15   d.emplace_back("Hello World");
16   draw(d);
17 }
```

RUNTIME POLYMORPHISM

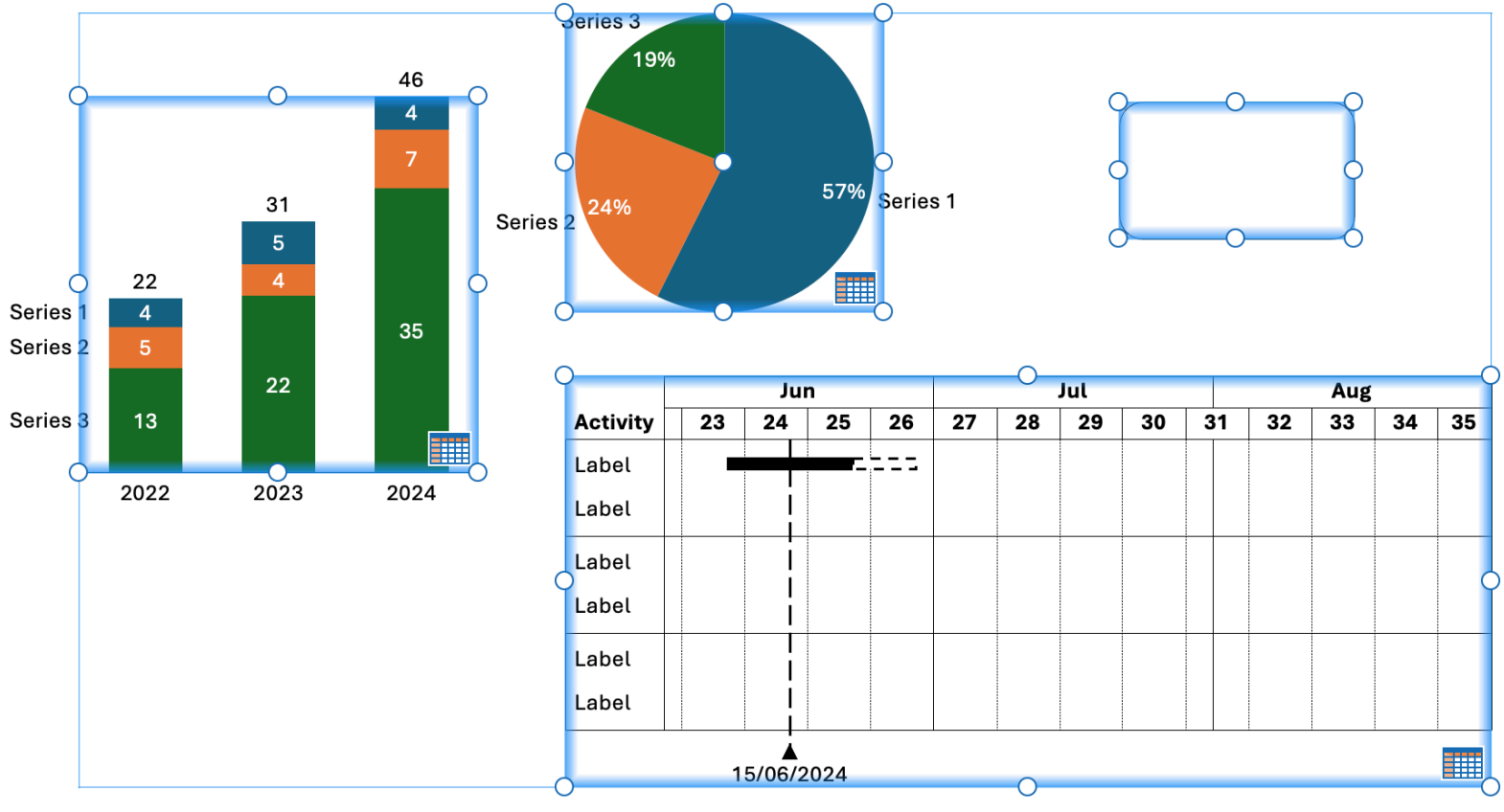
```
 1 void draw(int i) {}
 2 void draw(std::string s) {}
 3
 4 using document_t = vector<object_t>;
 5
 6 void draw(document_t const& d) {
 7   tc::for_each(d, [](auto const& o) {
 8     draw(o);
 9   });
10 }
11
12 void main() {
13   document_t d;
14   d.emplace_back(5);
15   d.emplace_back("Hello World");
16   draw(d);
17 }
```

RUNTIME POLYMORPHISM

Pre: Polymorphic use of stateful objects

1. *Interchangeable objects:* Owning base class pointer
2. *Objects with identity:* Intrusive containers
3. *Extensibility:* Hidden polymorphism

(WHEN TO AVOID) RUNTIME POLYMORPHISM



(WHEN TO AVOID) RUNTIME POLYMORPHISM

(WHEN TO AVOID) RUNTIME POLYMORPHISM

```
1  void update_data_model() {
2    uiobjects.emplace_back(drag_handle{ corner_rect() });
3  }
4
5  void mouse_move() {
6    // Check if mouse hovers! State!
7    tc::for_each(uiobjects, [](auto& o) { o.mouse_move(); });
8  }
9
10 void draw() {
11   // Draw hover texture
12   tc::for_each(uiobjects, [](auto& o) { o.draw(); });
13 }
```

(WHEN TO AVOID) RUNTIME POLYMORPHISM

```
1  void update_data_model() {
2    uiobjects.emplace_back(drag_handle{ corner_rect() });
3  }
4
5  void mouse_move() {
6    // Check if mouse hovers! State!
7    tc::for_each(uiobjects, [](auto& o) { o.mouse_move(); });
8  }
9
10 void draw() {
11   // Draw hover texture
12   tc::for_each(uiobjects, [](auto& o) { o.draw(); });
13 }
```

(WHEN TO AVOID) RUNTIME POLYMORPHISM

```
1  void update_data_model() {
2    uiobjects.emplace_back(drag_handle{ corner_rect() });
3  }
4
5  void mouse_move() {
6    // Check if mouse hovers! State!
7    tc::for_each(uiobjects, [](auto& o) { o.mouse_move(); });
8  }
9
10 void draw() {
11   // Draw hover texture
12   tc::for_each(uiobjects, [](auto& o) { o.draw(); });
13 }
```

(WHEN TO AVOID) RUNTIME POLYMORPHISM

```
 1 void mouse_move() {
 2   set_dirty(); // trigger redraw
 3 }
 4
 5 void draw() {
 6   if(corner_rect().contains(mouse_position())) {
 7     draw(hover_texture(), corner_rect());
 8   } else {
 9     draw(normal_texture(), corner_rect());
10   }
11 }
```

FINALLY ...

FINALLY …

`std::variant`

FINALLY ...

std::variant

```
1 struct result {};
2 using errormsg = std::string;
3
4 variant<result, errormsg>
5 search_img(std::string query);
```

VARIANT

```
1  struct result {};
2  using errormsg = std::string;
3
4  variant<result, errormsg>
5  search_img(std::string query);
6
7  void main() {
8    tc::fn_visit(
9      [](errormsg const& msg) {},
10     [](result const& s) {}
11   )(search_img("cat"));
12 }
```

VARIANT

```
 1 struct result {};
 2 using errormsg = std::string;
 3
 4 variant<result, errormsg>
 5 search_img(std::string query);
 6
 7 void main() {
 8   tc::fn_visit(
 9     [](errormsg const& msg) {},
10     [](result const& s) {}
11   )(search_img("cat"));
12 }
```

VARIANT

```
1  struct result {};
2  using errormsg = std::string;
3
4  variant<result, errormsg>
5  search_img(std::string query);
6
7  void main() {
8    tc::fn_visit(
9      [](errormsg const& msg) {
10       ShowAlert(msg);
11     },
12     [](result const& s) {
13       std::cout << "I have found something.";
14     }
15   )(search_img("cat"));
16 }
```

VARIANT

```
1  struct result {};
2  struct result2 {};
3  using errormsg = std::string;
4
5  variant<result, result2, errormsg>
6  search_img(std::string query);
7
8  void main() {
9    tc::fn_visit(
10     [](errormsg const& msg) {
11       ShowAlert(msg);
12     },
13     [](result const& s) {
14       std::cout << "I have found something.";
15     },
16     [](result2 const& s) {
17       std::cout << "I have found something different.";
18     }
19   )(search_img("cat"));
20 }
```

VARIANT

```
 1 struct result {};
 2 struct result2 {};
 3 struct result3 {};
 4
 5 void download(
 6   variant<result, result2, result3> const& t
 7 ) {
 8   tc::fn_visit(
 9     [](result const& s) {},
10     [](result2 const& s) {},
11     [](result3 const& s) {}
12   )(t);
13 }
```

VARIANT

```
 1 struct result {};
 2 struct result2 {};
 3 struct result3 {};
 4
 5 void download_thumbnail(
 6   variant<result, result2, result3> const& t
 7 ) {
 8   tc::fn_visit(
 9     [](result const& s) {},
10     [](result2 const& s) {},
11     [](result3 const& s) {}
12   )(t);
13 }
```

VARIANT

```
 1 struct result {};
 2 struct result2 {};
 3 struct result3 {};
 4
 5 void authenticate(
 6   variant<result, result2, result3> const& t
 7 ) {
 8   tc::fn_visit(
 9     [](result const& s) {},
10     [](result2 const& s) {},
11     [](result3 const& s) {}
12   )(t);
13 }
```

VARIANT

```
 1 struct result_t {
 2   template<typename T>
 3   result_t(T t) /* ... */
 4
 5   void authenticate();
 6   void download_image();
 7   void download_thumbnail();
 8
 9 private:
10   struct concept_t;
11   /* ... */
12   std::unique_ptr<concept_t> self_;
13 };
```

CONCLUSION

- Build generic library to maximize code reuse

CONCLUSION

- Build generic library to maximize code reuse
- Choose abstractions carefully

CONCLUSION

- Build generic library to maximize code reuse
- Choose abstractions carefully
- Different abstractions are interchangeable

CONCLUSION

- Build generic library to maximize code reuse
- Choose abstractions carefully
- Different abstractions are interchangeable
- But they don't have the same cost

CONCLUSION

- Build generic library to maximize code reuse
- Choose abstractions carefully
- Different abstractions are interchangeable
- But they don't have the same cost
- Revisit your choices occasionally

CONCLUSION

- Build generic library to maximize code reuse
- Choose abstractions carefully
- Different abstractions are interchangeable
- But they don't have the same cost
- Revisit your choices occasionally
- Your code may have "evolved"


THANK YOU