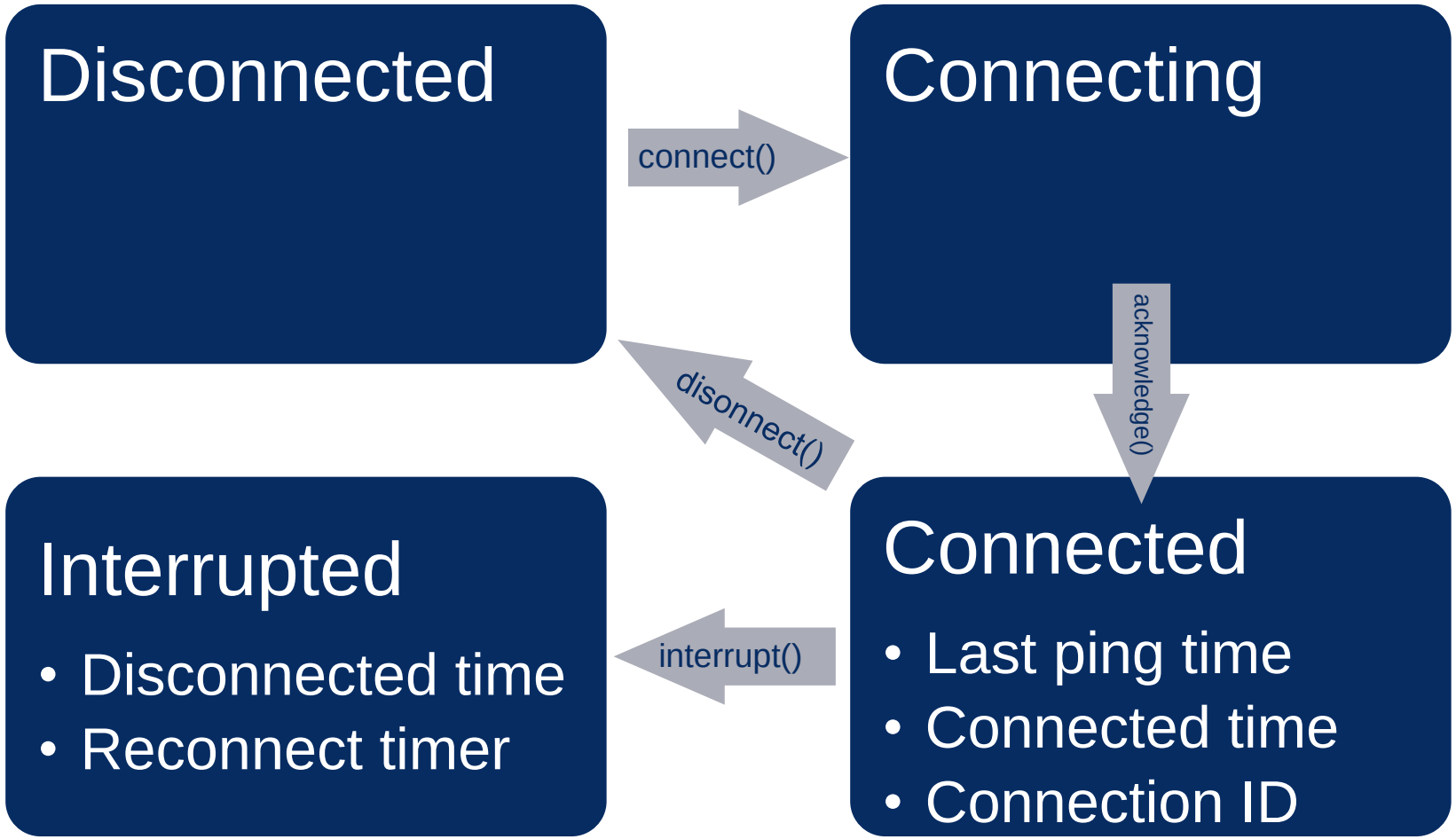
A variety of variants

Dvir Yitzchaki

CEVA®


STATE MACHINES
STATE MACHINES EVERYWHERE

State machines



Kalle Huttunen - Implementing State Machines with std::variant

State machines – first try

```
struct Connection {
  enum class State {
    DISCONNECTED,
    CONNECTING,
    CONNECTED,
    CONNECTION_INTERRUPTED
  };

  std::string m_serverAddress;
  ConnectionId m_id;
  std::chrono::system_clock::time_point m_connectedTime;
  std::optional<std::chrono::milliseconds> m_lastPingTime;
  std::chrono::system_clock::time_point m_disconnectedTime;
  Timer m_reconnectTimer;

  State m_state = State::DISCONNECTED;
};
```

State machines – first try

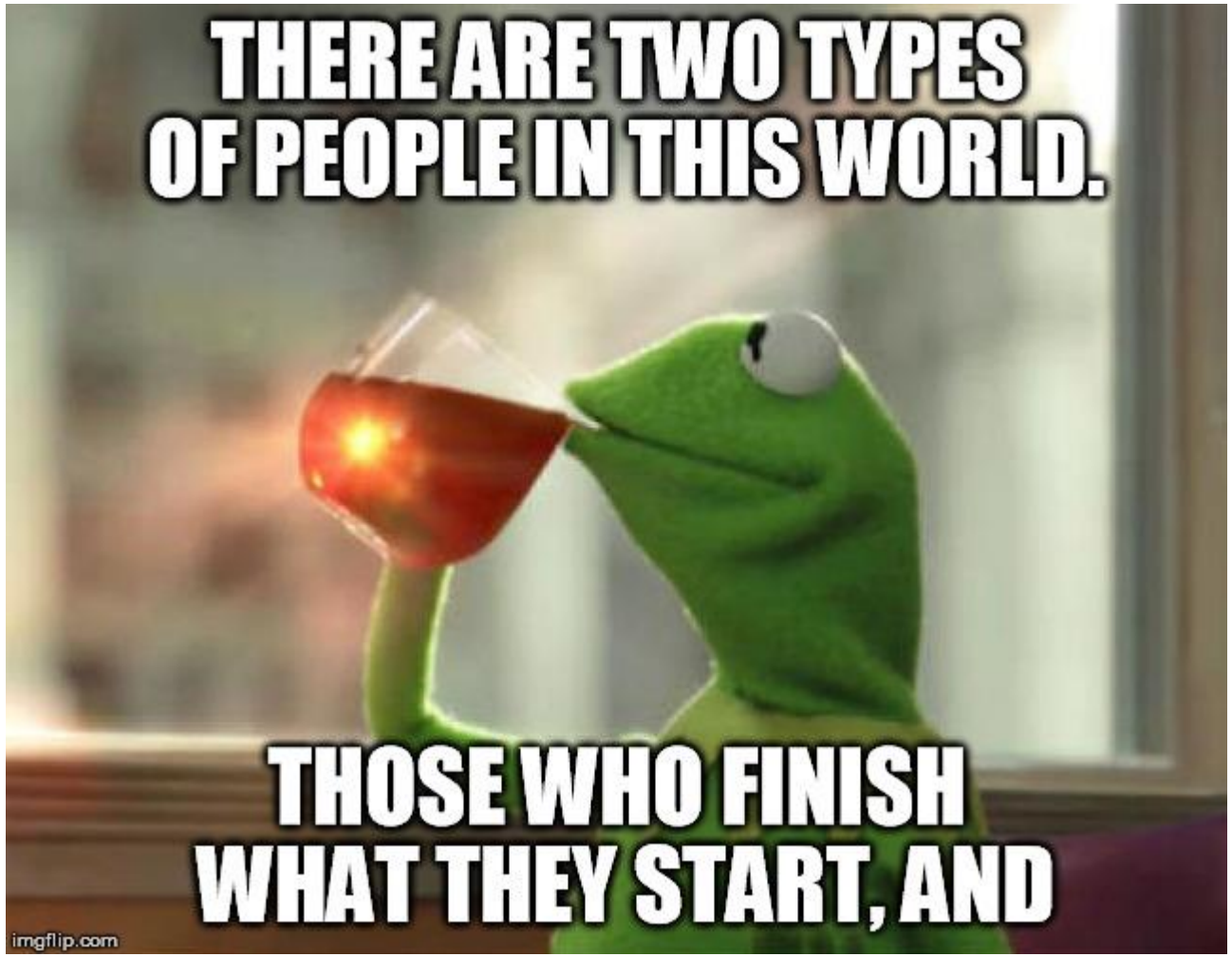
```
void interrupt() {
  switch (m_state) {
  case State::DISCONNECTED:
  case State::CONNECTING:
    m_state = State::DISCONNECTED;
    break;
  case State::CONNECTED:
    m_disconnectedTime = std::chrono::system_clock::now();
    notifyInterrupted(m_disconnectedTime);
    m_reconnectTimer = Timer{5000};
    m_state = State::CONNECTION_INTERRUPTED;
    break;
  case State::CONNECTION_INTERRUPTED:
    break;
  }
}
```

State machines – first try

```
void disconnect() {
  // maybe need to kill timer?
  m_state = State::DISCONNECTED;
}
```

State machines – problems

- Data is accessible where it shouldn’t be
- Can’t use RAII
- We want to make illegal states unrepresentable


THERE ARE TWO TYPES
OF PEOPLE IN THIS WORLD.
THOSE WHO FINISH
WHAT THEY START, AND
imgflip.com

Types

- WHAT IS A TYPE?
  - The set of values that can inhabit an expression
  - may be finite or "infinite"
  - characterized by cardinality
- Expressions have types
  - A program has a type

Using Types Effectively - Ben Deane - CppCon 2016

LET'S PLAY A GAME

To help us get thinking about types.

I'll tell you a type.

You tell me how many values it has.

There are no tricks: if it seems obvious, it is!

LEVEL 1

Types as sets of values

LEVEL 1

How many values?

```
bool;
```

LEVEL 1

How many values?

```
bool;
```

2 (true and false)

LEVEL 1

How many values?

```
char;
```

LEVEL 1

How many values?

```
char;
```

256

LEVEL 1

How many values?

```
void;
```

LEVEL 1

How many values?

```
void;
```

0

LEVEL 1

How many values?

```
enum FireSwampDangers : int8_t {
  FLAME_SPURTS,
  LIGHTNING_SAND,
  ROUSES
};
```

LEVEL 1

How many values?

```
enum FireSwampDangers : int8_t {
  FLAME_SPURTS,
  LIGHTNING_SAND,
  ROUSES
};
```

3

LEVEL 1

How many values?

```
template <typename T>
struct Foo {
  T m_t;
};
```

LEVEL 1

How many values?

```
template <typename T>
struct Foo {
  T m_t;
};
```

As much as T

END OF LEVEL 1

Algebraic data types

Algebraically, a type is the number of values that inhabit it.

These types are equivalent:

```
bool;

enum class InatorButtons {
  ON_OFF,
  SELF_DESTRUCT
};
```

Let's move on to level 2.

LEVEL 2

Aggregating Types

LEVEL 2

How many values?

```
std::pair<char, bool>;
```

LEVEL 2

How many values?

```
std::pair<char, bool>;
```

256 * 2 = 512

LEVEL 2

How many values?

```
struct Foo {
  char a;
  bool b;
};
```

LEVEL 2

How many values?

```
struct Foo {
  char a;
  bool b;
};
```

256 * 2 = 512

LEVEL 2

How many values?

```
std::tuple<bool, bool, bool>;
```

LEVEL 2

How many values?

```
std::tuple<bool, bool, bool>;
```

2 * 2 * 2 = 8

LEVEL 2

How many values?

```
template <typename T, typename U>
struct Foo {
  T m_t;
  U m_u;
};
```

LEVEL 2

How many values?

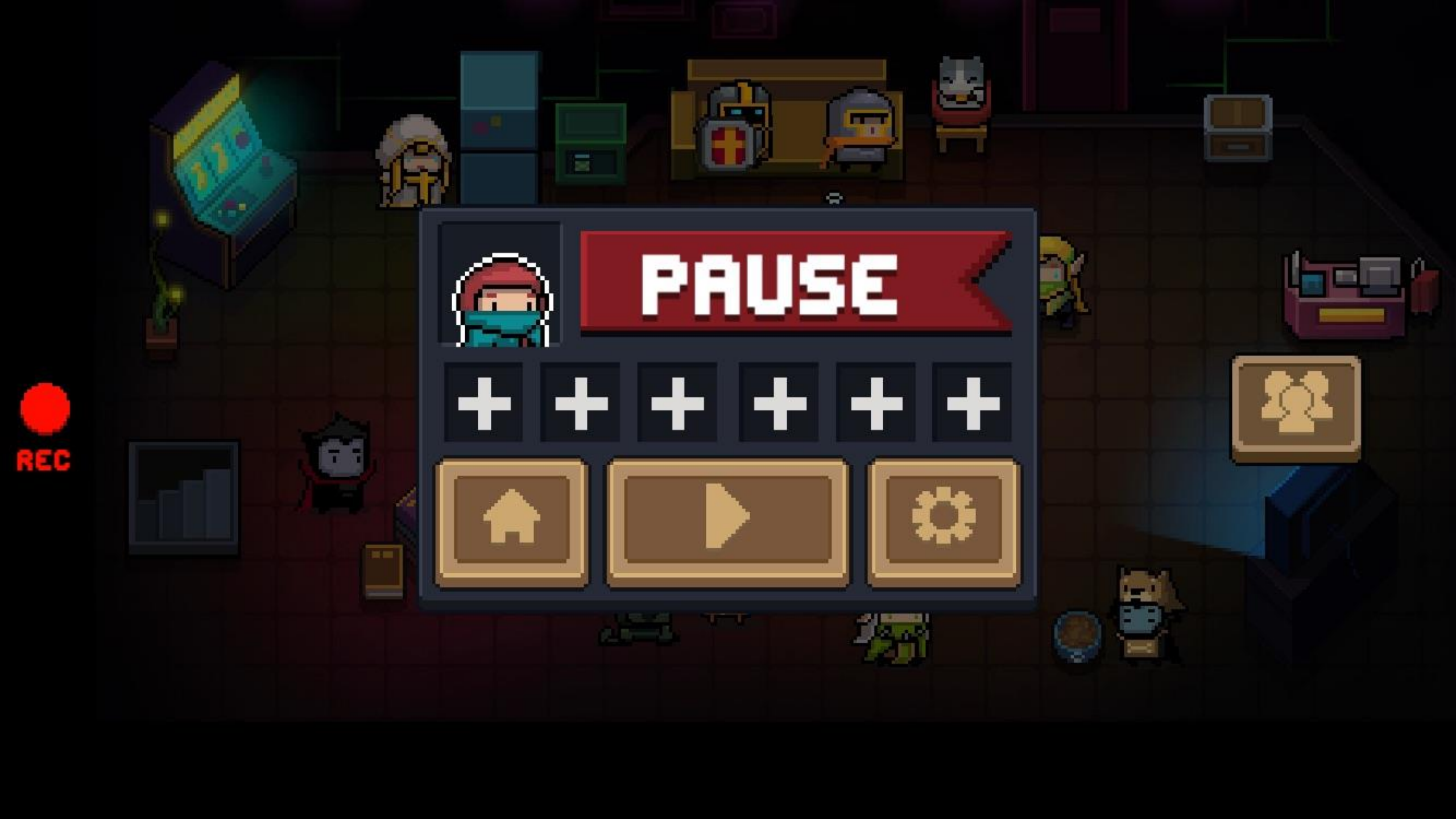
```
template <typename T, typename U>
struct Foo {
  T m_t;
  U m_u;
};
```

(# of values in T) * (# of values in U)

END OF LEVEL 2

Product type

- When two types are "concatenated" into one compound type, we multiply the # of inhabitants of the components.
- This kind of compounding gives us a product type.


PAUSE
REC

std::variant

```
std::variant<int, std::string> v;

v = 42;
std::cout << std::get<int>(v) << '\n'; // prints 42

try {
  std::cout << std::get<std::string>(v) << '\n'; // throws
}
catch (const std::bad_variant_access &e) {
  std::cout << e.what() << '\n'; // prints 'bad variant access'
}

// std::cout << std::get<long>(v) << '\n'; // doesn't compile

v = "forty two";
std::cout << std::get<std::string>(v) << '\n'; // prints 'forty two'
```

std::variant

```
if (auto p = std::get_if<std::string>(&v)) {
  std::cout << "we have a string: " << *p << '\n';}
else {
  std::cout << "we don't have a string\n";
}
```

std::visit

```
using Var = std::variant<int, std::string>;

Var duplicate(const Var& v) {
  struct Duplicater {
    Var operator()(int i) { return i * 2; }
    Var operator()(const std::string &s) {
      return s + s;
    }
  };
  return std::visit(Duplicater{}, v);
}
```

History

- 2002 – Boost.Variant started by Eric Friedman, Itay Maman
- 2004 – Release on Boost 1.31.0
- 2014 – std::variant proposed to C++ standard by Axel Naumann - N4218
- 2017 – std::variant released as part of C++17

Implementations

- Standard Template Library - http://en.cppreference.com/w/cpp/utility/variant
- Boost - http://www.boost.org/doc/libs/1_66_0/doc/html/variant.html
- Qt - http://doc.qt.io/qt-5/qvariant.html
- Michael Park – (back ported std::variant to C++11) https://github.com/mpark/variant
- Jonathan Müller - https://foonathan.net/doc/type_safe
- Mapbox - https://github.com/mapbox/variant
- Gnome - https://developer.gnome.org/glibmm/unstable/classGlib_1_1Variant.html

CONTINUE

LEVEL 3

Alternating Types

LEVEL 3

How many values?

```
std::optional<char>;
```

LEVEL 3

How many values?

```
std::optional<char>;
```

256 + 1 = 257

LEVEL 3

How many values?

```
std::variant<char, bool>;
```

LEVEL 3

How many values?

```
std::variant<char, bool>;
```

256 + 2 = 258

LEVEL 3

How many values?

```
template <typename T, typename U>
struct Foo {
  std::variant<T, U> m_v;
};
```

LEVEL 3

How many values?

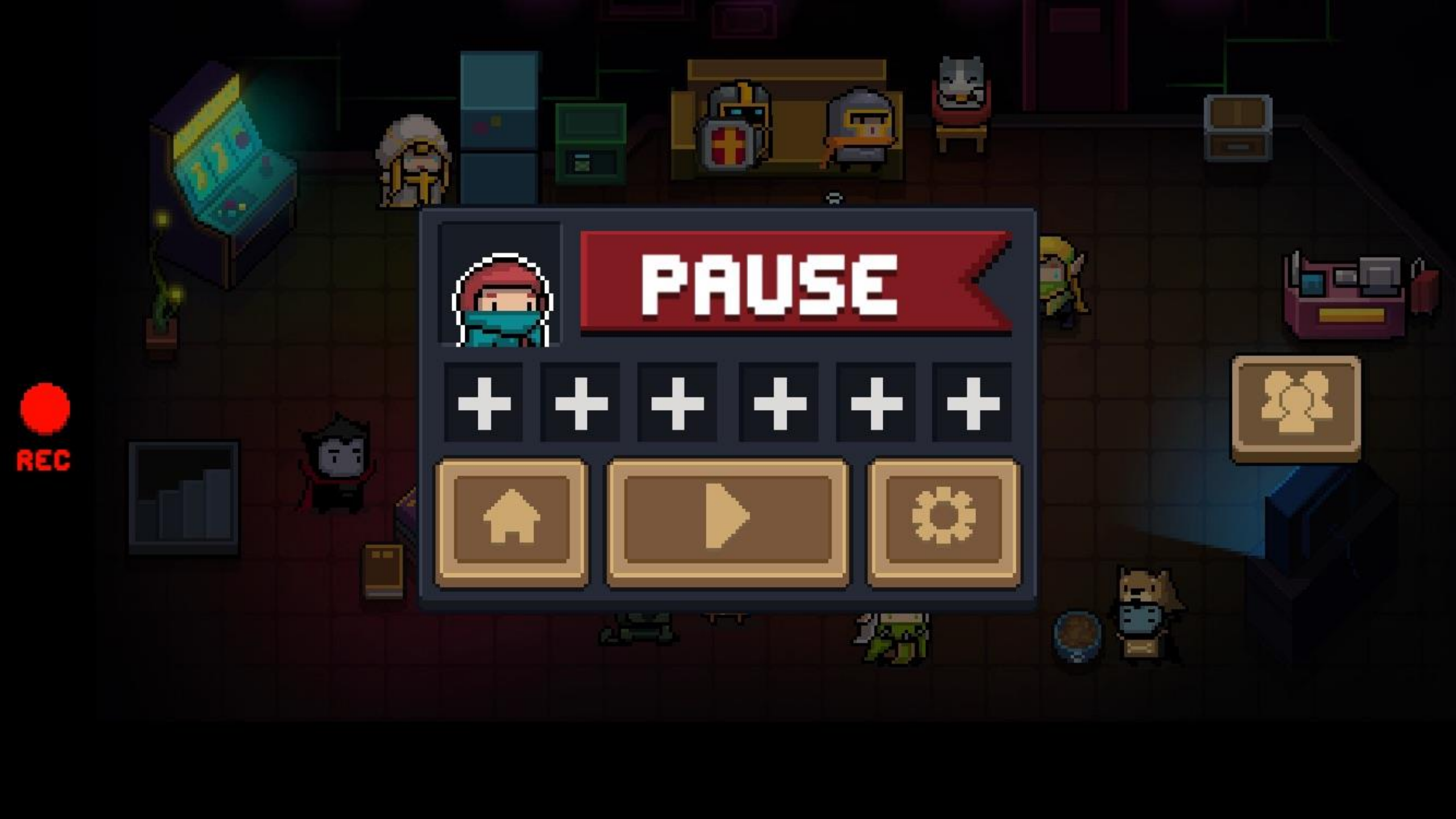
```
template <typename T, typename U>
struct Foo {
  std::variant<T, U> m_v;
};
```

(# of values in T) + (# of values in U)

END OF LEVEL 3

Sum type

- When two types are "alternated" into one compound type, we add the # of inhabitants of the components.
- This kind of compounding gives us a sum type.


PAUSE
REC

State machines – variant

```
struct Connection {
  struct Disconnected {};
  struct Connecting {};
  struct Connected {
    ConnectionId m_id;
    std::chrono::system_clock::time_point m_connectedTime;
    std::optional<std::chrono::milliseconds> m_lastPingTime;
  };
  struct ConnectionInterrupted {
    std::chrono::system_clock::time_point m_disconnectedTime;
    Timer m_reconnectTimer;
  };

  using State =
      std::variant<Disconnected, Connecting, Connected, ConnectionInterrupted>;

  State m_state;

  std::string m_serverAddress;
};
```

State machines – variant

```
void disconnect() {
  m_state = Disconnected();
  // if state was interrupted then ~Timer() would be called
}
```

State machines – variant

```
void interrupt() {
    struct InterruptedEvent {
      InterruptedEvent(Connection &c) : m_c(c) {}

      State operator()(const Disconnected &) { return Disconnected(); }

      State operator()(const Connecting &) { return Disconnected(); }

      State operator()(const Connected &) {
        const auto now = std::chrono::system_clock::now();
        m_c.notifyInterrupted(now);
        return ConnectionInterrupted(now, 5000);
      }

      State operator()(const ConnectionInterrupted &s) { return s; }

    private:
      Connection &m_c;
    };
    m_state = std::visit(InterruptedEvent(*this), m_state);
  }
```

State machines – variant

```
struct InterruptedEvent {
    InterruptedEvent(Connection &c) : m_c(c) {}

    template <typename T> State operator()(const T &) { return Disconnected(); }

    State operator()(const Connected &) {
      const auto now = std::chrono::system_clock::now();
      m_c.notifyInterrupted(now);
      return ConnectionInterrupted(now, 5000);
    }

    State operator()(const ConnectionInterrupted &s) { return s; }

  private:
    Connection &m_c;
  };

  void interrupt() { m_state = std::visit(InterruptedEvent(*this), m_state); }
```


WHEN FAMILY VISITS
Oh, no.
Now I have to act normal.
memegenerator.net

Lambda visitation

```
void interrupt() {
  m_state = std::visit(
      overload(
          [this](const Connected &) -> State {
            const auto now = std::chrono::system_clock::now();
            notifyInterrupted(now);
            return ConnectionInterrupted(now, 5000);
          },
          [](const ConnectionInterrupted &s) -> State { return s; },
          [](const auto & /*default*/) -> State { return Disconnected(); }),
      m_state);
}
```

Inheriting from lambdas – c++17

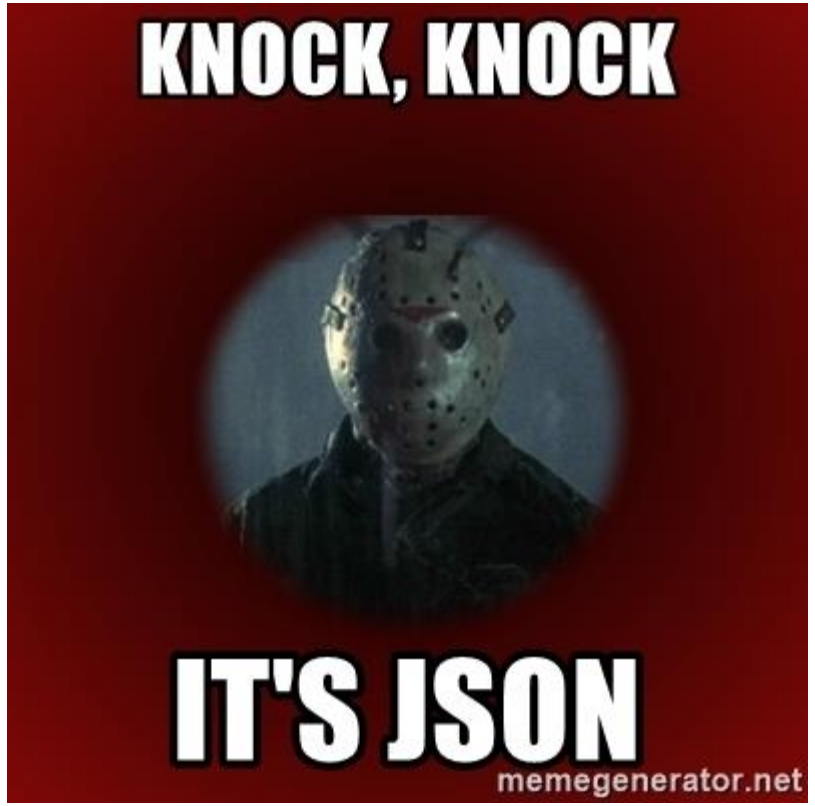
```
template <typename... Funcs> struct overload_set : Funcs... {
  using Funcs::operator()...; // exposes operator() from every base
};

template <typename... Funcs>
overload_set<Funcs...> overload(Funcs &&... funcs) {
  return {std::forward<Funcs>(funcs)...};
}
```

Inherting from lambdas – c++11

```
template <typename...> struct overload_set {
  void operator()() {}
};

template <typename Func, typename... Funcs>
struct overload_set<Func, Funcs...> : Func, overload_set<Funcs...> {
  using Func::operator();
  using overload_set<Funcs...>::operator();

  overload_set(const Func &f, const Funcs &... fs)
      : Func(f), overload_set<Funcs...>(fs...) {}
};

template <typename... Funcs>
overload_set<Funcs...> overload(Funcs &&... funcs) {
  return {std::forward<Funcs>(funcs)...};
}
```

constexpr if visitation

```
void interrupt() {
  m_state = std::visit(
      [this](const auto &state) -> State {
        // Needed to properly compare the types
        using T = std::decay_t<decltype(state)>;
        if constexpr (std::is_same_v<T, Connected>) {
          const auto now = std::chrono::system_clock::now();
          notifyInterrupted(now);
          return ConnectionInterrupted(now, 5000);
        } else if constexpr (std::is_same_v<T, ConnectionInterrupted>) {
          return state;
        } else {
          return Disconnected();
        }
      },
      m_state);
}
```


KNOCK, KNOCK
IT'S JSON
memegenerator.net

```
{
 "firstName": "John",
 "lastName": "Smith",
 "isAlive": true,
 "age": 27,
 "address":
 {
  "streetAddress": "21 2nd Street",
  "city": "New York"
 },
 "phoneNumbers":
 [
  {
   "type": "home",
   "number": "212 555-1234"
  },
  {
   "type": "office",
   "number": "646 555-4567"
  }
 ],
 "spouse": null
}
```

JSON

- JSON data types
  - Null
  - Number
  - String
  - Boolean
  - Array
  - Object

JSON – first try

```
using JsonValue =
    std::variant<nullptr_t,                                  // null
                 int,                                        // number
                 std::string,                                // string
                 bool,                                       // boolean
                 std::vector<JsonValue>,                     // array
                 std::vector<std::pair<std::string, JsonValue>> // object
                 >;
```

error C2065: 'JsonValue': undeclared identifier

JSON – add a level of indirection

```
struct JsonArray;
struct JsonObject;

using JsonValue = std::variant<nullptr_t,                    // null
                               int,                          // number
                               std::string,                  // string
                               bool,                         // boolean
                               std::unique_ptr<JsonArray>,   // array
                               std::unique_ptr<JsonObject>   // object
                               >;
```

JSON – add a level of indirection

```
template <typename T, typename... Args>
std::vector<T> make_vector(Args &&... args) {
  std::vector<T> res;
  (res.emplace_back(std::forward<Args>(args)), ...);
  return res;
}

struct JsonArray {
  std::vector<JsonValue> array;

  template <typename... Args>
  JsonArray(Args &&... args)
      : array(make_vector<JsonValue>(std::forward<Args>(args)...)) {}
};

struct JsonObject {
  std::vector<std::pair<std::string, JsonValue>> object;

  template <typename... Args>
  JsonObject(Args &&... args)
      : object(make_vector<std::pair<std::string, JsonValue>>(std::forward<Args>(args)...)) {}
};
```

JSON – construction

```
JsonValue construct() {
  using namespace std::string_literals;

  using p = std::pair<std::string, JsonValue>;
  return std::make_unique<JsonObject>(
      p{"firstName"s, "phn"s}, p{"lastName"s, "Smith"s},
      p{"isAlive"s, true}, p{"age"s, 27},
      p{"address"s,
         std::make_unique<JsonObject>(p{"streetAddress"s, "21 2nd Street"s},
                                      p{"city"s, "New York"s})},
      p{
          "phoneNumbers"s,
          std::make_unique<JsonArray>(
              std::make_unique<JsonObject>(p{"type"s, "home"s},
                                           p{"number"s, "212 555-1234"s}),
              std::make_unique<JsonObject>(p{"type"s, "office"s},
                                           p{"number"s, "646 555-4567"s})),
      },
      p{"spouse"s, nullptr});
}
```

```
{
 "firstName": "John",
 "lastName": "Smith",
 "isAlive": true,
 "age": 27,
 "address":
 {
  "streetAddress": "21 2nd Street",
  "city": "New York"
 },
 "phoneNumbers":
 [
  {
   "type": "home",
   "number": "212 555-1234"
  },
  {
   "type": "office",
   "number": "646 555-4567"
  }
 ],
 "spouse": null
}
```

JSON – visitation

```
std::ostream &operator<<(std::ostream &ost, const JsonValue &json) {
  std::visit(overload([&ost](const nullptr_t &) { ost << "null"; },
                      [&ost](const std::unique_ptr<JsonArray> &value) {
                        // should we check for null ptr?
                        ost << "[\n";
                        std::copy(begin(value->array), end(value->array),
                                  std::ostream_iterator<JsonValue>(ost, ",\n"));
                        ost << ']';
                      },
                      [&ost](const std::unique_ptr<JsonObject> &value) {
                        // should we check for null ptr?
                        ost << "{\n";
                        std::for_each(begin(value->object), end(value->object),
                                      [&](const auto &p) {
                                        ost << p.first << ": " << p.second << ",\n";
                                      });
                        ost << '}';
                      },
                      [&ost](const auto &value) { ost << value; }),
             json);
  return ost;
}
```

JSON – Boost::variant + recursive_wrapper

```
struct JsonArray;
struct JsonObject;

using JsonValue = boost::variant<nullptr_t, int, std::string, bool,
                                 boost::recursive_wrapper<JsonArray>,
                                 boost::recursive_wrapper<JsonObject>>;

struct JsonArray {
  std::vector<JsonValue> array;

  JsonArray(std::initializer_list<JsonValue> il)
      : array(il) {}
};

struct JsonObject {
  std::vector<std::pair<std::string, JsonValue>> object;

  JsonObject(std::initializer_list<std::pair<std::string, JsonValue>> il)
      : object(il) {}
};
```

JSON – Boost construction

```
JsonValue construct() {
  using namespace std::string_literals;
  return JsonObject{
      {"firstName"s, "John"s},
      {"lastName"s, "Smith"s},
      {"isAlive"s, true},
      {"age"s, 27},
      {"address"s, JsonObject{{"streetAddress"s, "21 2nd Street"s},
                              {"city"s, "New York"s}}},
      {
          "phoneNumbers"s,
          JsonArray{
              JsonObject{{"type"s, "home"s}, {"number"s, "212 555-1234"s}},
              JsonObject{{"type"s, "office"s}, {"number"s, "646 555-4567"s}}},
      },
      {"spouse"s, nullptr}};
}
```

```
{
 "firstName": "John",
 "lastName": "Smith",
 "isAlive": true,
 "age": 27,
 "address":
 {
  "streetAddress": "21 2nd Street",
  "city": "New York"
 },
 "phoneNumbers":
 [
  {
   "type": "home",
   "number": "212 555-1234"
  },
  {
   "type": "office",
   "number": "646 555-4567"
  }
 ],
 "spouse": null
}
```

JSON – Boost visitation

```
std::ostream &operator<<(std::ostream &ost, const JsonValue &json) {
  boost::apply_visitor(overload([&ost](const nullptr_t &) { ost << "null"; },
                       [&ost](const JsonArray &value) {
                         ost << "[\n";
                         std::copy(begin(value.array), end(value.array),
                                   std::ostream_iterator<JsonValue>(ost, ",\n"));
                         ost << ']';
                       },
                       [&ost](const JsonObject &value) {
                         ost << "{\n";
                         std::for_each(
                             begin(value.object), end(value.object), [&](const auto &pair) {
                               ost << pair.first << ": " << pair.second << ",\n";
                             });
                         ost << '}';
                       },
                       [&ost](const auto &value) { ost << value; }),
             json);
  return ost;
}
```

N4510 - Minimal incomplete type support for standard containers

```
struct JsonValue;

using JsonArray = std::vector<JsonValue>;
using JsonObject = std::vector<std::pair<std::string, JsonValue>>;

struct JsonValue {
  std::variant<nullptr_t,   // null
               int,         // number
               std::string, // string
               bool,        // boolean
               JsonArray,   // array
               JsonObject   // object
               >
      value;

  template <typename Arg, REQUIRES(!std::is_same_v<std::decay_t<Arg>, JsonValue>)>
  JsonValue(Arg &&arg) : value(std::forward<Arg>(arg)) {}
};
```

CONTINUE

LEVEL 4

Function Types

LEVEL 4

How many values?

```
bool f(bool);
```

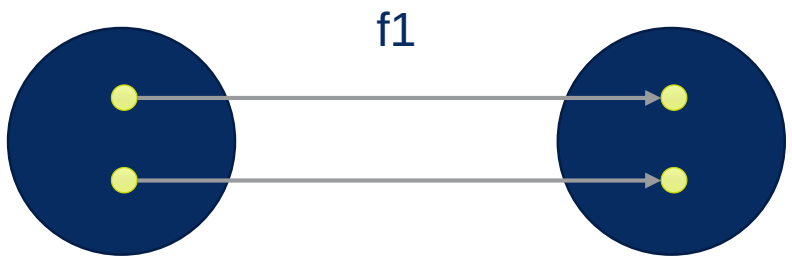
LEVEL 4

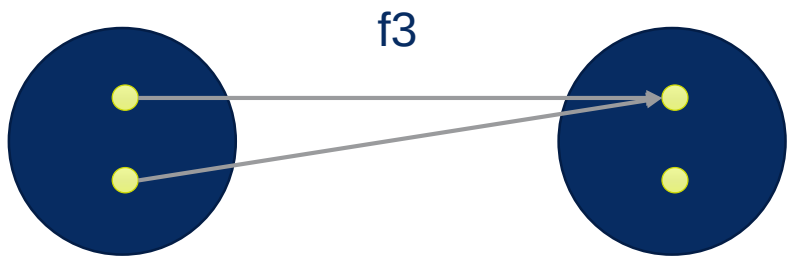
How many values?

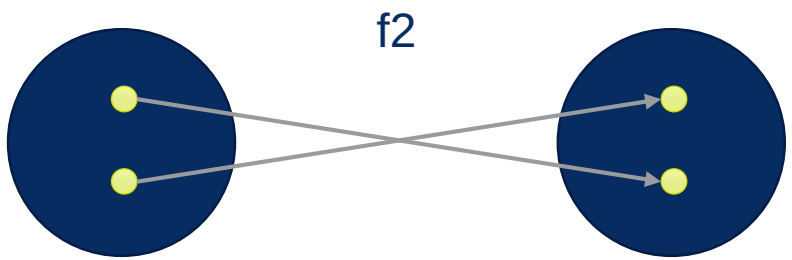
```
bool f(bool);
```

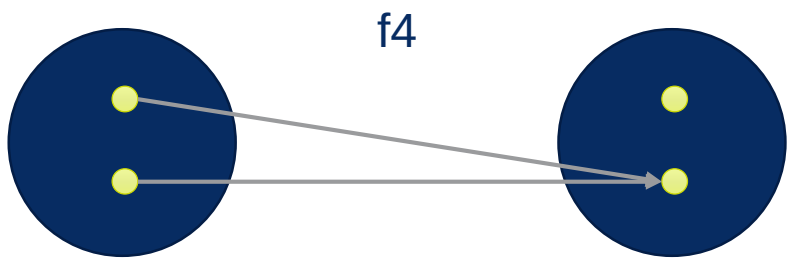
4

LEVEL 4









LEVEL 4

```
bool f1(bool b) { return b; }
bool f2(bool) { return true; }
bool f3(bool) { return false; }
bool f4(bool b) { return !b; }
```

LEVEL 4

How many values?

```
char f(bool);
```

LEVEL 4

How many values?

```
char f(bool);
```

256 * 256 = 65,536

LEVEL 4

How many values?

```
enum class Foo
{
  BAR,
  BAZ,
  QUUX
};
char f(Foo);
```

LEVEL 4

How many values?

```
enum class Foo
{
  BAR,
  BAZ,
  QUUX
};
char f(Foo);
```

256 * 256 * 256 = 16,777,216

LEVEL 4

How many values?

```
template <class T, class U>
    U f(T);
```

LEVEL 4

How many values?

```
template <class T, class U>
    U f(T);
```

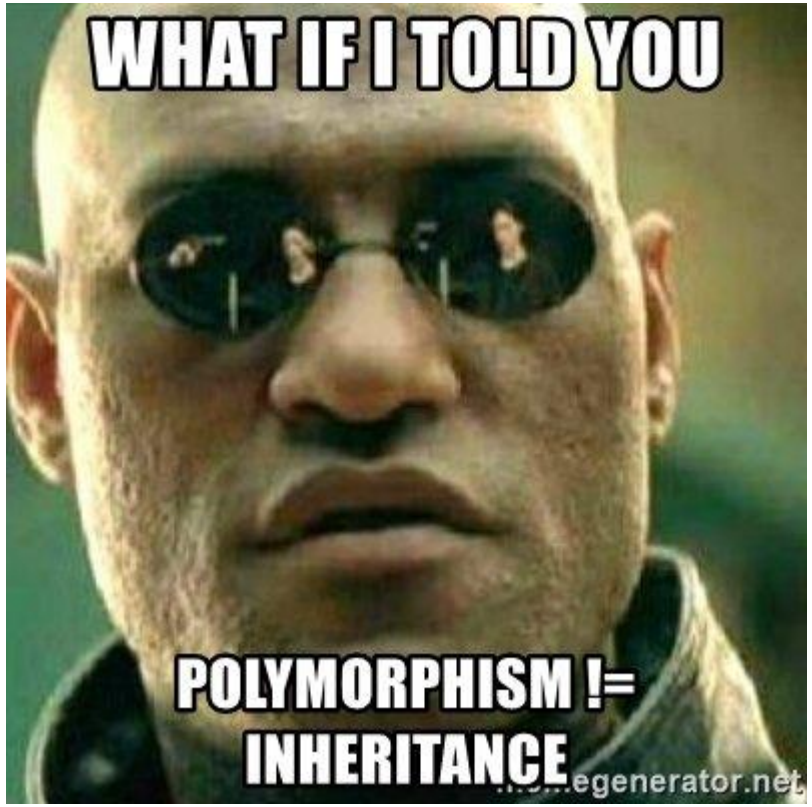
$$|U|^{|T|}$$

END OF LEVEL 4

Functions

- The number of values of a function is the number of different ways we can draw arrows between the inputs and the outputs.
- When we have a function from A to B, we raise the # of inhabitants of B to the power of the # of inhabitants of A.


Game Complete!
021675
MENU


WHAT IF I TOLD YOU
POLYMORPHISM !=
INHERITANCE

Inheritance – interface

```
struct Shape {
  virtual double area() const = 0;
  virtual ~Shape() = default;
};
```

Inheritance – derived

```
struct Square final : Shape {
  explicit Square(double length) : length{ length } {}
  double area() const override { return length * length; }
  double length;
};

struct Circle final : Shape {
  explicit Circle(double radius) : radius{ radius } {}
  double area() const override {
    constexpr double PI = 3.14159265358979323846;
    return PI * radius * radius;
  }
  double radius;
};
```

Inheritance - usage

```
const std::unique_ptr<Shape> shapes[] = {std::make_unique<Square>(4),
                                         std::make_unique<Circle>(2)};

const auto areaSum =
    std::accumulate(std::begin(shapes), std::end(shapes), 0.0,
                    [](double current, const std::unique_ptr<Shape> &shape) {
                      return current + shape->area();
                    });

std::cout << "area sum = " << areaSum << '\n';
```

Inheritance - usage

```
const std::unique_ptr<Shape> shapes[] = {std::make_unique<Square>(4),
                                         std::make_unique<Circle>(2)};

const auto areaSum =
    std::accumulate(std::begin(shapes), std::end(shapes), 0.0,
                    [](double current, const std::unique_ptr<Shape> &shape) {
                      return current + shape->area();
                    });

std::cout << "area sum = " << areaSum << '\n';
```

Output:

area sum = 28.5664

Variant – “derived”

```
struct Square {
  explicit Square(double length) : length{length} {}
  double length;
};

struct Circle {
  explicit Circle(double radius) : radius{radius} {}
  double radius;
};
```

Variant – bundling

```
struct Square {
  explicit Square(double length) : length{length} {}
  double length;
};

struct Circle {
  explicit Circle(double radius) : radius{radius} {}
  double radius;
};

using Shape = std::variant<Square, Circle>;
```

Variant – operation

```
double area(const Shape &shape) {
  return std::visit(
      overload(
          [](const Square &square) { return square.length * square.length; },
          [](const Circle &circle) {
            constexpr double PI = 3.14159265358979323846;
            return PI * circle.radius * circle.radius;
          }),
      shape);
}
```

Variant – usage

```
const Shape shapes[] = {Square{4}, Circle{2}};

const auto areaSum = std::accumulate(
    std::begin(shapes), std::end(shapes), 0.0,
    [](double current, const Shape &shape) { return current + area(shape); });

std::cout << "area sum = " << areaSum << '\n';
```

Variant – usage

```
const Shape shapes[] = {Square{4}, Circle{2}};

const auto areaSum = std::accumulate(
    std::begin(shapes), std::end(shapes), 0.0,
    [](double current, const Shape &shape) { return current + area(shape); });

std::cout << "area sum = " << areaSum << '\n';
```

Output:

area sum = 28.5664

Inheritance – adding functionality

```
struct Shape {
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  virtual ~Shape() = default;
};
```

Inheritance – adding functionality

```
struct Square final : Shape {
  explicit Square(double length) : length{ length } {}
  double area() const override { return length * length; }
  double perimeter() const override { return 4 * length; }
  double length;
};

struct Circle final : Shape {
  explicit Circle(double radius) : radius{ radius } {}
  double area() const override {
    return PI * radius * radius;
  }

  double perimeter() const override {
    return 2 * PI * radius;
  }
  double radius;

private:
  static constexpr double PI = 3.14159265358979323846;
};
```

Variant – adding functionality

```
struct Square {
  explicit Square(double length) : length{length} {}
  double length;
};

struct Circle {
  explicit Circle(double radius) : radius{radius} {}
  double radius;
};

using Shape = std::variant<Square, Circle>;
```

Variant – adding functionality

```
double perimeter(const Shape &shape) {
  return std::visit(
      overload([](const Square &square) { return 4 * square.length; },
               [](const Circle &circle) {
                 constexpr double PI = 3.14159265358979323846;
                 return 2 * PI * circle.radius;
               }),
      shape);
}
```

Inheritance – adding type

```
struct Circle final : Shape {
  explicit Circle(double radius) : radius{radius} {}
  double area() const override {
    constexpr double PI = 3.14159265358979323846;
    return PI * radius * radius;
  }
  double radius;
};
```

Variant – adding type

```
struct Triangle {
  Triangle(double base, double height) : base{base}, height{height} {}
  double base, height;
};

using Shape = std::variant<Square, Circle, Triangle>;
```

Variant – adding type

```
double area(const Shape &shape) {
  return std::visit(
      overload(
          [](const Square &square) { return square.length * square.length; },
          [](const Circle &circle) {
            constexpr double PI = 3.14159265358979323846;
            return PI * circle.radius * circle.radius;
          }),
      shape);
}
```

```
error: no matching function for call to '__invoke(overload_set<area(const
Shape&)::<lambda(const Square&)>, area(const Shape&)::<lambda(const Circle&)> >,
std::variant_alternative_t<2, std::variant<Square, Circle, Triangle> >&)'
```

Variant – adding type

```
double area(const Shape &shape) {
  return std::visit(
      overload(
          [](const Square &square) { return square.length * square.length; },
          [](const Circle &circle) {
            constexpr double PI = 3.14159265358979323846;
            return PI * circle.radius * circle.radius;
          },
          [](const Triangle &triangle) {
            return triangle.base * triangle.height / 2;
          }),
      shape);
}
```

Comparison

Inheritance	std::variant
Need not know all the derived types upfront (open-world assumption)	Must know all the cases upfront (closed-world assumption)
Dynamic Allocation (usually)	No dynamic allocation
Intrusive (must inherit from the base class)	Non-intrusive (third-party classes can participate)
Reference semantics (think how you copy a vector of pointers to base class?)	Value semantics (copying is trivial)
Algorithm scattered into classes	Algorithm in one place
Language supported (Clear errors if pure-virtual is not implemented)	Library supported (poor error messages)
Creates a first-class abstraction	It's just a container
Keeps fluent interfaces	Disables fluent interfaces. Repeated std::visit
Adding a new operation (generally) boils down to implementing a polymorphic method in all the classes	Adding a new operation simply requires writing a new free function

Source: http://cpptruths.blogspot.co.il/2018/02/inheritance-vs-stdvariant-based.html

THANK YOU


THE FUTURE!
memegen.com

C++ ADT

- Product types:
  - struct – language based
  - std::tuple – library based
- Sum types:
  - std::variant – library based
  - No language based type

lvariant

- A language based variant:

```
lvariant user_information {
  std::string name;
  int id;
};
```

David Sankel - Pattern Matching and Language Variants

Recursive

```
lvariant json_value {
  std::map<std::string, std::unique_ptr<json_value>> object;
  std::vector<std::unique_ptr<json_value>> array;
  std::string string;
  double number;
  bool boolean;
  std::monostate null;
};
```

Before

```
struct set_score {
  std::size_t value;
};

struct fire_missile {};

struct fire_laser {
  unsigned intensity;
};

struct rotate {
  double amount;
};

struct command {
  std::variant<set_score, fire_missile, fire_laser, rotate> value;
};
```

After

```
lvariant command {
  std::size_t set_score;
  std::monostate fire_missile;
  unsigned fire_laser;
  double rotate;
};
```

Before – visitation

```
std::ostream &operator<<(std::ostream &stream, const command &cmd) {
  std::visit(
      overload(
          [&](const set_score &ss) {
            stream << "Set the score to " << ss.value << ".\n";
          },
          [&](const fire_missile &) { stream << "Fire a missile.\n"; },
          [&](const fire_laser &fl) {
            stream << "Fire a laser with " << fl.intensity
                   << " intensity.\n";
          },
          [&](const rotate &r) {
            stream << "Rotate by " << r.amount << " degrees.\n";
          }),
      cmd.value);
  return ostream;
}
```

After – pattern matching

```
std::ostream &operator<<(std::ostream &stream, const command &cmd) {
  return inspect(cmd) {
    set_score value =>
      stream << "Set the score to " << value << ".\n";
    fire_missile _ =>
      stream << "Fire a missile.\n";
    fire_laser intensity =>
      stream << "Fire a laser with " << intensity << " intensity.\n";
    rotate degrees =>
      stream << "Rotate by " << degrees << " degrees.\n";
  };
}
```

Before – switch on enum

```
enum color { red, yellow, green, blue };
const Vec3 opengl_color = [&c] {
      switch (c) {
      case red:
        return Vec3(1.0, 0.0, 0.0);
        break;
      case yellow:
        return Vec3(1.0, 1.0, 0.0);
        break;
      case green:
        return Vec3(0.0, 1.0, 0.0);
        break;
      case blue:
        return Vec3(0.0, 0.0, 1.0);
        break;
      default:
        std::abort();
      }
    }();
```

After – pattern matching

```
enum color { red, yellow, green, blue };
const Vec3 opengl_color =
      inspect(c) {
        red => Vec3(1.0, 0.0, 0.0)
        yellow => Vec3(1.0, 1.0, 0.0)
        green => Vec3(0.0, 1.0, 0.0)
        blue => Vec3(0.0, 0.0, 1.0)
    };
```

Before – inspecting struct contents

```
struct player {
  std::string name;
  int hitpoints;
  int lives;
};

void takeDamage(player &p) {
  if (p.hitpoints == 0 && p.lives == 0)
    gameOver();
  else if (p.hitpoints == 0) {
    p.hitpoints = 10;
    p.lives--;
  } else if (p.hitpoints <= 3) {
    p.hitpoints--;
    messageAlmostDead();
  } else {
    p.hitpoints--;
  }
}
```

After – pattern matching

```
struct player {
  std::string name;
  int hitpoints;
  int lives;
};

void takeDamage(player &p) {
  inspect(p) {
    {hitpoints : 0, lives : 0}    => gameOver();
    {hitpoints:hp @0, lives : l}  => hp = 10, l--;
    {hitpoints:hp} if (hp <= 3)   => { hp--; messageAlmostDead(); }
    {hitpoints : hp} => hp--;
  }
}
```

THANK YOU