# Type-safe Unions in C++

Toby Allsopp
toby@mi6.gen.nz

Auckland C++ Meetup
19 July 2016

# This talk

# Overview

Introduction

Motivating Example

A Touch of Theory

Discriminated Unions

- Plain old `union`
- `union` + tag

Inline visitors

Multi-visitation

Questions

# A robot

- States

```
#include <iostream>

struct idle {};
struct turning {
  float targetHeading;
};

std::ostream& operator<<(std::ostream& os, const idle&) {
  return os << "idle{}";
}
std::ostream& operator<<(std::ostream& os, const turning& s) {
  return os << "turning{" << s.targetHeading << "}";
}
```

- Events

```
#include <iostream>
```

# State transitions

| Start state | Event | End state |
|---|---|---|
| Any | `start_turning{h}` | `turning{h}` |
| Any | `stop_turning` | `idle` |
| `turning{h}` | `heading_changed{h}` | `idle` |
| *s* | `heading_changed` | *s* |

# Types

A type is EITHER

- a set of $n$ primitive values $T = \{v_1, v_2, \ldots, v_n\}$,
- a product of $n$ other types $T = T_1 \times T_2 \times \cdots \times T_n$, or
- a sum of $n$ other types $T = T_1 \cup T_2 \cup \cdots \cup T_n$.

# Sum types

- To represent the state of the robot, we want a type whose values are either `idle` or `turning`.
- This is called a sum type because its cardinality is the sum of the cardinalities of its constituent types.
- We could also call it a union type because the set of its possible values is the union of the sets of values of its constituent types.

# Indiscriminate union

```
union u {
  int i;
  double d;
}
u x;
x.i = 3;
double d = x.d; // UNDEFINED BEHAVIOUR
```

- Can’t tell what was last stored but you’d better know!
- Can be used for bit twiddling on specific platforms
- No guarantees from the standard

# More union gotchas

```
union u {
  std::string s;
  std::vector<int> v;
} x;
x.s = "Hello"; // KABOOM!
new (&x.s) std::string("Hello"); // OK
x.s = "Goodbye"; // OK
u y = x; // NOPE - copy constructor is deleted
new (&x.v) std::vector<int>{1, 2, 3}; // LEAK!
x.s.~std::string(); // DO THIS FIRST
```

# union + class + enum

```
class svu {
 private:
  enum { STRING, VECTOR } tag;
  union {
    std::string s;
    std::vector<int> v;
  };
};
```

- Because it's so unsafe, let's bundle it up into a class and only expose safe operations.
- We need to keep track of which union member we have initialized - we call this a tag.

# Construction

```
private:
 void construct(const std::string& _s) {
   tag = STRING;
   new (&s) std::string(_s);
 }
 void construct(const std::vector<int>& _v) {
   tag = VECTOR;
   new (&v) std::vector<int>(_v);
 }
public:
 template <typename T>
 svu(const T& x) {
   construct(x);
 }
```

# Visitation

```
public:
 template <typename R, typename F>
 R visit(F&& f) {
   switch (tag) {
     case STRING: return f(s);
     case VECTOR: return f(v);
   }
 }
 // and a const version
```

- So the object you pass in has to have an `operator()` that takes a `string` and one that takes a `vector<int>`.
- Both operators must return something convertible to R.

# Visitation example

```
struct Visitor {
  int operator()(const string& s) { return s.size(); }
  int operator()(const vector<int> v) { return v[0]; }
};
svu x("Hello");
int i = x.visit<int>(Visitor()); // 5
int i = x.visit<int>([](const auto& v) {
                        return v.size();
                      });
```

# Destruction

```
private:
  void destruct() {
    visit<void>([](auto&& x) {
      using T = std::decay_t<decltype(x)>;
      x.~T();
    });
  }
public:
  ~svu() { destruct(); }
```

- decay_t?
- This has no dependencies on the particular types we're using.

# Assignment

```
svu& operator=(const svu& other) {
  destruct();
  other.visit<void>(
      [this](auto&& v) { construct(v); });
  return *this;
}
```

# Move semantics

# Inline visitors

- Recall how we defined a visitor earlier:

```
struct Visitor {
  int operator()(const string& s) {return s.size();}
  int operator()(const vector<int> v) {return v[0];}
};
```

- Wouldn't it be nice to not have to explicitly define a struct for this?

```
x.visit<int>(
    [](const string& s) { return s.size(); },
    [](const vector<int> v) { return v[0]; });
```

- We can do this with a little recursive class template...

# Overload set

- We start by declaring the class template. The template parameters are the types of the function-like objects that implement it:

```
template <typename... Fs>
class overload_set;
```

- Then we define the base case for the recursion - an overload set with zero functions:

```
template <>
class overload_set<> {
 public:
  void operator()() = delete;
};
```

# Overload set

- Finally we define the inductive case — an overload set with $n+1$ functions defined in terms of one with $n$:

```
template <typename F, typename... Fs>
class overload_set<F, Fs...>
    : private overload_set<Fs...>, private F {
 public:
  explicit overload_set(F&& f, Fs&&... fs)
      : overload_set<Fs...>(std::forward<Fs>(fs)...),
        F(std::forward<F>(f)) {}

  using F::operator();
  using overload_set<Fs...>::operator();
};
```

- Now we can add an overload of our `visit` function to create an `overload_set` if we pass more than one argument.

```
template <
    typename R, typename... Fs,
    typename = std::enable_if_t<sizeof...(Fs) >= 2>>
R visit(Fs&&... fs) {
  return visit<R>(
      overload_set<Fs...>(std::forward<Fs>(fs)...));
}
// and a const version
```

- In fact, we could make the original version private and use this one unconditionally — why not?

# Inline visitation

```
x.visit<int>(
    [](const std::string& s) { return s.size(); },
    [](const std::vector<int> v) { return v[0]; });
```

- Note how `s.size()` actually returns **`size_t`** but it gets implicitly converted to **`int`** — this is useful in many cases but consider `-Wconversion`.

# Multi-visitation

- What if we want to visit two objects at once?

```
int plux(const svu& u1, const svu& u2) {
  using namespace std;
  return u1.visit<int>(
      [&](const string& s1) {
        return u2.visit<int>(
            [&](const string& s2)      { return s1.size() + s2.size(); },
            [&](const vector<int>& v2) { return s1.size() + v2[0]; });
      },
      [&](const vector<int> v1) {
        return u2.visit<int>(
            [&](const string& s2)      { return v1.size() + s2[0]; },
            [&](const vector<int>& v2) { return v1[0] + v2[0]; });
      });
}
```

- That makes me want to claw my eyes out.

- I want to be able to write:

```
int plux2(const svu& u1, const svu& u2) {
  using namespace std;
  auto visitor = make_visitor<int>(
      [](const string& s1,     const string& s2)      { return s1.size() + s2.size(); },
      [](const string& s1,     const vector<int>& v2) { return s1.size() + v2[0]; },
      [](const vector<int> v1, const string& s2)      { return v1.size() + s2[0]; },
      [](const vector<int> v1, const vector<int>& v2) { return v1[0] + v2[0]; });
  return visitor(u1, u2);
}
```

- This is much nicer because
  - it scales gracefully to any number of visitees,
  - it allows wildcards in any position and
  - it only makes we want to claw one eye out.
- But how do we make one?

```
template <typename R, typename F>
class visitor {
 private:
  F m_f;

 public:
  explicit visitor(F&& f) : m_f(f) {}

  template <typename... Vs>
  auto operator()(const Vs&... args) {
    return collect(std::tuple<>(), args...);
  }
```

```
 private:
  template <typename T>
  auto collect(const T& t) {
    return apply(m_f, t);
  }

  template <typename T, typename V, typename... Vs>
  auto collect(const T& t, const V& arg,
               const Vs&... args) {
    return arg.template visit<R>([&](auto v) {
      return this->collect(
          std::tuple_cat(t, std::make_tuple(v)),
          args...);
    });
  }
};
```

```
template <typename Callable, typename Tuple,
          size_t... I>
auto apply_impl(Callable&& f, Tuple&& t,
                std::index_sequence<I...>) {
  return f(std::get<I>(t)...);
}

template <typename Callable, typename Tuple>
auto apply(Callable&& f, Tuple&& t) {
  using is = std::make_index_sequence<
      std::tuple_size<std::decay_t<Tuple>>::value>;
  return apply_impl(std::forward<Callable>(f),
                    std::forward<Tuple>(t), is());
}
```

```cpp
state transition(const state& s, const event& e) {
  return make_visitor<state>(make_overload_set(
      [](auto,      start_turning e) {
        return turning{e.target};
      },
      [](auto,      stop_turning) { return idle{}; },
      [](auto s,    heading_changed) { return s; },
      [](turning s, heading_changed e) -> state {
        if (std::abs(e.heading - s.target) < .1f) {
          return idle{};
        } else {
          return s;
        }
      }))(s, e);
}
```

Questions?

# Appendix

- Extra material
- References

# References

- Boost