Type-safe Unions in C++14

`https://github.com/toby-allsopp/auck_cpp-typesafe_unions`

Toby Allsopp
`toby@mi6.gen.nz`

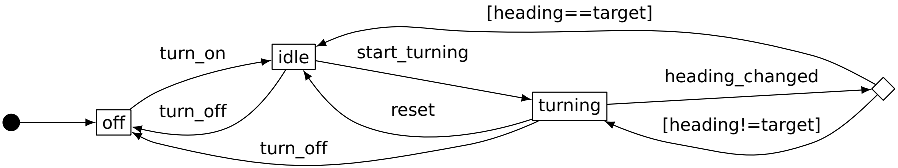
Auckland C++ Meetup
19 July 2016

Notes

- There is lots of code in this talk
- All code is on github
- Intended to conform to C++14
- Tested with GCC 6.1.1 and clang 3.8.0
- Might work with VS2015
- I'm not an expert on TMP; this code is almost certainly inefficient
  - at coding time
  - at compile time
  - at run time
- Feel free to shout out suggestions

Overview

A robot



- States

```
struct off {};
struct idle {};
struct turning { float target; };
```

- Events

```
struct turn_on {};
struct turn_off {};
struct start_turning { float target; };
struct heading_changed { float heading; };
struct reset { std::string reason; };
```

- We want to write a state transition function for the robot

```
state transition(const state& s, const event& e);
```

- But what types do we use for `state` and `event`?
- We need something that can hold either an `off`, an `idle` or a `turning` (and similarly for the events)
- Something like a **`union`**
- But better!

variant

- Boost has boost::variant
- C++17 will have std::variant
- We're going to build our own

```
using state = variant<off, idle, turning>;
using event =
    variant<turn_on, turn_off, start_turning, reset,
            heading_changed>;
```

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

```cpp
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

- We can define move copy constructors and assignment operators
- And we should for efficiency's sake
- But it's tedious — just chuck some `&&`s and `std::move`s around the place

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 other than one argument.

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

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_multivisitor<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?

- It starts out pretty simple...

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

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

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

- The tricky bit is that we need to accumulate the results of visiting each variant until we've visited them all.

```
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...);
    });
  }
};
```

- OK, now we have a tuple of values — how do we pass them to our function?
- C++17 has `std::apply` but we can copy a cheap imposter from `cppreference.com`...

```
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());
}
```

- So, that's all well and good if you want a variant that can hold a `string` or a `vector<int>`.
- But we wanted two different variants — one for states and one for events.
- So let's copy and paste and change the names, job done.

Hell, no!

We’re just getting warmed up.

Construction

- We need a `construct` function for each type in the variant.
- So we go to our old friend, the recursive class template.

```
template <typename I, I N, typename... Ts>
struct variant_construct;
```

- The base case, for a variant with zero types:

```
template <typename I, I N>
struct variant_construct<I, N> {
  static I construct();
};
```

Construction

- And the inductive $n+1$ case defined in terms of the n case:

```
template <typename I, I N, typename T, typename... Ts>
struct variant_construct<I, N, T, Ts...>
    : private variant_construct<I, N + 1, Ts...> {
  using super = variant_construct<I, N + 1, Ts...>;

  static I construct(void* storage, const T& value) {
    new (storage) T(value);
    return N;
  }
  using super::construct;
};
```

Visitation

```
template <typename I, I N, typename... Ts>
struct variant_visit;

template <typename I, I N>
struct variant_visit<I, N> {
  template <typename R, typename F>
  static R visit_helper_const(I tag, const void*, F&&) {
    throw std::logic_error("variant tag invalid");
  }
};

template <typename I, I N, typename T, typename... Ts>
struct variant_visit<I, N, T, Ts...> : private variant_visit<I, N + 1, Ts...> {
  using super = variant_visit<I, N + 1, Ts...>;

  template <typename R, typename F>
  static R visit_helper_const(I tag, const void* storage, F&& f) {
    if (tag == N) {
      return f(*reinterpret_cast<const T*>(storage));
    }
    return super::template visit_helper_const<R>(tag, storage, std::forward<F>(f));
  }
};
```

Storage

- I can't figure out how to use a **union** as storage.
- But we can use a very handy template:

```
typename std::aligned_union_t<0, Ts...> storage;
```

- The size needed for the tag depends on how many types are in the variant.

```
template <uintmax_t N, typename Enable = void>
struct smallest_unisnged_type;

template <uintmax_t N>
struct smallest_unisnged_type<
    N, typename std::enable_if_t<N <= std::numeric_limits<uint8_t>::max()>> {
  using type = uint8_t;
};

template <uintmax_t N>
using smallest_unisnged_type_t = typename smallest_unisnged_type<N>::type;
```

variant part 1

```
template <typename... Ts>
struct variant_helper {
  using tag_type = smallest_unisnged_type_t<sizeof...(Ts)>;
  using super_construct = variant_construct<tag_type, 0, Ts...>;
  using super_visit = variant_visit<tag_type, 0, Ts...>;
};

template <typename... Ts>
class variant : private variant_helper<Ts...>::super_construct,
                private variant_helper<Ts...>::super_visit {
  using helper = variant_helper<Ts...>;
  using super_construct = typename helper::super_construct;
  using super_visit = typename helper::super_visit;

  typename std::aligned_union_t<0, char, Ts...> storage;
  typename helper::tag_type tag;

  using super_construct::construct;

  void destruct() {
    std::move(*this).template visit<void>([this](auto&& v) {
      using T = std::decay_t<decltype(v)>;
      reinterpret_cast<T*>(&storage)->~T();
    });
  }
```

variant part 2

```
public:
  template <typename T, typename = decltype(construct(
                            &storage, std::forward<T>(std::declval<T>()))))>
  variant(T&& value) {
    tag = construct(&storage, std::forward<T>(value));
  }

  variant(const variant& other) {
    other.visit<void>(
        [this](auto&& value) { tag = construct(&storage, value); });
  }

  variant& operator=(const variant& other) {
    destruct();
    other.visit<void>(
        [this](auto&& value) { tag = construct(&storage, value); });
    return *this;
  }

  ~variant() { destruct(); }

  template <typename R, typename F>
  auto visit(F&& f) const& {
    return super_visit::template visit_helper_const<R>(tag, &storage,
                                                       std::forward<F>(f));
  }
};
```

Variant type

- Finally, we can define the types for our robot's state and events

```
using state = variant<off, idle, turning>;
using on = variant<idle, turning>;

using event = variant<turn_on, turn_off, start_turning, reset,
                      heading_changed>;
```

- That on type comes in handy for the state transitions.

State transitions

```
state transition(const state& s, const event& e) {
  return make_multivisitor<state>(
      [](off,      turn_on)          { return idle{}; },
      [](off,      auto)             { return off{}; },
      [](on,       turn_off)         { return off{}; },
      [](auto s,   turn_on)          { return s; },
      [](on,       reset)            { return idle{}; },
      [](on,       start_turning e)  { return turning{e.target}; },
      [](idle 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