Type-safe configuration library

Michał Dominiak

Wrocław University of Technology

Nokia Networks

griwes@griwes.info

Outline

- Introduction
- Why type-level?
- Boost.Any and Boost.TypeIndex
- Overload resolution control
- Traits
- Putting it together
- Default values
- Constrained configurations
- Boost.ProgramOptions wrapper

Introduction - what is this talk about?

- Shifting as much work as possible from runtime to compile time.
- Doing checks at type level.
- Avoiding code and constant repetition.
- Employing template metaprogramming in practice.
- My personal experience while implementing all of this.

Introduction - what is this talk *not* about?

- "Please use my library."
- Reading configuration files.

Introduction - what's in the title?

- For the purpose of this talk and this library, a *configuration* is a *set of key-value pairs*.
- Type-safe means all the possible checking is done at compile time, at type level, to avoid runtime type errors.

Motivation

- Existing solutions are not type-safe: Boost.PropertyTree, Boost.ProgramOptions.
- Strings are a terrible choice for keys.
- A solution needed for a specific use-case.

Original problem - `reaver::logger`

- ReaverLib - a personal set of libraries, used in multiple projects.
- Logger needed extensible logging levels, as they also determine the format of a line.
- Possibility 1: enums - not extensible
- Possibility 2: dynamically allocated levels - not really deterministic, near impossible to check statically
- Possibility 3: maybe enums + something else?
- Final choice: enums to define levels, types to define line format.

Motivation

- For the selected solution, needed a way to translate a type to a vector of "streamables".
- Trivial basic implementation - a map of `boost::any`s - but type safety not guaranteed .

Why type-level?

- Compile time checks are better than runtime checks, which might possibly only fire in production.
- Generated code is faster, since it needs less virtual calls, and that makes it possible to be inlined.
- Strings are terrible as keys - you need to either store them somewhere, or repeat in code, and can't really statically check for validity everywhere.
- Easy to define new, unique types; definining unique values is harder.

Internal implementation

- To avoid reinventing the basics, the internal implementation (at least at this point) uses two of Boost libraries.
- Boost.Any - unconstrained type-erasure wrapper.
- Boost.TypeIndex - a convenient replacement for standard RTTI features.

Boost.Any

- Allows storing any value.
- Possible to ask for the value back using `any_cast`.
- `any_cast` does a runtime check and throws `bad_any_cast` if it fails.
- Using Boost.Any is a quick and dirty solution for the necessary part of the implementation, to be replaced with something more sophisticated one day.

Boost.TypeIndex

- RTTI on steroids.
- Usable even in environments without RTTI.
- `boost::typeindex::type_id<T>()` is equivalent to `typeid(T)`.
- Human-readable `pretty_name()`.

Trivial implementation

- This is the most basic implementation of this piece of library, and allows only what I talked about so far - putting a value in, getting a value out.
- A helper class used: `unit`.

```
struct unit
{
};
```

Trivial implementation

```
#include <unordered_map>
#include <boost/type_index.hpp>
#include <boost/functional/hash.hpp>
#include <boost/any.hpp>
class configuration {
public:
    template<typename T, typename... Args> unit set(Args &&... args);
    template<typename T> typename T::type & get();
    template<typename T> const typename T::type & get() const;
private:
    std::unordered_map<boost::typeindex::type_index, boost::any,
        boost::hash<boost::typeindex::type_index>> _map;
};
```

Trivial implementation pt. 2

```
template<typename T, typename... Args> unit set(Args &&... args)
{
    _map[boost::typeindex::type_id<T>()] =
        typename T::type{ std::forward<Args>(args)... };
    return unit{};
}
template<typename T> typename T::type & get()
{
    return boost::any_cast<typename T::type &>(
        _map.at(boost::typeindex::type_id<T>()));
}
```

Overload resolution control

- *Beating overload resolution into submission* by Xeo.
- Fixes the problem of requiring exponential number of conditions for SFINAE.
- Allows ordering overloads.
- Slightly awkward when selecting overloads that take actual arguments - lambdas to the rescue.
- *(I'm going to skip the introductory part of the original blog post here; please follow the link at the end of this talk if you are interested in learning it.)*

Overload resolution control pt. 2

- Goal: avoid stating all the SFINAE conditions on all the overloads.
- Overload set must be ordered from the most constrained overloads to the least constrained ones.
- The technique orders overloads using type conversion ordering that exists in C++; it's similar to how function taking an int is selected before a function taking C varargs: in this case, it's inheritance.
- The amount of choices on next slide is arbitrarily limited to 11.

Overload resolution control pt. 3

```
template<unsigned I>
struct choice : choice<I + 1> {};

template<>
struct choice<10> {};

struct select_overload : choice<0> {};
```

Overload resolution control pt. 4

```
template<unsigned N,
    typename std::enable_if<N % 4 == 0, int>::type = 0>
void print_parity(choice<0>) { std::cout << "doubly even"; }
template<unsigned N,
    typename std::enable_if<N % 2 == 0, int>::type = 0>
void print_parity(choice<1>) { std::cout << "even"; }
template<unsigned N>
void print_parity(choice<2>) { std::cout << "odd"; }
print_parity<8>(select_overload{}); // prints "doubly even"
print_parity<2>(select_overload{}); // prints "even"
print_parity<5>(select_overload{}); // prints "odd"
```

Traits, the need for

- Allowing type conversions.
- Allowing custom functions constructing held data - `construct`.
- Selecting better matches first.
- Cases to consider:
  - Identity `construct`.
  - Value of correct type passed.
  - Exact `construct` match.
  - Single conversion available.
  - Matching `construct`.
  - Matching constructor.

Traits, the Old Way

- *Note: some of the traits are probably badly named. Sorry for that.*
- Initial trait implementation - following the old style, which used the technique similar to `choice<N>` and `select_overload`, but this time with `int` and `long`:

```
template<typename T>
struct _has_identity_construct {
private:
    template<typename U>
    static auto _test(int) -> decltype(
        U::construct(std::declval<typename U::type>()),
        void());
    template<typename U>
    static char _test(long);
public:
    static constexpr bool value = std::is_void<
        decltype(_test<T>(0))
    >::value;
};
```

Traits, the New Way: `void_t`

```
template<typename...>
struct voider
{
    using type = void;
};

template<typename... Args>
using void_t = typename voider<Args...>::type;
```

- The first type necessary due to CWG issue 1558: different compilers interpreted substitution failure in an unused alias template argument differently (either as an actual substitution failure or as well-formed code). When CWG 1558 is resolved w/ proposed resolution, void_t becomes:

```
template<typename... Args>
using void_t = void;
```

Traits, the New Way: `void_t`, pt. 2

- Let's rewrite the trait shown before, but this time using void_t instead of the overload selection hack:

```
template<typename Tag, typename = void>
struct _has_identity_construct
    : public std::false_type {};
template<typename T>
struct _has_identity_construct<T, void_t<decltype(
    T::construct(std::declval<typename T::type>()))>>
    : public std::true_type {};
```

- Much better!

Traits, the incomplete

- There’s one case in the list few slides ago that’s tricky to implement:
  - Exact `construct` match.
- How do you check if a function call is an exact match, generically, with an arbitrary number of function arguments?
- Find the result and get the address of the correct overload.
- To simplify the task, my implementation assumes that all overloads that might end up being checked take their argument by values, *to avoid having to check for references and cv-qualifiers*.
- This is not an optimal solution, but as far as I can tell (please prove me wrong!) there’s no easy way to generate all the possible cvref-qualified variants (preferably) for an arbitrary number of arguments.

Traits, the incomplete pt. 2

```
template<typename... Args>
struct _has_exact_match_impl : public std::false_type {};

template<typename T, typename... Args>
struct _has_exact_match_impl<void_t<
  decltype(static_cast<
      decltype(T::construct(std::declval<Args>()...)) (*)(Args...)
    >(&T::construct))>,
  T,
  Args...
> : public std::true_type {};

template<typename Tag, typename... From>
struct _has_exact_match
  : _has_exact_match_impl<void, Tag, std::remove_reference_t<From>...> {};
```

Helpers

```
template<typename... Args>
struct _is_same : std::false_type {};

template<typename T, typename U>
struct _is_same<T, U> : public std::is_same<
  typename std::remove_cv<
    typename std::remove_reference<T>::type
  >::type,
  typename std::remove_cv<
    typename std::remove_reference<U>::type
  >::type
> {};
```

Helpers pt. 2

```
template<
  template<typename...> class Trait,
  typename T, typename TypeList>
struct _apply_on_type_list;

template<
  template<typename...> class Trait,
  typename T, typename... Types>
struct _apply_on_type_list<Trait, T,
  std::tuple<Types...>> : Trait<T, Types...> {};
```

Putting it together

```
template<typename T, typename... Args>
unit set(Args &&... args)
{
    _set<T, std::tuple<Args...>>(
        select_overload{}
    )(
        std::forward<Args>(args)...
    );
    return {};
}
```

Putting it together pt. 2

```
template<typename T, typename TypeList,
  typename std::enable_if<
    _detail::_apply_on_type_list<
      _detail::_is_same, typename T::type, TypeList>::value
    && _detail::_has_identity_construct<T>::value,
  int>::type = 0>
auto _set(choice<0>)
{
  return [&](typename T::type value){
    _map[boost::typeindex::type_id<T>()] = static_cast<
      typename T::type
    >(T::construct(std::move(value)));
  };
}
```

Putting it together pt. 3

```
template<typename T, typename TypeList,
  typename std::enable_if<
    _detail::_apply_on_type_list<
      _detail::_is_same, typename T::type, TypeList
    >::value, int>::type = 0>
auto _set(choice<1>)
{
  return [&](typename T::type value)
  {
    _map[boost::typeindex::type_id<T>()]
      = std::move(value);
  };
}
```

Default values

```
template<typename T>
auto & get(T = {})
{
  return _get<T>(select_overload{})();
}

template<typename T, typename std::enable_if<
  std::is_void<decltype(T::default_value, void())>::value, int>::type = 0>
auto _get(choice<0>)
{
  return [&]() -> decltype(auto) {
    if (_map.find(boost::typeindex::type_id<T>()) == _map.end())
    {
      set<T>(typename T::type{ T::default_value });
    }
    return boost::any_cast<typename T::type &>(_map.at(boost::typeindex::type_id<T>()));
  };
}
template<typename T>
auto _get(choice<1>)
{
  return [&]() -> decltype(auto) {
    return boost::any_cast<typename T::type &>(_map.at(boost::typeindex::type_id<T>()));
  };
}
```

Constrained configurations: motivation

- Trying to write Boost.ProgramOptions wrapper.
- Easy Use™ idea: allow tags register themselves in a global registry; this way, for example, plugins adding their own command-line options would be trivial.
- Started writing type-erased wrappers over tags that would be used this way.
- Wait a second…
- …I can no longer do any kind of static checking on that thing!
- Worse yet, I can accidentally try to get a value that isn’t there in the configuration object resulting from wrapping ProgramOptions calls!
- Conclusion: I want a configuration type, that specifies exactly what values it contains.

Constrained configuration

- Prototype:

```
template<typename... Allowed>
class bound_configuration : public configuration {};
```

- Only keys among `Allowed` are allowed.
- All keys among `Allowed` have values associated with them – additional static safety check, this time for key errors, not for type errors.
- Constructible from `bound_configuration` whose `Allowed` are a superset of given `bound_configuration`’s `Allowed`.
- Constructible from unbound `configuration`, but might throw.
- Private constructor that violates the second invariant provided (to allow adding new keys and element to a given configuration).

Constrained configuration pt. 2

- `_is_a_subset` – checks whether the type list provided as the first argument is a subset of the type list provided as the second argument.
- Rvalue reference variants of functions omitted for clarity.
- Another helper – `swallow`:

```
struct swallow
{
    template<typename... Args>
    constexpr swallow(Args &&...)
    {
    }
};
```

Constrained configuration pt. 3

- Construction from unconstrained configuration:

```
bound_configuration(const configuration & config)
{
  swallow{ set<Allowed>(config.get<Allowed>())... };
}
```

- Construction from a constrained configuration that is a superset of the current configuration:

```
template<typename... Other, typename std::enable_if<
  detail::_is_a_subset<
    std::tuple<Allowed...>, std::tuple<Other...>
  >::value, int>::type = 0>
bound_configuration(const bound_configuration<Other...> & other)
{
  swallow{ set<Allowed>(other.template get<Allowed>())... };
}
```

Constrained configuration pt. 4

- The setters and getters are also properly constrained:

```
template<typename T, typename... Args,
  typename std::enable_if<
    _detail::_any_of<
      std::is_same<T, Allowed>::value...
    >::value, int
  >::type = 0>
unit set(Args &&... args)
{
  return configuration::set<T>(
    std::forward<Args>(args)...);
}
```

Adding keys to constrained configuration

- Since by default you can only construct an empty configuration, not being able to extend the value would result in it being pretty useless.
- You can extend a value by calling a member template add (that disallows duplicates):

```
template<typename T, typename... Args,
    typename std::enable_if<
        !_detail::_any_of<
            std::is_same<T, Allowed>::value...
        >::value, int
    >::type = 0>
auto add(Args &&... args) const &
{
    bound_configuration<Allowed..., T> ret = *this;
    ret.template set<T>(std::forward<Args>(args)...);
    return ret;
}
```

Boost.ProgramOptions

- Highly configurable parser of command line arguments and simple configuration files.
- Typed API, but relies on a Boost.Any-ish objects and manual casting to desired type, and uses strings as keys (…and depending on situation, those strings can be slightly different, yet refer to the same thing; we’ll see that in a moment).
- Bottom line, it’s a great tool for parsing command line arguments and a great candidate to be a low-level building block for a type-safe command line argument parser.

Boost.ProgramOptions wrapper

- Typical usage - defining options:

```
struct version : options::opt<version, void>
{
    static constexpr const char * name = "version,v";
};

struct output : options::opt<output, std::string>
{
    static constexpr const char * name = "output,o";
    static constexpr const char * description = "this is the description";
};

struct optional : options::opt<optional, boost::optional<int>>
{
    static constexpr const char * name = "optional,o";
};
```

Boost.ProgramOptions wrapper pt. 2

- Positional arguments are also possible:

```
struct positional : options::opt<command, std::string>
{
  static constexpr options::option_set options = {
    options::positional
  };
};
struct another : options::opt<value, std::size_t>
{
  static constexpr options::option_set options = {
    options::positional(1)
  };
};
```

- Missing functionality in the interface: specifying the count for the option (implemented in backend; a sane approach for exposing this feature needed).

Boost.ProgramOptions wrapper pt. 3

- Parsing the command line:

```
const char * argv[] = {
  "", "--count", "2", "--output", "foo"
};
auto parsed = options::parse_argv(5, argv,
  id<count>{}, id<output>{});

static_assert(std::is_same<
  decltype(parsed),
  bound_configuration<count, output>
>::value);
```

PO wrapper: internals

```
template<typename... Args>
auto parse_argv(int argc, const char * const * argv, id<Args>... args) {
    auto visible = _detail::_handle_visible(args...);
    auto hidden = _detail::_handle_hidden(args...);
    auto positional = _detail::_handle_positional(args...);

    boost::program_options::options_description all;
    all.add(visible).add(hidden);
    boost::program_options::variables_map variables;

    boost::program_options::store(
        boost::program_options::command_line_parser(argc, argv)
        .options(all).positional(positional)
        .style(boost::program_options::command_line_style::...)
        .run(), variables);

    return _detail::_get<Args...>(variables, bound_configuration<>{});
}
```

PO wrapper: internals pt. 2

```
template<typename... Args>
auto _handle_visible(id<Args>...)
{
    boost::program_options::options_description desc;
    swallow{ (Args::options.is_visible
        ? _handle<Args>(desc) : unit{})... };
    return desc;
}

template<typename... Args>
auto _handle_hidden(id<Args>...)
{
    boost::program_options::options_description desc;
    swallow{ (!Args::options.is_visible
        ? _handle<Args>(desc) : unit{})... };
    return desc;
 }
```

PO wrapper: `option_set`

```
struct option_set
{
    template<typename... Args>
    constexpr option_set(Args &&... args) { swallow{ initialize(std::forward<Args>(args))... }; }
    constexpr unit initialize(positional_type pos) {
        position_specified = true; position = pos;
        return {};
    }
    template<typename T>
    unit initialize(T &&) {
        static_assert(_detail::_false_type<T>(), "tried to use an unknown configuration option");
        return {};
    }
    bool position_specified = false;
    positional_type position;
    bool is_visible = true;
    bool allows_composing = true;
};
```

PO wrapper: `positional_type`

```
static constexpr struct positional_type
{
    constexpr positional_type() {}
    constexpr positional_type(bool specified, std::size_t position)
      : position_specified{ specified },
        required_position{ position } {}
    constexpr positional_type operator()(
        std::size_t required_position) const
    {
      return { true, required_position };
    }
    bool position_specified = false;
    std::size_t required_position = 0;
    std::size_t count = 1;
} positional;
```

PO wrapper: `option`

```cpp
template<typename CRTP, typename ValueType>
struct option
{
    option() {}
    using type = ValueType;
    static const char * const name;
    static constexpr const char * description = "";
    static constexpr std::size_t count = 1;
    static constexpr option_set options = {};
    static constexpr bool is_void = false;
};

template<typename CRTP>
struct option<CRTP, void> : option<CRTP, bool>
{
    static constexpr bool is_void = true;
};

template<typename CRTP, typename ValueType>
using opt = option<CRTP, ValueType>;

template<typename CRTP, typename ValueType>
const char * const option<CRTP, ValueType>::name
    = boost::typeindex::type_id<CRTP>().name();
```

PO wrapper: helpers

```
template<typename T, typename = void>
struct _po_type
{
    using type = typename T::type;
};

template<typename T>
struct _po_type<T, void_t<typename T::parsed_type>>
{
    using type = typename T::parsed_type;
};


std::string _name(const char * full_name)
{
    std::string buf{ full_name };
    return buf.substr(0, buf.find(','));
}
```

PO wrapper: _handle

```
template<typename T, typename std::enable_if<T::is_void, int>::type = 0>
unit _handle(po::options_description & desc)
{
  desc.add_options()(T::name, T::description);
  return {};
}


template<typename T, typename std::enable_if<!T::is_void, int>::type = 0>
unit _handle(po::options_description & desc)
{
  desc.add_options()(T::name, _handle_vector<T>(
    po::value<
      typename _remove_optional<typename _po_type<T>::type>::type
    >()), T::description);
  return {};
}
```

PO wrapper: `_handle_positional`

```
template<typename... Args>
auto _handle_positional(id<Args>...)
{
    po::positional_options_description desc;
    tpl::sort<
        tpl::filter<tpl::vector<Args...>, _is_positional
        >,
        _compare_positionals
    >::map([&](auto tpl_id){
        _handle_positional<typename decltype(tpl_id)::type>(desc);
    });
    return desc;
}

template<typename T>
unit _handle_positional(po::positional_options_description & desc)
{
    desc.add(T::name, T::options.position.count);
    return {};
}
```

PO wrapper: `_get`

```
template<typename... Args, typename Config>
auto _get(po::variables_map & map, Config && config)
{
    return _get_impl<Args...>(map,
        std::forward<Config>(config));
}
```

- ...this is not strictly accurate.
- The above is after all the `_get_impl` definitions, and before those, there's the following:

```
template<typename... Args, typename Config>
auto _get(po::variables_map& map, Config && config);
```

PO wrapper: `_get_impl`

```
template<typename Config>
auto _get_impl(po::variables_map & map, Config && config)
{
    return std::forward<Config>(config);
}

template<typename Head, typename... Tail, typename Config,
    typename std::enable_if<Head::is_void, int>::type = 0>
auto _get_impl(po::variables_map & map, Config && config)
{
    return _get<Tail...>(map,
        std::forward<Config>(config).template add<Head>(
            map.count(_name(Head::name))));
}
```

PO wrapper: `_get_impl` pt. 2

```
template<typename Head, typename... Tail, typename Config,
  typename std::enable_if<
    _is_optional<typename Head::type>::value
    || _is_vector<typename Head::type>::value,
    int>::type = 0>
auto _get_impl(po::variables_map & map, Config && config)
{
  return _get<Tail...>(map,
    std::forward<Config>(config).template add<Head>(
      map.count(_name(Head::name))
        ? map[_name(Head::name)].template as<
          typename _remove_optional<typename _po_type<Head>::type>::type>()
        : typename Head::type{}));
}
```

PO wrapper: `_get_impl` pt. 3

```
template<typename Head, typename... Tail, typename Config,
  typename std::enable_if<
    _has_default<Head>::value, int>::type = 0>
auto _get_impl(po::variables_map & map, Config && config)
{
  if (map.count(_name(Head::name)))
  {
    return _get<Tail...>(map, std::forward<Config>(config)
      .template add<Head>(map[_name(Head::name)].template as<
        typename _remove_optional<typename _po_type<Head>::type>::type>()));
  }

  return _get<Tail...>(map, std::forward<Config>(config)
    .template add<Head>(decltype(Head::default_value){
      Head::default_value
    }));
}
```

PO wrapper: `_get_impl`, pt. 4

```
template<typename Head, typename... Tail, typename Config,
  typename std::enable_if<
    !Head::is_void
    && !_is_optional<typename Head::type>::value
    && !_has_default<Head>::value
    && !_is_vector<typename Head::type>::value,
  int>::type = 0>
auto _get_impl(po::variables_map & map, Config && config)
{
  return _get<Tail...>(map, std::forward<Config>(config)
    .template add<Head>(map[_name(Head::name)]
    .template as<typename _po_type<Head>::type>()));
}
```

Links

- http://flamingdangerzone.com/cxx11/overload-ranking/ - *Beating overload resolution into submission*
- https://github.com/griwes/ReaverLib/tree/master/include/reaver/configuration

Q&A