Document number: P1081R0
Revises:
Date: 2018-05-07
Project: ISO JTC1/SC22/WG21: Programming Language C++
Audience: LEWG
Reply to: Vincent Reverdy and Collin Gress
University of Illinois at Urbana-Champaign
vince.rev@gmail.com

On empty structs in the standard library

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

Abstract

In this short paper, we discuss the need for empty structs in the standard library as multipurpose helper classes, and the different design options available.

Contents

1 Proposal [proposal]

1.1 The problem [proposal.problem]

An empty structure such as:

```
struct empty_struct {};
```

is a general purpose utility that can be used in many situations. Examples include a default type in contexts requiring a class, a blank type for variant, a default tag class, and an empty base class. The Boost libraries provide a structure called `blank` for this purpose. In C++17, the standard library provides two main empty structures:

— `tuple<>`: however, this is not a "perfectly" basic empty structure since it has a member function `swap` and tuple-related functions have been specialized on it

— `monostate`: however, it is currently considered to be a variant helper class, and is put in `<variant>` accordingly

As a consequence the standard library currently lacks a universal "default" empty structure. It also lacks a way to transform any type, or sequence of types into an empty structure. An example of use is conditional inheritance:

```
// Basic empty struct
struct empty_struct {};

// A base class
struct base {/* something */};

// A class derived from the base if T satisfies something
template <class T>
struct derived: conditional_t<
    is_integral_v<T>,
    base,
    empty_struct
> {/* something */};

// A class derived from the base if T... satisfy something
template <class... T>
struct multi_derived: conditional_t<
    is_integral_v<T>,
    base,
    empty_struct
>... {/* something */}; // Will fail

// A template empty struct
template <class... T>
struct empty_struct_template {};

// A class derived from the base if T... satisfy something
template <class... T>
struct multi_derived: conditional_t<
    is_integral_v<T>,
    base,
```

```
    empty_struct_template<T>
>... {/* something */}; // Will work
```

But is a "universal" empty structure even desirable, and what are the different possible options in terms of design?

1.2 Design options [proposal.solutions]

There are a few possible solutions:

- Do nothing and do not introduce a "universal" empty structure: let users keep creating their own.
- Make `monostate` the "universal" empty structure: in this case it has to be transfered from `<variant>` to `<type_traits>` or `<utility>`. What about the template version?
- Add a new empty structure for each new use case that is being standardized: for example, for conditional inheritance introduce a new `empty_base` and `empty_base_template` (or similar) in `<type_traits>` or `<utility>`. In that case, should `empty_base` be defined as a type alias for `empty_base_template<>`?
- Add a new "universal" empty structure, with a different name than `monostate` in `<type_traits>` or `<utility>` such as `empty_struct` and `empty_struct_template` (or similar). In that case, should `empty_struct` be defined as a type alias for `empty_struct_template<>`?

In any case, the names of the template and non-template versions of a "universal" empty structure have to be bikeshedded. Examples of names include: `empty_struct`, `empty_base`, `empty_tag`, `empty`, `blank`, `nothing`, `none`, `null`, `nil`, `vacant`, `primal`...

1.3 Implementation [proposal.implementation]

Regardless of the chosen design, and once the name is adjusted, implementation is straightforward and will consist of something along the line of:

```
template <class...> struct empty_struct_template {};     // The template empty struct
struct empty_struct {};                                   // The non-template empty struct, version 0
using empty_struct = empty_struct_template<>;             // The non-template empty struct, version 1
```

Additional functionalities like comparison operators or hash specialization could also be added to the design.

1.4 Impact on the standard [proposal.impact]

This proposal is a pure library extension. It does not require changes to any standard classes or functions. All the extensions belong to the `<type_traits>` header or to the `<utility>` header depending on design choices. Also, depending on the preferred option, `monostate` may have to be changed from the `<variant>` header to the `<type_traits>` header or to the `<utility>` header.

1.5 Acknowledgements [proposal.acknowledgements]

The authors would like to thank the participants to the related discussion on the future-proposals group. This work has been made possible thanks to the National Science Foundation through the awards CCF-1647432 and SI2-SSE-1642411.

1.6 References [proposal.references]

A few additional type manipulation utilities, Vincent Reverdy, *Github* (March 2018)

N4727, Working Draft, Standard for Programming Language C++, Richard Smith, *ISO/IEC JTC1/SC22/WG21* (February 2018)

General purpose utilities for template metaprogramming and type manipulation, ISO C++ Standard - Future Proposals, *Google Groups* (March 2018)

Boost blank, Eric Friedman, *The Boost C++ Libraries* (2003)

On empty structs in the standard library

Vincent Reverdy
Collin Gress

Overview

Summary

A "universal" default empty structure is useful in multiple contexts. The standard library currently does not have one.

Examples

```
// Variant
struct monostate {}; // empty struct
struct S {S(int i) : i(i) {} int i;};
variant<monostate, S> variant;

// Conditional inheritance
struct empty_base {}; // empty struct
template <class T> struct derived
: conditional_t<is_class_v<T> && !is_final_v<T>, T, empty_base> {};

// Need for a template version
template <class...> struct empty_base_template {}; // empty struct
template <class... T> struct derived
: conditional_t<is_class_v<T> && !is_final_v<T>, T, empty_base_template<T>>... {};
```

Currently

- `tuple<>`: in `<tuple>`, has a `swap` member, is not intended to be a "universal" empty structure
- `monostate`: in `<variant>`, is considered to be a helper class for `variant`
- `blank`: the "universal" empty structure of the Boost libraries

Question

Need for a "universal" empty structure in `<type_traits>` or `<utility>`?

Design options

Option 1: do nothing

Do nothing and do not introduce a "universal" empty structure: let users keep creating their own.

Option 2: promote monostate

Make `monostate` the "universal" empty structure and transfer it from `<variant>` to `<type_traits>` or `<utility>`.

Option 3: one empty structure per use case

Add a new empty structure for each new use case that is being standardized such as a new `empty_base` for conditional inheritance.

Option 4: a universal empty structure on top of monostate

Add a new "universal" empty structure, with a different name than `monostate` in `<type_traits>` or `<utility>`.

Design of the template version

Motivation

```
// Need for a template version
template <class...> struct empty_base_template {};
template <class... T> struct derived
: conditional_t<is_class_v<T> && !is_final_v<T>, T, empty_base_template<T>>... {};
```

Design question

- `struct empty_struct {}`: the non-template and the template versions are two independent structures
- `using empty_struct = empty_struct_template<>`: the non-template version is an alias of the template version

Conclusion

What option?

- Option 1: do nothing
- Option 2: promote monostate
- Option 3: one empty structure per use case
- Option 4: a universal empty structure on top of monostate

Non-template and template versions

- the non-template and the template versions are two independent structures?
- the non-template version is an alias of the template version?

Bikeshedding

- `empty_struct`
- `empty`
- `blank`
- `nothing`
- `none`
- `null`
- `nil`
- `vacant`
- `primal`

- . . . and what about the template version?

Additional functionalities

- Comparison operators?
- Hash specialization?

Thank you for your attention