# Boost.Variant

Eric Friedman

Itay Maman

## Table of Contents

# Introduction

## Abstract

The `variant` class template is a safe, generic, stack-based discriminated union container, offering a simple solution for manipulating an object from a heterogeneous set of types in a uniform manner. Whereas standard containers such as `std::vector` may be thought of as "**multi-value, single type**," `variant` is "**multi-type, single value**."

Notable features of `boost::variant` include:

- Full value semantics, including adherence to standard overload resolution rules for conversion operations.

- Compile-time type-safe value visitation via `boost::apply_visitor`.

---

- Run-time checked explicit value retrieval via `boost::get`.

- Support for recursive variant types via both `boost::make_recursive_variant` and `boost::recursive_wrapper`.

- Efficient implementation -- stack-based when possible (see the section called ""Never-Empty" Guarantee" for more details).

# Motivation

## Problem

Many times, during the development of a C++ program, the programmer finds himself in need of manipulating several distinct types in a uniform manner. Indeed, C++ features direct language support for such types through its `union` keyword:

```
union { int i; double d; } u;
u.d = 3.14;
u.i = 3; // overwrites u.d (OK: u.d is a POD type)
```

C++'s `union` construct, however, is nearly useless in an object-oriented environment. The construct entered the language primarily as a means for preserving compatibility with C, which supports only POD (Plain Old Data) types, and so does not accept types exhibiting non-trivial construction or destruction:

```
union {
  int i;
  std::string s; // illegal: std::string is not a POD type!
} u;
```

Clearly another approach is required. Typical solutions feature the dynamic-allocation of objects, which are subsequently manipulated through a common base type (often a virtual base class [Hen01 [46]] or, more dangerously, a `void*`). Objects of concrete type may be then retrieved by way of a polymorphic downcast construct (e.g., `dynamic_cast`, `boost::any_cast`, etc.).

However, solutions of this sort are highly error-prone, due to the following:

- *Downcast errors cannot be detected at compile-time.* Thus, incorrect usage of downcast constructs will lead to bugs detectable only at run-time.

- *Addition of new concrete types may be ignored.* If a new concrete type is added to the hierarchy, existing downcast code will continue to work as-is, wholly ignoring the new type. Consequently, the programmer must manually locate and modify code at numerous locations, which often results in run-time errors that are difficult to find.

Furthermore, even when properly implemented, these solutions tend to incur a relatively significant abstraction penalty due to the use of the heap, virtual function calls, and polymorphic downcasts.

## Solution: A Motivating Example

The `boost::variant` class template addresses these issues in a safe, straightforward, and efficient manner. The following example demonstrates how the class can be used:

```
#include "boost/variant.hpp"
#include <iostream>

class my_visitor : public boost::static_visitor<int>
{
public:
    int operator()(int i) const
    {
        return i;
    }

    int operator()(const std::string & str) const
    {
        return str.length();
    }
};

int main()
{
    boost::variant< int, std::string > u("hello world");
    std::cout << u; // output: hello world

    int result = boost::apply_visitor( my_visitor(), u );
    std::cout << result; // output: 11 (i.e., length of "hello world")
}
```

# Tutorial

## Basic Usage

A discriminated union container on some set of types is defined by instantiating the `boost::variant` class template with the desired types. These types are called **bounded types** and are subject to the requirements of the *BoundedType* concept. Any number of bounded types may be specified, up to some implementation-defined limit (see `BOOST_VARIANT_LIMIT_TYPES`).

For example, the following declares a discriminated union container on `int` and `std::string`:

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

By default, a `variant` default-constructs its first bounded type, so `v` initially contains `int(0)`. If this is not desired, or if the first bounded type is not default-constructible, a `variant` can be constructed directly from any value convertible to one of its bounded types. Similarly, a `variant` can be assigned any value convertible to one of its bounded types, as demonstrated in the following:

```
v = "hello";
```

Now `v` contains a `std::string` equal to `"hello"`. We can demonstrate this by **streaming** `v` to standard output:

```
std::cout << v << std::endl;
```

Usually though, we would like to do more with the content of a `variant` than streaming. Thus, we need some way to access the contained value. There are two ways to accomplish this: `apply_visitor`, which is safest and very powerful, and `get<T>`, which is sometimes more convenient to use.

For instance, suppose we wanted to concatenate to the string contained in `v`. With **value retrieval** by `get`, this may be accomplished quite simply, as seen in the following:

```
std::string& str = boost::get<std::string>(v);
str += " world! ";
```

As desired, the `std::string` contained by `v` now is equal to `"hello world! "`. Again, we can demonstrate this by streaming `v` to standard output:

```
std::cout << v << std::endl;
```

While use of `get` is perfectly acceptable in this trivial example, `get` generally suffers from several significant shortcomings. For instance, if we were to write a function accepting a `variant<int, std::string>`, we would not know whether the passed `variant` contained an `int` or a `std::string`. If we insisted upon continued use of `get`, we would need to query the `variant` for its contained type. The following function, which "doubles" the content of the given `variant`, demonstrates this approach:

```
void times_two( boost::variant< int, std::string > & operand )
{
    if ( int* pi = boost::get<int>( &operand ) )
        *pi *= 2;
    else if ( std::string* pstr = boost::get<std::string>( &operand ) )
        *pstr += *pstr;
}
```

However, such code is quite brittle, and without careful attention will likely lead to the introduction of subtle logical errors detectable only at runtime. For instance, consider if we wished to extend `times_two` to operate on a `variant` with additional bounded types. Specifically, let's add `std::complex<double>` to the set. Clearly, we would need to at least change the function declaration:

```
void times_two( boost::variant< int, std::string, std::complex<double> > & operand )
{
    // as above...?
}
```

Of course, additional changes are required, for currently if the passed `variant` in fact contained a `std::complex` value, `times_two` would silently return -- without any of the desired side-effects and without any error. In this case, the fix is obvious. But in more complicated programs, it could take considerable time to identify and locate the error in the first place.

Thus, real-world use of `variant` typically demands an access mechanism more robust than `get`. For this reason, `variant` supports compile-time checked **visitation** via `apply_visitor`. Visitation requires that the programmer explicitly handle (or ignore) each bounded type. Failure to do so results in a compile-time error.

Visitation of a `variant` requires a visitor object. The following demonstrates one such implementation of a visitor implementating behavior identical to `times_two`:

```
class times_two_visitor
    : public boost::static_visitor<>
{
public:

    void operator()(int & i) const
    {
        i *= 2;
    }

    void operator()(std::string & str) const
    {
        str += str;
    }

};
```

With the implementation of the above visitor, we can then apply it to `v`, as seen in the following:

```
boost::apply_visitor( times_two_visitor(), v );
```

As expected, the content of `v` is now a `std::string` equal to `"hello world! hello world! "`. (We'll skip the verification this time.)

In addition to enhanced robustness, visitation provides another important advantage over `get`: the ability to write generic visitors. For instance, the following visitor will "double" the content of *any* `variant` (provided its bounded types each support operator+=):

```
class times_two_generic
    : public boost::static_visitor<>
{
public:

    template <typename T>
    void operator()( T & operand ) const
    {
        operand += operand;
    }

};
```

Again, `apply_visitor` sets the wheels in motion:

```
boost::apply_visitor( times_two_generic(), v );
```

While the initial setup costs of visitation may exceed that required for `get`, the benefits quickly become significant. Before concluding this section, we should explore one last benefit of visitation with `apply_visitor`: **delayed visitation**. Namely, a special form of `apply_visitor` is available that does not immediately apply the given visitor to any `variant` but rather returns a function object that operates on any `variant` given to it. This behavior is particularly useful when operating on sequences of `variant` type, as the following demonstrates:

```
std::vector< boost::variant<int, std::string> > vec;
vec.push_back( 21 );
vec.push_back( "hello " );

times_two_generic visitor;
std::for_each(
      vec.begin(), vec.end()
    , boost::apply_visitor(visitor)
    );
```

# Advanced Topics

This section discusses several features of the library often required for advanced uses of `variant`. Unlike in the above section, each feature presented below is largely independent of the others. Accordingly, this section is not necessarily intended to be read linearly or in its entirety.

## Preprocessor macros

While the `variant` class template's variadic parameter list greatly simplifies use for specific instantiations of the template, it significantly complicates use for generic instantiations. For instance, while it is immediately clear how one might write a function accepting a specific `variant` instantiation, say `variant<int, std::string>`, it is less clear how one might write a function accepting any given `variant`.

Due to the lack of support for true variadic template parameter lists in the C++98 standard, the preprocessor is needed. While the Preprocessor library provides a general and powerful solution, the need to repeat BOOST_VARIANT_LIMIT_TYPES unnecessarily clutters otherwise simple code. Therefore, for common use-cases, this library provides its own macro **BOOST_VARIANT_ENUM_PARAMS**.

This macro simplifies for the user the process of declaring `variant` types in function templates or explicit partial specializations of class templates, as shown in the following:

```
// general cases
template <typename T> void some_func(const T &);
template <typename T> class some_class;

// function template overload
template <BOOST_VARIANT_ENUM_PARAMS(typename T)>
void some_func(const boost::variant<BOOST_VARIANT_ENUM_PARAMS(T)> &);

// explicit partial specialization
template <BOOST_VARIANT_ENUM_PARAMS(typename T)>
class some_class< boost::variant<BOOST_VARIANT_ENUM_PARAMS(T)> >;
```

## Using a type sequence to specify bounded types

While convenient for typical uses, the `variant` class template's variadic template parameter list is limiting in two significant dimensions. First, due to the lack of support for true variadic template parameter lists in C++, the number of parameters must be limited to some implementation-defined maximum (namely, BOOST_VARIANT_LIMIT_TYPES). Second, the nature of parameter lists in general makes compile-time manipulation of the lists excessively difficult.

To solve these problems, make_variant_over< *Sequence* > exposes a `variant` whose bounded types are the elements of `Sequence` (where `Sequence` is any type fulfilling the requirements of MPL's *Sequence* concept). For instance,

```
typedef mpl::vector< std::string > types_initial;
typedef mpl::push_front< types_initial, int >::type types;

boost::make_variant_over< types >::type v1;
```

behaves equivalently to

```
boost::variant< int, std::string > v2;
```

**Portability**: Unfortunately, due to standard conformance issues in several compilers, make_variant_over is not universally available. On these compilers the library indicates its lack of support for the syntax via the definition of the preprocessor symbol BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT.

## Recursive `variant` types

Recursive types facilitate the construction of complex semantics from simple syntax. For instance, nearly every programmer is familiar with the canonical definition of a linked list implementation, whose simple definition allows sequences of unlimited length:

```
template <typename T>
struct list_node
{
    T data;
    list_node * next;
};
```

The nature of `variant` as a generic class template unfortunately precludes the straightforward construction of recursive `variant` types. Consider the following attempt to construct a structure for simple mathematical expressions:

```
struct add;
struct sub;
template <typename OpTag> struct binary_op;

typedef boost::variant<
      int
    , binary_op<add>
    , binary_op<sub>
    > expression;

template <typename OpTag>
struct binary_op
{
    expression left;   // variant instantiated here...
    expression right;

    binary_op( const expression & lhs, const expression & rhs )
        : left(lhs), right(rhs)
    {
    }

}; // ...but binary_op not complete until here!
```

While well-intentioned, the above approach will not compile because `binary_op` is still incomplete when the `variant` type `expression` is instantiated. Further, the approach suffers from a more significant logical flaw: even if C++ syntax were different such that the above example could be made to "work," `expression` would need to be of infinite size, which is clearly impossible.

To overcome these difficulties, `variant` includes special support for the `boost::recursive_wrapper` class template, which breaks the circular dependency at the heart of these problems. Further, `boost::make_recursive_variant` provides a more convenient syntax for declaring recursive `variant` types. Tutorials for use of these facilities is described in the section called "Recursive types with recursive_wrapper" and the section called "Recursive types with make_recursive_variant".

## Recursive types with `recursive_wrapper`

The following example demonstrates how `recursive_wrapper` could be used to solve the problem presented in the section called "Recursive variant types":

```
typedef boost::variant<
      int
    , boost::recursive_wrapper< binary_op<add> >
    , boost::recursive_wrapper< binary_op<sub> >
    > expression;
```

Because `variant` provides special support for `recursive_wrapper`, clients may treat the resultant `variant` as though the wrapper were not present. This is seen in the implementation of the following visitor, which calculates the value of an `expression` without any reference to `recursive_wrapper`:

```
class calculator : public boost::static_visitor<int>
{
public:

    int operator()(int value) const
    {
        return value;
    }

    int operator()(const binary_op<add> & binary) const
    {
        return boost::apply_visitor( calculator(), binary.left )
             + boost::apply_visitor( calculator(), binary.right );
    }

    int operator()(const binary_op<sub> & binary) const
    {
        return boost::apply_visitor( calculator(), binary.left )
             - boost::apply_visitor( calculator(), binary.right );
    }

};
```

Finally, we can demonstrate `expression` in action:

```
void f()
{
    // result = ((7-3)+8) = 12
    expression result(
        binary_op<add>(
            binary_op<sub>(7,3)
          , 8
          )
      );

    assert( boost::apply_visitor(calculator(),result) == 12 );
}
```

## Recursive types with `make_recursive_variant`

For some applications of recursive `variant` types, a user may be able to sacrifice the full flexibility of using `recursive_wrapper` with `variant` for the following convenient syntax:

```
typedef boost::make_recursive_variant<
      int
    , std::vector< boost::recursive_variant_ >
    >::type int_tree_t;
```

Use of the resultant `variant` type is as expected:

```
std::vector< int_tree_t > subresult;
subresult.push_back(3);
subresult.push_back(5);

std::vector< int_tree_t > result;
result.push_back(1);
result.push_back(subresult);
result.push_back(7);

int_tree_t var(result);
```

To be clear, one might represent the resultant content of `var` as `( 1 ( 3 5 ) 7 )`.

Finally, note that a type sequence can be used to specify the bounded types of a recursive `variant` via the use of `boost::make_recursive_variant_over`, whose semantics are the same as `make_variant_over` (which is described in the section called "Using a type sequence to specify bounded types").

**Portability**: Unfortunately, due to standard conformance issues in several compilers, `make_recursive_variant` is not universally supported. On these compilers the library indicates its lack of support via the definition of the preprocessor symbol `BOOST_VARI-ANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT`. Thus, unless working with highly-conformant compilers, maximum portability will be achieved by instead using `recursive_wrapper`, as described in the section called "Recursive types with recursive_wrapper".

## Binary visitation

As the tutorial above demonstrates, visitation is a powerful mechanism for manipulating `variant` content. Binary visitation further extends the power and flexibility of visitation by allowing simultaneous visitation of the content of two different `variant` objects.

Notably this feature requires that binary visitors are incompatible with the visitor objects discussed in the tutorial above, as they must operate on two arguments. The following demonstrates the implementation of a binary visitor:

```
class are_strict_equals
    : public boost::static_visitor<bool>
{
public:

    template <typename T, typename U>
    bool operator()( const T &, const U & ) const
    {
        return false; // cannot compare different types
    }

    template <typename T>
    bool operator()( const T & lhs, const T & rhs ) const
    {
        return lhs == rhs;
    }

};
```

As expected, the visitor is applied to two `variant` arguments by means of `apply_visitor`:

```
boost::variant< int, std::string > v1( "hello" );

boost::variant< double, std::string > v2( "hello" );
assert( boost::apply_visitor(are_strict_equals(), v1, v2) );

boost::variant< int, const char * > v3( "hello" );
assert( !boost::apply_visitor(are_strict_equals(), v1, v3) );
```

Finally, we must note that the function object returned from the "delayed" form of `apply_visitor` also supports binary visitation, as the following demonstrates:

```
typedef boost::variant<double, std::string> my_variant;

std::vector< my_variant > seq1;
seq1.push_back("pi is close to ");
seq1.push_back(3.14);

std::list< my_variant > seq2;
seq2.push_back("pi is close to ");
seq2.push_back(3.14);

are_strict_equals visitor;
assert( std::equal(
      seq1.begin(), seq1.end(), seq2.begin()
    , boost::apply_visitor( visitor )
    ) );
```

# Reference

## Concepts

### BoundedType

The requirements on a **bounded type** are as follows:

- CopyConstructible [20.1.3].

- Destructor upholds the no-throw exception-safety guarantee.

- Complete at the point of `variant` template instantiation. (See `boost::recursive_wrapper`<T> for a type wrapper that accepts incomplete types to enable recursive `variant` types.)

Every type specified as a template argument to `variant` must at minimum fulfill the above requirements. In addition, certain features of `variant` are available only if its bounded types meet the requirements of these following additional concepts:

- Assignable: `variant` is itself *Assignable* if and only if every one of its bounded types meets the requirements of the concept. (Note that top-level `const`-qualified types and reference types do *not* meet these requirements.)

- DefaultConstructible [20.1.4]: `variant` is itself DefaultConstructible if and only if its first bounded type (i.e., `T1`) meets the requirements of the concept.

- EqualityComparable: `variant` is itself EqualityComparable if and only if every one of its bounded types meets the requirements of the concept.

- LessThanComparable: `variant` is itself LessThanComparable if and only if every one of its bounded types meets the requirements of the concept.

- *OutputStreamable*: `variant` is itself *OutputStreamable* if and only if every one of its bounded types meets the requirements of the concept.

### StaticVisitor

The requirements on a **static visitor** of a type `T` are as follows:

- Must allow invocation as a function by overloading `operator()`, unambiguously accepting any value of type `T`.

- Must expose inner type `result_type`. (See `boost::visitor_ptr` for a solution to using functions as visitors.)

• If `result_type` is not `void`, then each operation of the function object must return a value implicitly convertible to `result_type`.

### Examples

The following class satisfies the requirements of a static visitor of several types (i.e., explicitly: `int` and `std::string`; or, e.g., implicitly: `short` and `const char *`; etc.):

```
class my_visitor
    : public boost::static_visitor<int>
{
public:

    int operator()(int i)
    {
        return i * 2;
    }

    int operator()(const std::string& s)
    {
        return s.length();
    }

};
```

Another example is the following class, whose function-call operator is a member template, allowing it to operate on values of many types. Thus, the following class is a visitor of any type that supports streaming output (e.g., `int`, `double`, `std::string`, etc.):

```
class printer
    : public boost::static_visitor<>
{
    template <typename T>
    void operator()(const T& t)
    {
        std::cout << t << std::endl;
    }
};
```

## *OutputStreamable*

The requirements on an **output streamable** type `T` are as follows:

• For any object `t` of type `T`, `std::cout << t` must be a valid expression.

# Header <boost/variant.hpp>

This header exists simply as a convenience to the user, including all of the headers in the `boost/variant` directory.

# Header <boost/variant/variant_fwd.hpp>

Provides forward declarations of the `boost::variant`, `boost::make_variant_over`, `boost::make_recursive_variant`, and `boost::make_recursive_variant_over` class templates and the `boost::recursive_variant_` tag type. Also defines several preprocessor symbols, as described below.

```
BOOST_VARIANT_LIMIT_TYPES
BOOST_VARIANT_ENUM_PARAMS(param)
BOOST_VARIANT_ENUM_SHIFTED_PARAMS(param)
BOOST_VARIANT_NO_REFERENCE_SUPPORT
BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT
BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT
```

## Macro BOOST_VARIANT_LIMIT_TYPES

BOOST_VARIANT_LIMIT_TYPES — Expands to the length of the template parameter list for variant.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>

BOOST_VARIANT_LIMIT_TYPES
```

## Description

**Note**: Conforming implementations of variant must allow at least ten template arguments. That is, BOOST_VARIANT_LIMIT_TYPES must be greater or equal to 10.

## Macro BOOST_VARIANT_ENUM_PARAMS

BOOST_VARIANT_ENUM_PARAMS — Enumerate parameters for use with `variant`.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>

BOOST_VARIANT_ENUM_PARAMS(param)
```

## Description

Expands to a comma-separated sequence of length `BOOST_VARIANT_LIMIT_TYPES`, where each element in the sequence consists of the concatenation of *param* with its zero-based index into the sequence. That is, `param ## 0, param ## 1, ..., param ## BOOST_VARIANT_LIMIT_TYPES - 1`.

**Rationale**: This macro greatly simplifies for the user the process of declaring `variant` types in function templates or explicit partial specializations of class templates, as shown in the tutorial.

## Macro BOOST_VARIANT_ENUM_SHIFTED_PARAMS

BOOST_VARIANT_ENUM_SHIFTED_PARAMS — Enumerate all but the first parameter for use with variant.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>

BOOST_VARIANT_ENUM_SHIFTED_PARAMS(param)
```

## Description

Expands to a comma-separated sequence of length BOOST_VARIANT_LIMIT_TYPES - 1, where each element in the sequence consists of the concatenation of *param* with its one-based index into the sequence. That is, param ## 1, ..., param ## BOOST_VARIANT_LIMIT_TYPES - 1.

**Note**: This macro results in the same expansion as BOOST_VARIANT_ENUM_PARAMS -- but without the first term.

## Macro BOOST_VARIANT_NO_REFERENCE_SUPPORT

BOOST_VARIANT_NO_REFERENCE_SUPPORT — Indicates variant does not support references as bounded types.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>

BOOST_VARIANT_NO_REFERENCE_SUPPORT
```

## Description

Defined only if variant does not support references as bounded types.

## Macro BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT

BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT — Indicates absence of support for specifying the bounded types of a variant by the elements of a type sequence.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>

BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT
```

## Description

Defined only if make_variant_over and make_recursive_variant_over are not supported for some reason on the target compiler.

# Macro BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT

BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT — Indicates make_recursive_variant operates in an implementation-defined manner.

# Synopsis

```
// In header: <boost/variant/variant_fwd.hpp>


BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT
```

### Description

Defined only if make_recursive_variant does not operate as documented on the target compiler, but rather in an implementation-defined manner.

**Implementation Note**: If BOOST_VARIANT_NO_FULL_RECURSIVE_VARIANT_SUPPORT is defined for the target compiler, the current implementation uses the MPL lambda mechanism to approximate the desired behavior. (In most cases, however, such compilers do not have full lambda support either.)

# Header <boost/variant/variant.hpp>

```
namespace boost {
  template<typename T1, typename T2 = unspecified, ...,
           typename TN = unspecified>
    class variant;

  template<typename Sequence> class make_variant_over;
  template<typename T1, typename T2, ..., typename TN>
    void swap(variant<T1, T2, ..., TN> &, variant<T1, T2, ..., TN> &);
  template<typename ElemType, typename Traits, typename T1, typename T2, ...,
           typename TN>
    std::basic_ostream<ElemType,Traits> &
    operator<<(std::basic_ostream<ElemType,Traits> &,
               const variant<T1, T2, ..., TN> &);
}
```

# Class template variant

boost::variant — Safe, generic, stack-based discriminated union container.

# Synopsis

```cpp
// In header: <boost/variant/variant.hpp>

template<typename T1, typename T2 = unspecified, ...,
         typename TN = unspecified>
class variant {
public:
  // types
  typedef unspecified types;

  // construct/copy/destruct
  variant();
  variant(const variant &);
  template<typename T> variant(T &);
  template<typename T> variant(const T &);
  template<typename U1, typename U2, ..., typename UN>
    variant(variant<U1, U2, ..., UN> &);
  template<typename U1, typename U2, ..., typename UN>
    variant(const variant<U1, U2, ..., UN> &);
  ~variant();

  // modifiers
  void swap(variant &);
  variant & operator=(const variant &);
  template<typename T> variant & operator=(const T &);

  // queries
  int which() const;
  bool empty() const;
  const std::type_info & type() const;

  // relational
  bool operator==(const variant &) const;
  template<typename U> void operator==(const U &) const;
  bool operator<(const variant &) const;
  template<typename U> void operator<(const U &) const;
};
```

### Description

The `variant` class template (inspired by Andrei Alexandrescu's class of the same name [Ale01A [46]]) is an efficient, recursive-capable, bounded discriminated union value type capable of containing any value type (either POD or non-POD). It supports construction from any type convertible to one of its bounded types or from a source `variant` whose bounded types are each convertible to one of the destination `variant`'s bounded types. As well, through `apply_visitor`, `variant` supports compile-time checked, type-safe visitation; and through `get`, `variant` supports run-time checked, type-safe value retrieval.

**Notes**:

- The bounded types of the `variant` are exposed via the nested typedef `types`, which is an MPL-compatible Sequence containing the set of types that must be handled by any visitor to the `variant`.

- All members of `variant` satisfy at least the basic guarantee of exception-safety. That is, all operations on a `variant` remain defined even after previous operations have failed.

- Each type specified as a template argument to `variant` must meet the requirements of the *BoundedType* concept.

- Each type specified as a template argument to `variant` must be distinct after removal of qualifiers. Thus, for instance, both `variant<int, int>` and `variant<int, const int>` have undefined behavior.

- Conforming implementations of `variant` must allow at least ten types as template arguments. The exact number of allowed arguments is exposed by the preprocessor macro `BOOST_VARIANT_LIMIT_TYPES`. (See `make_variant_over` for a means to specify the bounded types of a `variant` by the elements of an MPL or compatible Sequence, thus overcoming this limitation.)

**`variant` public construct/copy/destruct**

1.
```
variant();
```

| | |
|---|---|
| Requires: | The first bounded type of the `variant` (i.e., `T1`) must fulfill the requirements of the *DefaultConstructible* [20.1.4] concept. |
| Postconditions: | Content of `*this` is the default value of the first bounded type (i.e, `T1`). |
| Throws: | May fail with any exceptions arising from the default constructor of `T1`. |

2.
```
variant(const variant & other);
```

| | |
|---|---|
| Postconditions: | Content of `*this` is a copy of the content of `other`. |
| Throws: | May fail with any exceptions arising from the copy constructor of `other`'s contained type. |

3.
```
template<typename T> variant(T & operand);
```

| | |
|---|---|
| Requires: | `T` must be unambiguously convertible to one of the bounded types (i.e., `T1`, `T2`, etc.). |
| Postconditions: | Content of `*this` is the best conversion of `operand` to one of the bounded types, as determined by standard overload resolution rules. |
| Throws: | May fail with any exceptions arising from the conversion of `operand` to one of the bounded types. |

4.
```
template<typename T> variant(const T & operand);
```

| | |
|---|---|
| Notes: | Same semantics as previous constructor, but allows construction from temporaries. |

5.
```
template<typename U1, typename U2, ..., typename UN>
  variant(variant<U1, U2, ..., UN> & operand);
```

| | |
|---|---|
| Requires: | *Every* one of `U1`, `U2`, ..., `UN` must have an unambiguous conversion to one of the bounded types (i.e., `T1`, `T2`, ..., `TN`). |
| Postconditions: | If `variant<U1, U2, ..., UN>` is itself one of the bounded types, then content of `*this` is a copy of `operand`. Otherwise, content of `*this` is the best conversion of the content of `operand` to one of the bounded types, as determined by standard overload resolution rules. |
| Throws: | If `variant<U1, U2, ..., UN>` is itself one of the bounded types, then may fail with any exceptions arising from the copy constructor of `variant<U1, U2, ..., UN>`. Otherwise, may fail with any exceptions arising from the conversion of the content of `operand` to one of the bounded types. |

6.
```
template<typename U1, typename U2, ..., typename UN>
  variant(const variant<U1, U2, ..., UN> & operand);
```

| | |
|---|---|
| Notes: | Same semantics as previous constructor, but allows construction from temporaries. |

7.
```
~variant();
```

| | |
|---|---|
| Effects: | Destroys the content of `*this`. |
| Throws: | Will not throw. |

## variant **modifiers**

1.
```
void swap(variant & other);
```

| | |
|---|---|
| Requires: | Every bounded type must fulfill the requirements of the Assignable concept. |
| Effects: | Interchanges the content of *this and other. |
| Throws: | If the contained type of other is the same as the contained type of *this, then may fail with any exceptions arising from the swap of the contents of *this and other. Otherwise, may fail with any exceptions arising from either of the copy constructors of the contained types. Also, in the event of insufficient memory, may fail with std::bad_alloc (why?). |

2.
```
variant & operator=(const variant & rhs);
```

| | |
|---|---|
| Requires: | Every bounded type must fulfill the requirements of the Assignable concept. |
| Effects: | If the contained type of rhs is the same as the contained type of *this, then assigns the content of rhs into the content of *this. Otherwise, makes the content of *this a copy of the content of rhs, destroying the previous content of *this. |
| Throws: | If the contained type of rhs is the same as the contained type of *this, then may fail with any exceptions arising from the assignment of the content of rhs into the content *this. Otherwise, may fail with any exceptions arising from the copy constructor of the contained type of rhs. Also, in the event of insufficient memory, may fail with std::bad_alloc (why?). |

3.
```
template<typename T> variant & operator=(const T & rhs);
```

| | |
|---|---|
| Requires: | • T must be unambiguously convertible to one of the bounded types (i.e., T1, T2, etc.). <br><br>• Every bounded type must fulfill the requirements of the Assignable concept. |
| Effects: | If the contained type of *this is T, then assigns rhs into the content of *this. Otherwise, makes the content of *this the best conversion of rhs to one of the bounded types, as determined by standard overload resolution rules, destroying the previous content of *this. |
| Throws: | If the contained type of *this is T, then may fail with any exceptions arising from the assignment of rhs into the content *this. Otherwise, may fail with any exceptions arising from the conversion of rhs to one of the bounded types. Also, in the event of insufficient memory, may fail with std::bad_alloc (why?). |

## variant **queries**

1.
```
int which() const;
```

| | |
|---|---|
| Returns: | The zero-based index into the set of bounded types of the contained type of *this. (For instance, if called on a variant<int, std::string> object containing a std::string, which() would return 1.) |
| Throws: | Will not throw. |

2.
```
bool empty() const;
```

| | |
|---|---|
| Returns: | false: variant always contains exactly one of its bounded types. (See the section called ""Never-Empty" Guarantee" for more information.) |
| Rationale: | Facilitates generic compatibility with boost::any. |
| Throws: | Will not throw. |

3.
```
const std::type_info & type() const;
```

| | |
|---|---|
| Returns: | typeid(x), where x is the the content of *this. |
| Throws: | Will not throw. |
| Notes: | Not available when BOOST_NO_TYPEID is defined. |

**`variant` relational**

1.
```cpp
bool operator==(const variant & rhs) const;
template<typename U> void operator==(const U &) const;
```

Equality comparison.

| | |
|---|---|
| Notes: | The overload returning `void` exists only to prohibit implicit conversion of the operator's right-hand side to `variant`; thus, its use will (purposefully) result in a compile-time error. |
| Requires: | Every bounded type of the `variant` must fulfill the requirements of the EqualityComparable concept. |
| Returns: | `true` iff `which() == rhs.which()` *and* `content_this == content_rhs`, where `content_this` is the content of `*this` and `content_rhs` is the content of `rhs`. |
| Throws: | If `which() == rhs.which()` then may fail with any exceptions arising from `operator==(T,T)`, where `T` is the contained type of `*this`. |

2.
```cpp
bool operator<(const variant & rhs) const;
template<typename U> void operator<(const U &) const;
```

LessThan comparison.

| | |
|---|---|
| Notes: | The overload returning `void` exists only to prohibit implicit conversion of the operator's right-hand side to `variant`; thus, its use will (purposefully) result in a compile-time error. |
| Requires: | Every bounded type of the `variant` must fulfill the requirements of the LessThanComparable concept. |
| Returns: | If `which() == rhs.which()` then: `content_this < content_rhs`, where `content_this` is the content of `*this` and `content_rhs` is the content of `rhs`. Otherwise: `which() < rhs.which()`. |
| Throws: | If `which() == rhs.which()` then may fail with any exceptions arising from `operator<(T,T)`, where `T` is the contained type of `*this`. |

# Function template swap

boost::swap

# Synopsis

```
// In header: <boost/variant/variant.hpp>


template<typename T1, typename T2, ..., typename TN>
  void swap(variant<T1, T2, ..., TN> & lhs, variant<T1, T2, ..., TN> & rhs);
```

## Description

Effects:      Swaps lhs with rhs by application of variant::swap.

Throws:      May fail with any exception arising from variant::swap.

# Function template operator<<

boost::operator<< — Provides streaming output for `variant` types.

# Synopsis

```cpp
// In header: <boost/variant/variant.hpp>


template<typename ElemType, typename Traits, typename T1, typename T2, ...,
         typename TN>
  std::basic_ostream<ElemType,Traits> &
  operator<<(std::basic_ostream<ElemType,Traits> & out,
             const variant<T1, T2, ..., TN> & rhs);
```

## Description

Requires:      Every bounded type of the `variant` must fulfill the requirements of the *OutputStreamable* concept.

Effects:      Calls `out << x`, where `x` is the content of `rhs`.

Notes:      Not available when `BOOST_NO_IOSTREAM` is defined.

# Class template make_variant_over

boost::make_variant_over — Exposes a `variant` whose bounded types are the elements of the given type sequence.

# Synopsis

```
// In header: <boost/variant/variant.hpp>

template<typename Sequence>
class make_variant_over {
public:
  // types
  typedef variant< unspecified > type;
};
```

## Description

`type` has behavior equivalent in every respect to `variant< Sequence[0], Sequence[1], ... >` (where `Sequence[i]` denotes the *i*-th element of `Sequence`), except that no upper limit is imposed on the number of types.

**Notes**:

- `Sequence` must meet the requirements of MPL's *Sequence* concept.

- Due to standard conformance problems in several compilers, `make_variant_over` may not be supported on your compiler. See BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT for more information.

# Header <boost/variant/recursive_variant.hpp>

```
namespace boost {
  typedef unspecified recursive_variant_;

  template<typename T1, typename T2 = unspecified, ...,
           typename TN = unspecified>
    class make_recursive_variant;
  template<typename Sequence> class make_recursive_variant_over;
}
```

# Class template make_recursive_variant

boost::make_recursive_variant — Simplifies declaration of recursive `variant` types.

# Synopsis

```
// In header: <boost/variant/recursive_variant.hpp>

template<typename T1, typename T2 = unspecified, ...,
         typename TN = unspecified>
class make_recursive_variant {
public:
  // types
  typedef boost::variant< unspecified > type;
};
```

## Description

`type` has behavior equivalent in every respect to some `variant< U1, U2, ..., UN >`, where each type `U`*i* is the result of the corresponding type `T`*i* undergone a transformation function. The following pseudo-code specifies the behavior of this transformation (call it `substitute`):

- If `T`*i* is `boost::recursive_variant_` then: `variant< U1, U2, ..., UN >`;

- Else if `T`*i* is of the form `X * ` then: `substitute(X) *`;

- Else if `T`*i* is of the form `X & ` then: `substitute(X) &`;

- Else if `T`*i* is of the form `R (*)( X1, X2, ..., XN )` then: `substitute(R) (*)( substitute(X1), substitute(X2), ..., substitute(XN) )`;

- Else if `T`*i* is of the form `F < X1, X2, ..., XN >` then: `F< substitute(X1), substitute(X2), ..., substitute(XN) >`;

- Else: `T`*i*.

Note that cv-qualifiers are preserved and that the actual process is generally a bit more complicated. However, the above does convey the essential idea as well as describe the extent of the substititions.

Use of `make_recursive_variant` is demonstrated in the section called "Recursive types with make_recursive_variant".

**Portability**: Due to standard conformance issues in several compilers, `make_recursive_variant` is not universally supported. On these compilers the library indicates its lack of support via the definition of the preprocessor symbol `BOOST_VARIANT_NO_FULL_RE-CURSIVE_VARIANT_SUPPORT`.

# Class template make_recursive_variant_over

boost::make_recursive_variant_over — Exposes a recursive `variant` whose bounded types are the elements of the given type sequence.

# Synopsis

```
// In header: <boost/variant/recursive_variant.hpp>

template<typename Sequence>
class make_recursive_variant_over {
public:
  // types
  typedef variant< unspecified > type;
};
```

## Description

`type` has behavior equivalent in every respect to `make_recursive_variant`< Sequence[0], Sequence[1], ... >::type (where `Sequence[i]` denotes the *i*-th element of `Sequence`), except that no upper limit is imposed on the number of types.

**Notes**:

- `Sequence` must meet the requirements of MPL's *Sequence* concept.

- Due to standard conformance problems in several compilers, `make_recursive_variant_over` may not be supported on your compiler. See `BOOST_VARIANT_NO_TYPE_SEQUENCE_SUPPORT` for more information.

# Header <boost/variant/recursive_wrapper.hpp>

```
namespace boost {
  template<typename T> class recursive_wrapper;
  template<typename T> class is_recursive_wrapper;
  template<typename T> class unwrap_recursive_wrapper;
}
```

# Class template recursive_wrapper

boost::recursive_wrapper — Solves circular dependencies, enabling recursive types.

# Synopsis

```cpp
// In header: <boost/variant/recursive_wrapper.hpp>

template<typename T>
class recursive_wrapper {
public:
  // types
  typedef T type;

  // construct/copy/destruct
  recursive_wrapper();
  recursive_wrapper(const recursive_wrapper &);
  recursive_wrapper(const T &);
  ~recursive_wrapper();

  // modifiers
  void swap(recursive_wrapper &);
  recursive_wrapper & operator=(const recursive_wrapper &);
  recursive_wrapper & operator=(const T &);

  // queries
  T & get();
  const T & get() const;
  T * get_pointer();
  const T * get_pointer() const;
};
```

## Description

The `recursive_wrapper` class template has an interface similar to a simple value container, but its content is allocated dynamically. This allows `recursive_wrapper` to hold types `T` whose member data leads to a circular dependency (e.g., a data member of `T` has a data member of type `T`).

The application of `recursive_wrapper` is easiest understood in context. See the section called "Recursive types with recursive_wrapper" for a demonstration of a common use of the class template.

**Notes**:

• Any type specified as the template argument to `recursive_wrapper` must be capable of construction via `operator new`. Thus, for instance, references are not supported.

**`recursive_wrapper` public construct/copy/destruct**

1.
```cpp
recursive_wrapper();
```

Default constructor.

Initializes `*this` by default construction of `T`.

| | |
|---|---|
| Requires: | `T` must fulfill the requirements of the *DefaultConstructible* [20.1.4] concept. |
| Throws: | May fail with any exceptions arising from the default constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`. |

2.
```cpp
recursive_wrapper(const recursive_wrapper & other);
```

Copy constructor.

Copies the content of `other` into `*this`.

Throws:      May fail with any exceptions arising from the copy constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`.

3.
```cpp
recursive_wrapper(const T & operand);
```

Value constructor.

Copies `operand` into `*this`.

Throws:      May fail with any exceptions arising from the copy constructor of `T` or, in the event of insufficient memory, with `std::bad_alloc`.

4.
```cpp
~recursive_wrapper();
```

Destructor.

Deletes the content of `*this`.

Throws:      Will not throw.

## `recursive_wrapper` modifiers

1.
```cpp
void swap(recursive_wrapper & other);
```

Exchanges contents of `*this` and `other`.

Throws:      Will not throw.

2.
```cpp
recursive_wrapper & operator=(const recursive_wrapper & rhs);
```

Copy assignment operator.

Assigns the content of `rhs` to the content of `*this`.

Requires:      `T` must fulfill the requirements of the Assignable concept.

Throws:      May fail with any exceptions arising from the assignment operator of `T`.

3.
```cpp
recursive_wrapper & operator=(const T & rhs);
```

Value assignment operator.

Assigns `rhs` into the content of `*this`.

Requires:      `T` must fulfill the requirements of the Assignable concept.

Throws:      May fail with any exceptions arising from the assignment operator of `T`.

## `recursive_wrapper` queries

1.
```cpp
T & get();
const T & get() const;
```

Returns a reference to the content of `*this`.

Throws:      Will not throw.

2.
```cpp
T * get_pointer();
const T * get_pointer() const;
```

Returns a pointer to the content of `*this`.

Throws:     Will not throw.

# Class template is_recursive_wrapper

boost::is_recursive_wrapper — Determines whether the specified type is a specialization of `recursive_wrapper`.

# Synopsis

```
// In header: <boost/variant/recursive_wrapper.hpp>

template<typename T>
class is_recursive_wrapper {
public:
  // types
  typedef unspecified type;

  // static constants
  static const bool value = unspecified;
};
```

## Description

Value is true iff `T` is a specialization of `recursive_wrapper`.

**Note**: `is_recursive_wrapper` is a model of MPL's *IntegralConstant* concept.

# Class template unwrap_recursive_wrapper

boost::unwrap_recursive_wrapper — Unwraps the specified argument if given a specialization of `recursive_wrapper`.

# Synopsis

```
// In header: <boost/variant/recursive_wrapper.hpp>

template<typename T>
class unwrap_recursive_wrapper {
public:
  // types
  typedef unspecified type;
};
```

## Description

`type` is equivalent to `T::type` if `T` is a specialization of `recursive_wrapper`. Otherwise, `type` is equivalent to `T`.

# Header <boost/variant/apply_visitor.hpp>

```
namespace boost {
  template<typename Visitor> class apply_visitor_delayed_t;
  template<typename Visitor, typename Variant>
    typename Visitor::result_type apply_visitor(Visitor &, Variant &);
  template<typename Visitor, typename Variant>
    typename Visitor::result_type apply_visitor(const Visitor &, Variant &);
  template<typename BinaryVisitor, typename Variant1, typename Variant2>
    typename BinaryVisitor::result_type
    apply_visitor(BinaryVisitor &, Variant1 &, Variant2 &);
  template<typename BinaryVisitor, typename Variant1, typename Variant2>
    typename BinaryVisitor::result_type
    apply_visitor(const BinaryVisitor &, Variant1 &, Variant2 &);
  template<typename Visitor>
    apply_visitor_delayed_t<Visitor> apply_visitor(Visitor &);
}
```

# Class template apply_visitor_delayed_t

boost::apply_visitor_delayed_t — Adapts a visitor for use as a function object.

# Synopsis

```cpp
// In header: <boost/variant/apply_visitor.hpp>

template<typename Visitor>
class apply_visitor_delayed_t {
public:
  // types
  typedef typename Visitor::result_type result_type;

  // construct/copy/destruct
  explicit apply_visitor_delayed_t(Visitor &);

  // function object interface
  template<typename Variant> result_type operator()(Variant &);
  template<typename Variant1, typename Variant2>
    result_type operator()(Variant1 &, Variant2 &);
};
```

### Description

Adapts the function given at construction for use as a function object. This is useful, for example, when one needs to operate on each element of a sequence of variant objects using a standard library algorithm such as `std::for_each`.

See the "visitor-only" form of apply_visitor for a simple way to create apply_visitor_delayed_t objects.

#### apply_visitor_delayed_t public construct/copy/destruct

1.
```cpp
explicit apply_visitor_delayed_t(Visitor & visitor);
```

Effects:    Constructs the function object with the given visitor.

#### apply_visitor_delayed_t function object interface

1.
```cpp
template<typename Variant> result_type operator()(Variant & operand);
template<typename Variant1, typename Variant2>
  result_type operator()(Variant1 & operand1, Variant2 & operand2);
```

Function call operator.

Invokes apply_visitor on the stored visitor using the given operands.

# Function apply_visitor

boost::apply_visitor — Allows compile-time checked type-safe application of the given visitor to the content of the given variant, ensuring that all types are handled by the visitor.

# Synopsis

```
// In header: <boost/variant/apply_visitor.hpp>


template<typename Visitor, typename Variant>
  typename Visitor::result_type
  apply_visitor(Visitor & visitor, Variant & operand);
template<typename Visitor, typename Variant>
  typename Visitor::result_type
  apply_visitor(const Visitor & visitor, Variant & operand);
template<typename BinaryVisitor, typename Variant1, typename Variant2>
  typename BinaryVisitor::result_type
  apply_visitor(BinaryVisitor & visitor, Variant1 & operand1,
                Variant2 & operand2);
template<typename BinaryVisitor, typename Variant1, typename Variant2>
  typename BinaryVisitor::result_type
  apply_visitor(const BinaryVisitor & visitor, Variant1 & operand1,
                Variant2 & operand2);
template<typename Visitor>
  apply_visitor_delayed_t<Visitor> apply_visitor(Visitor & visitor);
```

## Description

The behavior of `apply_visitor` is dependent on the number of arguments on which it operates (i.e., other than the visitor). The function behaves as follows:

- Overloads accepting one operand invoke the unary function call operator of the given visitor on the content of the given `variant` operand.

- Overloads accepting two operands invoke the binary function call operator of the given visitor on the content of the given `variant` operands.

- The overload accepting only a visitor returns a `generic function object` that accepts either one or two arguments and invokes `apply_visitor` using these arguments and `visitor`, thus behaving as specified above. (This behavior is particularly useful, for example, when one needs to operate on each element of a sequence of variant objects using a standard library algorithm.)

| | |
|---|---|
| Returns: | The overloads acccepting operands return the result of applying the given visitor to the content of the given operands. The overload accepting only a visitor return a function object, thus delaying application of the visitor to any operands. |
| Requires: | The given visitor must fulfill the *StaticVisitor* concept requirements with respect to each of the bounded types of the given `variant`. |
| Throws: | The overloads accepting operands throw only if the given visitor throws when applied. The overload accepting only a visitor will not throw. (Note, however, that the returned function object may throw when invoked.) |

# Header <boost/variant/get.hpp>

```
namespace boost {
  class bad_get;
  template<typename U, typename T1, typename T2, ..., typename TN>
    U * get(variant<T1, T2, ..., TN> *);
  template<typename U, typename T1, typename T2, ..., typename TN>
    const U * get(const variant<T1, T2, ..., TN> *);
  template<typename U, typename T1, typename T2, ..., typename TN>
    U & get(variant<T1, T2, ..., TN> &);
  template<typename U, typename T1, typename T2, ..., typename TN>
    const U & get(const variant<T1, T2, ..., TN> &);
}
```

# Class bad_get

boost::bad_get — The exception thrown in the event of a failed application of `boost::get` on the given operand value.

# Synopsis

```
// In header: <boost/variant/get.hpp>


class bad_get : public std::exception {
public:
  virtual const char * what() const;
};
```

## Description

```
virtual const char * what() const;
```

# Function get

boost::get — Retrieves a value of a specified type from a given `variant`.

# Synopsis

```
// In header: <boost/variant/get.hpp>


template<typename U, typename T1, typename T2, ..., typename TN>
  U * get(variant<T1, T2, ..., TN> * operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  const U * get(const variant<T1, T2, ..., TN> * operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  U & get(variant<T1, T2, ..., TN> & operand);
template<typename U, typename T1, typename T2, ..., typename TN>
  const U & get(const variant<T1, T2, ..., TN> & operand);
```

## Description

The `get` function allows run-time checked, type-safe retrieval of the content of the given `variant`. The function succeeds only if the content is of the specified type U, with failure indicated as described below.

**Warning**: After either `operand` or its content is destroyed (e.g., when the given `variant` is assigned a value of different type), the returned reference is invalidated. Thus, significant care and caution must be extended when handling the returned reference.

| | |
|---|---|
| Notes: | As part of its guarantee of type-safety, `get` enforces `const`-correctness. Thus, the specified type U must be `const`-qualified whenever `operand` or its content is likewise `const`-qualified. The converse, however, is not required: that is, the specified type U may be `const`-qualified even when `operand` and its content are not. |
| Returns: | If passed a pointer, `get` returns a pointer to the value content if it is of the specified type U; otherwise, a null pointer is returned. If passed a reference, `get` returns a reference to the value content if it is of the specified type U; otherwise, an exception is thrown (see below). |
| Throws: | Overloads taking a `variant` pointer will not throw; the overloads taking a `variant` reference throw `bad_get` if the content is not of the specified type U. |
| Rationale: | While visitation via `apply_visitor` is generally preferred due to its greater safety, `get` may may be more convenient in some cases due to its straightforward usage. |

# Header <boost/variant/bad_visit.hpp>

```
namespace boost {
  class bad_visit;
}
```

# Class bad_visit

boost::bad_visit — The exception thrown in the event of a visitor unable to handle the visited value.

# Synopsis

```
// In header: <boost/variant/bad_visit.hpp>


class bad_visit : public std::exception {
public:
  virtual const char * what() const;
};
```

## Description

```
virtual const char * what() const;
```

# Header <boost/variant/static_visitor.hpp>

```
namespace boost {
  template<typename ResultType> class static_visitor;
}
```

# Class template static_visitor

boost::static_visitor — Convenient base type for static visitors.

# Synopsis

```
// In header: <boost/variant/static_visitor.hpp>


template<typename ResultType>
class static_visitor {
public:
  // types
  typedef ResultType result_type;  // Exposes result_type member as required by StaticVisitor ↵
concept.
};
```

## Description

Denotes the intent of the deriving class as meeting the requirements of a static visitor of some type. Also exposes the inner type `result_type` as required by the *StaticVisitor* concept.

**Notes**: `static_visitor` is intended for use as a base type only and is therefore noninstantiable.

# Header <boost/variant/visitor_ptr.hpp>

```
namespace boost {
  template<typename T, typename R> class visitor_ptr_t;
  template<typename R, typename T> visitor_ptr_t<T,R> visitor_ptr(R (*)(T));
}
```

# Class template visitor_ptr_t

boost::visitor_ptr_t — Adapts a function pointer for use as a static visitor.

# Synopsis

```
// In header: <boost/variant/visitor_ptr.hpp>

template<typename T, typename R>
class visitor_ptr_t : public static_visitor<R> {
public:
  // construct/copy/destruct
  explicit visitor_ptr_t(R (*)(T));

  // static visitor interfaces
  R operator()(unspecified-forwarding-type);
  template<typename U> void operator()(const U&);
};
```

## Description

Adapts the function given at construction for use as a static visitor of type `T` with result type `R`.

### visitor_ptr_t public construct/copy/destruct

1.
```
explicit visitor_ptr_t(R (*)(T));
```

Effects:      Constructs the visitor with the given function.

### visitor_ptr_t static visitor interfaces

1.
```
R operator()(unspecified-forwarding-type operand);
template<typename U> void operator()(const U&);
```

Effects:      If passed a value or reference of type `T`, it invokes the function given at construction, appropriately forwarding `operand`.
Returns:      Returns the result of the function invocation.
Throws:       The overload taking a value or reference of type `T` throws if the invoked function throws. The overload taking all other values *always* throws `bad_visit`.

## Function template visitor_ptr

boost::visitor_ptr — Returns a visitor object that adapts function pointers for use as a static visitor.

## Synopsis

```
// In header: <boost/variant/visitor_ptr.hpp>


template<typename R, typename T> visitor_ptr_t<T,R> visitor_ptr(R (*)(T));
```

### Description

Constructs and returns a `visitor_ptr_t` adaptor over the given function.

Returns:     Returns a `visitor_ptr_t` visitor object that, when applied, invokes the given function.
Throws:     Will not throw. (Note, however, that the returned visitor object may throw when applied.)

# Design Overview

## "Never-Empty" Guarantee

## The Guarantee

All instances `v` of type `variant<T1,T2,...,TN>` guarantee that `v` has constructed content of one of the types `Ti`, even if an operation on `v` has previously failed.

This implies that `variant` may be viewed precisely as a union of *exactly* its bounded types. This "never-empty" property insulates the user from the possibility of undefined `variant` content and the significant additional complexity-of-use attendant with such a possibility.

## The Implementation Problem

While the never-empty guarantee might at first seem "obvious," it is in fact not even straightforward how to implement it in general (i.e., without unreasonably restrictive additional requirements on bounded types).

The central difficulty emerges in the details of `variant` assignment. Given two instances `v1` and `v2` of some concrete `variant` type, there are two distinct, fundamental cases we must consider for the assignment `v1 = v2`.

First consider the case that `v1` and `v2` each contains a value of the same type. Call this type `T`. In this situation, assignment is perfectly straightforward: use `T::operator=`.

However, we must also consider the case that `v1` and `v2` contain values *of distinct types*. Call these types `T` and `U`. At this point, since `variant` manages its content on the stack, the left-hand side of the assignment (i.e., `v1`) must destroy its content so as to permit in-place copy-construction of the content of the right-hand side (i.e., `v2`). In the end, whereas `v1` began with content of type `T`, it ends with content of type `U`, namely a copy of the content of `v2`.

The crux of the problem, then, is this: in the event that copy-construction of the content of `v2` fails, how can `v1` maintain its "never-empty" guarantee? By the time copy-construction from `v2` is attempted, `v1` has already destroyed its content!

## The "Ideal" Solution: False Hopes

Upon learning of this dilemma, clever individuals may propose the following scheme hoping to solve the problem:

1. Provide some "backup" storage, appropriately aligned, capable of holding values of the contained type of the left-hand side.

2. Copy the memory (e.g., using `memcpy`) of the storage of the left-hand side to the backup storage.

3. Attempt a copy of the right-hand side content to the (now-replicated) left-hand side storage.

4. In the event of an exception from the copy, restore the backup (i.e., copy the memory from the backup storage back into the left-hand side storage).

5. Otherwise, in the event of success, now copy the memory of the left-hand side storage to another "temporary" aligned storage.

6. Now restore the backup (i.e., again copying the memory) to the left-hand side storage; with the "old" content now restored, invoke the destructor of the contained type on the storage of the left-hand side.

7. Finally, copy the memory of the temporary storage to the (now-empty) storage of the left-hand side.

While complicated, it appears such a scheme could provide the desired safety in a relatively efficient manner. In fact, several early iterations of the library implemented this very approach.

Unfortunately, as Dave Abraham's first noted, the scheme results in undefined behavior:

> "That's a lot of code to read through, but if it's doing what I think it's doing, it's undefined behavior.

> "Is the trick to move the bits for an existing object into a buffer so we can tentatively construct a new object in that memory, and later move the old bits back temporarily to destroy the old object?

> "The standard does not give license to do that: only one object may have a given address at a time. See 3.8, and particularly paragraph 4."

Additionally, as close examination quickly reveals, the scheme has the potential to create irreconcilable race-conditions in concurrent environments.

Ultimately, even if the above scheme could be made to work on certain platforms with particular compilers, it is still necessary to find a portable solution.

## An Initial Solution: Double Storage

Upon learning of the infeasibility of the above scheme, Anthony Williams proposed in [Wil02] [46] a scheme that served as the basis for a portable solution in some pre-release implementations of `variant`.

The essential idea to this scheme, which shall be referred to as the "double storage" scheme, is to provide enough space within a `variant` to hold two separate values of any of the bounded types.

With the secondary storage, a copy the right-hand side can be attempted without first destroying the content of the left-hand side; accordingly, the content of the left-hand side remains available in the event of an exception.

Thus, with this scheme, the `variant` implementation needs only to keep track of which storage contains the content -- and dispatch any visitation requests, queries, etc. accordingly.

The most obvious flaw to this approach is the space overhead incurred. Though some optimizations could be applied in special cases to eliminate the need for double storage -- for certain bounded types or in some cases entirely (see the section called "Enabling Optimizations" for more details) -- many users on the Boost mailing list strongly objected to the use of double storage. In particular, it was noted that the overhead of double storage would be at play at all times -- even if assignment to `variant` never occurred. For this reason and others, a new approach was developed.

## Current Approach: Temporary Heap Backup

Despite the many objections to the double storage solution, it was realized that no replacement would be without drawbacks. Thus, a compromise was desired.

To this end, Dave Abrahams suggested to include the following in the behavior specification for `variant` assignment: "`variant` assignment from one type to another may incur dynamic allocation." That is, while `variant` would continue to store its content *in*

*situ* after construction and after assignment involving identical contained types, `variant` would store its content on the heap after assignment involving distinct contained types.

The algorithm for assignment would proceed as follows:

1. Copy-construct the content of the right-hand side to the heap; call the pointer to this data `p`.

2. Destroy the content of the left-hand side.

3. Copy `p` to the left-hand side storage.

Since all operations on pointers are nothrow, this scheme would allow `variant` to meet its never-empty guarantee.

The most obvious concern with this approach is that while it certainly eliminates the space overhead of double storage, it introduces the overhead of dynamic-allocation to `variant` assignment -- not just in terms of the initial allocation but also as a result of the continued storage of the content on the heap. While the former problem is unavoidable, the latter problem may be avoided with the following "temporary heap backup" technique:

1. Copy-construct the content of the *left*-hand side to the heap; call the pointer to this data `backup`.

2. Destroy the content of the left-hand side.

3. Copy-construct the content of the right-hand side in the (now-empty) storage of the left-hand side.

4. In the event of failure, copy `backup` to the left-hand side storage.

5. In the event of success, deallocate the data pointed to by `backup`.

With this technique: 1) only a single storage is used; 2) allocation is on the heap in the long-term only if the assignment fails; and 3) after any *successful* assignment, storage within the `variant` is guaranteed. For the purposes of the initial release of the library, these characteristics were deemed a satisfactory compromise solution.

There remain notable shortcomings, however. In particular, there may be some users for which heap allocation must be avoided at all costs; for other users, any allocation may need to occur via a user-supplied allocator. These issues will be addressed in the future (see the section called "Future Direction: Policy-based Implementation"). For now, though, the library treats storage of its content as an implementation detail. Nonetheless, as described in the next section, there *are* certain things the user can do to ensure the greatest efficiency for `variant` instances (see the section called "Enabling Optimizations" for details).

# Enabling Optimizations

As described in the section called "The Implementation Problem", the central difficulty in implementing the never-empty guarantee is the possibility of failed copy-construction during `variant` assignment. Yet types with nothrow copy constructors clearly never face this possibility. Similarly, if one of the bounded types of the `variant` is nothrow default-constructible, then such a type could be used as a safe "fallback" type in the event of failed copy construction.

Accordingly, `variant` is designed to enable the following optimizations once the following criteria on its bounded types are met:

- For each bounded type `T` that is nothrow copy-constructible (as indicated by `boost::has_nothrow_copy`), the library guarantees `variant` will use only single storage and in-place construction for `T`.

- If *any* bounded type is nothrow default-constructible (as indicated by `boost::has_nothrow_constructor`), the library guarantees `variant` will use only single storage and in-place construction for *every* bounded type in the `variant`. Note, however, that in the event of assignment failure, an unspecified nothrow default-constructible bounded type will be default-constructed in the left-hand side operand so as to preserve the never-empty guarantee.

**Caveat**: On most platforms, the Type Traits templates `has_nothrow_copy` and `has_nothrow_constructor` by default return `false` for all `class` and `struct` types. It is necessary therefore to provide specializations of these templates as appropriate for user-defined types, as demonstrated in the following:

```
// ...in your code (at file scope)...

namespace boost {

  template <>
  struct has_nothrow_copy< myUDT >
    : mpl::true_
  {
  };

}
```

**Implementation Note**: So as to make the behavior of `variant` more predictable in the aftermath of an exception, the current implementation prefers to default-construct `boost::blank` if specified as a bounded type instead of other nothrow default-constructible bounded types. (If this is deemed to be a useful feature, it will become part of the specification for `variant`; otherwise, it may be obsoleted. Please provide feedback to the Boost mailing list.)

## Future Direction: Policy-based Implementation

As the previous sections have demonstrated, much effort has been expended in an attempt to provide a balance between performance, data size, and heap usage. Further, significant optimizations may be enabled in `variant` on the basis of certain traits of its bounded types.

However, there will be some users for whom the chosen compromise is unsatisfactory (e.g.: heap allocation must be avoided at all costs; if heap allocation is used, custom allocators must be used; etc.). For this reason, a future version of the library will support a policy-based implementation of `variant`. While this will not eliminate the problems described in the previous sections, it will allow the decisions regarding tradeoffs to be decided by the user rather than the library designers.

# Miscellaneous Notes

# Boost.Variant vs. Boost.Any

As a discriminated union container, the Variant library shares many of the same features of the Any library. However, since neither library wholly encapsulates the features of the other, one library cannot be generally recommended for use over the other.

That said, Boost.Variant has several advantages over Boost.Any, such as:

- Boost.Variant guarantees the type of its content is one of a finite, user-specified set of types.

- Boost.Variant provides *compile-time* checked visitation of its content. (By contrast, the current version of Boost.Any provides no visitation mechanism at all; but even if it did, it would need to be checked at run-time.)

- Boost.Variant enables generic visitation of its content. (Even if Boost.Any did provide a visitation mechanism, it would enable visitation only of explicitly-specified types.)

- Boost.Variant offers an efficient, stack-based storage scheme (avoiding the overhead of dynamic allocation).

Of course, Boost.Any has several advantages over Boost.Variant, such as:

- Boost.Any, as its name implies, allows virtually any type for its content, providing great flexibility.

- Boost.Any provides the no-throw guarantee of exception safety for its swap operation.

- Boost.Any makes little use of template metaprogramming techniques (avoiding potentially hard-to-read error messages and significant compile-time processor and memory demands).

# Portability

The library aims for 100% ANSI/ISO C++ conformance. However, this is strictly impossible due to the inherently non-portable nature of the Type Traits library's `type_with_alignment` facility. In practice though, no compilers or platforms have been discovered where this reliance on undefined behavior has been an issue.

Additionally, significant effort has been expended to ensure proper functioning despite various compiler bugs and other conformance problems. To date the library testsuite has been compiled and tested successfully on at least the following compilers for basic and advanced functionality:

|  | **Basic** | `variant<T&>` | `make_variant_over` | `make_recursive_variant` |
|---|---|---|---|---|
| Borland C++ 5.5.1 and 5.6.4 | X | X | | |
| Comeau C++ 4.3.0 | X | X | X | X |
| GNU GCC 3.3.1 | X | X | X | X |
| GNU GCC 2.95.3 | X | X | | X |
| Intel C++ 7.0 | X | | X | X |
| Metrowerks CodeWarrior 8.3 | X | | X | X |
| Microsoft Visual C++ 7.1 | X | X | X | X |
| Microsoft Visual C++ 6 SP5 and 7 | X | | | |

Finally, the current state of the testsuite in CVS may be found on the Test Summary page. Please note, however, that this page reports on day-to-day changes to inter-release code found in the Boost CVS and thus likely does not match the state of code found in Boost releases.

# Troubleshooting

Due to the heavy use of templates in the implementation of `variant`, it is not uncommon when compiling to encounter problems related to template instantiaton depth, compiler memory, etc. This section attempts to provide advice to common problems experienced on several popular compilers.

(This section is still in progress, with additional advice/feedback welcome. Please post to the Boost-Users list with any useful experiences you may have.)

## "Template instantiation depth exceeds maximum"

### GNU GCC

The compiler option `-ftemplate-depth-NN` can increase the maximum allowed instantiation depth. (Try `-ftemplate-depth-50`.)

## "Internal heap limit reached"

### Microsoft Visual C++

The compiler option `/ZmNNN` can increase the memory allocation limit. The `NNN` is a scaling percentage (i.e., `100` denotes the default limit). (Try `/Zm200`.)

# Acknowledgments

Eric Friedman and Itay Maman designed the initial submission; Eric was the primary implementer.

Eric is also the library maintainer and has expanded upon the initial submission -- adding `make_recursive_variant`, `make_variant_over`, support for reference content, etc.

Andrei Alexandrescu's work in [Ale01a [46]] and [Ale02 [46]] inspired the library's design.

Jeff Garland was the formal review manager.

Douglas Gregor, Dave Abrahams, Anthony Williams, Fernando Cacciola, Joel de Guzman, Dirk Schreib, Brad King, Giovanni Bajo, Eugene Gladyshev, and others provided helpful feedback and suggestions to refine the semantics, interface, and implementation of the library.

# References

[Abr00] David Abrahams. "Exception-Safety in Generic Components." M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming '98, Proc. of a Dagstuhl Seminar, Lecture Notes on Computer Science, Vol. 1766, pp. 69-79. Springer-Verlag Berlin Heidelberg. 2000.

[Abr01] David Abrahams. "Error and Exception Handling." Boost technical article. 2001-2003.

[Ale01a] Andrei Alexandrescu. "An Implementation of Discriminated Unions in C++." *OOPSLA 2001*, Second Workshop on C++ Template Programming. Tampa Bay, 14 October 2001.

[Ale01b] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, C++ In-Depth series. 2001.

[Ale02] Andrei Alexandrescu. "Generic<Programming>: Discriminated Unions" series: Part 1, Part 2, Part 3. *C/C++ Users Journal*. 2002.

[Boo02] Various Boost members. "Proposal --- A type-safe union." Boost public discussion. 2002.

[C++98] *International Standard, Programming Languages – C++*. ISO/IEC:14882. 1998.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1995.

[Gre02] Douglas Gregor. "BOOST_USER: variant." Boost Wiki paper. 2002.

MPL Aleksey Gurtovoy. *Boost Metaprogramming Library*. 2002.

Any Kevlin Henney. *Boost Any Library*. 2001.

Preprocessor Paul Mensonides and Vesa Karvonen. *Boost Preprocessor Library*. 2002.

Type Traits Steve Cleary, Beman Dawes, Aleksey Gurtovoy, Howard Hinnant, Jesse Jones, Mat Marcus, John Maddock, Jeremy Siek. *Boost Type Traits Library*. 2001.

[Sut00] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, C++ In-Depth series. 2000.

[Wil02] Anthony Williams. Double-Storage Proposal. 2002.