Help

```
// Copyright 2016-2018 by Martin Moene
//
// https://github.com/martinmoene/variant-lite
//
// Distributed under the Boost Software License, Version 1.0.
// (See accompanying file LICENSE.txt or copy at http://www.boost.org/LICENSE_1_

#pragma once

#ifndef NONSTD_VARIANT_LITE_HPP
#define NONSTD_VARIANT_LITE_HPP

#define variant_lite_MAJOR  1
#define variant_lite_MINOR  1
#define variant_lite_PATCH  0

#define variant_lite_VERSION  variant_STRINGIFY(variant_lite_MAJOR) "." variant_

#define variant_STRINGIFY(  x )  variant_STRINGIFY_( x )
#define variant_STRINGIFY_( x )  #x

// variant-lite configuration:

#define variant_VARIANT_DEFAULT  0
#define variant_VARIANT_NONSTD   1
#define variant_VARIANT_STD      2

#if !defined( variant_CONFIG_SELECT_VARIANT )
# define variant_CONFIG_SELECT_VARIANT  ( variant_HAVE_STD_VARIANT ? variant_VAR
#endif

#ifndef  variant_CONFIG_OMIT_VARIANT_SIZE_V_MACRO
# define variant_CONFIG_OMIT_VARIANT_SIZE_V_MACRO  0
#endif

#ifndef  variant_CONFIG_OMIT_VARIANT_ALTERNATIVE_T_MACRO
# define variant_CONFIG_OMIT_VARIANT_ALTERNATIVE_T_MACRO  0
#endif
```

```
// Control presence of exception handling (try and auto discover):

#ifndef variant_CONFIG_NO_EXCEPTIONS
# if defined(__cpp_exceptions) || defined(__EXCEPTIONS) || defined(_CPPUNWIND)
#  define variant_CONFIG_NO_EXCEPTIONS  0
# else
#  define variant_CONFIG_NO_EXCEPTIONS  1
# endif
#endif

// C++ language version detection (C++20 is speculative):
// Note: VC14.0/1900 (VS2015) lacks too much from C++14.

#ifndef   variant_CPLUSPLUS
# if defined(_MSVC_LANG ) && !defined(__clang__)
#  define variant_CPLUSPLUS  (_MSC_VER == 1900 ? 201103L : _MSVC_LANG )
# else
#  define variant_CPLUSPLUS  __cplusplus
# endif
#endif

#define variant_CPP98_OR_GREATER  ( variant_CPLUSPLUS >= 199711L )
#define variant_CPP11_OR_GREATER  ( variant_CPLUSPLUS >= 201103L )
#define variant_CPP11_OR_GREATER_ ( variant_CPLUSPLUS >= 201103L )
#define variant_CPP14_OR_GREATER  ( variant_CPLUSPLUS >= 201402L )
#define variant_CPP17_OR_GREATER  ( variant_CPLUSPLUS >= 201703L )
#define variant_CPP20_OR_GREATER  ( variant_CPLUSPLUS >= 202000L )

// Use C++17 std::variant if available and requested:

#if variant_CPP17_OR_GREATER && defined(__has_include )
# if __has_include( <variant> )
#  define variant_HAVE_STD_VARIANT  1
# else
#  define variant_HAVE_STD_VARIANT  0
# endif
#else
# define  variant_HAVE_STD_VARIANT  0
#endif

#define  variant_USES_STD_VARIANT  ( (variant_CONFIG_SELECT_VARIANT == variant_V
```

```
//
// in_place: code duplicated in any-lite, expected-lite, optional-lite, value-pt
//

#ifndef nonstd_lite_HAVE_IN_PLACE_TYPES
#define nonstd_lite_HAVE_IN_PLACE_TYPES  1

// C++17 std::in_place in <utility>:

#if variant_CPP17_OR_GREATER

#include <utility>

namespace nonstd {

using std::in_place;
using std::in_place_type;
using std::in_place_index;
using std::in_place_t;
using std::in_place_type_t;
using std::in_place_index_t;

#define nonstd_lite_in_place_t(      T)  std::in_place_t
#define nonstd_lite_in_place_type_t( T)  std::in_place_type_t<T>
#define nonstd_lite_in_place_index_t(K)  std::in_place_index_t<K>

#define nonstd_lite_in_place(      T)    std::in_place_t{}
#define nonstd_lite_in_place_type( T)    std::in_place_type_t<T>{}
#define nonstd_lite_in_place_index(K)    std::in_place_index_t<K>{}

} // namespace nonstd

#else // variant_CPP17_OR_GREATER

#include <cstddef>

namespace nonstd {
namespace detail {

template< class T >
```

```
struct in_place_type_tag {};

template< std::size_t K >
struct in_place_index_tag {};

} // namespace detail

struct in_place_t {};

template< class T >
inline in_place_t in_place( detail::in_place_type_tag<T> = detail::in_place_type
{
    return in_place_t();
}

template< std::size_t K >
inline in_place_t in_place( detail::in_place_index_tag<K> = detail::in_place_ind
{
    return in_place_t();
}

template< class T >
inline in_place_t in_place_type( detail::in_place_type_tag<T> = detail::in_place
{
    return in_place_t();
}

template< std::size_t K >
inline in_place_t in_place_index( detail::in_place_index_tag<K> = detail::in_pla
{
    return in_place_t();
}

// mimic templated typedef:

#define nonstd_lite_in_place_t(      T)  nonstd::in_place_t(&)( nonstd::detail::
#define nonstd_lite_in_place_type_t( T)  nonstd::in_place_t(&)( nonstd::detail::
#define nonstd_lite_in_place_index_t(K)  nonstd::in_place_t(&)( nonstd::detail::

#define nonstd_lite_in_place(      T)    nonstd::in_place_type<T>
#define nonstd_lite_in_place_type( T)    nonstd::in_place_type<T>
```

```
#define nonstd_lite_in_place_index(K)    nonstd::in_place_index<K>

} // namespace nonstd

#endif // variant_CPP17_OR_GREATER
#endif // nonstd_lite_HAVE_IN_PLACE_TYPES

//
// Use C++17 std::variant:
//

#if variant_USES_STD_VARIANT

#include <functional>   // std::hash<>
#include <
href../../../../../common/math/jlparser/include/jlparser/variant_h_src.pdfvarian

#if ! variant_CONFIG_OMIT_VARIANT_SIZE_V_MACRO
# define variant_size_V(T)  nonstd::variant_size<T>::value
#endif

#if ! variant_CONFIG_OMIT_VARIANT_ALTERNATIVE_T_MACRO
# define variant_alternative_T(K,T)  typename nonstd::variant_alternative<K,T >:
#endif

namespace nonstd {

    using std::variant;
    using std::monostate;
    using std::bad_variant_access;
    using std::variant_size;
    using std::variant_size_v;
    using std::variant_alternative;
    using std::variant_alternative_t;
    using std::hash;

    using std::visit;
    using std::holds_alternative;
    using std::get;
    using std::get_if;
    using std::operator==;
```

```
    using std::operator!=;
    using std::operator<;
    using std::operator<=;
    using std::operator>;
    using std::operator>=;
    using std::swap;

    constexpr auto variant_npos = std::variant_npos;
}

#else // variant_USES_STD_VARIANT

#include <cstddef>
#include <limits>
#include <new>
#include <utility>

#if variant_CONFIG_NO_EXCEPTIONS
# include <cassert>
#else
# include <stdexcept>
#endif

// variant-lite type and visitor argument count configuration (script/generate_h

#define variant_CONFIG_MAX_TYPE_COUNT  16
#define variant_CONFIG_MAX_VISITOR_ARG_COUNT  5

// variant-lite alignment configuration:

#ifndef  variant_CONFIG_MAX_ALIGN_HACK
# define variant_CONFIG_MAX_ALIGN_HACK  0
#endif

#ifndef  variant_CONFIG_ALIGN_AS
// no default, used in #if defined()
#endif

#ifndef  variant_CONFIG_ALIGN_AS_FALLBACK
# define variant_CONFIG_ALIGN_AS_FALLBACK  double
#endif
```

```
// half-open range [lo..hi):
#define variant_BETWEEN( v, lo, hi ) ( (lo) <= (v) && (v) < (hi) )

// Compiler versions:
//
// MSVC++ 6.0  _MSC_VER == 1200 (Visual Studio 6.0)
// MSVC++ 7.0  _MSC_VER == 1300 (Visual Studio .NET 2002)
// MSVC++ 7.1  _MSC_VER == 1310 (Visual Studio .NET 2003)
// MSVC++ 8.0  _MSC_VER == 1400 (Visual Studio 2005)
// MSVC++ 9.0  _MSC_VER == 1500 (Visual Studio 2008)
// MSVC++ 10.0 _MSC_VER == 1600 (Visual Studio 2010)
// MSVC++ 11.0 _MSC_VER == 1700 (Visual Studio 2012)
// MSVC++ 12.0 _MSC_VER == 1800 (Visual Studio 2013)
// MSVC++ 14.0 _MSC_VER == 1900 (Visual Studio 2015)
// MSVC++ 14.1 _MSC_VER >= 1910 (Visual Studio 2017)

#if defined(_MSC_VER ) && !defined(__clang__)
# define variant_COMPILER_MSVC_VER      (_MSC_VER )
# define variant_COMPILER_MSVC_VERSION  (_MSC_VER / 10 - 10 * ( 5 + (_MSC_VER <
#else
# define variant_COMPILER_MSVC_VER      0
# define variant_COMPILER_MSVC_VERSION  0
#endif

#define variant_COMPILER_VERSION( major, minor, patch )  ( 10 * ( 10 * (major) +

#if defined(__clang__)
# define variant_COMPILER_CLANG_VERSION  variant_COMPILER_VERSION(__clang_major_
#else
# define variant_COMPILER_CLANG_VERSION  0
#endif

#if defined(__GNUC__) && !defined(__clang__)
# define variant_COMPILER_GNUC_VERSION  variant_COMPILER_VERSION(__GNUC__, __GNU
#else
# define variant_COMPILER_GNUC_VERSION  0
#endif

#if variant_BETWEEN( variant_COMPILER_MSVC_VER, 1300, 1900 )
# pragma warning( push )
```

```
# pragma warning( disable: 4345 )   // initialization behavior changed
#endif

// Presence of language and library features:

#define variant_HAVE( feature )  ( variant_HAVE_##feature )

#ifdef _HAS_CPP0X
# define variant_HAS_CPP0X  _HAS_CPP0X
#else
# define variant_HAS_CPP0X  0
#endif

// Unless defined otherwise below, consider VC14 as C++11 for variant-lite:

#if variant_COMPILER_MSVC_VER >= 1900
# undef  variant_CPP11_OR_GREATER
# define variant_CPP11_OR_GREATER  1
#endif

#define variant_CPP11_90   (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V
#define variant_CPP11_100  (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V
#define variant_CPP11_110  (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V
#define variant_CPP11_120  (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V
#define variant_CPP11_140  (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V
#define variant_CPP11_141  (variant_CPP11_OR_GREATER_ || variant_COMPILER_MSVC_V

#define variant_CPP14_000  (variant_CPP14_OR_GREATER)
#define variant_CPP17_000  (variant_CPP17_OR_GREATER)

// Presence of C++11 language features:

#define variant_HAVE_CONSTEXPR_11       variant_CPP11_140
#define variant_HAVE_INITIALIZER_LIST   variant_CPP11_120
#define variant_HAVE_NOEXCEPT           variant_CPP11_140
#define variant_HAVE_NULLPTR            variant_CPP11_100
#define variant_HAVE_OVERRIDE           variant_CPP11_140

// Presence of C++14 language features:

#define variant_HAVE_CONSTEXPR_14       variant_CPP14_000
```

```
// Presence of C++17 language features:

// no flag

// Presence of C++ library features:

#define variant_HAVE_CONDITIONAL        variant_CPP11_120
#define variant_HAVE_REMOVE_CV          variant_CPP11_120
#define variant_HAVE_STD_ADD_POINTER    variant_CPP11_90
#define variant_HAVE_TYPE_TRAITS        variant_CPP11_90

#define variant_HAVE_TR1_TYPE_TRAITS    (!! variant_COMPILER_GNUC_VERSION )
#define variant_HAVE_TR1_ADD_POINTER    (!! variant_COMPILER_GNUC_VERSION )

// C++ feature usage:

#if variant_HAVE_CONSTEXPR_11
# define variant_constexpr constexpr
#else
# define variant_constexpr /*constexpr*/
#endif

#if variant_HAVE_CONSTEXPR_14
# define variant_constexpr14 constexpr
#else
# define variant_constexpr14 /*constexpr*/
#endif

#if variant_HAVE_NOEXCEPT
# define variant_noexcept noexcept
#else
# define variant_noexcept /*noexcept*/
#endif

#if variant_HAVE_NULLPTR
# define variant_nullptr nullptr
#else
# define variant_nullptr NULL
#endif
```

```
#if variant_HAVE_OVERRIDE
# define variant_override override
#else
# define variant_override /*override*/
#endif

// additional includes:

#if variant_CPP11_OR_GREATER
# include <functional>      // std::hash
#endif

#if variant_HAVE_INITIALIZER_LIST
# include <initializer_list>
#endif

#if variant_HAVE_TYPE_TRAITS
# include <type_traits>
#elif variant_HAVE_TR1_TYPE_TRAITS
# include <tr1/type_traits>
#endif

// Method enabling

#if variant_CPP11_OR_GREATER

#define variant_REQUIRES_0(...) \
    template< bool B = (__VA_ARGS__), typename std::enable_if<B, int>::type = 0

#define variant_REQUIRES_T(...) \
    , typename = typename std::enable_if< (__VA_ARGS__), nonstd::variants::detai

#define variant_REQUIRES_R(R, ...) \
    typename std::enable_if< (__VA_ARGS__), R>::type

#define variant_REQUIRES_A(...) \
    , typename std::enable_if< (__VA_ARGS__), void*>::type = nullptr

#endif

//
```

```
// variant:
//

namespace nonstd { namespace variants {

// C++11 emulation:

namespace std11 {

#if variant_HAVE_STD_ADD_POINTER

using std::add_pointer;

#elif variant_HAVE_TR1_ADD_POINTER

using std::tr1::add_pointer;

#else

template< class T > struct remove_reference     { typedef T type; };
template< class T > struct remove_reference<T&> { typedef T type; };

template< class T > struct add_pointer
{
    typedef typename remove_reference<T>::type * type;
};

#endif // variant_HAVE_STD_ADD_POINTER

#if variant_HAVE_REMOVE_CV

using std::remove_cv;

#else

template< class T > struct remove_const          { typedef T type; };
template< class T > struct remove_const<const T> { typedef T type; };

template< class T > struct remove_volatile             { typedef T type; };
template< class T > struct remove_volatile<volatile T> { typedef T type; };
```

```
template< class T >
struct remove_cv
{
    typedef typename remove_volatile<typename remove_const<T>::type>::type type;
};

#endif // variant_HAVE_REMOVE_CV

#if variant_HAVE_CONDITIONAL

using std::conditional;

#else

template< bool Cond, class Then, class Else >
struct conditional;

template< class Then, class Else >
struct conditional< true , Then, Else > { typedef Then type; };

template< class Then, class Else >
struct conditional< false, Then, Else > { typedef Else type; };

#endif // variant_HAVE_CONDITIONAL

} // namespace std11

/// type traits C++17:

namespace std17 {

#if variant_CPP17_OR_GREATER

using std::is_swappable;
using std::is_nothrow_swappable;

#elif variant_CPP11_OR_GREATER

namespace detail {

using std::swap;
```

```
struct is_swappable
{
    template< typename T, typename = decltype( swap( std::declval<T&>(), std::de
    static std::true_type test( int );

    template< typename >
    static std::false_type test(...);
};

struct is_nothrow_swappable
{
    // wrap noexcept(epr) in separate function as work-around for VC140 (VS2015)

    template< typename T >
    static constexpr bool test()
    {
        return noexcept( swap( std::declval<T&>(), std::declval<T&>() ) );
    }

    template< typename T >
    static auto test( int ) -> std::integral_constant<bool, test<T>()>{}

    template< typename >
    static std::false_type test(...);
};

} // namespace detail

// is [nothow] swappable:

template< typename T >
struct is_swappable : decltype( detail::is_swappable::test<T>(0) ){};

template< typename T >
struct is_nothrow_swappable : decltype( detail::is_nothrow_swappable::test<T>(0)

#endif // variant_CPP17_OR_GREATER

} // namespace std17
```

```
// detail:

namespace detail {

// for variant_REQUIRES_T():

/*enum*/ class enabler{};

// typelist:

#define variant_TL1( T1 ) detail::typelist< T1, detail::nulltype >
#define variant_TL2( T1, T2) detail::typelist< T1, variant_TL1( T2) >
#define variant_TL3( T1, T2, T3) detail::typelist< T1, variant_TL2( T2, T3) >
#define variant_TL4( T1, T2, T3, T4) detail::typelist< T1, variant_TL3( T2, T3,
#define variant_TL5( T1, T2, T3, T4, T5) detail::typelist< T1, variant_TL4( T2,
#define variant_TL6( T1, T2, T3, T4, T5, T6) detail::typelist< T1, variant_TL5(
#define variant_TL7( T1, T2, T3, T4, T5, T6, T7) detail::typelist< T1, variant_T
#define variant_TL8( T1, T2, T3, T4, T5, T6, T7, T8) detail::typelist< T1, varia
#define variant_TL9( T1, T2, T3, T4, T5, T6, T7, T8, T9) detail::typelist< T1, v
#define variant_TL10( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10) detail::typelist<
#define variant_TL11( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11) detail::type
#define variant_TL12( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12) detail:
#define variant_TL13( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13) de
#define variant_TL14( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T1
#define variant_TL15( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T1
#define variant_TL16( T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T1


// variant parameter unused type tags:

template< class T >
struct TX : T
{
                        inline TX<T> operator+ (           ) const { return TX<T
                        inline TX<T> operator- (           ) const { return TX<T

                        inline TX<T> operator! (           ) const { return TX<T
                        inline TX<T> operator~ (           ) const { return TX<T

                        inline TX<T>*operator& (           ) const { return vari
```

```
    template< class U > inline TX<T> operator* ( U const & ) const { return TX<T
    template< class U > inline TX<T> operator/ ( U const & ) const { return TX<T

    template< class U > inline TX<T> operator% ( U const & ) const { return TX<T
    template< class U > inline TX<T> operator+ ( U const & ) const { return TX<T
    template< class U > inline TX<T> operator- ( U const & ) const { return TX<T

    template< class U > inline TX<T> operator<<( U const & ) const { return TX<T
    template< class U > inline TX<T> operator>>( U const & ) const { return TX<T

                        inline bool  operator==( T const & ) const { return fals
                        inline bool  operator< ( T const & ) const { return fals

    template< class U > inline TX<T> operator& ( U const & ) const { return TX<T
    template< class U > inline TX<T> operator| ( U const & ) const { return TX<T
    template< class U > inline TX<T> operator^ ( U const & ) const { return TX<T

    template< class U > inline TX<T> operator&&( U const & ) const { return TX<T
    template< class U > inline TX<T> operator||( U const & ) const { return TX<T
};

struct S0{}; typedef TX<S0> T0;
struct S1{}; typedef TX<S1> T1;
struct S2{}; typedef TX<S2> T2;
struct S3{}; typedef TX<S3> T3;
struct S4{}; typedef TX<S4> T4;
struct S5{}; typedef TX<S5> T5;
struct S6{}; typedef TX<S6> T6;
struct S7{}; typedef TX<S7> T7;
struct S8{}; typedef TX<S8> T8;
struct S9{}; typedef TX<S9> T9;
struct S10{}; typedef TX<S10> T10;
struct S11{}; typedef TX<S11> T11;
struct S12{}; typedef TX<S12> T12;
struct S13{}; typedef TX<S13> T13;
struct S14{}; typedef TX<S14> T14;
struct S15{}; typedef TX<S15> T15;


struct nulltype{};
```

```
template< class Head, class Tail >
struct typelist
{
    typedef Head head;
    typedef Tail tail;
};

// typelist max element size:

template< class List >
struct typelist_max;

template<>
struct typelist_max< nulltype >
{
    enum V { value = 0 } ;
    typedef void type;
};

template< class Head, class Tail >
struct typelist_max< typelist<Head, Tail> >
{
private:
    enum TV { tail_value = size_t( typelist_max<Tail>::value ) };

    typedef typename typelist_max<Tail>::type tail_type;

public:
    enum V { value = (sizeof( Head ) > tail_value) ? sizeof( Head ) : std::size_

    typedef typename std11::conditional< (sizeof( Head ) > tail_value), Head, ta
};

#if variant_CPP11_OR_GREATER

// typelist max alignof element type:

template< class List >
struct typelist_max_alignof;

template<>
```

```
struct typelist_max_alignof< nulltype >
{
    enum V { value = 0 } ;
};

template< class Head, class Tail >
struct typelist_max_alignof< typelist<Head, Tail> >
{
private:
    enum TV { tail_value = size_t( typelist_max_alignof<Tail>::value ) };

public:
    enum V { value = (alignof( Head ) > tail_value) ? alignof( Head ) : std::siz
};

#endif

// typelist size (length):

template< class List >
struct typelist_size
{
   enum V { value = 1 };
};

template<> struct typelist_size< T0 > { enum V { value = 0 }; };
template<> struct typelist_size< T1 > { enum V { value = 0 }; };
template<> struct typelist_size< T2 > { enum V { value = 0 }; };
template<> struct typelist_size< T3 > { enum V { value = 0 }; };
template<> struct typelist_size< T4 > { enum V { value = 0 }; };
template<> struct typelist_size< T5 > { enum V { value = 0 }; };
template<> struct typelist_size< T6 > { enum V { value = 0 }; };
template<> struct typelist_size< T7 > { enum V { value = 0 }; };
template<> struct typelist_size< T8 > { enum V { value = 0 }; };
template<> struct typelist_size< T9 > { enum V { value = 0 }; };
template<> struct typelist_size< T10 > { enum V { value = 0 }; };
template<> struct typelist_size< T11 > { enum V { value = 0 }; };
template<> struct typelist_size< T12 > { enum V { value = 0 }; };
template<> struct typelist_size< T13 > { enum V { value = 0 }; };
template<> struct typelist_size< T14 > { enum V { value = 0 }; };
template<> struct typelist_size< T15 > { enum V { value = 0 }; };
```

```
template<> struct typelist_size< nulltype > { enum V { value = 0 } ; };

template< class Head, class Tail >
struct typelist_size< typelist<Head, Tail> >
{
    enum V { value = typelist_size<Head>::value + typelist_size<Tail>::value };
};

// typelist index of type:

template< class List, class T >
struct typelist_index_of;

template< class T >
struct typelist_index_of< nulltype, T >
{
    enum V { value = -1 };
};

template< class Tail, class T >
struct typelist_index_of< typelist<T, Tail>, T >
{
    enum V { value = 0 };
};

template< class Head, class Tail, class T >
struct typelist_index_of< typelist<Head, Tail>, T >
{
private:
    enum TV { nextVal = typelist_index_of<Tail, T>::value };

public:
    enum V { value = nextVal == -1 ? -1 : 1 + nextVal } ;
};

// typelist type at index:

template< class List, std::size_t i>
struct typelist_type_at;
```

```
template< class Head, class Tail >
struct typelist_type_at< typelist<Head, Tail>, 0 >
{
    typedef Head type;
};

template< class Head, class Tail, std::size_t i >
struct typelist_type_at< typelist<Head, Tail>, i >
{
    typedef typename typelist_type_at<Tail, i - 1>::type type;
};

#if variant_CONFIG_MAX_ALIGN_HACK

// Max align, use most restricted type for alignment:

#define variant_UNIQUE(  name )       variant_UNIQUE2( name, __LINE__ )
#define variant_UNIQUE2( name, line ) variant_UNIQUE3( name, line )
#define variant_UNIQUE3( name, line ) name ## line

#define variant_ALIGN_TYPE( type ) \
    type variant_UNIQUE( _t ); struct_t< type > variant_UNIQUE( _st )

template< class T >
struct struct_t { T _; };

union max_align_t
{
    variant_ALIGN_TYPE( char );
    variant_ALIGN_TYPE( short int );
    variant_ALIGN_TYPE( int );
    variant_ALIGN_TYPE( long int  );
    variant_ALIGN_TYPE( float  );
    variant_ALIGN_TYPE( double );
    variant_ALIGN_TYPE( long double );
    variant_ALIGN_TYPE( char * );
    variant_ALIGN_TYPE( short int * );
    variant_ALIGN_TYPE( int *  );
    variant_ALIGN_TYPE( long int * );
    variant_ALIGN_TYPE( float * );
```

```
    variant_ALIGN_TYPE( double * );
    variant_ALIGN_TYPE( long double * );
    variant_ALIGN_TYPE( void * );

#ifdef HAVE_LONG_LONG
    variant_ALIGN_TYPE( long long );
#endif

    struct Unknown;

    Unknown ( * variant_UNIQUE(_) )( Unknown );
    Unknown * Unknown::* variant_UNIQUE(_);
    Unknown ( Unknown::* variant_UNIQUE(_) )( Unknown );

    struct_t< Unknown ( * )( Unknown)         > variant_UNIQUE(_);
    struct_t< Unknown * Unknown::*            > variant_UNIQUE(_);
    struct_t< Unknown ( Unknown::* )(Unknown) > variant_UNIQUE(_);
};

#undef variant_UNIQUE
#undef variant_UNIQUE2
#undef variant_UNIQUE3

#undef variant_ALIGN_TYPE

#elif defined( variant_CONFIG_ALIGN_AS ) // variant_CONFIG_MAX_ALIGN_HACK

// Use user-specified type for alignment:

#define variant_ALIGN_AS( unused ) \
    variant_CONFIG_ALIGN_AS

#else // variant_CONFIG_MAX_ALIGN_HACK

// Determine POD type to use for alignment:

#define variant_ALIGN_AS( to_align ) \
    typename detail::type_of_size< detail::alignment_types, detail::alignment_of

template< typename T >
struct alignment_of;
```

```
template< typename T >
struct alignment_of_hack
{
    char c;
    T t;
    alignment_of_hack();
};

template< size_t A, size_t S >
struct alignment_logic
{
    enum V { value = A < S ? A : S };
};

template< typename T >
struct alignment_of
{
    enum V { value = alignment_logic<
        sizeof( alignment_of_hack<T> ) - sizeof(T), sizeof(T) >::value };
};

template< typename List, size_t N >
struct type_of_size
{
    typedef typename std11::conditional<
        N == sizeof( typename List::head ),
            typename List::head,
            typename type_of_size<typename List::tail, N >::type >::type type;
};

template< size_t N >
struct type_of_size< nulltype, N >
{
    typedef variant_CONFIG_ALIGN_AS_FALLBACK type;
};

template< typename T>
struct struct_t { T _; };

#define variant_ALIGN_TYPE( type ) \
```

```
    typelist< type , typelist< struct_t< type >

struct Unknown;

typedef
    variant_ALIGN_TYPE( char ),
    variant_ALIGN_TYPE( short ),
    variant_ALIGN_TYPE( int ),
    variant_ALIGN_TYPE( long ),
    variant_ALIGN_TYPE( float ),
    variant_ALIGN_TYPE( double ),
    variant_ALIGN_TYPE( long double ),

    variant_ALIGN_TYPE( char *),
    variant_ALIGN_TYPE( short * ),
    variant_ALIGN_TYPE( int * ),
    variant_ALIGN_TYPE( long * ),
    variant_ALIGN_TYPE( float * ),
    variant_ALIGN_TYPE( double * ),
    variant_ALIGN_TYPE( long double * ),

    variant_ALIGN_TYPE( Unknown ( * )( Unknown ) ),
    variant_ALIGN_TYPE( Unknown * Unknown::*     ),
    variant_ALIGN_TYPE( Unknown ( Unknown::* )( Unknown ) ),

    nulltype
    > > > > > > >    > > > > > > >
    > > > > > > >    > > > > > > >
    > > > > > >
    alignment_types;

#undef variant_ALIGN_TYPE

#endif // variant_CONFIG_MAX_ALIGN_HACK

#if variant_CPP11_OR_GREATER

template< typename T>
inline std::size_t hash( T const & v )
{
    return std::hash<T>()( v );
```

```
}

inline std::size_t hash( T0 const & ) { return 0; }
inline std::size_t hash( T1 const & ) { return 0; }
inline std::size_t hash( T2 const & ) { return 0; }
inline std::size_t hash( T3 const & ) { return 0; }
inline std::size_t hash( T4 const & ) { return 0; }
inline std::size_t hash( T5 const & ) { return 0; }
inline std::size_t hash( T6 const & ) { return 0; }
inline std::size_t hash( T7 const & ) { return 0; }
inline std::size_t hash( T8 const & ) { return 0; }
inline std::size_t hash( T9 const & ) { return 0; }
inline std::size_t hash( T10 const & ) { return 0; }
inline std::size_t hash( T11 const & ) { return 0; }
inline std::size_t hash( T12 const & ) { return 0; }
inline std::size_t hash( T13 const & ) { return 0; }
inline std::size_t hash( T14 const & ) { return 0; }
inline std::size_t hash( T15 const & ) { return 0; }


#endif // variant_CPP11_OR_GREATER




template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
struct helper
{
    typedef signed char type_index_t;
    typedef variant_TL16( T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,

    template< class U >
    static U * as( void * data )
    {
        return reinterpret_cast<U*>( data );
    }

    template< class U >
    static U const * as( void const * data )
    {
```

```
        return reinterpret_cast<const U*>( data );
    }

    static type_index_t to_index_t( std::size_t index )
    {
        return static_cast<type_index_t>( index );
    }

    static void destroy( type_index_t index, void * data )
    {
        switch ( index )
        {
        case 0: as<T0>( data )->~T0(); break;
        case 1: as<T1>( data )->~T1(); break;
        case 2: as<T2>( data )->~T2(); break;
        case 3: as<T3>( data )->~T3(); break;
        case 4: as<T4>( data )->~T4(); break;
        case 5: as<T5>( data )->~T5(); break;
        case 6: as<T6>( data )->~T6(); break;
        case 7: as<T7>( data )->~T7(); break;
        case 8: as<T8>( data )->~T8(); break;
        case 9: as<T9>( data )->~T9(); break;
        case 10: as<T10>( data )->~T10(); break;
        case 11: as<T11>( data )->~T11(); break;
        case 12: as<T12>( data )->~T12(); break;
        case 13: as<T13>( data )->~T13(); break;
        case 14: as<T14>( data )->~T14(); break;
        case 15: as<T15>( data )->~T15(); break;

        }
    }

#if variant_CPP11_OR_GREATER
    template< class T, class... Args >
    static type_index_t construct_t( void * data, Args&&... args )
    {
        new( data ) T( std::forward<Args>(args)... );

        return to_index_t( detail::typelist_index_of< variant_types, T>::value )
    }
```

```
template< std::size_t K, class... Args >
static type_index_t construct_i( void * data, Args&&... args )
{
    using type = typename detail::typelist_type_at< variant_types, K >::type

    construct_t< type >( data, std::forward<Args>(args)... );

    return to_index_t( K );
}

static type_index_t move_construct( type_index_t const from_index, void * fr
{
    switch ( from_index )
    {
    case 0: new( to_value ) T0( std::move( *as<T0>( from_value ) ) ); break;
    case 1: new( to_value ) T1( std::move( *as<T1>( from_value ) ) ); break;
    case 2: new( to_value ) T2( std::move( *as<T2>( from_value ) ) ); break;
    case 3: new( to_value ) T3( std::move( *as<T3>( from_value ) ) ); break;
    case 4: new( to_value ) T4( std::move( *as<T4>( from_value ) ) ); break;
    case 5: new( to_value ) T5( std::move( *as<T5>( from_value ) ) ); break;
    case 6: new( to_value ) T6( std::move( *as<T6>( from_value ) ) ); break;
    case 7: new( to_value ) T7( std::move( *as<T7>( from_value ) ) ); break;
    case 8: new( to_value ) T8( std::move( *as<T8>( from_value ) ) ); break;
    case 9: new( to_value ) T9( std::move( *as<T9>( from_value ) ) ); break;
    case 10: new( to_value ) T10( std::move( *as<T10>( from_value ) ) ); bre
    case 11: new( to_value ) T11( std::move( *as<T11>( from_value ) ) ); bre
    case 12: new( to_value ) T12( std::move( *as<T12>( from_value ) ) ); bre
    case 13: new( to_value ) T13( std::move( *as<T13>( from_value ) ) ); bre
    case 14: new( to_value ) T14( std::move( *as<T14>( from_value ) ) ); bre
    case 15: new( to_value ) T15( std::move( *as<T15>( from_value ) ) ); bre

    }
    return from_index;
}

static type_index_t move_assign( type_index_t const from_index, void * from_
{
    switch ( from_index )
    {
    case 0: *as<T0>( to_value ) = std::move( *as<T0>( from_value ) ); break;
    case 1: *as<T1>( to_value ) = std::move( *as<T1>( from_value ) ); break;
```

```
        case 2: *as<T2>( to_value ) = std::move( *as<T2>( from_value ) ); break;
        case 3: *as<T3>( to_value ) = std::move( *as<T3>( from_value ) ); break;
        case 4: *as<T4>( to_value ) = std::move( *as<T4>( from_value ) ); break;
        case 5: *as<T5>( to_value ) = std::move( *as<T5>( from_value ) ); break;
        case 6: *as<T6>( to_value ) = std::move( *as<T6>( from_value ) ); break;
        case 7: *as<T7>( to_value ) = std::move( *as<T7>( from_value ) ); break;
        case 8: *as<T8>( to_value ) = std::move( *as<T8>( from_value ) ); break;
        case 9: *as<T9>( to_value ) = std::move( *as<T9>( from_value ) ); break;
        case 10: *as<T10>( to_value ) = std::move( *as<T10>( from_value ) ); bre
        case 11: *as<T11>( to_value ) = std::move( *as<T11>( from_value ) ); bre
        case 12: *as<T12>( to_value ) = std::move( *as<T12>( from_value ) ); bre
        case 13: *as<T13>( to_value ) = std::move( *as<T13>( from_value ) ); bre
        case 14: *as<T14>( to_value ) = std::move( *as<T14>( from_value ) ); bre
        case 15: *as<T15>( to_value ) = std::move( *as<T15>( from_value ) ); bre

        }
        return from_index;
    }
#endif

    static type_index_t copy_construct( type_index_t const from_index, const voi
    {
        switch ( from_index )
        {
        case 0: new( to_value ) T0( *as<T0>( from_value ) ); break;
        case 1: new( to_value ) T1( *as<T1>( from_value ) ); break;
        case 2: new( to_value ) T2( *as<T2>( from_value ) ); break;
        case 3: new( to_value ) T3( *as<T3>( from_value ) ); break;
        case 4: new( to_value ) T4( *as<T4>( from_value ) ); break;
        case 5: new( to_value ) T5( *as<T5>( from_value ) ); break;
        case 6: new( to_value ) T6( *as<T6>( from_value ) ); break;
        case 7: new( to_value ) T7( *as<T7>( from_value ) ); break;
        case 8: new( to_value ) T8( *as<T8>( from_value ) ); break;
        case 9: new( to_value ) T9( *as<T9>( from_value ) ); break;
        case 10: new( to_value ) T10( *as<T10>( from_value ) ); break;
        case 11: new( to_value ) T11( *as<T11>( from_value ) ); break;
        case 12: new( to_value ) T12( *as<T12>( from_value ) ); break;
        case 13: new( to_value ) T13( *as<T13>( from_value ) ); break;
        case 14: new( to_value ) T14( *as<T14>( from_value ) ); break;
        case 15: new( to_value ) T15( *as<T15>( from_value ) ); break;
```

```
        }
        return from_index;
    }

    static type_index_t copy_assign( type_index_t const from_index, const void *
    {
        switch ( from_index )
        {
        case 0: *as<T0>( to_value ) = *as<T0>( from_value ); break;
        case 1: *as<T1>( to_value ) = *as<T1>( from_value ); break;
        case 2: *as<T2>( to_value ) = *as<T2>( from_value ); break;
        case 3: *as<T3>( to_value ) = *as<T3>( from_value ); break;
        case 4: *as<T4>( to_value ) = *as<T4>( from_value ); break;
        case 5: *as<T5>( to_value ) = *as<T5>( from_value ); break;
        case 6: *as<T6>( to_value ) = *as<T6>( from_value ); break;
        case 7: *as<T7>( to_value ) = *as<T7>( from_value ); break;
        case 8: *as<T8>( to_value ) = *as<T8>( from_value ); break;
        case 9: *as<T9>( to_value ) = *as<T9>( from_value ); break;
        case 10: *as<T10>( to_value ) = *as<T10>( from_value ); break;
        case 11: *as<T11>( to_value ) = *as<T11>( from_value ); break;
        case 12: *as<T12>( to_value ) = *as<T12>( from_value ); break;
        case 13: *as<T13>( to_value ) = *as<T13>( from_value ); break;
        case 14: *as<T14>( to_value ) = *as<T14>( from_value ); break;
        case 15: *as<T15>( to_value ) = *as<T15>( from_value ); break;

        }
        return from_index;
    }
};

} // namespace detail

//
// Variant:
//

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
class variant;

// 19.7.8 Class monostate
```

```
class monostate{};

// 19.7.9 monostate relational operators

inline variant_constexpr bool operator< ( monostate, monostate ) variant_noexcep
inline variant_constexpr bool operator> ( monostate, monostate ) variant_noexcep
inline variant_constexpr bool operator<=( monostate, monostate ) variant_noexcep
inline variant_constexpr bool operator>=( monostate, monostate ) variant_noexcep
inline variant_constexpr bool operator==( monostate, monostate ) variant_noexcep
inline variant_constexpr bool operator!=( monostate, monostate ) variant_noexcep

// 19.7.4 variant helper classes

// obtain the size of the variant's list of alternatives at compile time

template< class T >
struct variant_size; /* undefined */

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
struct variant_size< variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T
{
    enum _ { value = detail::typelist_size< variant_TL16(T0, T1, T2, T3, T4, T5,
};

#if variant_CPP14_OR_GREATER
template< class T >
constexpr std::size_t variant_size_v = variant_size<T>::value;
#endif

#if ! variant_CONFIG_OMIT_VARIANT_SIZE_V_MACRO
# define variant_size_V(T)  nonstd::variant_size<T>::value
#endif

// obtain the type of the alternative specified by its index, at compile time:

template< std::size_t K, class T >
struct variant_alternative; /* undefined */

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
struct variant_alternative< K, variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T
{
```

```
    typedef typename detail::typelist_type_at<variant_TL16(T0, T1, T2, T3, T4, T
};

#if variant_CPP11_OR_GREATER
template< std::size_t K, class T >
using variant_alternative_t = typename variant_alternative<K, T>::type;
#endif

#if ! variant_CONFIG_OMIT_VARIANT_ALTERNATIVE_T_MACRO
# define variant_alternative_T(K,T)  typename nonstd::variant_alternative<K,T >:
#endif

// NTS:implement specializes the std::uses_allocator type trait
// std::uses_allocator<nonstd::variant>

// index of the variant in the invalid state (constant)

#if variant_CPP11_OR_GREATER
variant_constexpr std::size_t variant_npos = static_cast<std::size_t>( -1 );
#else
static const std::size_t variant_npos = static_cast<std::size_t>( -1 );
#endif

#if ! variant_CONFIG_NO_EXCEPTIONS

// 19.7.11 Class bad_variant_access

class bad_variant_access : public std::exception
{
public:
#if variant_CPP11_OR_GREATER
    virtual const char* what() const variant_noexcept variant_override
#else
    virtual const char* what() const throw()
#endif
    {
        return "bad variant access";
    }
};

#endif // variant_CONFIG_NO_EXCEPTIONS
```

```
// 19.7.3 Class template variant

template<
    class T0,
    class T1 = detail::T1,
    class T2 = detail::T2,
    class T3 = detail::T3,
    class T4 = detail::T4,
    class T5 = detail::T5,
    class T6 = detail::T6,
    class T7 = detail::T7,
    class T8 = detail::T8,
    class T9 = detail::T9,
    class T10 = detail::T10,
    class T11 = detail::T11,
    class T12 = detail::T12,
    class T13 = detail::T13,
    class T14 = detail::T14,
    class T15 = detail::T15
    >
class variant
{
    typedef detail::helper< T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T1
    typedef variant_TL16( T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,

public:
    // 19.7.3.1 Constructors

    variant() : type_index( 0 ) { new( ptr() ) T0(); }

    variant( T0 const & t0 ) : type_index( 0 ) { new( ptr() ) T0( t0 ); }
    variant( T1 const & t1 ) : type_index( 1 ) { new( ptr() ) T1( t1 ); }
    variant( T2 const & t2 ) : type_index( 2 ) { new( ptr() ) T2( t2 ); }
    variant( T3 const & t3 ) : type_index( 3 ) { new( ptr() ) T3( t3 ); }
    variant( T4 const & t4 ) : type_index( 4 ) { new( ptr() ) T4( t4 ); }
    variant( T5 const & t5 ) : type_index( 5 ) { new( ptr() ) T5( t5 ); }
    variant( T6 const & t6 ) : type_index( 6 ) { new( ptr() ) T6( t6 ); }
    variant( T7 const & t7 ) : type_index( 7 ) { new( ptr() ) T7( t7 ); }
    variant( T8 const & t8 ) : type_index( 8 ) { new( ptr() ) T8( t8 ); }
    variant( T9 const & t9 ) : type_index( 9 ) { new( ptr() ) T9( t9 ); }
```

```
    variant( T10 const & t10 ) : type_index( 10 ) { new( ptr() ) T10( t10 ); }
    variant( T11 const & t11 ) : type_index( 11 ) { new( ptr() ) T11( t11 ); }
    variant( T12 const & t12 ) : type_index( 12 ) { new( ptr() ) T12( t12 ); }
    variant( T13 const & t13 ) : type_index( 13 ) { new( ptr() ) T13( t13 ); }
    variant( T14 const & t14 ) : type_index( 14 ) { new( ptr() ) T14( t14 ); }
    variant( T15 const & t15 ) : type_index( 15 ) { new( ptr() ) T15( t15 ); }


#if variant_CPP11_OR_GREATER
    variant( T0 && t0 ) : type_index( 0 ) { new( ptr() ) T0( std::move(t0) ); }
    variant( T1 && t1 ) : type_index( 1 ) { new( ptr() ) T1( std::move(t1) ); }
    variant( T2 && t2 ) : type_index( 2 ) { new( ptr() ) T2( std::move(t2) ); }
    variant( T3 && t3 ) : type_index( 3 ) { new( ptr() ) T3( std::move(t3) ); }
    variant( T4 && t4 ) : type_index( 4 ) { new( ptr() ) T4( std::move(t4) ); }
    variant( T5 && t5 ) : type_index( 5 ) { new( ptr() ) T5( std::move(t5) ); }
    variant( T6 && t6 ) : type_index( 6 ) { new( ptr() ) T6( std::move(t6) ); }
    variant( T7 && t7 ) : type_index( 7 ) { new( ptr() ) T7( std::move(t7) ); }
    variant( T8 && t8 ) : type_index( 8 ) { new( ptr() ) T8( std::move(t8) ); }
    variant( T9 && t9 ) : type_index( 9 ) { new( ptr() ) T9( std::move(t9) ); }
    variant( T10 && t10 ) : type_index( 10 ) { new( ptr() ) T10( std::move(t10)
    variant( T11 && t11 ) : type_index( 11 ) { new( ptr() ) T11( std::move(t11)
    variant( T12 && t12 ) : type_index( 12 ) { new( ptr() ) T12( std::move(t12)
    variant( T13 && t13 ) : type_index( 13 ) { new( ptr() ) T13( std::move(t13)
    variant( T14 && t14 ) : type_index( 14 ) { new( ptr() ) T14( std::move(t14)
    variant( T15 && t15 ) : type_index( 15 ) { new( ptr() ) T15( std::move(t15)

#endif

    variant(variant const & other)
    : type_index( other.type_index )
    {
        (void) helper_type::copy_construct( other.type_index, other.ptr(), ptr()
    }

#if variant_CPP11_OR_GREATER

    variant( variant && other ) noexcept(
        std::is_nothrow_move_constructible<T0>::value &&
        std::is_nothrow_move_constructible<T1>::value &&
        std::is_nothrow_move_constructible<T2>::value &&
        std::is_nothrow_move_constructible<T3>::value &&
```

```
    std::is_nothrow_move_constructible<T4>::value &&
    std::is_nothrow_move_constructible<T5>::value &&
    std::is_nothrow_move_constructible<T6>::value &&
    std::is_nothrow_move_constructible<T7>::value &&
    std::is_nothrow_move_constructible<T8>::value &&
    std::is_nothrow_move_constructible<T9>::value &&
    std::is_nothrow_move_constructible<T10>::value &&
    std::is_nothrow_move_constructible<T11>::value &&
    std::is_nothrow_move_constructible<T12>::value &&
    std::is_nothrow_move_constructible<T13>::value &&
    std::is_nothrow_move_constructible<T14>::value &&
    std::is_nothrow_move_constructible<T15>::value)
    : type_index( other.type_index )
{
    (void) helper_type::move_construct( other.type_index, other.ptr(), ptr()
}

template< std::size_t K >
using type_at_t = typename detail::typelist_type_at< variant_types, K >::typ

template< class T, class... Args
    variant_REQUIRES_T( std::is_constructible< T, Args...>::value )
>
explicit variant( nonstd_lite_in_place_type_t(T), Args&&... args)
{
    type_index = variant_npos_internal();
    type_index = helper_type::template construct_t<T>( ptr(), std::forward<A
}

template< class T, class U, class... Args
    variant_REQUIRES_T( std::is_constructible< T, std::initializer_list<U>&,
>
explicit variant( nonstd_lite_in_place_type_t(T), std::initializer_list<U> i
{
    type_index = variant_npos_internal();
    type_index = helper_type::template construct_t<T>( ptr(), il, std::forwa
}

template< std::size_t K, class... Args
    variant_REQUIRES_T( std::is_constructible< type_at_t<K>, Args...>::value
>
```

```
    explicit variant( nonstd_lite_in_place_index_t(K), Args&&... args )
    {
        type_index = variant_npos_internal();
        type_index = helper_type::template construct_i<K>( ptr(), std::forward<A
    }

    template< size_t K, class U, class... Args
        variant_REQUIRES_T( std::is_constructible< type_at_t<K>, std::initialize
    >
    explicit variant( nonstd_lite_in_place_index_t(K), std::initializer_list<U>
    {
        type_index = variant_npos_internal();
        type_index = helper_type::template construct_i<K>( ptr(), il, std::forwa
    }

#endif // variant_CPP11_OR_GREATER

    // 19.7.3.2 Destructor

    ~variant()
    {
        if ( ! valueless_by_exception() )
        {
            helper_type::destroy( type_index, ptr() );
        }
    }

    // 19.7.3.3 Assignment

    variant & operator=( variant const & other )
    {
        return copy_assign( other );
    }

#if variant_CPP11_OR_GREATER

    variant & operator=( variant && other ) noexcept(
        std::is_nothrow_move_assignable<T0>::value &&
        std::is_nothrow_move_assignable<T1>::value &&
        std::is_nothrow_move_assignable<T2>::value &&
        std::is_nothrow_move_assignable<T3>::value &&
```

```
            std::is_nothrow_move_assignable<T4>::value &&
            std::is_nothrow_move_assignable<T5>::value &&
            std::is_nothrow_move_assignable<T6>::value &&
            std::is_nothrow_move_assignable<T7>::value &&
            std::is_nothrow_move_assignable<T8>::value &&
            std::is_nothrow_move_assignable<T9>::value &&
            std::is_nothrow_move_assignable<T10>::value &&
            std::is_nothrow_move_assignable<T11>::value &&
            std::is_nothrow_move_assignable<T12>::value &&
            std::is_nothrow_move_assignable<T13>::value &&
            std::is_nothrow_move_assignable<T14>::value &&
            std::is_nothrow_move_assignable<T15>::value)
            {
            return move_assign( std::move( other ) );
        }

        variant & operator=( T0 &&      t0 ) { return assign_value<0>( std::move( t0
        variant & operator=( T1 &&      t1 ) { return assign_value<1>( std::move( t1
        variant & operator=( T2 &&      t2 ) { return assign_value<2>( std::move( t2
        variant & operator=( T3 &&      t3 ) { return assign_value<3>( std::move( t3
        variant & operator=( T4 &&      t4 ) { return assign_value<4>( std::move( t4
        variant & operator=( T5 &&      t5 ) { return assign_value<5>( std::move( t5
        variant & operator=( T6 &&      t6 ) { return assign_value<6>( std::move( t6
        variant & operator=( T7 &&      t7 ) { return assign_value<7>( std::move( t7
        variant & operator=( T8 &&      t8 ) { return assign_value<8>( std::move( t8
        variant & operator=( T9 &&      t9 ) { return assign_value<9>( std::move( t9
        variant & operator=( T10 &&      t10 ) { return assign_value<10>( std::move(
        variant & operator=( T11 &&      t11 ) { return assign_value<11>( std::move(
        variant & operator=( T12 &&      t12 ) { return assign_value<12>( std::move(
        variant & operator=( T13 &&      t13 ) { return assign_value<13>( std::move(
        variant & operator=( T14 &&      t14 ) { return assign_value<14>( std::move(
        variant & operator=( T15 &&      t15 ) { return assign_value<15>( std::move(


#endif

        variant & operator=( T0 const & t0 ) { return assign_value<0>( t0 ); }
        variant & operator=( T1 const & t1 ) { return assign_value<1>( t1 ); }
        variant & operator=( T2 const & t2 ) { return assign_value<2>( t2 ); }
        variant & operator=( T3 const & t3 ) { return assign_value<3>( t3 ); }
        variant & operator=( T4 const & t4 ) { return assign_value<4>( t4 ); }
```

```
    variant & operator=( T5 const & t5 ) { return assign_value<5>( t5 ); }
    variant & operator=( T6 const & t6 ) { return assign_value<6>( t6 ); }
    variant & operator=( T7 const & t7 ) { return assign_value<7>( t7 ); }
    variant & operator=( T8 const & t8 ) { return assign_value<8>( t8 ); }
    variant & operator=( T9 const & t9 ) { return assign_value<9>( t9 ); }
    variant & operator=( T10 const & t10 ) { return assign_value<10>( t10 ); }
    variant & operator=( T11 const & t11 ) { return assign_value<11>( t11 ); }
    variant & operator=( T12 const & t12 ) { return assign_value<12>( t12 ); }
    variant & operator=( T13 const & t13 ) { return assign_value<13>( t13 ); }
    variant & operator=( T14 const & t14 ) { return assign_value<14>( t14 ); }
    variant & operator=( T15 const & t15 ) { return assign_value<15>( t15 ); }


    std::size_t index() const
    {
        return variant_npos_internal() == type_index ? variant_npos : static_cas
    }

    // 19.7.3.4 Modifiers

#if variant_CPP11_OR_GREATER
    template< class T, class... Args
        variant_REQUIRES_T( std::is_constructible< T, Args...>::value )
    >
    T& emplace( Args&&... args )
    {
        helper_type::destroy( type_index, ptr() );
        type_index = variant_npos_internal();
        type_index = helper_type::template construct_t<T>( ptr(), std::forward<A

        return *as<T>();
    }

    template< class T, class U, class... Args
        variant_REQUIRES_T( std::is_constructible< T, std::initializer_list<U>&,
    >
    T& emplace( std::initializer_list<U> il, Args&&... args )
    {
        helper_type::destroy( type_index, ptr() );
        type_index = variant_npos_internal();
        type_index = helper_type::template construct_t<T>( ptr(), il, std::forwa
```

```
        return *as<T>();
    }

    template< size_t K, class... Args
        variant_REQUIRES_T( std::is_constructible< type_at_t<K>, Args...>::value
    >
    variant_alternative_t<K, variant> & emplace( Args&&... args )
    {
        return this->template emplace< type_at_t<K> >( std::forward<Args>(args).
    }

    template< size_t K, class U, class... Args
        variant_REQUIRES_T( std::is_constructible< type_at_t<K>, std::initialize
    >
    variant_alternative_t<K, variant> & emplace( std::initializer_list<U> il, Ar
    {
        return this->template emplace< type_at_t<K> >( il, std::forward<Args>(ar
    }

#endif // variant_CPP11_OR_GREATER

    // 19.7.3.5 Value status

    bool valueless_by_exception() const
    {
        return type_index == variant_npos_internal();
    }

    // 19.7.3.6 Swap

    void swap( variant & other )
#if variant_CPP11_OR_GREATER
        noexcept(
            std::is_nothrow_move_constructible<T0>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T1>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T2>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T3>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T4>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T5>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T6>::value && std17::is_nothrow_s
```

```
            std::is_nothrow_move_constructible<T7>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T8>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T9>::value && std17::is_nothrow_s
            std::is_nothrow_move_constructible<T10>::value && std17::is_nothrow_
            std::is_nothrow_move_constructible<T11>::value && std17::is_nothrow_
            std::is_nothrow_move_constructible<T12>::value && std17::is_nothrow_
            std::is_nothrow_move_constructible<T13>::value && std17::is_nothrow_
            std::is_nothrow_move_constructible<T14>::value && std17::is_nothrow_
            std::is_nothrow_move_constructible<T15>::value && std17::is_nothrow_

        )
#endif
    {
        if ( valueless_by_exception() && other.valueless_by_exception() )
        {
            // no effect
        }
        else if ( type_index == other.type_index )
        {
            this->swap_value( type_index, other );
        }
        else
        {
#if variant_CPP11_OR_GREATER
            variant tmp( std::move( *this ) );
            *this = std::move( other );
            other = std::move( tmp );
#else
            variant tmp( *this );
            *this = other;
            other = tmp;
#endif
        }
    }

    //
    // non-standard:
    //

    template< class T >
    static variant_constexpr std::size_t index_of() variant_noexcept
```

```
    {
        return to_size_t( detail::typelist_index_of<variant_types, typename std1
    }

    template< class T >
    T & get()
    {
        const std::size_t i = index_of<T>();

#if variant_CONFIG_NO_EXCEPTIONS
        assert( i == index() );
#else
        if ( i != index() )
        {
            throw bad_variant_access();
        }
#endif
        return *as<T>();
    }

    template< class T >
    T const & get() const
    {
        const std::size_t i = index_of<T>();

#if variant_CONFIG_NO_EXCEPTIONS
        assert( i == index() );
#else
        if ( i != index() )
        {
            throw bad_variant_access();
        }
#endif
        return *as<const T>();
    }

    template< std::size_t K >
    typename variant_alternative< K, variant >::type &
    get()
    {
        return this->template get< typename detail::typelist_type_at< variant_ty
```

```
    }

    template< std::size_t K >
    typename variant_alternative< K, variant >::type const &
    get() const
    {
        return this->template get< typename detail::typelist_type_at< variant_ty
    }

private:
    typedef typename helper_type::type_index_t type_index_t;

    void * ptr() variant_noexcept
    {
        return &data;
    }

    void const * ptr() const variant_noexcept
    {
        return &data;
    }

    template< class U >
    U * as()
    {
        return reinterpret_cast<U*>( ptr() );
    }

    template< class U >
    U const * as() const
    {
        return reinterpret_cast<U const *>( ptr() );
    }

    template< class U >
    static variant_constexpr std::size_t to_size_t( U index )
    {
        return static_cast<std::size_t>( index );
    }

    variant_constexpr type_index_t variant_npos_internal() const variant_noexcep
```

```
    {
        return static_cast<type_index_t>( -1 );
    }

    variant & copy_assign( variant const & other )
    {
        if ( valueless_by_exception() && other.valueless_by_exception() )
        {
            // no effect
        }
        else if ( ! valueless_by_exception() && other.valueless_by_exception() )
        {
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
        }
        else if ( index() == other.index() )
        {
            type_index = helper_type::copy_assign( other.type_index, other.ptr()
        }
        else
        {
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
            type_index = helper_type::copy_construct( other.type_index, other.pt
        }
        return *this;
    }

#if variant_CPP11_OR_GREATER

    variant & move_assign( variant && other )
    {
        if ( valueless_by_exception() && other.valueless_by_exception() )
        {
            // no effect
        }
        else if ( ! valueless_by_exception() && other.valueless_by_exception() )
        {
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
        }
```

```
        else if ( index() == other.index() )
        {
            type_index = helper_type::move_assign( other.type_index, other.ptr()
        }
        else
        {
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
            type_index = helper_type::move_construct( other.type_index, other.pt
        }
        return *this;
    }

    template< std::size_t K, class T >
    variant & assign_value( T && value )
    {
        if( index() == K )
        {
            *as<T>() = std::forward<T>( value );
        }
        else
        {
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
            new( ptr() ) T( std::forward<T>( value ) );
            type_index = K;
        }
        return *this;
    }

#endif // variant_CPP11_OR_GREATER

    template< std::size_t K, class T >
    variant & assign_value( T const & value )
    {
        if( index() == K )
        {
            *as<T>() = value;
        }
        else
        {
```

```
            helper_type::destroy( type_index, ptr() );
            type_index = variant_npos_internal();
            new( ptr() ) T( value );
            type_index = K;
        }
        return *this;
    }

    void swap_value( type_index_t index, variant & other )
    {
        using std::swap;
        switch( index )
        {
            case 0: swap( this->get<0>(), other.get<0>() ); break;
            case 1: swap( this->get<1>(), other.get<1>() ); break;
            case 2: swap( this->get<2>(), other.get<2>() ); break;
            case 3: swap( this->get<3>(), other.get<3>() ); break;
            case 4: swap( this->get<4>(), other.get<4>() ); break;
            case 5: swap( this->get<5>(), other.get<5>() ); break;
            case 6: swap( this->get<6>(), other.get<6>() ); break;
            case 7: swap( this->get<7>(), other.get<7>() ); break;
            case 8: swap( this->get<8>(), other.get<8>() ); break;
            case 9: swap( this->get<9>(), other.get<9>() ); break;
            case 10: swap( this->get<10>(), other.get<10>() ); break;
            case 11: swap( this->get<11>(), other.get<11>() ); break;
            case 12: swap( this->get<12>(), other.get<12>() ); break;
            case 13: swap( this->get<13>(), other.get<13>() ); break;
            case 14: swap( this->get<14>(), other.get<14>() ); break;
            case 15: swap( this->get<15>(), other.get<15>() ); break;

        }
    }

private:
    enum { data_size  = detail::typelist_max< variant_types >::value };

#if variant_CPP11_OR_GREATER

    enum { data_align = detail::typelist_max_alignof< variant_types >::value };

    using aligned_storage_t = typename std::aligned_storage< data_size, data_ali
```

```
    aligned_storage_t data;

#elif variant_CONFIG_MAX_ALIGN_HACK

    typedef union { unsigned char data[ data_size ]; } aligned_storage_t;

    detail::max_align_t hack;
    aligned_storage_t data;

#else
    typedef typename detail::typelist_max< variant_types >::type max_type;

    typedef variant_ALIGN_AS( max_type ) align_as_type;

    typedef union { align_as_type data[ 1 + ( data_size - 1 ) / sizeof(align_as_
    aligned_storage_t data;

// #   undef variant_ALIGN_AS

#endif // variant_CONFIG_MAX_ALIGN_HACK

    type_index_t type_index;
};

// 19.7.5 Value access

template< class T, class T0, class T1, class T2, class T3, class T4, class T5, c
inline bool holds_alternative( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T
{
    return v.index() == variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11
}

template< class R, class T0, class T1, class T2, class T3, class T4, class T5, c
inline R & get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T
{
    return v.template get<R>();
}

template< class R, class T0, class T1, class T2, class T3, class T4, class T5, c
inline R const & get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11,
{
```

```
    return v.template get<R>();
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename variant_alternative< K, variant<T0, T1, T2, T3, T4, T5, T6, T7,
get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T1
{
#if variant_CONFIG_NO_EXCEPTIONS
    assert( K == v.index() );
#else
    if ( K != v.index() )
    {
        throw bad_variant_access();
    }
#endif
    return v.template get<K>();
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename variant_alternative< K, variant<T0, T1, T2, T3, T4, T5, T6, T7,
get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T1
{
#if variant_CONFIG_NO_EXCEPTIONS
    assert( K == v.index() );
#else
    if ( K != v.index() )
    {
        throw bad_variant_access();
    }
#endif
    return v.template get<K>();
}

#if variant_CPP11_OR_GREATER

template< class R, class T0, class T1, class T2, class T3, class T4, class T5, c
inline R && get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12,
{
    return std::move(v.template get<R>());
}
```

```
template< class R, class T0, class T1, class T2, class T3, class T4, class T5, c
inline R const && get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11,
{
    return std::move(v.template get<R>());
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename variant_alternative< K, variant<T0, T1, T2, T3, T4, T5, T6, T7,
get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T1
{
#if variant_CONFIG_NO_EXCEPTIONS
    assert( K == v.index() );
#else
    if ( K != v.index() )
    {
        throw bad_variant_access();
    }
#endif
    return std::move(v.template get<K>());
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename variant_alternative< K, variant<T0, T1, T2, T3, T4, T5, T6, T7,
get( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T1
{
#if variant_CONFIG_NO_EXCEPTIONS
    assert( K == v.index() );
#else
    if ( K != v.index() )
    {
        throw bad_variant_access();
    }
#endif
    return std::move(v.template get<K>());
}

#endif // variant_CPP11_OR_GREATER

template< class T, class T0, class T1, class T2, class T3, class T4, class T5, c
inline typename std11::add_pointer<T>::type
get_if( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
```

```
{
    return ( pv->index() == variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
}

template< class T, class T0, class T1, class T2, class T3, class T4, class T5, c
inline typename std11::add_pointer<const T>::type
get_if( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
{
    return ( pv->index() == variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10,
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename std11::add_pointer< typename variant_alternative<K, variant<T0,
get_if( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
{
    return ( pv->index() == K ) ? &get<K>( *pv ) : variant_nullptr;
}

template< std::size_t K, class T0, class T1, class T2, class T3, class T4, class
inline typename std11::add_pointer< const typename variant_alternative<K, varian
get_if( variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14,
{
    return ( pv->index() == K ) ? &get<K>( *pv )  : variant_nullptr;
}

// 19.7.10 Specialized algorithms

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
#if variant_CPP11_OR_GREATER
    variant_REQUIRES_T(
        std::is_move_constructible<T0>::value && std17::is_swappable<T0>::value
        std::is_move_constructible<T1>::value && std17::is_swappable<T1>::value
        std::is_move_constructible<T2>::value && std17::is_swappable<T2>::value
        std::is_move_constructible<T3>::value && std17::is_swappable<T3>::value
        std::is_move_constructible<T4>::value && std17::is_swappable<T4>::value
        std::is_move_constructible<T5>::value && std17::is_swappable<T5>::value
        std::is_move_constructible<T6>::value && std17::is_swappable<T6>::value
        std::is_move_constructible<T7>::value && std17::is_swappable<T7>::value
        std::is_move_constructible<T8>::value && std17::is_swappable<T8>::value
        std::is_move_constructible<T9>::value && std17::is_swappable<T9>::value
        std::is_move_constructible<T10>::value && std17::is_swappable<T10>::valu
```

```
        std::is_move_constructible<T11>::value && std17::is_swappable<T11>::valu
        std::is_move_constructible<T12>::value && std17::is_swappable<T12>::valu
        std::is_move_constructible<T13>::value && std17::is_swappable<T13>::valu
        std::is_move_constructible<T14>::value && std17::is_swappable<T14>::valu
        std::is_move_constructible<T15>::value && std17::is_swappable<T15>::valu
         )
#endif
>
inline void swap(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
#if variant_CPP11_OR_GREATER
    noexcept( noexcept( a.swap( b ) ) )
#endif
{
    a.swap( b );
}

// 19.7.7 Visitation

// Variant 'visitor' implementation

namespace detail
{

template< typename R, typename VT >
struct VisitorApplicatorImpl
{
    template< typename Visitor, typename T >
    static R apply(Visitor const& v, T const& arg)
    {
        return v(arg);
    }
};

template< typename R, typename VT >
struct VisitorApplicatorImpl<R, TX<VT> >
{
    template< typename Visitor, typename T >
    static R apply(Visitor const&, T)
    {
```

```
        return R();
    }
};

template<typename R>
struct VisitorApplicator;

template< typename R, typename Visitor, typename V1 >
struct VisitorUnwrapper;

#if variant_CPP11_OR_GREATER
template< size_t NumVars, typename R, typename Visitor, typename ... T >
#else
template< size_t NumVars, typename R, typename Visitor, typename T1, typename T2
#endif
struct TypedVisitorUnwrapper;

template< typename R, typename Visitor, typename T2 >
struct TypedVisitorUnwrapper<2, R, Visitor, T2>
{
    const Visitor& visitor;
    T2 const& val2;

    TypedVisitorUnwrapper(const Visitor& visitor_, T2 const& val2_)
        : visitor(visitor_)
        , val2(val2_)

    {
    }

    template<typename T>
    R operator()(const T& val1) const
    {
        return visitor(val1, val2);
    }
};

template< typename R, typename Visitor, typename T2, typename T3 >
struct TypedVisitorUnwrapper<3, R, Visitor, T2, T3>
{
    const Visitor& visitor;
```

```
    T2 const& val2;
    T3 const& val3;

    TypedVisitorUnwrapper(const Visitor& visitor_, T2 const& val2_, T3 const& va
        : visitor(visitor_)
        , val2(val2_)
        , val3(val3_)

    {
    }

    template<typename T>
    R operator()(const T& val1) const
    {
        return visitor(val1, val2, val3);
    }
};

template< typename R, typename Visitor, typename T2, typename T3, typename T4 >
struct TypedVisitorUnwrapper<4, R, Visitor, T2, T3, T4>
{
    const Visitor& visitor;
    T2 const& val2;
    T3 const& val3;
    T4 const& val4;

    TypedVisitorUnwrapper(const Visitor& visitor_, T2 const& val2_, T3 const& va
        : visitor(visitor_)
        , val2(val2_)
        , val3(val3_)
        , val4(val4_)

    {
    }

    template<typename T>
    R operator()(const T& val1) const
    {
        return visitor(val1, val2, val3, val4);
    }
};
```

```
template< typename R, typename Visitor, typename T2, typename T3, typename T4, t
struct TypedVisitorUnwrapper<5, R, Visitor, T2, T3, T4, T5>
{
    const Visitor& visitor;
    T2 const& val2;
    T3 const& val3;
    T4 const& val4;
    T5 const& val5;

    TypedVisitorUnwrapper(const Visitor& visitor_, T2 const& val2_, T3 const& va
        : visitor(visitor_)
        , val2(val2_)
        , val3(val3_)
        , val4(val4_)
        , val5(val5_)

    {
    }

    template<typename T>
    R operator()(const T& val1) const
    {
        return visitor(val1, val2, val3, val4, val5);
    }
};



template<typename R, typename Visitor, typename V2>
struct VisitorUnwrapper
{
    const Visitor& visitor;
    const V2& r;

    VisitorUnwrapper(const Visitor& visitor_, const V2& r_)
        : visitor(visitor_)
        , r(r_)
    {
    }
```

```
    template< typename T1 >
    R operator()(T1 const& val1) const
    {
        typedef TypedVisitorUnwrapper<2, R, Visitor, T1> visitor_type;
        return VisitorApplicator<R>::apply(visitor_type(visitor, val1), r);
    }

    template< typename T1, typename T2 >
    R operator()(T1 const& val1, T2 const& val2) const
    {
        typedef TypedVisitorUnwrapper<3, R, Visitor, T1, T2> visitor_type;
        return VisitorApplicator<R>::apply(visitor_type(visitor, val1, val2), r)
    }

    template< typename T1, typename T2, typename T3 >
    R operator()(T1 const& val1, T2 const& val2, T3 const& val3) const
    {
        typedef TypedVisitorUnwrapper<4, R, Visitor, T1, T2, T3> visitor_type;
        return VisitorApplicator<R>::apply(visitor_type(visitor, val1, val2, val
    }

    template< typename T1, typename T2, typename T3, typename T4 >
    R operator()(T1 const& val1, T2 const& val2, T3 const& val3, T4 const& val4)
    {
        typedef TypedVisitorUnwrapper<5, R, Visitor, T1, T2, T3, T4> visitor_typ
        return VisitorApplicator<R>::apply(visitor_type(visitor, val1, val2, val
    }

    template< typename T1, typename T2, typename T3, typename T4, typename T5 >
    R operator()(T1 const& val1, T2 const& val2, T3 const& val3, T4 const& val4,
    {
        typedef TypedVisitorUnwrapper<6, R, Visitor, T1, T2, T3, T4, T5> visitor
        return VisitorApplicator<R>::apply(visitor_type(visitor, val1, val2, val
    }

};


template<typename R>
struct VisitorApplicator
```

```
{
    template<typename Visitor, typename V1>
    static R apply(const Visitor& v, const V1& arg)
    {
        switch( arg.index() )
        {
            case 0: return apply_visitor<0>(v, arg);
            case 1: return apply_visitor<1>(v, arg);
            case 2: return apply_visitor<2>(v, arg);
            case 3: return apply_visitor<3>(v, arg);
            case 4: return apply_visitor<4>(v, arg);
            case 5: return apply_visitor<5>(v, arg);
            case 6: return apply_visitor<6>(v, arg);
            case 7: return apply_visitor<7>(v, arg);
            case 8: return apply_visitor<8>(v, arg);
            case 9: return apply_visitor<9>(v, arg);
            case 10: return apply_visitor<10>(v, arg);
            case 11: return apply_visitor<11>(v, arg);
            case 12: return apply_visitor<12>(v, arg);
            case 13: return apply_visitor<13>(v, arg);
            case 14: return apply_visitor<14>(v, arg);
            case 15: return apply_visitor<15>(v, arg);

            default: return R();
        }
    }

    template<size_t Idx, typename Visitor, typename V1>
    static R apply_visitor(const Visitor& v, const V1& arg)
    {

#if variant_CPP11_OR_GREATER
        typedef typename variant_alternative<Idx, typename std::decay<V1>::type>
#else
        typedef typename variant_alternative<Idx, V1>::type value_type;
#endif
        return VisitorApplicatorImpl<R, value_type>::apply(v, get<Idx>(arg));
    }

#if variant_CPP11_OR_GREATER
    template<typename Visitor, typename V1, typename V2, typename ... V>
```

```
    static R apply(const Visitor& v, const V1& arg1, const V2& arg2, const V ...
    {
        typedef VisitorUnwrapper<R, Visitor, V1> Unwrapper;
        Unwrapper unwrapper(v, arg1);
        return apply(unwrapper, arg2, args ...);
    }
#else

    template< typename Visitor, typename V1, typename V2 >
    static R apply(const Visitor& v, V1 const& arg1, V2 const& arg2)
    {
        typedef VisitorUnwrapper<R, Visitor, V1> Unwrapper;
        Unwrapper unwrapper(v, arg1);
        return apply(unwrapper, arg2);
    }

    template< typename Visitor, typename V1, typename V2, typename V3 >
    static R apply(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& a
    {
        typedef VisitorUnwrapper<R, Visitor, V1> Unwrapper;
        Unwrapper unwrapper(v, arg1);
        return apply(unwrapper, arg2, arg3);
    }

    template< typename Visitor, typename V1, typename V2, typename V3, typename
    static R apply(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& a
    {
        typedef VisitorUnwrapper<R, Visitor, V1> Unwrapper;
        Unwrapper unwrapper(v, arg1);
        return apply(unwrapper, arg2, arg3, arg4);
    }

    template< typename Visitor, typename V1, typename V2, typename V3, typename
    static R apply(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& a
    {
        typedef VisitorUnwrapper<R, Visitor, V1> Unwrapper;
        Unwrapper unwrapper(v, arg1);
        return apply(unwrapper, arg2, arg3, arg4, arg5);
    }

#endif
```

```
};

#if variant_CPP11_OR_GREATER
template< size_t NumVars, typename Visitor, typename ... V >
struct VisitorImpl
{
    typedef decltype(std::declval<Visitor>()(get<0>(static_cast<const V&>(std::d
    typedef VisitorApplicator<result_type> applicator_type;
};
#endif
} // detail

#if variant_CPP11_OR_GREATER
// No perfect forwarding here in order to simplify code
template< typename Visitor, typename ... V >
inline auto visit(Visitor const& v, V const& ... vars) -> typename detail::Visit
{
    typedef detail::VisitorImpl<sizeof ... (V), Visitor, V... > impl_type;
    return impl_type::applicator_type::apply(v, vars...);
}
#else

template< typename R, typename Visitor, typename V1 >
inline R visit(const Visitor& v, V1 const& arg1)
{
    return detail::VisitorApplicator<R>::apply(v, arg1);
}

template< typename R, typename Visitor, typename V1, typename V2 >
inline R visit(const Visitor& v, V1 const& arg1, V2 const& arg2)
{
    return detail::VisitorApplicator<R>::apply(v, arg1, arg2);
}

template< typename R, typename Visitor, typename V1, typename V2, typename V3 >
inline R visit(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& arg3)
{
    return detail::VisitorApplicator<R>::apply(v, arg1, arg2, arg3);
}

template< typename R, typename Visitor, typename V1, typename V2, typename V3, t
```

```
inline R visit(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& arg3,
{
    return detail::VisitorApplicator<R>::apply(v, arg1, arg2, arg3, arg4);
}

template< typename R, typename Visitor, typename V1, typename V2, typename V3, t
inline R visit(const Visitor& v, V1 const& arg1, V2 const& arg2, V3 const& arg3,
{
    return detail::VisitorApplicator<R>::apply(v, arg1, arg2, arg3, arg4, arg5);
}

#endif

// 19.7.6 Relational operators

namespace detail {

template< class Variant >
struct Comparator
{
    static inline bool equal( Variant const & v, Variant const & w )
    {
        switch( v.index() )
        {
            case 0: return get<0>( v ) == get<0>( w );
            case 1: return get<1>( v ) == get<1>( w );
            case 2: return get<2>( v ) == get<2>( w );
            case 3: return get<3>( v ) == get<3>( w );
            case 4: return get<4>( v ) == get<4>( w );
            case 5: return get<5>( v ) == get<5>( w );
            case 6: return get<6>( v ) == get<6>( w );
            case 7: return get<7>( v ) == get<7>( w );
            case 8: return get<8>( v ) == get<8>( w );
            case 9: return get<9>( v ) == get<9>( w );
            case 10: return get<10>( v ) == get<10>( w );
            case 11: return get<11>( v ) == get<11>( w );
            case 12: return get<12>( v ) == get<12>( w );
            case 13: return get<13>( v ) == get<13>( w );
            case 14: return get<14>( v ) == get<14>( w );
            case 15: return get<15>( v ) == get<15>( w );
```

```
            default: return false;
        }
    }

    static inline bool less_than( Variant const & v, Variant const & w )
    {
        switch( v.index() )
        {
            case 0: return get<0>( v ) < get<0>( w );
            case 1: return get<1>( v ) < get<1>( w );
            case 2: return get<2>( v ) < get<2>( w );
            case 3: return get<3>( v ) < get<3>( w );
            case 4: return get<4>( v ) < get<4>( w );
            case 5: return get<5>( v ) < get<5>( w );
            case 6: return get<6>( v ) < get<6>( w );
            case 7: return get<7>( v ) < get<7>( w );
            case 8: return get<8>( v ) < get<8>( w );
            case 9: return get<9>( v ) < get<9>( w );
            case 10: return get<10>( v ) < get<10>( w );
            case 11: return get<11>( v ) < get<11>( w );
            case 12: return get<12>( v ) < get<12>( w );
            case 13: return get<13>( v ) < get<13>( w );
            case 14: return get<14>( v ) < get<14>( w );
            case 15: return get<15>( v ) < get<15>( w );

            default: return false;
        }
    }
};

} //namespace detail

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator==(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
{
    if      ( v.index() != w.index()     ) return false;
    else if ( v.valueless_by_exception() ) return true;
    else                                   return detail::Comparator< variant<T0
}
```

```
template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator!=(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
{
    return ! ( v == w );
}

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator<(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
{
    if      ( w.valueless_by_exception() ) return false;
    else if ( v.valueless_by_exception() ) return true;
    else if ( v.index() < w.index()      ) return true;
    else if ( v.index() > w.index()      ) return false;
    else                                   return detail::Comparator< variant<T0
}

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator>(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
{
    return w < v;
}

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator<=(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
{
    return ! ( v > w );
}

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
inline bool operator>=(
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
    variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15
```

```
{
    return ! ( v < w );
}

} // namespace variants

using namespace variants;

} // namespace nonstd

#if variant_CPP11_OR_GREATER

// 19.7.12 Hash support

namespace std {

template<>
struct hash< nonstd::monostate >
{
    std::size_t operator()( nonstd::monostate ) const variant_noexcept
    {
        return 42;
    }
};

template< class T0, class T1, class T2, class T3, class T4, class T5, class T6,
struct hash< nonstd::variant<T0, T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T
{
    std::size_t operator()( nonstd::variant<T0, T1, T2, T3, T4, T5, T6, T7, T8,
    {
        namespace nvd = nonstd::variants::detail;

        switch( v.index() )
        {
            case 0: return nvd::hash( 0 ) ^ nvd::hash( get<0>( v ) );
            case 1: return nvd::hash( 1 ) ^ nvd::hash( get<1>( v ) );
            case 2: return nvd::hash( 2 ) ^ nvd::hash( get<2>( v ) );
            case 3: return nvd::hash( 3 ) ^ nvd::hash( get<3>( v ) );
            case 4: return nvd::hash( 4 ) ^ nvd::hash( get<4>( v ) );
            case 5: return nvd::hash( 5 ) ^ nvd::hash( get<5>( v ) );
            case 6: return nvd::hash( 6 ) ^ nvd::hash( get<6>( v ) );
```

```
            case 7: return nvd::hash( 7 ) ^ nvd::hash( get<7>( v ) );
            case 8: return nvd::hash( 8 ) ^ nvd::hash( get<8>( v ) );
            case 9: return nvd::hash( 9 ) ^ nvd::hash( get<9>( v ) );
            case 10: return nvd::hash( 10 ) ^ nvd::hash( get<10>( v ) );
            case 11: return nvd::hash( 11 ) ^ nvd::hash( get<11>( v ) );
            case 12: return nvd::hash( 12 ) ^ nvd::hash( get<12>( v ) );
            case 13: return nvd::hash( 13 ) ^ nvd::hash( get<13>( v ) );
            case 14: return nvd::hash( 14 ) ^ nvd::hash( get<14>( v ) );
            case 15: return nvd::hash( 15 ) ^ nvd::hash( get<15>( v ) );

            default: return false;
        }
    }
};

} //namespace std

#endif // variant_CPP11_OR_GREATER

#if variant_BETWEEN( variant_COMPILER_MSVC_VER, 1300, 1900 )
# pragma warning( pop )
#endif

#endif // variant_USES_STD_VARIANT

#endif // NONSTD_VARIANT_LITE_HPP
```