Boost.TypeIndex 4.0

Table of Contents

Motivation

Sometimes getting and storing information about a type at runtime is required. For such cases a construction like `&typeid(T)` or C++11 class `std::type_index` is usually used, which is where problems start:

- `typeid(T)` and `std::type_index` require Run Time Type Info (RTTI)
- some implementations of `typeid(T)` erroneously do not strip const, volatile and references from type
- some compilers have bugs and do not correctly compare `std::type_info` objects across shared libraries
- only a few implementations of Standard Library currently provide `std::type_index`
- no easy way to store type info without stripping const, volatile and references
- no nice and portable way to get human readable type names
- no way to easily make your own type info class

Boost.TypeIndex library was designed to work around all those issues.

Note

`T` means type here. Think of it as of `T` in `template <class T>`

Getting started

`boost::typeindex::type_info` is a drop-in replacement for `std::type_info` and `boost::typeindex::type_index` is a drop-in replacement for `std::type_index`. Unlike Standard Library versions those classes can work without RTTI.

`type_index` provides the full set of comparison operators, hashing functions and ostream operators, so it can be used with any container class.

How to use

To start using Boost.TypeIndex:

Replace this:	With the following:	More Info
`#include <typeinfo>` `#include <typeindex>`	`#include <boost/type_index.hpp>`	more...
`std::type_index`	`boost::typeindex::type_index`	more...

`typeid(T)`
`typeid(T).name() // not human readable`
`typeid(variable)`
	`boost::typeindex::type_id<T>()`
	`boost::typeindex::type_id<T>().pretty_name() // human readable`
	`boost::typeindex::type_id_runtime(variable)`
		more...
		more...
		more...

`// attempt to save const, volatile, reference` `typeid(please_save_modifiers<T>)`	`// cvr = const, volatile, reference` `boost::typeindex::type_id_with_cvr<T>()`	more...

`// when reference to `std::type_info` is required`
`const std::type_info& v1 = typeid(int);`
 	
`// other cases`
`const std::type_info* v2 = &typeid(int);`
	`const boost::typeindex::type_info& v1`
	`    = boost::typeindex::type_id<int>().type_info();`
	
	`boost::typeindex::type_index v2`
	`    = boost::typeindex::type_id<int>();`
		more...
		more...

If you are using `type_id_runtime()` methods and RTTI is disabled, make sure that classes that are passed to `type_id_runtime()` are marked with `BOOST_TYPE_INDEX_REGISTER_CLASS` macro.

Example with Boost.Any

Here is how TypeIndex could be used in `boost/any.hpp`:

Before	With TypeIndex
`#include <typeinfo>`	`#include <boost/type_index.hpp>`
`virtu↵ al const std::type_info & type() const BOOST_NO↵ EXCEPT {     // requires RTTI     return typeid(ValueType); }`	`virtual const boost::typein↵ dex::type_info & type() const BOOST_NOEXCEPT {     // now works even with RTTI disabled     return boost::typein↵ dex::type_id<ValueType>().type_info(); }`

Example with Boost.Variant

Here is how TypeIndex could be used in `boost/variant/variant.hpp`:

Before	With TypeIndex

`#if !defined(BOOST_NO_TYPEID)`
`#include <typeinfo> // for typeid, ↵`
`std::type_info`
`#endif // BOOST_NO_TYPEID`
	`#include <boost/type_index.hpp>`

`#if !defined(BOOST_NO_TYPEID)`
 	
`class reflect`
`    : public static_visit↵`
`or<const std::type_info&>`
`{`
`public: // visitor interfaces`
 	
`    template <typename T>`
`    const std::type_info& operat↵`
`or()(const T&) const BOOST_NOEXCEPT`
`    {`
`        return typeid(T);`
`    }`
 	
`};`
 	
`#endif // BOOST_NO_TYPEID`
	`class reflect`
	`    : public static_visit↵`
	`or<const boost::typeindex::type_info&>`
	`{`
	`public: // visitor interfaces`
	
	`    template <typename T>`
	`   const boost::typeindex::type_info& operat↵`
	`or()(const T&) const BOOST_NOEXCEPT`
	`    {`
	`        return boost::typein↵`
	`dex::type_id<T>().type_info();`
	`    }`
	
	`};`

`#if !defined(BOOST_NO_TYPEID)`
`    const std::type_info& type() const`
`    {`
`        detail::variant::reflect visitor;`
`        return this->apply_visitor(visitor);`
`    }`
`#endif`
	`const boost::typein↵`
	`dex::type_info& type() const`
	`{`
	`    detail::variant::reflect visitor;`
	`    return this->apply_visitor(visitor);`
	`}`

Configuring and building the library

TypeIndex is a header only library and it does not use Boost libraries that require building. You just need to `#include <boost/type_index.hpp>` to start using it.

The library supports a number of configuration macros, defining which require full rebuild of all the projects that use TypeIndex:

Table 1. Configuration macros

Macro name	Short description
`BOOST_TYPE_INDEX_USER_TYPEINDEX`	Macro that allows you to use your own implementation of TypeIndex instead of the default all around the projects and libraries.
`BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY`	Macro that must be defined if you are mixing RTTI-on and RTTI-off.
`BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING` and `BOOST_TYPE_INDEX_FUNCTION_SIGNATURE`	Macros that allow you to specify parsing options and type name generating macro for RTTI-off cases.

You can define configuration macros in the `bjam` command line using one of the following approaches:

```
    b2 variant=release define=BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY stage
```

```
   b2 variant=release "define=BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING='(39, 1, true, \"T = ↵
\")'" stage
```

However, it may be more convenient to define configuration macros in the "boost/config/user.hpp" file in order to automatically define them both for the library and user's projects.

How it works

`type_index` is just a typedef for `boost::typeindex::stl_type_index` or `boost::typeindex::ctti_type_index`.

Depending on the `typeid()` availability TypeIndex library will choose an optimal class for `type_index`. In cases when at least basic support for `typeid()` is available `stl_type_index` will be used.

`BOOST_TYPE_INDEX_REGISTER_CLASS` macro is a helper macro that places some virtual helper functions or expands to nothing.

Issues with cross module type comparison on a bugged compilers are bypassed by directly comparing strings with type (latest versions of those compilers resolved that issue using exactly the same approach).

Examples

Getting human readable and mangled type names

The following example shows how short (mangled) and human readable type names could be obtained from a type. Works with and without RTTI.

```
#include <boost/type_index.hpp>
#include <iostream>

template <class T>
void foo(T) {
    std::cout << "\n Short name: " << boost::typeindex::type_id<T>().raw_name();
    std::cout << "\n Readable name: " << boost::typeindex::type_id<T>().pretty_name();
}

struct user_defined_type{};

namespace ns1 { namespace ns2 {
    struct user_defined_type{};
}} // namespace ns1::ns2

namespace {
    struct in_anon_type{};
} // anonymous namespace

int main() {
    // Call to
    foo(1);
    // will output something like this:
    //
    // (RTTI on)                                        (RTTI off)
    // Short name: i                                    Short name: int]
    // Readable name: int                               Readable name: int

    user_defined_type t;
    foo(t);
    // Will output:
    //
    // (RTTI on)                                        (RTTI off)
    // Short name: 17user_defined_type                  user_defined_type]
    // Readable name: user_defined_type                 user_defined_type

    ns1::ns2::user_defined_type t_in_ns;
    foo(t_in_ns);
    // Will output:
    //
    // (RTTI on)                                        (RTTI off)
    // Short name: N3ns13ns217user_defined_typeE        ns1::ns2::user_defined_type]
    // Readable name: ns1::ns2::user_defined_type       ns1::ns2::user_defined_type

    in_anon_type anon_t;
    foo(anon_t);
    // Will output:
    //
    // (RTTI on)                                        (RTTI off)
    // Short name: N12_GLOBAL__N_112in_anon_typeE       {anonymous}::in_anon_type]
    // Readable name: (anonymous namespace)::in_anon_type   {anonymous}::in_anon_type
}
```

Short names are very compiler dependant: some compiler will output `.H`, others `i`.

Readable names may also differ between compilers: `struct user_defined_type`, `user_defined_type`.

Warning

With RTTI off different classes with same names in anonymous namespace may collapse. See 'RTTI emulation limitations'.

Storing information about a type in container

The following example shows how an information about a type could be stored. Example works with and without RTTI.

```
#include <boost/type_index.hpp>
#include <boost/unordered_set.hpp>
#include <boost/functional/hash.hpp>
#include <cassert>

int main() {
    boost::unordered_set<boost::typeindex::type_index> types;

    // Storing some `boost::type_info`s
    types.insert(boost::typeindex::type_id<int>());
    types.insert(boost::typeindex::type_id<float>());

    // `types` variable contains two `boost::type_index`es:
    assert(types.size() == 2);

    // Const, volatile and reference will be striped from the type:
    bool is_inserted = types.insert(boost::typeindex::type_id<const int>()).second;
    assert(!is_inserted);
    assert(types.erase(boost::typeindex::type_id<float&>()) == 1);

    // We have erased the `float` type, only `int` remains
    assert(*types.begin() == boost::typeindex::type_id<int>());
}
```

Getting through the inheritance to receive a real type name

The following example shows that `type_info` is able to store the real type, successfully getting through all the inheritances.

Example works with and without RTTI."

```
#include <boost/type_index.hpp>
#include <iostream>

struct A {
    BOOST_TYPE_INDEX_REGISTER_CLASS
    virtual ~A(){}
};
struct B: public A { BOOST_TYPE_INDEX_REGISTER_CLASS };
struct C: public B { BOOST_TYPE_INDEX_REGISTER_CLASS };

void print_real_type(const A& a) {
    std::cout << boost::typeindex::type_id_runtime(a).pretty_name() << '\n';
}

int main() {
    C c;
    const A& c_as_a = c;
    print_real_type(c_as_a);    // Outputs `struct C`
    print_real_type(B());       // Outputs `struct B`
}
```

Exact type matching: storing type with const, volatile and reference qualifiers

The following example shows that `type_index` (and `type_info`) is able to store the exact type, without stripping const, volatile and references. Example works with and without RTTI.

In this example we'll create a class that stores a pointer to function and remembers the exact type of the parameter the function accepts. When the call to the bound function is made, he actual input parameter type is checked against the stored parameter type and an exception is thrown in case of mismatch.

```
#include <boost/type_index.hpp>
#include <iostream>
#include <stdexcept>
#include <cassert>

class type_erased_unary_function {
    void*                           function_ptr_;
    boost::typeindex::type_index    exact_param_t_;

public:
    template <class ParamT>
    type_erased_unary_function(void(*ptr)(ParamT))
        : function_ptr_(reinterpret_cast<void*>(ptr)) // ptr - is a pointer to function return↵
ing `void` and accepting parameter of type `ParamT`
        , exact_param_t_(boost::typeindex::type_id_with_cvr<ParamT>())
    {}

    template <class ParamT>
    void call(ParamT v) {
        if (exact_param_t_ != boost::typeindex::type_id_with_cvr<ParamT>()) {
            throw std::runtime_error("Incorrect `ParamT`");
        }

        return (reinterpret_cast<void(*)(ParamT)>(function_ptr_))(v);
    }
};

void foo(int){}

int main() {
    type_erased_unary_function func(&foo);
    func.call(100); // OK, `100` has type `int`

    try {
        int i = 100;

        // An attempt to convert stored function to a function accepting reference
        func.call<int&>(i); // Will throw, because types `int&` and `int` missmatch

        assert(false);
    } catch (const std::runtime_error& /*e*/) {}
}
```

Table of raw_name() and pretty_name() outputs with and without RTTI

The following example shows how different type names look when we explicitly use classes for RTTI and RTT off.

This example requires RTTI. For a more portable example see 'Getting human readable and mangled type names':

```cpp
#include <boost/type_index/stl_type_index.hpp>
#include <boost/type_index/ctti_type_index.hpp>
#include <iostream>

template <class T>
void print(const char* name) {
    boost::typeindex::stl_type_index sti = boost::typeindex::stl_type_index::type_id<T>();
    boost::typeindex::ctti_type_index cti = boost::typeindex::ctti_type_index::type_id<T>();
    std::cout << "\t[" /* start of the row */
        << "[" << name << "]"
        << "[`" << sti.raw_name() << "`] "
        << "[`" << sti.pretty_name() << "`] "
        << "[`" << cti.raw_name() << "`] "
    << "]\n" /* end of the row */ ;
}

struct user_defined_type{};

namespace ns1 { namespace ns2 {
    struct user_defined_type{};
}} // namespace ns1::ns2

namespace {
    struct in_anon_type{};
} // anonymous namespace

namespace ns3 { namespace { namespace ns4 {
    struct in_anon_type{};
}}} // namespace ns3::{anonymous}::ns4


template <class T0, class T1>
class templ {};

template <>
class templ<int, int> {};

int main() {
    std::cout << "[table:id Table of names\n";
    std::cout << "\t[[Type] [RTTI & raw_name] [RTTI & pretty_name] [noRTTI & raw_name]]\n";

    print<user_defined_type>("User defined type");
    print<in_anon_type>("In anonymous namespace");
    print<ns3::ns4::in_anon_type>("In ns3::{anonymous}::ns4 namespace");
    print<templ<short, int> >("Template class");
    print<templ<int, int> >("Template class (full specialization)");
    print<templ<
        templ<char, signed char>,
        templ<int, user_defined_type>
    > >("Template class with templae classes");


    std::cout << "]\n";
}
```

Code from the example will produce the following table:

Table 2. Table of names

Type	RTTI & raw_name	RTTI & pretty_name	noRTTI & raw_name
User defined type	`17user_defined_type`	`user_defined_type`	`user_defined_type]`
In anonymous namespace	`N12_GLOBAL__N_112in_anon_typeE`	`(anonymous namespace)::in_anon_type`	`{anonymous}::in_anon_type]`
In ns3::{anonymous}::ns4 namespace	`N3ns312_GLOBAL__N_13ns412in_anon_typeE`	`ns3::(anonymous namespace)::ns4::in_anon_type`	`ns3::{anonymous}::ns4::in_anon_type]`
Template class	`5templIsiE`	`templ<short, int>`	`templ<short int, int>]`
Template class (full specialization)	`5templIiiE`	`templ<int, int>`	`templ<int, int>]`
Template class with templae classes	`5templIS_IcaES_Ii17user_defined_typeEE`	`templ<templ<char, signed char>, templ<int, user_defined_type> >`	`templ<templ<char, signed char>, templ<int, user_defined_type> >]`

We have not show the "noRTTI & pretty_name" column in the table becuse it is almost equal to "noRTTI & raw_name" column.

Warning

With RTTI off different classes with same names in anonymous namespace may collapse. See 'RTTI emulation limitations'.

Boost.TypeIndex Header Reference

Header <boost/type_index.hpp>

Includes minimal set of headers required to use the Boost.TypeIndex library.

By inclusion of this file most optimal type index classes will be included and used as a boost::typeindex::type_index and boost::typeindex::type_info.

```
BOOST_TYPE_INDEX_REGISTER_CLASS
BOOST_TYPE_INDEX_FUNCTION_SIGNATURE
BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING
BOOST_TYPE_INDEX_USER_TYPEINDEX
BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY
```

```
namespace boost {
  namespace typeindex {
    typedef platform_specific type_index;
    typedef type_index::type_info_t type_info;
    template<typename T> type_index type_id();
    template<typename T> type_index type_id_with_cvr();
    template<typename T> type_index type_id_runtime(const T &);
  }
}
```

Type definition type_index

type_index

Synopsis

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


typedef platform_specific type_index;
```

Description

Depending on a compiler flags, optimal implementation of type_index will be used as a default boost::typeindex::type_index.

Could be a boost::typeindex::stl_type_index, boost::typeindex::ctti_type_index or user defined type_index class.

See boost::typeindex::type_index_facade for a full description of type_index functions.

Type definition type_info

type_info

Synopsis

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


typedef type_index::type_info_t type_info;
```

Description

Depending on a compiler flags, optimal implementation of type_info will be used as a default boost::typeindex::type_info.

Could be a std::type_info, boost::typeindex::detail::ctti_data or some user defined class.

type_info **is not** copyable or default constructible. It is **not** assignable too!

Function template type_id

boost::typeindex::type_id

Synopsis

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


template<typename T> type_index type_id();
```

Description

Function to get boost::typeindex::type_index for a type T. Removes const, volatile && and & modifiers from T.

Example:

```
type_index ti = type_id<int&>();
std::cout << ti.pretty_name();  // Outputs 'int'
```

Template Parameters:	`T` Type for which type_index must be created.
Returns:	boost::typeindex::type_index with information about the specified type T.
Throws:	Nothing.

Function template type_id_with_cvr

boost::typeindex::type_id_with_cvr

Synopsis

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


template<typename T> type_index type_id_with_cvr();
```

Description

Function for constructing boost::typeindex::type_index instance for type T. Does not remove const, volatile, & and && modifiers from T.

If T has no const, volatile, & and && modifiers, then returns exactly the same result as in case of calling `type_id<T>()`.

Example:

```
type_index ti = type_id_with_cvr<int&>();
std::cout << ti.pretty_name();  // Outputs 'int&'
```

Template Parameters:	`T` Type for which type_index must be created.
Returns:	boost::typeindex::type_index with information about the specified type T.
Throws:	Nothing.

Function template type_id_runtime

boost::typeindex::type_id_runtime

Synopsis

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


template<typename T> type_index type_id_runtime(const T & runtime_val);
```

Description

Function that works exactly like C++ typeid(rtti_val) call, but returns boost::type_index.

Retunrs runtime information about specified type.

Requirements: RTTI available or Base and Derived classes must be marked with BOOST_TYPE_INDEX_REGISTER_CLASS.

Example:

```
struct Base { virtual ~Base(){} };
struct Derived: public Base  {};
...
Derived d;
Base& b = d;
type_index ti = type_id_runtime(b);
std::cout << ti.pretty_name();  // Outputs 'Derived'
```

Parameters:	`runtime_val` Varaible which runtime type must be returned.
Returns:	boost::typeindex::type_index with information about the specified variable.
Throws:	Nothing.

Macro BOOST_TYPE_INDEX_REGISTER_CLASS

BOOST_TYPE_INDEX_REGISTER_CLASS

Synopsis

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

BOOST_TYPE_INDEX_REGISTER_CLASS
```

Description

BOOST_TYPE_INDEX_REGISTER_CLASS is used to help to emulate RTTI. Put this macro into the public section of polymorphic class to allow runtime type detection.

Depending on the typeid() availability this macro will expand to nothing or to virtual helper function `virtual const type_info& boost_type_info_type_id_runtime_() const noexcept`.

Example:

```
class A {
public:
    BOOST_TYPE_INDEX_REGISTER_CLASS
    virtual ~A(){}
};

struct B: public A {
    BOOST_TYPE_INDEX_REGISTER_CLASS
};

struct C: public B {
    BOOST_TYPE_INDEX_REGISTER_CLASS
};

...

C c1;
A* pc1 = &c1;
assert(boost::typeindex::type_id<C>() == boost::typeindex::type_id_runtime(*pc1));
```

Macro BOOST_TYPE_INDEX_FUNCTION_SIGNATURE

BOOST_TYPE_INDEX_FUNCTION_SIGNATURE

Synopsis

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

BOOST_TYPE_INDEX_FUNCTION_SIGNATURE
```

Description

BOOST_TYPE_INDEX_FUNCTION_SIGNATURE is used by boost::typeindex::ctti_type_index class to deduce the name of a type. If your compiler is not recognized by the TypeIndex library and you wish to work with boost::typeindex::ctti_type_index, you may define this macro by yourself.

BOOST_TYPE_INDEX_FUNCTION_SIGNATURE must be defined to a compiler specific macro that outputs the **whole** function signature **including template parameters**.

If your compiler is not recognised and BOOST_TYPE_INDEX_FUNCTION_SIGNATURE is not defined, then a compile-time error will arise at any attempt to use boost::typeindex::ctti_type_index classes.

See BOOST_TYPE_INDEX_REGISTER_CTTI_PARSING_PARAMS and BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING for an information of how to tune the implementation to make a nice pretty_name() output.

Macro BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING

BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING

Synopsis

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

BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING
```

Description

This is a helper macro for making correct pretty_names() with RTTI off.

BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING macro may be defined to '(begin_skip, end_skip, runtime_skip, runtime_skip_until)' with parameters for adding a support for compilers, that by default are not recognized by TypeIndex library.

Example:

Imagine the situation when

```
boost::typeindex::ctti_type_index::type_id<int>().pretty_name()
```

returns the following string:

```
"static const char *boost::detail::ctti<int>::n() [T = int]"
```

and

```
boost::typeindex::ctti_type_index::type_id<short>().pretty_name()
```

returns the following:

```
"static const char *boost::detail::ctti<short>::n() [T = short]"
```

As we may see first 39 characters are "static const char *boost::detail::ctti<" and they do not depend on the type T. After first 39 characters we have a human readable type name which is duplicated at the end of a string. String always ends on ']', which consumes 1 character.

Now if we define `BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING` to `(39, 1, false, "")` we'll be getting

```
"int>::n() [T = int"
```

for `boost::typeindex::ctti_type_index::type_id<int>().pretty_name()` and

```
"short>::n() [T = short"
```

for `boost::typeindex::ctti_type_index::type_id<short>().pretty_name()`.

Now we need to take additional care of the characters that go before the last mention of our type. We'll do that by telling the macro that we need to cut off everything that goes before the "T = " including the "T = " itself:

```
(39, 1, true, "T = ")
```

In case of GCC or Clang command line we need to add the following line while compiling all the sources:

```
-DBOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING='(39, 1, true, "T = ")'
```

See RTTI emulation limitations for more info.

Macro BOOST_TYPE_INDEX_USER_TYPEINDEX

BOOST_TYPE_INDEX_USER_TYPEINDEX

Synopsis

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

BOOST_TYPE_INDEX_USER_TYPEINDEX
```

Description

BOOST_TYPE_INDEX_USER_TYPEINDEX can be defined to the path to header file with user provided implementation of type_index.

See Making a custom type_index section of documentation for usage example.

Macro BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY

BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY

Synopsis

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

BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY
```

Description

BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY is a helper macro that must be defined if mixing RTTI on/off modules. See Mixing sources with RTTI on and RTTI off section of documentation for more info.

Header <boost/type_index/ctti_type_index.hpp>

Contains boost::typeindex::ctti_type_index class.

boost::typeindex::ctti_type_index class can be used as a drop-in replacement for std::type_index.

It is used in situations when typeid() method is not available or BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY macro is defined.

```
namespace boost {
  namespace typeindex {
    class ctti_type_index;

    // Helper method for getting detail::ctti_data of a tempalte patameter T.
    template<typename T> unspecified ctti_construct();
  }
}
```

Class ctti_type_index

boost::typeindex::ctti_type_index

Synopsis

```
// In header: <boost/type_index/ctti_type_index.hpp>


class ctti_type_index {
public:
  // types
  typedef unspecified type_info_t;

  // construct/copy/destruct
  ctti_type_index() noexcept;
  ctti_type_index(const type_info_t &) noexcept;

  // private member functions
  std::size_t get_raw_name_length() const noexcept;

  // public member functions
  const type_info_t & type_info() const noexcept;
  const char * raw_name() const noexcept;
  std::string pretty_name() const;
  std::size_t hash_code() const noexcept;

  // public static functions
  template<typename T> static ctti_type_index type_id() noexcept;
  template<typename T> static ctti_type_index type_id_with_cvr() noexcept;
  template<typename T>
    static ctti_type_index type_id_runtime(const T &) noexcept;
};
```

Description

This class is a wrapper that pretends to work exactly like stl_type_index, but does not require RTTI support. **For description of functions see** type_index_facade.

This class produces slightly longer type names, so consider using stl_type_index in situations when typeid() is working.

`ctti_type_index` public construct/copy/destruct

1.
```
ctti_type_index() noexcept;
```

2.
```
ctti_type_index(const type_info_t & data) noexcept;
```

`ctti_type_index` private member functions

1.
```
std::size_t get_raw_name_length() const noexcept;
```

`ctti_type_index` public member functions

1.
```
const type_info_t & type_info() const noexcept;
```

2.
```
const char * raw_name() const noexcept;
```

3.
```
std::string pretty_name() const;
```

4.
```
std::size_t hash_code() const noexcept;
```

`ctti_type_index` public static functions

1.
```
template<typename T> static ctti_type_index type_id() noexcept;
```

2.
```
template<typename T> static ctti_type_index type_id_with_cvr() noexcept;
```

3.
```
template<typename T>
  static ctti_type_index type_id_runtime(const T & variable) noexcept;
```

Header <boost/type_index/stl_type_index.hpp>

Contains boost::typeindex::stl_type_index class.

boost::typeindex::stl_type_index class can be used as a drop-in replacement for std::type_index.

It is used in situations when RTTI is enabled or typeid() method is available. When typeid() is disabled or BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY macro is defined boost::typeindex::ctti is usually used instead of boost::typeindex::stl_type_index.

```
namespace boost {
  namespace typeindex {
    class stl_type_index;
  }
}
```

Class stl_type_index

boost::typeindex::stl_type_index

Synopsis

```
// In header: <boost/type_index/stl_type_index.hpp>


class stl_type_index : public boost::typeindex::type_index_facade< stl_type_index, std::type_info >
{
public:
  // types
  typedef std::type_info type_info_t;

  // construct/copy/destruct
  stl_type_index() noexcept;
  stl_type_index(const type_info_t &) noexcept;

  // public member functions
  const type_info_t & type_info() const noexcept;
  const char * raw_name() const noexcept;
  const char * name() const noexcept;
  std::string pretty_name() const;
  std::size_t hash_code() const noexcept;
  bool equal(const stl_type_index &) const noexcept;
  bool before(const stl_type_index &) const noexcept;

  // public static functions
  template<typename T> static stl_type_index type_id() noexcept;
  template<typename T> static stl_type_index type_id_with_cvr() noexcept;
  template<typename T>
    static stl_type_index type_id_runtime(const T &) noexcept;
};
```

Description

This class is a wrapper around std::type_info, that workarounds issues and provides much more rich interface. **For description of functions see** type_index_facade.

This class requires typeid() to work. For cases when RTTI is disabled see ctti_type_index.

`stl_type_index` public construct/copy/destruct

1.
```
stl_type_index() noexcept;
```

2.
```
stl_type_index(const type_info_t & data) noexcept;
```

`stl_type_index` public member functions

1.
```
const type_info_t & type_info() const noexcept;
```

2.
```
const char * raw_name() const noexcept;
```

3.
```
const char * name() const noexcept;
```

4. ```
   std::string pretty_name() const;
   ```

5. ```
   std::size_t hash_code() const noexcept;
   ```

6. ```
   bool equal(const stl_type_index & rhs) const noexcept;
   ```

7. ```
   bool before(const stl_type_index & rhs) const noexcept;
   ```

`stl_type_index` public static functions

1. ```
   template<typename T> static stl_type_index type_id() noexcept;
   ```

2. ```
   template<typename T> static stl_type_index type_id_with_cvr() noexcept;
   ```

3. ```
   template<typename T>
     static stl_type_index type_id_runtime(const T & value) noexcept;
   ```

Header <boost/type_index/type_index_facade.hpp>

```
namespace boost {
  namespace typeindex {
    template<typename Derived, typename TypeInfo> class type_index_facade;

    // noexcept comparison operators for type_index_facade classes.
    bool operator==,!=,<,...(const type_index_facade & lhs,
                             const type_index_facade & rhs);

    // noexcept comparison operators for type_index_facade and it's TypeInfo classes.
    bool operator==,!=,<,...(const type_index_facade & lhs,
                             const TypeInfo & rhs);

    // noexcept comparison operators for type_index_facade's TypeInfo and type_in↵
dex_facade classes.
    bool operator==,!=,<,...(const TypeInfo & lhs,
                             const type_index_facade & rhs);

    // Ostream operator that will output demangled name.
    template<typename CharT, typename TriatT, typename Derived,
             typename TypeInfo>
      std::basic_ostream< CharT, TriatT > &
      operator<<(std::basic_ostream< CharT, TriatT > & ostr,
                 const type_index_facade< Derived, TypeInfo > & ind);
    template<typename Derived, typename TypeInfo>
      std::size_t hash_value(const type_index_facade< Derived, TypeInfo > &);
  }
}
```
```

Class template type_index_facade

boost::typeindex::type_index_facade

Synopsis

```
// In header: <boost/type_index/type_index_facade.hpp>

template<typename Derived, typename TypeInfo>
class type_index_facade {
public:
  // types
  typedef TypeInfo type_info_t;

  // public member functions
  const char * name() const noexcept;
  std::string pretty_name() const;
  bool equal(const Derived &) const noexcept;
  bool before(const Derived &) const noexcept;
  std::size_t hash_code() const noexcept;

  // protected member functions
  const char * raw_name() const noexcept;
  const type_info_t & type_info() const noexcept;

  // protected static functions
  template<typename T> static Derived type_id() noexcept;
  template<typename T> static Derived type_id_with_cvr() noexcept;
  template<typename T> static Derived type_id_runtime(const T &) noexcept;
};
```

Description

This class takes care about the comparison operators, hash functions and ostream operators. Use this class as a public base class for defining new type_info-conforming classes.

Example:

```
class stl_type_index: public type_index_facade<stl_type_index, std::type_info>
{
public:
    typedef std::type_info type_info_t;
private:
    const type_info_t* data_;

public:
    stl_type_index(const type_info_t& data) noexcept
        : data_(&data)
    {}
// ...
};
```

Note

Take a look at the protected methods. They are **not defined** in type_index_facade. Protected member functions raw_name() **must** be defined in Derived class. All the other methods are mandatory.

See Also:

'Making a custom type_index' section for more information about creating your own type_index using type_index_facade.

Template Parameters

1. ```
   typename Derived
   ```

   Class derived from `type_index_facade`.

2. ```
   typename TypeInfo
   ```

   Class that will be used as a base type_info class.

`type_index_facade` public member functions

1. ```
   const char * name() const noexcept;
   ```

   Override: This function **may** be redefined in Derived class. Overrides **must** not throw.
   Returns: Name of a type. By default retuns Derived::raw_name().

2. ```
   std::string pretty_name() const;
   ```

   Override: This function **may** be redefined in Derived class. Overrides may throw.
   Returns: Human redable type name. By default retuns Derived::name().

3. ```
   bool equal(const Derived & rhs) const noexcept;
   ```

   Override: This function **may** be redefined in Derived class. Overrides **must** not throw.
   Returns: True if two types are equal. By default compares types by raw_name().

4. ```
   bool before(const Derived & rhs) const noexcept;
   ```

   Override: This function **may** be redefined in Derived class. Overrides **must** not throw.
   Returns: True if rhs is greater than this. By default compares types by raw_name().

5. ```
   std::size_t hash_code() const noexcept;
   ```

   Override: This function **may** be redefined in Derived class. Overrides **must** not throw.

   > **Note**
   >
   > <boost/functional/hash.hpp> has to be included if this function is used.

   Returns: Hash code of a type. By default hashes types by raw_name().

`type_index_facade` protected member functions

1. ```
   const char * raw_name() const noexcept;
   ```

   Override: This function **must** be redefined in Derived class. Overrides **must** not throw.
   Returns: Pointer to unredable/raw type name.

2.
```
const type_info_t & type_info() const noexcept;
```

Override: This function **may** be redefined in Derived class. Overrides **must** not throw.
Returns: Const reference to underlying low level type_info_t.

`type_index_facade` protected static functions

1.
```
template<typename T> static Derived type_id() noexcept;
```

This is a factory method that is used to create instances of Derived classes. boost::typeindex::type_id() will call this method, if Derived has same type as boost::typeindex::type_index.

Override: This function **may** be redefined and made public in Derived class. Overrides **must** not throw. Overrides **must** remove const, volatile && and & modifiers from T.
Template Parameters: `T` Type for which type_index must be created.
Returns: type_index for type T.

2.
```
template<typename T> static Derived type_id_with_cvr() noexcept;
```

This is a factory method that is used to create instances of Derived classes. boost::typeindex::type_id_with_cvr() will call this method, if Derived has same type as boost::typeindex::type_index.

Override: This function **may** be redefined and made public in Derived class. Overrides **must** not throw. Overrides **must not** remove const, volatile && and & modifiers from T.
Template Parameters: `T` Type for which type_index must be created.
Returns: type_index for type T.

3.
```
template<typename T>
  static Derived type_id_runtime(const T & variable) noexcept;
```

This is a factory method that is used to create instances of Derived classes. boost::typeindex::type_id_runtime(const T&) will call this method, if Derived has same type as boost::typeindex::type_index.

Override: This function **may** be redefined and made public in Derived class.
Parameters: `variable` Variable which runtime type will be stored in type_index.
Returns: type_index with runtime type of variable.

Function template hash_value

boost::typeindex::hash_value

Synopsis

```
// In header: <boost/type_index/type_index_facade.hpp>


template<typename Derived, typename TypeInfo>
  std::size_t hash_value(const type_index_facade< Derived, TypeInfo > & lhs);
```

Description

This free function is used by Boost's unordered containers.

Note

<boost/functional/hash.hpp> has to be included if this function is used.

Making a custom type_index

Sometimes there may be a need to create your own type info system. This may be usefull if you wish to store some more info about types (PODness, size of a type, pointers to common functions...) or if you have an idea of a more compact types representations.

Basics

The following example shows how a user defined type_info can be created and used. Example works with and without RTTI.

Consider situation when user uses only those types in `typeid()`:

```
#include <vector>
#include <string>

namespace my_namespace {

class my_class;
struct my_struct;

typedef std::vector<my_class> my_classes;
typedef std::string my_string;

} // namespace my_namespace
```

In that case user may wish to save space in binary and create it's own type system. For that case `detail::typenum<>` meta function is added. Depending on the input type T this function will return different numeric values.

```
#include <boost/type_index/type_index_facade.hpp>

namespace my_namespace { namespace detail {
    template <class T> struct typenum;
    template <> struct typenum<void>{       enum {value = 0}; };
    template <> struct typenum<my_class>{   enum {value = 1}; };
    template <> struct typenum<my_struct>{  enum {value = 2}; };
    template <> struct typenum<my_classes>{ enum {value = 3}; };
    template <> struct typenum<my_string>{  enum {value = 4}; };

    // my_typeinfo structure is used to save type number
    struct my_typeinfo {
        const char* const type_;
    };

    const my_typeinfo infos[5] = {
        {"void"}, {"my_class"}, {"my_struct"}, {"my_classes"}, {"my_string"}
    };

    template <class T>
    inline const my_typeinfo& my_typeinfo_construct() {
        return infos[typenum<T>::value];
    }
}} // my_namespace::detail
```

`my_type_index` is a user created type_index class. If in doubt during this phase, you can always take a look at the `<boost/type_index/ctti_type_index.hpp>` or `<boost/type_index/stl_type_index.hpp>` files. Documentation for `type_index_facade` could be also useful.

See implementation of `my_type_index`:

```
namespace my_namespace {

class my_type_index: public boost::typeindex::type_index_facade<my_type_index, de↵
tail::my_typeinfo> {
    const detail::my_typeinfo* data_;

public:
    typedef detail::my_typeinfo type_info_t;

    inline my_type_index() BOOST_NOEXCEPT
        : data_(&detail::my_typeinfo_construct<void>())
    {}

    inline my_type_index(const type_info_t& data) BOOST_NOEXCEPT
        : data_(&data)
    {}

    inline const type_info_t&  type_info() const BOOST_NOEXCEPT {
        return *data_;
    }

    inline const char*  raw_name() const BOOST_NOEXCEPT {
        return data_->type_;
    }

    inline std::string  pretty_name() const {
        return data_->type_;
    }

    template <class T>
    inline static my_type_index type_id() BOOST_NOEXCEPT {
        return detail::my_typeinfo_construct<T>();
    }

    template <class T>
    inline static my_type_index type_id_with_cvr() BOOST_NOEXCEPT {
        return detail::my_typeinfo_construct<T>();
    }

    template <class T>
    inline static my_type_index type_id_runtime(const T& variable) BOOST_NOEXCEPT;
};

} // namespace my_namespace
```

Note that we have used the boost::typeindex::type_index_facade class as base. That class took care about all the helper function and operators (comparison, hashing, ostreaming and others).

Finally we can use the my_type_index class for getting type indexes:

```
my_type_index
    cl1 = my_type_index::type_id<my_class>(),
    st1 = my_type_index::type_id<my_struct>(),
    st2 = my_type_index::type_id<my_struct>(),
    vec = my_type_index::type_id<my_classes>()
;

assert(cl1 != st1);
assert(st2 == st1);
assert(vec.pretty_name() == "my_classes");
assert(cl1.pretty_name() == "my_class");
```

Getting type infos at runtime

Usually to allow runtime type info we need to register class with some macro. Let's see how a `MY_TYPEINDEX_REGISTER_CLASS` macro could be implemented for our `my_type_index` class:

```
namespace my_namespace { namespace detail {

    template <class T>
    inline const my_typeinfo& my_typeinfo_construct_ref(const T*) {
        return my_typeinfo_construct<T>();
    }

#define MY_TYPEINDEX_REGISTER_CLASS                                             \
    virtual const my_namespace::detail::my_typeinfo& type_id_runtime() const {  \
        return my_namespace::detail::my_typeinfo_construct_ref(this);           \
    }

}} // namespace my_namespace::detail
```

Now when we have a MY_TYPEINDEX_REGISTER_CLASS, let's implement a `my_type_index::type_id_runtime` method:

```
namespace my_namespace {
    template <class T>
    my_type_index my_type_index::type_id_runtime(const T& variable) BOOST_NOEXCEPT {
        // Classes that were marked with `MY_TYPEINDEX_REGISTER_CLASS` will have a
        // `type_id_runtime()` method.
        return variable.type_id_runtime();
    }
}
```

Consider the situation, when `my_class` and `my_struct` are polymorphic classes:

```
namespace my_namespace {

class my_class {
public:
    MY_TYPEINDEX_REGISTER_CLASS
    virtual ~my_class() {}
};

struct my_struct: public my_class {
    MY_TYPEINDEX_REGISTER_CLASS
};

} // namespace my_namespace
```

Now the following example will compile and work.

```
my_struct str;
my_class& reference = str;
assert(my_type_index::type_id<my_struct>() == my_type_index::type_id_runtime(reference));
```

Using new type infos all around the code

There is an easy way to force `boost::typeindex::type_id` to use your own type_index class.

All we need to do is just define `BOOST_TYPE_INDEX_USER_TYPEINDEX` to the full path to header file of your type index class:

```
// BOOST_TYPE_INDEX_USER_TYPEINDEX must be defined *BEFORE* first inclusion of <boost/type_in↵
dex.hpp>
#define BOOST_TYPE_INDEX_USER_TYPEINDEX <boost/../libs/type_index/ex↵
amples/user_defined_typeinfo.hpp>
#include <boost/type_index.hpp>
```

You'll also need to add some typedefs and macro to your "user_defined_typeinfo.hpp" header file:

```
#define BOOST_TYPE_INDEX_REGISTER_CLASS MY_TYPEINDEX_REGISTER_CLASS
namespace boost { namespace typeindex {
    typedef my_namespace::my_type_index type_index;
}}
```

That's it! Now all TypeIndex global methods and typedefs will be using your class:

```
boost::typeindex::type_index worldwide = boost::typeindex::type_id<my_classes>();
assert(worldwide.pretty_name() == "my_classes");
assert(worldwide == my_type_index::type_id<my_classes>());
```

Space and Performance

- `ctti_type_index` uses macro for getting full text representation of function name which could lead to code bloat, so prefer using `stl_type_index` type when possible.
- All the type_index classes hold a single pointer and are fast to copy.
- Calls to `const char* raw_name()` do not require dynamic memory allocation and usually just return a pointer to an array of chars in a read-only section of the binary image.
- Comparison operators are optimized as much as possible and execute a single `std::strcmp` in worst case.
- Calls to `std::string pretty_name()` usually require dynamic memory allocation and some computations, so they are not recommended for usage in performance critical sections.

Code bloat

Without RTTI TypeIndex library will switch from using `boost::typeindex::stl_type_index` class to `boost::typeindex::ctti_type_index`. `boost::typeindex::ctti_type_index` uses macro for getting full text representation of function name for each type that is passed to `type_id()` and `type_id_with_cvr()` functions.

This leads to big strings in binary file:

```
static const char* boost::detail::ctti<T>::n() [with T = int]
static const char* boost::detail::ctti<T>::n() [with T = user_defined_type]
```

While using RTTI, you'll get the following (more compact) string in binary file:

```
i
17user_defined_type
```

RTTI emulation limitations

TypeIndex has been tested and successfully work on many compilers.

Warning

With RTTI off classes with exactly the same names defined in different modules in anonymous namespaces may collapse:

```
// In A.cpp
namespace { struct user_defined{}; }
type_index foo_a() { return type_id<user_defined>(); }

// In B.cpp
namespace { struct user_defined{}; }
type_index foo_b() { return type_id<user_defined>(); }

// In main.cpp
assert(foo_a() != foo_b()); // will fail on some compilers
```

Compilers that have that limitation: GCC, CLANG.

Define the BOOST_TYPE_INDEX_FUNCTION_SIGNATURE macro

If you get the following error during compilation

```
TypeIndex library could not detect your compiler.
Please make the BOOST_TYPE_INDEX_FUNCTION_SIGNATURE macro use
correct compiler macro for getting the whole function name.
Define BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING to correct value after that.
```

then you are using a compiler that was not tested with this library.

`BOOST_TYPE_INDEX_FUNCTION_SIGNATURE` must be defined to a compiler specific macro, that outputs the **whole** function signature including template parameters.

Fixing pretty_name() output

If the output of `boost::typeindex::ctti_type_index::type_id<int>().name()` * returns not just `int` but also a lot of text around the `int` * or does not return type at all then you are usng a compiler that was not tested with this library and you need to setup the `BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING` macro.

Here is a short instruction:

1. get the output of `boost::typeindex::ctti_type_index::type_id<int>().name()`
2. define `BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING` to `(skip_at_begin, skip_at_end, false, "")`, where
   - `skip_at_begin` is equal to characters count before the first occurrence of `int` in output
   - `skip_at_end` is equal to characters count after last occurrence of `int` in output
3. check that `boost::typeindex::ctti_type_index::type_id<int>().name_demangled()` returns "int"
4. if it does not return `int`, then define BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING to `(skip_at_begin, skip_at_end, true, "T = ")`, where

- `skip_at_begin` is equal to `skip_at_begin` at step 2
- `skip_at_end` is equal to `skip_at_end` at step 2
- `"T = "` is equal to characters that are right before the `int` in output

5. (optional, but highly recomended) create ticket with feature request to add your compiler to supported compilers list. Include parameters provided to `BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING` macro.

Consider the following example:

`boost::typeindex::ctti_type_index::type_id<int>().raw_name()` returns "const char *__cdecl boost::detail::ctti<int>::n(void)". Then you shall set `skip_at_begin` to `sizeof("const char *__cdecl boost::detail::ctti<") - 1` and `skip_at_end` to `sizeof(">::n(void)") - 1`.

```
#define BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING (39, 6, false, "")
```

Another example:

`boost::typeindex::ctti_type_index::type_id<int>().raw_name()` returns "static const char *boost::detail::ctti<int>::n() [T = int]"". Then you shall set `skip_at_begin` to `sizeof("static const char *boost::detail::ctti<") - 1` and `skip_at_end` to `sizeof("]") - 1` and last parameter of macto to "T = ".

```
#define BOOST_TYPE_INDEX_CTTI_USER_DEFINED_PARSING (39, 1, true, "T = ")
```

Mixing sources with RTTI on and RTTI off

Linking a binary from source files that were compiled with different RTTI flags is not a very good idea and may lead to a lot of surprises. However if there is a very strong need, TypeIndex library provides a solution for mixing sources: just define `BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY` macro. This would lead to usage of same type_index class (`boost::typeindex::ctti_type_index` or `boost::typeindex::stl_type_index`) all around the project.

Note

Do not forget to rebuild **all** the projects with `BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY` macro defined

You must know that linking RTTI on and RTTI off binaries may succeed even without defining the `BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY` macro, but that does not mean that you'll get a working binary. Such actions may break the One Definition Rule. Take a look at the table below, that shows how the `boost::type_index get_integer();` function will look like with different RTTI flags:

RTTI on	**RTTI off**
`boost::typeindex::stl_type_index get_integer();`	`boost::typeindex::ctti_type_index get_integer();`

Such differences are usually not detected by linker and lead to errors at runtime.

Warning

Even with `BOOST_TYPE_INDEX_FORCE_NO_RTTI_COMPATIBILITY` defined there is no guarantee that everything will be OK. Libraries that use their own workarounds for disabled RTTI may fail to link or to work correctly.

Acknowledgements

In order of helping and advising:

- Peter Dimov for writing source codes in late 2007, that gave me an idea of how to emulate RTTI.
- Agustín Bergé K-ballo for helping with docs and fixing a lot of typos.
- Niall Douglas for generating a lot of great ideas, reviewing the sources and being the review manager for the library.