

---

# Boost.Typeof

Arkadiy Vertleyb

Peder Holt

Copyright © 2004, 2005 Arkadiy Vertleyb, Peder Holt

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt) )

## Table of Contents

Motivation .....	2
Tutorial .....	4
Reference .....	7
AUTO, AUTO_TPL .....	7
COMPLIANT .....	7
INCREMENT_REGISTRATION_GROUP .....	7
INTEGRAL .....	8
LIMIT_FUNCTION_ARITY .....	8
MESSAGES .....	9
LIMIT_SIZE .....	9
REGISTER_TYPE .....	9
REGISTER_TEMPLATE .....	9
TEMPLATE .....	10
TYPEOF, TYPEOF_TPL .....	11
TYPEOF_NESTED_TYPEDEF, TYPEOF_NESTED_TYPEDEF_TPL .....	12
Other considerations and tips .....	13
Native typeof support and emulation .....	13
The three participating parties .....	14
Supported features .....	14
What needs to be registered? .....	15
Limitations .....	16
Contributed By: .....	17
Acknowledgements .....	18

# Motivation

Today many template libraries supply object generators to simplify object creation by utilizing the C++ template argument deduction facility. Consider `std::pair`. In order to instantiate this class template and create a temporary object of this instantiation, one has to supply template parameters, as well as parameters to the constructor:

```
std::pair<int, double>(5, 3.14159);
```

To avoid this duplication, STL supplies the `std::make_pair` object generator. When it is used, the types of template parameters are deduced from supplied function arguments:

```
std::make_pair(5, 3.14159);
```

For the temporary objects it is enough. However, when a named object needs to be allocated, the problem appears again:

```
std::pair<int, double> p(5, 3.14159);
```

The object generator no longer helps:

```
std::pair<int, double> p = std::make_pair(5, 3.14159);
```

It would be nice to deduce the type of the object (on the left) from the expression it is initialized with (on the right), but the current C++ syntax does not allow for this.

The above example demonstrates the essence of the problem but does not demonstrate its scale. Many libraries, especially expression template libraries, create objects of really complex types, and go a long way to hide this complexity behind object generators. Consider a nit Boost.Lambda functor:

```
_1 > 15 && _2 < 20
```

If one wanted to allocate a named copy of such an innocently looking functor, she would have to specify something like this:

```
lambda_functor<
    lambda_functor_base<
        logical_action<and_action>,
        tuple<
            lambda_functor<
                lambda_functor_base<
                    relational_action<greater_action>,
                    tuple<
                        lambda_functor<placeholder<1> >,
                        int const
                    >
                >
            >,
            lambda_functor<
                lambda_functor_base<
                    relational_action<less_action>,
                    tuple<
                        lambda_functor<placeholder<2> >,
                        int const
                    >
                >
            >
        >
    >
>
f = _1 > 15 && _2 < 20;
```

Not exactly elegant. To solve this problem (as well as some other problems), the C++ standard committee is considering a few additions to the standard language, such as `typeof/decltype` and `auto` (see <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf>).

The `typeof` operator (or `decltype`, which is a slightly different flavor of `typeof`) allows one to determine the type of an expression at compile time. Using `typeof`, the above example can be simplified drastically:

```
typeof(_1 > 15 && _2 < 20) f = _1 > 15 && _2 < 20;
```

Much better, but some duplication still exists. The `auto` type solves the rest of the problem:

```
auto f = _1 > 15 && _2 < 20;
```

The purpose of the Boost.Typeof library is to provide a library-based solution, which could be used until the language-based facility is added to the Standard and becomes widely available.

# Tutorial

To start using typeof include the typeof header:

```
#include <boost/typeof/typeof.hpp>
```

To deduce the type of an expression at compile time use the BOOST\_TYPEOF macro:

```
namespace ex1
{
    typedef BOOST_TYPEOF(1 + 0.5) type;

    BOOST_STATIC_ASSERT((is_same<type, double>::value));
}
```

In the dependent context use BOOST\_TYPEOF\_TPL instead of BOOST\_TYPEOF:

```
namespace ex2
{
    template<class T, class U>
    BOOST_TYPEOF_TPL(T() + U()) add(const T& t, const U& u)
    {
        return t + u;
    };

    typedef BOOST_TYPEOF(add('a', 1.5)) type;

    BOOST_STATIC_ASSERT((is_same<type, double>::value));
}
```

The above examples are possible because the Typeof Library knows about primitive types, such as int, double, char, etc. The Typeof Library also knows about most types and templates defined by the Standard C++ Library, but the appropriate headers need to be included to take advantage of this:

```
#include <boost/typeof/std/utility.hpp>

namespace ex3
{
    BOOST_AUTO(p, make_pair(1, 2));

    BOOST_STATIC_ASSERT((is_same<BOOST_TYPEOF(p), pair<int, int> >::value));
}
```

Here <boost/typeof/std/utility.hpp> includes <utility> and contains knowledge about templates defined there. This naming convention applies in general, for example to let the Typeof Library handle std::vector, include <boost/typeof/std/vector.hpp>, etc.

To deduce the type of a variable from the expression, this variable is initialized with, use the BOOST\_AUTO macro (or BOOST\_AUTO\_TPL in a dependent context:

```
#include <boost/typeof/std/string.hpp>

namespace ex4
{
    BOOST_AUTO(p, new int[20]);

    BOOST_STATIC_ASSERT((is_same<BOOST_TYPEOF(p), int*>::value));
}
```

Both `BOOST_TYPEOF` and `BOOST_AUTO` strip top-level qualifiers. Therefore, to allocate for example a reference, it has to be specified explicitly:

```
namespace ex5
{
    string& hello()
    {
        static string s = "hello";
        return s;
    }

    BOOST_AUTO(&s, hello());
}
```

To better understand this syntax, note that this gets expanded into:

```
BOOST_TYPEOF(hello()) &s = hello();
```

If you define your own type, the Typeof Library cannot handle it unless you let it know about this type. You tell the Typeof Library about a type (or template) by the means of "registering" this type/template.

Any source or header file where types/templates are registered has to contain the following line before any registration is done:

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()
```

After this a type can be registered:

```
namespace ex6
{
    struct MyType
    {};
}

BOOST_TYPEOF_REGISTER_TYPE(ex6::MyType)
```

The registration must be done from the context of global namespace; fully qualified type name has to be used.

Any number of types can be registered in one file, each on a separate line.

Once your type is registered, the Typeof Library can handle it in any context:

```
namespace ex6
{
    typedef BOOST_TYPEOF(make_pair(1, MyType())) type;

    BOOST_STATIC_ASSERT((is_same<type, pair<int, MyType> >::value));
}
```

A template is registered by specifying its fully qualified name, and describing its parameters. In the simplest case, when all parameters are type parameters, only their number needs to be specified:

```
namespace ex7
{
    template<class T, class U>
    struct MyTemplate
    {};
}

BOOST_TYPEOF_REGISTER_TEMPLATE(ex7::MyTemplate, 2)

namespace ex7
{
    typedef BOOST_TYPEOF(make_pair(1, MyTemplate<int, ex6::MyType>())) type;

    BOOST_STATIC_ASSERT((is_same<type,
        pair<int, MyTemplate<int, ex6::MyType> >
        >::value>));
}
```

When a template has integral template parameters, all parameters need to be described in the preprocessor sequence:

```
namespace ex8
{
    template<class T, int n>
    struct MyTemplate
    {};
}

BOOST_TYPEOF_REGISTER_TEMPLATE(ex8::MyTemplate, (class)(int))

namespace ex8
{
    typedef BOOST_TYPEOF(make_pair(1, MyTemplate<ex7::MyTemplate<ex6::MyType, int>, 0>())) type;

    BOOST_STATIC_ASSERT((is_same<type,
        pair<int, MyTemplate<ex7::MyTemplate<ex6::MyType, int>, 0> >
        >::value>));
}
```

Please see the reference for more details.

# Reference

## AUTO, AUTO\_TPL

The `BOOST_AUTO` macro emulates the proposed `auto` keyword in C++.

### Usage

```
BOOST_AUTO( var , expr )
BOOST_AUTO_TPL( var , expr )
```

### Arguments

`var`     a variable to be initialized with the expression

`expr`    a valid c++ expression

### Remarks

If you want to use `auto` in a template-context, use `BOOST_AUTO_TPL( expr )`, which takes care of the `typename` keyword inside the `auto` expression.

### Sample Code

```
int main()
{
    length::meter a(5);
    force::newton b(6);
    BOOST_AUTO(c, a * b);
}
```

## COMPLIANT

The `BOOST_TYPEOF_COMPLIANT` macro can be used to force the emulation mode. Define it if your compiler by default uses another mode, such as native `typeof` or Microsoft-specific trick, but you want to use the emulation mode, for example for portability reasons.

## INCREMENT\_REGISTRATION\_GROUP

The `BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP` macro ensures that type registrations in different header files receive unique identifiers.

### Usage

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP( )
```

### Remarks

specified once in every `cpp/hpp` file where any registration is performed, before any registration.

## Sample Code

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

class X;
BOOST_TYPEOF_REGISTER_TYPE(X)
```

## INTEGRAL

The `BOOST_TYPEOF_INTEGRAL` macro is used when registering an integral template parameter using `BOOST_TYPEOF_REGISTER_TEMPLATE`.

Useful for enums and dependent integral template parameters.

## Usage

```
BOOST_TYPEOF_INTEGRAL(x)
```

## Arguments

`x` a fully qualified integral type or enum

## Remarks

A short syntax has been implemented for the built in types (int, bool, long, unsigned long, etc.) Other non-type template parameters (e.g. pointer to member) are not supported.

## Sample Code

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

namespace foo
{
    enum color {red, green, blue};

    template<color C0, typename T1>
    class class_with_enum {};

    template<typename T0, T0 I1>
    class class_with_dependent_non_type {};
}

BOOST_TYPEOF_REGISTER_TEMPLATE(foo::class_with_enum,
    (BOOST_TYPEOF_INTEGRAL(foo::color))
    (typename)
)

BOOST_TYPEOF_REGISTER_TEMPLATE(foo::class_with_dependent_non_type,
    (typename)
    (BOOST_TYPEOF_INTEGRAL(P0))
)
```

## LIMIT\_FUNCTION\_ARITY

The `BOOST_TYPEOF_LIMIT_FUNCTION_ARITY` macro defines how many parameters are supported for functions, and applies to functions, function pointers, function references, and member function pointers. The default value is 10. Redefine if you want the Typeof Library to handle functions with more parameters.



## MESSAGES

Define `BOOST_TYPEOF_MESSAGE` before including `boost/typeof/typeof.hpp` to include messages "using typeof emulation" and "using native typeof". By default, these messages will not be displayed.

## LIMIT\_SIZE

The `BOOST_TYPEOF_LIMIT_SIZE` macro defines the size of the compile-time sequence used to encode a type. The default value is 50. Increase it if you want the Typeof Library to handle very complex types, although this possibility is limited by the maximum number of template parameters supported by your compiler. On the other hand, if you work only with very simple types, decreasing this number may help to boost compile-time performance.

## REGISTER\_TYPE

The `BOOST_TYPEOF_REGISTER_TYPE` macro informs the Typeof Library about the existence of a type

### Usage

```
BOOST_TYPEOF_REGISTER_TYPE(x)
```

### Arguments

`x` a fully qualified type

### Remarks

Must be used in the global namespace

### Sample Code

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

namespace foo
{
    class bar {};
    enum color {red, green, blue};
}

BOOST_TYPEOF_REGISTER_TYPE(foo::bar)
BOOST_TYPEOF_REGISTER_TYPE(foo::color)
```

## REGISTER\_TEMPLATE

The `BOOST_TYPEOF_REGISTER_TEMPLATE` macro informs the Typeof Library about the existence of a template and describes its parameters

### Usage

```
BOOST_TYPEOF_REGISTER_TEMPLATE(x, n)
BOOST_TYPEOF_REGISTER_TEMPLATE(x, seq)
```

### Arguments

`x` a fully qualified template

`n` the number of template arguments. Only valid if all template arguments are typenames

`seq` a sequence of template arguments. Must be used when integral or template template parameters are present

## Remarks

Must be used in the global namespace.

The library allows registration of templates with type, integral, and template template parameters:

- A type template parameter is described by the `(class)` or `(typename)` sequence element
- A template parameter of a well-known integral type can be described by simply supplying its type, like `(unsigned int)`. The following well-known integral types are supported:
  - `[signed/unsigned] char`
  - `[unsigned] short`
  - `[unsigned] int`
  - `[unsigned] long`
  - `unsigned`
  - `bool`
  - `size_t`
- Enums and typedefs of integral types, need to be described explicitly with the `BOOST_TYPEOF_INTEGRAL` macro, like `(BOOST_TYPEOF_INTEGRAL(MyEnum))`
- Template template parameters are described with the `BOOST_TYPEOF_TEMPLATE` macro, like: `(BOOST_TYPEOF_TEMPLATE((class)(unsigned int)))`. In case of all type parameters this can be shortened to something like `(BOOST_TYPEOF_TEMPLATE(2))`. The nested template template parameters are not supported.

## Sample Code

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

namespace foo
{
    template<typename T0, typename T1>
    class simple_template {};

    template<typename T0, int I1>
    class class_with_integral_constant {};
}

BOOST_TYPEOF_REGISTER_TEMPLATE(foo::simple_template, 2)
BOOST_TYPEOF_REGISTER_TEMPLATE(foo::class_with_integral_constant, (typename)(int))
```

## TEMPLATE

The `BOOST_TYPEOF_TEMPLATE` macro is used when registering template template parameters using `BOOST_TYPEOF_REGISTER_TEMPLATE`.

## Usage

```
BOOST_TYPEOF_TEMPLATE(n)
BOOST_TYPEOF_TEMPLATE(seq)
```

## Arguments

`n` the number of template arguments. Only valid if all template arguments are typenames

`seq` a sequence of template arguments. Must be used when there are integral constants in the nested template

## Remarks

Can not be used to register nested template template parameters.

## Sample Code

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

namespace foo
{
    enum color {red, green, blue};

    template<color C0, template<typename> class T1>
    class nested_template_class {};

    template<template<typename, unsigned char> class T1>
    class nested_with_integral {};
}

BOOST_TYPEOF_REGISTER_TEMPLATE(foo::nested_template_class,
    (foo::color)
    (BOOST_TYPEOF_TEMPLATE(1))
)

BOOST_TYPEOF_REGISTER_TEMPLATE(foo::nested_with_integral,
    (BOOST_TYPEOF_TEMPLATE((typename)(unsigned char)))
)
```

# typeof, typeof\_TPL

The `BOOST_TYPEOF` macro calculates the type of an expression, but removes the top-level qualifiers, `const&`

## Usage

```
BOOST_TYPEOF(expr)
BOOST_TYPEOF_TPL(expr)
```

## Arguments

`expr` a valid c++ expression that can be bound to `const T&`

## Remarks

If you want to use `typeof` in a template-context, use `BOOST_TYPEOF_TPL(expr)`, which takes care of `typename` inside the `typeof` expression.

## Sample Code

```
template<typename A, typename B>
struct result_of_conditional
{
    typedef BOOST_TYPEOF_TPL(true?A():B()) type;
};

template<typename A, typename B>
result_of_conditional<A, B>::type min(const A& a, const B& b)
{
    return a < b ? a : b;
}
```

## TYPEOF\_NESTED\_TYPEDEF, TYPEOF\_NESTED\_TYPEDEF\_TPL

The `TYPEOF_NESTED_TYPEDEF` macro works in much the same way as the 'TYPEOF' macro does, but workarounds several compiler deficiencies.

## Usage

```
BOOST_TYPEOF_NESTED_TYPEDEF(name, expr)
BOOST_TYPEOF_NESTED_TYPEDEF_TPL(name, expr)
```

## Arguments

**name**    a valid identifier to nest the `typeof` operation inside

**expr**

          a valid c++ expression that can be bound to `const T&`

## Remarks

'`typeof_nested_typedef`' nests the '`typeof`' operation inside a struct. By doing this, the '`typeof`' operation can be split into two steps, deconfusing several compilers (notably VC7.1 and VC8.0) on the way. This also removes the limitation imposed by `BOOST_TYPEOF_LIMIT_SIZE` and allows you to use '`typeof`' on much larger expressions.

If you want to use `typeof_nested_typedef` in a template-context, use `BOOST_TYPEOF_NESTED_TYPEDEF_TPL(name, expr)`, which takes care of `typename` inside the `typeof` expression.

'`typeof_nested_typedef`' can not be used at function/block scope.

## Sample Code

```
template<typename A, typename B>
struct result_of_conditional
{
    BOOST_TYPEOF_NESTED_TYPEDEF_TPL(nested, true?A():B())
    typedef typename nested::type type;
};

template<typename A, typename B>
result_of_conditional<A, B>::type min(const A& a, const B& b)
{
    return a < b ? a : b;
}
```

# Other considerations and tips

## Native typeof support and emulation

Many compilers support `typeof` already, most noticeable GCC and Metrowerks.

Igor Chesnokov discovered a method that allows to implement `typeof` on the VC series of compilers. It uses a bug in the Microsoft compiler that allows a nested class of base to be defined in a class derived from base:

```
template<int ID> struct typeof_access
{
    struct id2type; //not defined
};

template<class T, int ID> struct typeof_register : typeof_access
{
    // define base's nested class here
    struct typeof_access::id2type
    {
        typedef T type;
    };
};

//Type registration function
typeof_register<T, compile-time-constant> register_type(const T&);

//Actually register type by instantiating typeof_register for the correct type
sizeof(register_type(some-type));

//Use the base class to access the type.
typedef typeof_access::id2type::type type;
```

Peder Holt adapted this method to VC7.0, where the nested class is a template class that is specialized in the derived class.

In VC8.0, it seemed that all the bug-featire had been fixed, but Steven Watanabe managed to implement a more rigorous version of the VC7.0 fix that enables 'typeof' to be supported 'natively' here as well.

For many other compilers neither native `typeof` support nor the trick described above is an option. For such compilers the emulation method is the only way of implementing `typeof`.

According to a rough estimate, at the time of this writing the introduction of the `typeof`, `auto`, etc., into the C++ standard may not happen soon. Even after it's done, some time still has to pass before most compilers implement this feature. But even after that, there always are legacy compilers to support (for example now, in 2005, many people are still using VC6, long after VC7.x, and even VC8.0 beta became available).

Considering extreme usefulness of the feature right now, it seems to make sense to implement it at the library level.

The emulation mode seems to be important even if a better option is present on some particular compiler. If a library author wants to develop portable code using `typeof`, she needs to use emulation mode and register her types and templates. Those users who have a better option can still take advantage of it, since the registration macros are defined as no-op on such compilers, while the users for whom emulation is the only option will use it.

The other consideration applies to the users of VC7.1. Even though the more convenient `typeof` trick is available, the possibility of upgrade to VC8, where emulation remains the only option, should be considered.

The emulation mode can be forced on the compilers that don't use it by default by defining the `BOOST_TYPEOF_COMPLIANT` symbol:

```
g++ -D BOOST_TYPEOF_COMPLIANT -I \boost\boost_1_32_0 main.cpp
```

## The three participating parties

The Lambda example from the Motivation section requires the following registration:

```
#include BOOST_TYPEOF_INCREMENT_REGISTRATION_GROUP()

BOOST_TYPEOF_REGISTER_TEMPLATE(boost::tuples::tuple, 2);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::lambda_functor, 1);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::lambda_functor_base, 2);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::relational_action, 1);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::logical_action, 1);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::other_action, 1);
BOOST_TYPEOF_REGISTER_TYPE(boost::lambda::greater_action);
BOOST_TYPEOF_REGISTER_TYPE(boost::lambda::less_action);
BOOST_TYPEOF_REGISTER_TYPE(boost::lambda::and_action);
BOOST_TYPEOF_REGISTER_TEMPLATE(boost::lambda::placeholder, (int));
```

It may seem that the price for the ability to discover the expression's type is too high: rather large amount of registration is required. However note that all of the above registration is done only once, and after that, any combination of the registered types and templates would be handled. Moreover, this registration is typically done not by the end-user, but rather by a layer on top of some library (in this example -- Boost.Lambda).

When thinking about this, it's helpful to consider three parties: the typeof facility, the library (probably built on expression templates principle), and the end-user. The typeof facility is responsible for registering fundamental types. The library can register its own types and templates.

In the best-case scenario, if the expressions always consist of only fundamental types and library-defined types and templates, a library author can achieve the impression that the typeof is natively supported for her library. On the other hand, the more often expressions contain user-defined types, the more responsibility is put on the end-user, and therefore the less attractive this approach becomes.

Thus, the ratio of user-defined types in the expressions should be the main factor to consider when deciding whether or not to apply the typeof facility.

## Supported features

The Typeof library pre-registers fundamental types. For these types, and for any other types/templates registered by the user library or end-user, any combination of the following is supported:

- Pointers;
- References (except top-level);
- Consts (except top-level);
- Volatiles (except top-level);
- Arrays;
- Functions, function pointers, and references;
- Pointers to member functions;
- Pointers to data members.

For example the following type:

```
int& (*) (const volatile char*, double[5], void(*) (short))
```

is supported right away, and something like:

```
void (MyClass::*)(int MyClass::*, MyClass[10]) const
```

is supported provided MyClass is registered.

The Typeof Library also provides registration files for most STL classes/templates. These files are located in the std subdirectory, and named after corresponding STL headers. These files are not included by the typeof system and have to be explicitly included by the user, as needed:

```
#include <boost/typeof/std/functional.hpp>
BOOST_AUTO(fun, std::bind2nd(std::less<int>(), 21)); //create named function object for future use.
```

## What needs to be registered?

It is possible to take advantage of the compiler when registering types for the Typeof Library. Even though there is currently no direct support for typeof in the language, the compiler is aware of what the type of an expression is, and gives an error if it encounters an expression that has not been handled correctly. In the typeof context, this error message will contain clues to what types needs to be registered with the Typeof Library in order for BOOST\_TYPEOF to work.

```
struct X {};
```

```
template<typename A, bool B>
struct Y {};
```

```
std::pair<X, Y<int, true> > a;
```

```
BOOST_AUTO(a, b);
```

We get the following error message from VC7.1

```
error C2504: 'boost::type_of::'anonymous-namespace':::encode_type_impl<V, Type_Not_Registered_With_Typeof_System>' : base
class undefined
with
[
    V=boost::type_of::'anonymous-namespace':::encode_type_impl<boost::mpl::vector0<boost::mpl::na>, std::pair<X, Y<int, true>>>::V0,
    Type_Not_Registered_With_Typeof_System=X
]
```

Inspecting this error message, we see that the compiler complains about x

```
BOOST_TYPEOF_REGISTER_TYPE(X); //register X with the typeof system
```

Recompiling, we get a new error message from VC7.1

```
error C2504: 'boost::type_of::'anonymous-namespace':::encode_type_impl<V, Type_Not_Registered_With_Typeof_System>' : base
class undefined
with
[
    V=boost::type_of::'anonymous-namespace':::encode_type_impl<boost::mpl::vector0<boost::mpl::na>, std::pair<X, Y<int, true>>>::V1,
    Type_Not_Registered_With_Typeof_System=Y<int, true>
]
```

Inspecting this error message, we see that the compiler complains about `Y<int, true>`. Since `Y` is a template, and contains integral constants, we need to take more care when registering:

```
BOOST_TYPEOF_REGISTER_TEMPLATE(Y, (typename)(bool)); //register template class Y
```

It is a good idea to look up the exact definition of `Y` when it contains integral constants. For simple template classes containing only typenamees, you can rely solely on the compiler error.

The above code now compiles.

This technique can be used to get an overview of which types needs to be registered for a given project in order to support `typeof`.

## Limitations

Nested template template parameters are not supported, like:

```
template<template<template<class> class> class> class Tpl>  
class A; // can't register!
```

Classes and templates nested inside other templates also can't be registered because of the issue of nondeduced context. This limitation is most noticeable with regards to standard iterators in Dinkumware STL, which are implemented as nested classes. Instead, instantiations can be registered:

```
BOOST_TYPEOF_REGISTER_TYPE(std::list<int>::const_iterator)
```



## Contributed By:

- Compliant compilers -- Arkadiy Vertleyb, Peder Holt
- MSVC 6.5, 7.0, 7.1 -- Igor Chesnokov, Peder Holt

# Acknowledgements

The idea of representing a type as multiple compile-time integers, and passing these integers across function boundaries using `sizeof()`, was taken from Steve Dewhurst's article "A Bitwise `typeof` Operator", CUJ 2002. This article can also be viewed online, at <http://www.semantics.org/localarchive.html>.

Special thank you to Paul Mensonides, Vesa Karvonen, and Aleksey Gurtovoy for the Boost Preprocessor Library and MPL. Without these two libraries, this `typeof` implementation would not exist.

The following people provided support, gave valuable comments, or in any other way contributed to the library development (in alphabetical order):

- David Abrahams
- Andrey Beliaikov
- Joel de Guzman
- Daniel James
- Vesa Karvonen
- Andy Little
- Paul Mensonides
- Alexander Nasonov
- Tobias Schwinger
- Martin Wille