Toward Boost.Conversion 0.6.0

Vicente J. Botet Escriba

Table of Contents

Warning

Conversion is not a part of the Boost libraries.

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by `<>` to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.

- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of the core Conversion files
#include <boost/conversion.hpp>

using namespace boost;
```

Motivation

I've needed recently to convert from boost::chrono::time_point<Clock, Duration> to boost::posix_time::ptime and from boost::chrono::duration<Rep, Period> to boost::posix_time::time_duration. This kind of conversions are needed quite often when you use code from two different libraries that have implemented the same concept using of course different representations and have hard coded the library interface to its own implementation. Well this is a normal situation we can't avoid. Life is life.

Quite often we need to convert unrelated types `Source` and `Target`. As these classes are unrelated, neither of them offers conversion operators to the other. Usually we get it by defining a specific function such as

```
Target ConvertToTarget(Source& v);
```

In my case I started by defining

```
template <typename Rep, typename Period>
boost::posix_time::time_duration convert_to_posix_time_time_duration(
        const boost::chrono::duration<Rep, Period>& from);

template <typename Clock, typename Duration>
posix_time::ptime convert_to_posix_time_ptime(const chrono::time_point<Clock, Duration>& from);
```

Imagine now that you need to convert a `std::pair<Source, Source>` to a `std::pair<Target, Target>`. The standard defines conversions of pairs if the related types are C++ convertible:

```
template <typename T1, typename T2>
struct pair {
    ...
    template<class U, class V>
    //requires Constructible<T1, const U&> && Constructible<T2, const V&>
    std::pair(const pair<U, V>& p);

    template<class U , class V>
    //requires HasAssign<T1, const U&> && HasAssign<T2, const V&>
    std::pair& operator=(const std::pair<U , V>& p);
    ...
};
```

As the types `Target` and `Source` are not C++ convertible other than using a specific function, xe need to use a workaround.

Well we can again define a specific function

```
std::pair<Target,Target> ConvertToPairOfTarget(std::pair<Source,Source>& v) {
    return std::make_pair(ConvertToTarget(v.fisrt), ConvertToTarget(v.second));
}
```

While the `ConvertToTarget` could be specific, the `ConvertToPairOfTarget` should be generic

```
template <typename Target1, typename Target2, typename Source1, typename Source2)
std::pair<Target1,Target2> ConvertToPair(std::pair<Source1,Source2>& v);
```

In order to do that we need that the pair template parameters define a common function, let it call `convert_to`,

```
template <typename Target, typename Source)
Target convert_to(Source& v);
```

so `ConvertToPair` can be defined as

```
template <typename Target1, typename Target2, typename Source1, typename Source2)
std::pair<Target1,Target2> ConvertToPair(std::pair<Source1,Source2>& v) {
    return std::make_pair(convert_to<Target1>(v.fisrt), convert_to<Target2>(v.second));
}
```

We need to specialize the `convert_to` function for the specific classes `Source` and `Target`. We can do it as follows

```
Target  convert_to(Source& v) {return ConvertToTarget(v);}
```

In my case I needed

```
template <typename Rep, typename Period>
boost::posix_time::time_duration convert_to(const boost::chrono::duration<Rep, Period>& from)
{
        return convert_to_posix_time_time_duration(from);
}

template <typename Clock, typename Duration>
boost::posix_time::ptime convert_to(const boost::chrono::time_point<Clock, Duration>& from)
{
        return convert_to_posix_time_ptime(from);
}
```

So now I can convert

```
std::pair<chrono::time_point<Clock, Duration>, boost::chrono::duration<Rep, Period> >
```

to

```
std::pair<boost::posix_time::ptime, boost::posix_time::time_duration>
```

using the `ConvertToPair` function.

What about converting `std::pair<Source,std::pair<Source,Source> >` to `std::pair<Target,std::pair<Target,Target> >`? The issue now is that `convert_to(std::make_pair<to, std::make_pair<to,to> >)` does not compiles because the conversion of `std::pair` is named `ConvertToPair`. So we need to specialize the function `convert_to` for pairs.

```
template <typename T1, typename T2, typename S1, typename S2)
static std::pair<T1,T2> convert_to(std::pair<Source1,Source2>& from) {
{
    return std::pair<T1,T2>(convert_to<T1>(from.first), convert_to<T2>(from.second));
}
```

There is still a last point. The preceding design works well with unrelated classes, but what about classes that already define some kind of conversion, using a constructor or a conversion operator. Do we need to make specialization for these conversion? The answer is no. We need just to define the default implementation of `convert_to` function to just return the explicit conversion.

```
template < typename Target, typename Source>
Target convert_to(const Source& from)
{
    return Target(from);
}
```

What have we learned? Classes or algorithms relying on a conversion by copy-construction or by the conversion operator can be made more generic by relaying in a function that explicitly states this conversion. Thus, instead of requiring

```
Target(from)
```

we could require

```
convert_to<Target>(from)
```

The same applies to classes or algorithms relying on the assignment operator. So instead of requiring

```
to = from
```

we could use

```
assign_to(to, from);
```

The default implementation of `assign_to` relies on the assignment operator

```
template < typename Target, typename Source >
To& assign_to(Target& to, const Source& from)
{
    to = from;
    return to;
}
```

For classes that are explicitly convertible and having a self assignment operator it is easy to make a specialization of `assign_to` as follows.

```
to = convert_to<Target>(from);
```

The rationale is that if there was not a copy constructor from a Source seems reasonable to think that there will not be an assignment operator. So in most of the cases, once we have specialized the convert_to function we recover a reasonable implementation for the `assign_to` function.

When doing multiple assignments we use to do

```
a = b = c;
```

With `assign_to` we could do

```
assign_to(a, assign_to(b, c));
```

and if we find this not really readable we can try with

```
mca(a) = mca(b) = c;
```

The behavior of `mca` recall the tie function of **Boost.Tuple**, but instead of allowing multiple assignations, allows a single `assign_to` call.

We can even generalize this, so classes or algorithms relying on a member function can be made more generic by relaying on a function. The default function implementation could just to call to the member function with the equivalent prototype, but this is out of the scope of this library.

So one of the advantages of using this common functions is uniformity. The other is that now we are able to find all the explicit conversions to one type, as we can do with explicit casts.

Description

Boost.Conversion manages with generic explicit conversion between unrelated types.

The template function `convert_to` allows to convert a source type to a target type, using argument dependent lookup (ADL) to select a specialized `convert_to` function if available. If no specialized `convert_to` function is available, `boost::conversion::convert_to` is used.

The generic `convert_to` function requires that the elements to be converted are assignable and copy constructible. It is implemented using the `Target` copy construction from a `Source` or the `Source` conversion operator `Target` - this is sometimes unavailable.

For standard types, we can not add a specialized `convert_to` function on the namespace std. The alternative to using argument dependent lookup in this situation is to provide a template specialization of `boost::conversion::convert_to` for every pair of standard types that requires a specialized `convert_to`.

Boost.Conversion provides:

- a generic `convert_to` function which can be specialized by the user to make explicit conversion between unrelated types.
- a generic `assign_to` function which can be specialized by the user to make explicit assignation between unrelated types.
- a generic `mca` function returning a wrapper which replace assignments by a call to `assign_to` and conversion operators by a call `convert_to`.
- a generic `convert_to_via` function which convert a type `From` to another `To` using a temporary one `Via`.
- a generic `pack` function used to pack Source and target constructor arguments.
- conversion between `std::complex` of explicitly convertible types.
- conversion between `std::pair` of explicitly convertible types.
- conversion between `boost::optional` of explicitly convertible types.
- conversion between `boost::rational` of explicitly convertible types.
- conversion between `boost::interval` of explicitly convertible types.
- conversion between `boost::chrono::time_point` and `boost::ptime`.
- conversion between `boost::chrono::duration` and `boost::time_duration`.

- conversion between `boost::array` of explicitly convertible types.
- conversion between Boost.Fusion sequences of explicitly convertible types.
- conversion between `std::vector` of explicitly convertible types.

Users' Guide

Getting Started

Installing Conversion

Getting Boost.Conversion

You can get the last stable release of Boost.Conversion by downloading `conversion.zip` from the Boost Vault Utilities directory

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.Conversion

There is no need to compile **Boost.Conversion**, since it's a header only library. Just include your Boost header directory in your compiler include path.

Requirements

The genric part of **Boost.Conversion** depends only on Boost.Config. Of course it depends on the specific libraries when specific conversion are used.

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

Currently, **Boost.Conversion** has been tested in the following compilers/platforms:

Windows with

- MSVC 10.0

Cygwin 1.5 with

- GCC 3.4.4

Cygwin 1.7 with

- GCC 4.3.4

MinGW with

- GCC 4.4.0
- GCC 4.5.0
- GCC 4.5.0 -std=c++0x

- GCC 4.6.0
- GCC 4.6.0 -std=c++0x

Ubuntu 10.10

- GCC 4.4.5
- GCC 4.4.5 -std=c++0x
- GCC 4.5.1
- GCC 4.5.1 -std=c++0x
- clang 2.8

Note

Please let us know how this works on other platforms/compilers.

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Hello World!

Tutorial

Using generic conversions

When you need to make a generic explicit conversion or assignation you just need to include the file `boost/conversion/convert_to.hpp` or `boost/conversion/assign_to.hpp` and just use the boost conversion function.

```
#include <boost/conversion/convert_to.hpp>

// ...

int i = convert_to<int>(3.5);
```

Using specific conversions

When you need to make a specific conversion you will need to include the specific conversion file. E.g.

```
#include <boost/conversion/std/pair.hpp>

std::pair<int,int> pint(0,1);
std::pair<double,double> pdouble=boost::convert_to<std::pair<double,double> >(pint);
```

Do not forget to include this files when you use a generic class or algorithm using the generic `convert_to` or `assign_to`, otherwise your program should not compile. E.g. if you want to convert a pair of `chrono::time_point<>` to a pair of `posix_time::ptime` do not forget to include in addition to the `boost/conversion/std/pair.hpp` the file `boost/conversion/boost/chrono_posix_time.hpp`

How to specialize the conversion functions?

You can add an overload of `convert_to` and `assign_to` functions as you will do for the swap function for example, but we have to use a trick to allow ADL and avoid infinite recursion and ambiguity. This trick consists in adding a unused parameter representing the target type. E.g. if you want to add an explicit conversion from a type `A` to a type `B` do the following:

```
namespace  my_own_namespace {
    B convert_to(const A& from, boost::dummy::type_tag<B> const&);
}
```

How to partially specialize the conversion functions for standard types?

As it has been explained in the introduction, we can not use ADL for standard types, as we can not add new functions on the standad std namespace. For these types we need to specialize the `boost::conversion::convert_to` function.

```
template < typename Target, typename Source >
Target convert_to(const Source& val, boost::dummy::type_tag<Target> const&)
```

With compilers supporting partial specialization of function templates there is no major problem. For the others, we need to use a trick; as it allows partial specialization of classes we can define `convert_to` by as relying to a specific function of a class, as follows:

```
namespace  boost { namespace conversion {
    namespace partial_specialization_workaround {
        template < typename Target, typename Source >
        struct convert_to {
        static Target apply(const Source& val);
        };
    }

    template < typename Target, typename Source >
    Target convert_to(const Source& val, dummy::type_tag<Target> const&=dummy::type_tag<Tar↵
get>()) {
        return partial_specialization_workaround::convert_to<Target,Source>::apply(val);
    }
}}
```

So now we can specialize `partial_specialization_workaround::convert_to` for pairs as follows:

```
namespace partial_specialization_workaround {
    template <typename Target1, typename Target2, typename Source1, typename Source2)
    struct convert_to< std::pair<Target1,Target2>, std::pair<Source1,Source2> > {
        inline static std::pair<Target1,Target2> apply(std::pair<Source1,Source2>& v) {
        {
            return std::pair<T1,T2>(convert_to<T1>(from.first), convert_to<T2>(from.second));
        }
    };

}
```

The same applies to the generic `assign_to` function.

```
namespace partial_specialization_workaround {
    template < typename Target, typename Source >
    struct assign_to {
        inline static To& apply(Target& to, const Source& from)
        {
            to = from;
            return to;
        }
    };
}
template < typename Target, typename Source >
To& assign_to(Target& to, const Source& from, dummy::type_tag<Target> const&) {
    return partial_specialization_workaround::assign_to<Target,Source>::apply(to, from);
}
```

How to convert to types needing some constructors arguments?

Sometimes we need the conversion construct the resulting type with some arguments. This could be the case for example of std::vector, for which we need to pass an allocator to the constructor. In order to maintain the same signature, the library provides a `pack` function that will wrap the Source and the Target constructor parameters in a single parameter. So the overloading must be done on the result of this `pack` function.

Examples

chrono::time_point and posix_time::ptime

```
#ifndef BOOST_CONVERT_TO_CHRONO_TIME_POINT_TO_POSIX_TIME_PTIME_HPP
#define BOOST_CONVERT_TO_CHRONO_TIME_POINT_TO_POSIX_TIME_PTIME_HPP

#include <boost/chrono/chrono.hpp>
#include <boost/date_time/posix_time/posix_time_types.hpp>
#include <boost/date_time/posix_time/conversion.hpp>
#include <boost/conversion/convert_to.hpp>
#include <boost/conversion/assign_to.hpp>
#include <boost/config.hpp>

namespace boost {
    #ifdef BOOST_NO_FUNCTION_TEMPLATE_ORDERING
    namespace conversion { namespace partial_specialization_workaround {
        template < class Clock, class Duration>
        struct convert_to<posix_time::ptime, chrono::time_point<Clock, Duration> > {
          inline static posix_time::ptime apply(const chrono::time_point<Clock, Duration>& from)
            {
                typedef chrono::time_point<Clock, Duration> time_point_t;
                typedef chrono::nanoseconds duration_t;
                typedef duration_t::rep rep_t;
                rep_t d = chrono::duration_cast<duration_t>(from.time_since_epoch()).count();
                rep_t sec = d/1000000000;
                rep_t nsec = d%1000000000;
                return  posix_time::from_time_t(0)+
                        posix_time::seconds(static_cast<long>(sec))+
#ifdef BOOST_DATE_TIME_HAS_NANOSECONDS
                        posix_time::nanoseconds(nsec);
#else
                        posix_time::microseconds((nsec+500)/1000);
#endif
            }
        };
        template < class Clock, class Duration>
        struct assign_to<posix_time::ptime, chrono::time_point<Clock, Duration> > {
            inline static posix_time::ptime& ap↵
ply(posix_time::ptime& to, const chrono::time_point<Clock, Duration>& from)
            {
                to = boost::convert_to<posix_time::ptime>(from);
                return to;
            }
        };

        template < class Clock, class Duration>
        struct convert_to<chrono::time_point<Clock, Duration>, posix_time::ptime> {
          inline static chrono::time_point<Clock, Duration> apply(const posix_time::ptime& from)
            {
              posix_time::time_duration const time_since_epoch=from-posix_time::from_time_t(0);
                chrono::time_point<Clock, Duration> t=chrono::sys↵
tem_clock::from_time_t(time_since_epoch.total_seconds());
                long long nsec=time_since_epoch.fraction↵
al_seconds()*(1000000000/time_since_epoch.ticks_per_second());
                return  t+chrono::duration_cast<Duration>(chrono::nanoseconds(nsec));
            }
        };
        template < class Clock, class Duration>
        struct assign_to<chrono::time_point<Clock, Duration>, posix_time::ptime> {
          inline static chrono::time_point<Clock, Duration>& apply(chrono::time_point<Clock, Dur↵
ation>& to, const posix_time::ptime& from)
```

```
            {
                to = boost::convert_to<chrono::time_point<Clock, Duration> >(from);
                return to;
            }
        };
    }}
    #else
    namespace chrono {
        template < class Clock, class Duration>
        inline posix_time::ptime convert_to(const chrono::time_point<Clock, Duration>& from
                    , boost::dummy::type_tag<posix_time::ptime> const&)
        {
                typedef chrono::time_point<Clock, Duration> time_point_t;
                typedef chrono::nanoseconds duration_t;
                typedef duration_t::rep rep_t;
                rep_t d = chrono::duration_cast<duration_t>(from.time_since_epoch()).count();
                rep_t sec = d/1000000000;
                rep_t nsec = d%1000000000;
                return  posix_time::from_time_t(0)+
                        posix_time::seconds(static_cast<long>(sec))+
#ifdef BOOST_DATE_TIME_HAS_NANOSECONDS
                        posix_time::nanoseconds(nsec);
#else
                        posix_time::microseconds((nsec+500)/1000);
#endif
        }


        template < class Clock, class Duration>
        inline chrono::time_point<Clock, Duration>& assign_to(chrono::time_point<Clock, Dura↵
tion>& to, const posix_time::ptime& from
                        , boost::dummy::type_tag<chrono::time_point<Clock, Duration> > const&
        )
        {
            to = boost::convert_to<chrono::time_point<Clock, Duration> >(from);
            return to;
        }

    }

    namespace posix_time {
        template < class TP>
        inline TP convert_to(const ptime& from
                , boost::dummy::type_tag<TP > const&)
        {
            time_duration const time_since_epoch=from-from_time_t(0);
            TP t=chrono::system_clock::from_time_t(time_since_epoch.total_seconds());
            long long nsec=time_since_epoch.fraction↵
al_seconds()*(1000000000/time_since_epoch.ticks_per_second());
            return  t+chrono::duration_cast<typename TP::duration>(chrono::nanoseconds(nsec));
        }


        template < class Clock, class Duration>
        inline ptime& assign_to(ptime& to, const chrono::time_point<Clock, Duration>& from
                    , boost::dummy::type_tag<posix_time::ptime> const&
        )
        {
            to = boost::convert_to<ptime>(from);
            return to;
        }
```

```
    }
    #endif
}

#endif
```

boost::optional

```
#ifndef BOOST_CONVERT_TO_OPTIONAL_HPP
#define BOOST_CONVERT_TO_OPTIONAL_HPP

#include <boost/optional.hpp>
#include <boost/none.hpp>
#include <boost/conversion/convert_to.hpp>
#include <boost/conversion/assign_to.hpp>
#include <boost/config.hpp>

namespace boost {

    #ifdef BOOST_NO_FUNCTION_TEMPLATE_ORDERING
    namespace conversion { namespace partial_specialization_workaround {
        template < class Target, class Source>
        struct convert_to< optional<Target>, optional<Source> > {
            inline static optional<Target> apply(optional<Source> const & from)
            {
              return (from?optional<Target>(boost::convert_to<Target>(from.get())):optional<Tar↵
get>());
            }
        };
        template < class Target, class Source>
        struct assign_to< optional<Target>, optional<Source> > {
            inline static optional<Target>& apply(optional<Target>& to, const option↵
al<Source>& from)
            {
                to = from?boost::convert_to<Target>(from.get()):optional<Target>();
                return to;
            }
        };

    }}
    #else
    template < class Target, class Source>
    inline optional<Target> convert_to(optional<Source> const & from
                , boost::dummy::type_tag<optional<Target> > const&)
    {
       return (from?optional<Target>(boost::convert_to<Target>(from.get())):optional<Target>());
    }

    template < class Target, class Source>
    inline optional<Target>& assign_to(optional<Target>& to, const optional<Source>& from
                , boost::dummy::type_tag<optional<Target> > const&
                )
    {
        to = from?boost::convert_to<Target>(from.get()):optional<Target>();
        return to;
```

```
    }
    #endif
}

#endif
```

std::pair

```
#ifndef BOOST_CONVERT_TO_PAIR_HPP
#define BOOST_CONVERT_TO_PAIR_HPP

#include <utility>
//#include <boost/conversion/convert_to.hpp>
#include <boost/conversion/assign_to.hpp>

namespace boost { namespace conversion {

    // std namespace can not be overloaded
    namespace partial_specialization_workaround {
        template < class T1, class T2, class U1, class U2>
        struct convert_to< std::pair<T1,T2>, std::pair<U1,U2> > {
            inline static std::pair<T1,T2> apply(std::pair<U1,U2> const & from)
            {
                return std::pair<T1,T2>(boost::convert_to<T1>(from.first), boost::con↵
vert_to<T2>(from.second));
            }
        };
        template < class T1, class T2, class U1, class U2>
        struct assign_to< std::pair<T1,T2>, std::pair<U1,U2> > {
            inline static std::pair<T1,T2>& ap↵
ply(std::pair<T1,T2>& to, const std::pair<U1,U2>& from)
            {
                to.first = boost::convert_to<T1>(from.first);
                to.second = boost::convert_to<T2>(from.second);
                return to;
            }
        };
    }
}}

#endif
```

External Resources

Boost.Convert	Vladimir Batov. Not yet reviewed
Boost.Conversion.LexicalCast	general literal text conversions, such as an int represented as a string, or vice-versa from Kevlin Henney
Boost.NumericConversion	Optimized Policy-based Numeric Conversions from Fernando Cacciola.
N2380 - Explicit Conversion Operator Draft Working Paper (revision 2)	Lois Goldthwaite, Michael Wong, Jens Mauer, Alisdair Meredith.
N2200 - Operator Overloading	Gary Powell, Doug Gregor, Jaakko Jarvi.

N1671 - Overloading operator.() & operator.*()	Gary Powell, Doug Gregor, Jaakko Jarvi.
N1676 - Non-member overloaded copy assignment operator	Bronek Kozicki.
N1694 - A Proposal to Extend the Function Call Operator	Bronek Kozicki.

Reference

Core

Header <boost/conversion.hpp>

Include all the core conversion public header files. Note that you will need to include explicitly the C++ standard or Boost specific files when using specific classes.

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

Header <boost/conversion/include.hpp>

Include all the core conversion public header files. Note that you will need to include explicitly the C++ standard or Boost specific files when using specific classes.

```
#include <boost/conversion/convert_to.hpp>
#include <boost/conversion/assign_to.hpp>
#include <boost/conversion/convert_to_via.hpp>
#include <boost/conversion/ca_wrapper.hpp>
#include <boost/conversion/pack.hpp>
```

Header <boost/conversion/convert_to.hpp>

```
namespace boost {
    namespace dummy {
        template <typename T> struct base_tag {};
        template <typename T> struct type_tag : public base_tag<T> {};
    }
    template <typename T> struct type_tag;
    namespace conversion {
        namespace partial_specialization_workaround {
            template < typename To, typename From >
            struct convert_to {
                static To apply(const From& val);
            };
        }

        template < typename To, typename From >
        To convert_to(const From& from, dummy::type_tag<To> const&);
    }

    template <typename Target, typename Source>
    Target convert_to(Source const& from, dummy::base_tag<To> const&=dummy::base_tag<To>());
}
```

Defines a free function `convert_to` which converts the `from` parameter to a `To` type. The default implementation applies the conversion `To` operator of the `From` class or the copy constructor of the `To` class. Of course if both exist the conversion is ambiguous. A user adapting another type could need to specialize the `convert_to` free function if the default behavior is not satisfactory.

The user can add the `convert_to` overloading on the namespace of a specific Source. But sometimes as it is the case for the standard classes, we can not add new functions on the std namespace, so we need a different technique.

The technique consists in partially specialize on the function `convert_to` on the `boost::conversion` namespace. For compilers for which we can not partially specialize a function a trick is used: instead of calling directly to the `convert_to` member function, `convert_to` calls to the static operation `apply` on a class with the same name in the namespace __partial_specializationworkaround _. Thus the user can specialize partially this class.

Function `convert_to()`

```
template < typename To, typename From >
To convert_to(const From& from, dummy::type_tag<To> const&);
```

Effects: Converts the from parameter to an instance of the To type, using by default the conversion operator or copy constructor.

Throws: Whatever the underlying conversion `To` operator of the `From` class or the copy constructor of the `To` class throws.

Header `<boost/conversion/assign_to.hpp>`

```
namespace boost {
    namespace conversion {
        namespace partial_specialization_workaround {
            template < typename To, typename From >
            struct assign_to {
                static void apply(To& to, const From& from);
            };
            template < typename To, typename From, std::size_t N  >
            struct assign_to<To[N],From[N]> {
                static void apply(To (&to)[N], const From(& from)[N]);
            };
        }

        template < typename To, typename From >
        void assign_to(To& to, const From& from, dummy::type_tag<To> const&);
    }

    template <typename Target, typename Source>
    Target& assign_to(Tar↵
get& to, const Source& from, dummy::base_tag<To> const&=dummy::base_tag<To>()) {
        return ::boost_conversion_impl::assign_to_impl<Target, Source>(to, from);
    }
}
```

Defines a free function `assign_to` which assigns the `from` parameter to the `to` parameter. The default implementation applies the the assignment operator of the `To` class. A user adapting another type could need to specialize the `assign_to` free function if the default behavior is not satisfactory.

The user can add the `assign_to` overloading on the namespace of the Source or Target classes. But sometimes as it is the case for the standard classes, we can not add new functions on the std namespace, so we need a different technique.

The technique consists in partially specialize on the function `assign_to` on the `boost::conversion` namespace. For compilers for which we can not partially specialize a function a trick is used: instead of calling directly to the `assign_to` member function, `assign_to` calls to the static operation apply on a class with the same name in the namespace __partial_specializationworkaround _. Thus the user can specialize partially this class.

Function `assign_to()`

```
template < typename To, typename From >
void assign_to(To& to, const From& from);
```

Effects: Assigns the `from` parameter to the `to` parameter, using by default the assignment operator of the `To` class.

Throws: Whatever the underlying the assignment operator of the `To` class throws.

Header `<boost/conversion/convert_to_via.hpp>`

```
namespace boost {

    template < typename To, typename Via, typename From >
    To convert_to_via(const From& val) {
        convert_to<To>(convert_to<Via>(val));
    }
}
```

Function `convert_to_via<>()`

Effects: Converts the from parameter to an instance of the To type using a intermediary Via type.

Throws: Whatever the underlying conversions functions throw.

Header `<boost/conversion/ca_wrapper.hpp>`

```
namespace boost {
    template <typename T> 'implementation_dependent<T>' mca(T& r);
}
```

Function `mca<>()`

```
template <typename T> 'implementation_dependent<T>' mca(T& r);
```

Effects: Returns a implementation dependent class able to transform conversion by `convert_to` calls and assignments by `assign_to` calls.

Throws: Nothing.

Template Class `implementation_dependent`

```
template <typename T>
class implementation_dependent {
public:
    implementation_dependent(T& r);
    implementation_dependent(implementation_dependent const& r);
    template <typename U> operator U();
    template <typename U> T& operator =(implementation_dependent<U> const& u);
    template <typename U> T& operator =(U const& u);
};
```

Constructor

```
implementation_dependent(T& r);
```

Effects: Stores the reference to type.

Throws: Nothing

Conversion operator

```
template <typename U> operator U();
```

Effects: Call to the `convert_to<U>` on the stored reference.

Throws: Whatever `convert_to` throws.

Assignment operator

```
template <typename U> T& operator =(U const& u);
```

Effects: Call to the `assign_to<T,U>` with the stored reference and the passed parameter `u`.

Throws: Whatever `assign_to` throws.

Header `<boost/conversion/pack.hpp>`

The result of the `pack` function is equivalent to a fusion sequence containing reference_warpper's instead of C++ reference (&) as this are not allowed.

```
namespace boost { namespace conversion {
    namespace result_of
    template <typename T1, typename T2, ...>
    struct pack {
        typedef fusion::sequence<
            reference_wrapper<T1>, reference_wrapper<T2, ...>
        > type;
    };
    }
    template <typename T1, typename T2, ...>
    typename result_of_pack<T1 const, T2 const>::type pack(
            T1 const& t1, T2 const& t2);

    template <typename T1, typename T2, ...>
    typename result_of_pack<T1 const, T2, ...>::type pack(T1 const& t1, T2 & t2, ...);

    ...

} }
```

Function `pack<>()`

```
template <typename T1, typename T2>
typename result_of_pack<T1 const, T2 const>::type pack(
        T1 const& t1, T2 const& t2);

template <typename T1, typename T2>
typename result_of_pack<T1 const, T2>::type pack(
        T1 const& t1, T2 & t2);
```

Effects: Returns a packed type from the template parameters.

Throws: Nothing.

C++ Standard classes specializations

Header `<boost/conversion/std/complex.hpp>`

Include this file when using conversions between complex of convertible types.

```
namespace boost {
    namespace conversion {
    namespace partial_specialization_workaround {
        template < class T, class U>
        struct convert_to< std::complex<T>, std::complex<U> > {
            static std::complex<T> apply(std::complex<U> const & from);
        };
        template < class T, class U>
        struct convert_to< std::complex<T>, std::complex<U> > {
            static std::complex<T>& apply(const std::complex<U>& from, std::complex<T>& to);
        };

    }}
}
```

Header `<boost/conversion/std/pair.hpp>`

Include this file when using conversions between pairs of convertible types.

```
namespace boost {
    namespace conversion {
    namespace partial_specialization_workaround {
        template < class T1, class T2, class U1, class U2>
        struct convert_to< std::pair<T1,T2>, std::pair<U1,U2> > {
            static std::pair<T1,T2> apply(std::pair<U1,U2> const & from);
        };
        template < class T1, class T2, class U1, class U2>
        struct assign_to< std::pair<T1,T2>, std::pair<U1,U2> > {
            static std::pair<T1,T2>& apply(std::pair<T1,T2>& to, const std::pair<U1,U2>& from);
        };
    }}
}
```

Header `<boost/conversion/std/vector.hpp>`

Include this file when using conversions between std::vector of convertible types.

```
namespace boost {
    namespace conversion {
    namespace partial_specialization_workaround {
        template < class T1, class A1, class T2, class A2>
        struct convert_to< std::vector<T1,A1>, std::vector<T2,A2> > {
            inline static std::vector<T1,A1> apply(std::vector<T2,A2> const & from);
        };
        template < class T1, class A1, class T2, class A2>
        struct assign_to< std::vector<T1,A1>, std::vector<T2,A2> > {
            inline static std::vector<T1,A1>& apply(
                std::vector<T1,A1>& to,
                std::vector<T2,A2> const & from);
        };

        template < class T1, class A1, class T2, class A2>
        struct convert_to< std::vector<T1,A1>,
                typename result_of_pack<std::vector<T2,A2> const, A1 const>::type
            > {
            inline static std::vector<T1,A1> apply(
               typename result_of_pack<std::vector<T2,A2> const, A1 const>::type const & pack);
        };
    }}
}
```

Header <boost/conversion/std/string.hpp>

Include this file when using conversions to std::string.

```
namespace boost {
    namespace conversion {
    namespace partial_specialization_workaround {
        template<typename T, typename CharT, typename Traits, typename Alloc>
        struct convert_to< std::basic_string<CharT,Traits,Alloc>, T > {
            static std::basic_string<CharT,Traits,Alloc> apply(T const & from);
        }
        template<typename T, typename CharT, typename Traits, typename Alloc>
        struct assign_to< std::basic_string<CharT,Traits,Alloc>, T > {
            static std::basic_string<CharT,Traits,Alloc>& apply(
                std::basic_string<CharT,Traits,Alloc>& to, const T& from);
        };
        template<typename T, typename CharT, typename Traits, typename Alloc>
        struct convert_to< T, std::basic_string<CharT,Traits,Alloc>> {
            static T apply(std::basic_string<CharT,Traits,Alloc> const & from);
        };
        template<typename T, typename CharT, typename Traits, typename Alloc>
        struct assign_to< T, std::basic_string<CharT,Traits,Alloc>> {
            static void apply(T& to
                const std::basic_string<CharT,Traits,Alloc>& from);
        };
    }}
}
```

Boost classes specializations

Header <boost/conversion/boost/rational.hpp>

Include this file when using conversions between rational of convertible types.

```
namespace boost {
    template < class T, class U>
    rational<T> convert_to(rational<U> const & from
                        , boost::dummy::type_tag<rational<T> > const&);

    template < class T, class U>
    rational<T>& assign_to(rational<T>& to, const rational<U>& from
                        , boost::dummy::type_tag<rational<T> > const&);
}
```

Header <boost/conversion/boost/chrono_posix_time.hpp>

Include this file when using conversions between chrono and posix_time time and duration types.

```
namespace boost {
namespace chrono {
    template < class Clock, class Duration>
    posix_time::ptime convert_to(const chrono::time_point<Clock, Duration>& from
                , boost::dummy::type_tag<posix_time::ptime> const&);

    template < class Clock, class Duration>
    chrono::time_point<Clock, Duration>& assign_to(chrono::time_point<Clock, Dura↵
tion>& to, const posix_time::ptime& from
                      , boost::dummy::type_tag<chrono::time_point<Clock, Duration> > const&);

}
namespace posix_time {
    template < class Clock, class Duration>
    chrono::time_point<Clock, Duration> convert_to(const ptime& from
            , boost::dummy::type_tag<chrono::time_point<Clock, Duration> > const&);

    template < class Clock, class Duration>
    ptime& assign_to(ptime& to, const chrono::time_point<Clock, Duration>& from
                , boost::dummy::type_tag<posix_time::ptime> const&);
}

namespace chrono {
    template < class Rep, class Period>
    inline posix_time::time_duration convert_to(chrono::duration<Rep, Period> const & from
                      , boost::dummy::type_tag<posix_time::time_duration> const&);

    template < class Rep, class Period>
    inline chrono::duration<Rep, Period> & assign_to(chrono::duration<Rep, Peri↵
od> & to, const posix_time::time_duration& from
                        , boost::dummy::type_tag<chrono::duration<Rep, Period> > const&);
}

namespace posix_time {
    template < class Rep, class Period>
    inline chrono::duration<Rep, Period> convert_to(time_duration const & from
                        , boost::dummy::type_tag<chrono::duration<Rep, Period> > const&);

    template < class Rep, class Period>
   inline time_duration& assign_to(time_duration& to, const chrono::duration<Rep, Period>& from
                      , boost::dummy::type_tag<posix_time::time_duration> const&);
}
}
```

Header <boost/conversion/boost/interval.hpp>

Include this file when using conversions between intervals of convertible types.

```
namespace boost {
namespace numeric {
    template < class T, class PT, class U, class PU>
    interval<T,PT> convert_to(interval<U,PU> const & from
                    , boost::dummy::type_tag<interval<T,PT> > const&);

    template < class T, class PT, class U, class PU>
    inline interval<T,PT>& assign_to(interval<T,PT>& to, const interval<U,PU>& from
                    , boost::dummy::type_tag<interval<T,PT> > const&);
}
}
```

Header <boost/conversion/boost/optional.hpp>

Include this file when using conversions between optional of convertible types.

```
namespace boost {
    template < class Target, class Source>
    optional<Target> convert_to(optional<Source> const & from
                , boost::dummy::type_tag<optional<Target> > const&);

    template < class Target, class Source>
    optional<Target>& assign_to(optional<Target>& to, const optional<Source>& from
                , boost::dummy::type_tag<optional<Target> > const&);
}
```

Header <boost/conversion/boost/array.hpp>

Include this file when using conversions between arrays of convertible types.

```
namespace boost {
    template < typename T1, typename T2, std::size_t N>
    array<T1,N> convert_to(array<T2,N> const & from
                        , boost::dummy::type_tag<array<T1,N> > const&);

    template < typename T1, typename T2, std::size_t N>
    array<T1,N>& assign_to(array<T1,N>& to, array<T2,N> const & from
    , boost::dummy::type_tag<array<T1,N> > const&);
}
```

Header <boost/conversion/boost/tuple.hpp>

Include this file when using conversions between fusion::tuple of convertible types.

```
namespace boost {
namespace fusion {
    template < class T1, class T2, class U1, class U2>
    tuple<T1,T2> convert_to(tuple<U1,U2> const & from
                    , boost::dummy::type_tag<tuple<T1,T2> > const&);

    template < class T1, class T2, class U1, class U2>
    tuple<T1,T2>& assign_to(tuple<T1,T2>& to, tuple<U1,U2> const & from
                    , boost::dummy::type_tag<tuple<T1,T2> > const&);

    template < class T1, class T2, class T3, class U1, class U2, class U3>
    tuple<T1,T2,T3> convert_to(tuple<U1,U2,U3> const & from
                    , boost::dummy::type_tag<tuple<T1,T2,T3> > const&);

    template < class T1, class T2, class T3, class U1, class U2, class U3>
    tuple<T1,T2,T3> assign_to(tuple<T1,T2,T3>& to, boost::fusion::tuple<U1,U2,U3> const & from
                    , boost::dummy::type_tag<tuple<T1,T2,T3> > const&);
}
}
```

Appendices

Appendix A: History

Version 0.5.1, Fefruary 20, 2011

Bug

- Fix bug on chain mcs assignement.

Version 0.5.0, May 30, 2010

New Features:

- Added a pack funcrion able to pack the Source and the Target constructor arguments in one parameter.
- Added conversion between std::vector of explicitly convertible types.
- Added is_convertible_to metafunction. Inherits: If an imaginary lvalue of type From is convertible to type To using convert_to then inherits from true_type, otherwise inherits from false_type.

```
template <class From, class To>
struct is_convertible : public true_type-or-false_type {};
```

- Added is_asignable_to metafunction.

Version 0.4.0, October 27, 2009

Applying the same technique that boost::swap applies making use of ADL

New Features:

A `Source` class is convertible to a `Target` class if:

- Either: A function with the signature `convert_to<Target>(Source const&, boost::dummy::type_tag<To> const&)` is available via argument dependent lookup
- Or: A template specialization of `boost::conversion::convert_to<Target, Source>` exists for `Target` and `Source`

- Or: `Target` is copy constructible from `Source` (default implementation)

Version 0.3.0, October 22, 2009

Changing the order of `to` and `from` parameters on `assign_to` function + Added `mca` function

Incompatibility:

- Changing the order of `to` and `from` parameters on `assign_to`.
- Now `boost/conversion/convert_to.hpp` and `boost/conversion/assign_to.hpp` files are separated.

New Features:

- Added `<boost/conversion.hpp>` global file.
- Added `mca()` function.
- Added `convert_to_via` function.

Test:

- Added test for the new features

Version 0.2.0, Mai 16, 2009

Adding array + fusion::tuples + Adaptation to Boost 1.39

New Features:

- conversion between `boost::array` of explicitly convertible types.
- conversion between Boost.Fusion sequences of explicitly convertible types.

Version 0.1.0, April 16, 2009

Announcement of Conversions

Features:

- a generic `convert_to` function which can be specialized by the user to make explicit conversion between unrelated types.
- a generic `assign_to` function which can be specialized by the user to make explicit assignation between unrelated types.
- conversion between C-arrays of explicitly convertible types.
- conversion between `std::complex` of explicitly convertible types.
- conversion between `std::pair` of explicitly convertible types.
- conversion between `std::string` and Streamable types.
- conversion between `boost::optional` of explicitly convertible types.
- conversion between `boost::rational` of explicitly convertible types.
- conversion between `boost::interval` of explicitly convertible types.
- conversion between `boost::chrono::time_point` and `boost::ptime`.
- conversion between `boost::chrono::duration` and `boost::time_duration`.

Appendix B: Rationale

Trick to avoid recursion on the convert_to calls

The implementation of this utility contains various workarounds:

- `conversion_impl` is put outside the boost namespace, to avoid infinite recursion (causing stack overflow) when converting objects of a primitive type.
- `conversion_impl` has a using-directive `using namespace boost::conversion;`, rather than a using-declaration, because some compilers (including MSVC 7.1, Borland 5.9.3, and Intel 8.1) don't do argument-dependent lookup when it has a using-declaration instead.
- `boost::convert_to` has an additional template argument, a tag, to avoid ambiguity between the `boost::conversion::convert_to` and `boost::convert_to` and the when converting from objects of a Boost type that does not have its own `boost::convert_to` overload. This additional argument is a reference to a base tag class `dummy::base_tag<Target> const&` for the `boost::convert_to` and a reference derived tag class `dummy::type_tag<To> const&` for all others.

```
namespace dummy {
    template <typename T> struct base_tag {};
    template <typename T> struct type_tag : public base_tag<T> {};
}
```

In this way

```
template <typename Target, typename Source>
Target boost::convert_to(Source const& from, dummy::base_tag<Target> const& p=dummy::base_tag<Tar↵
get>()) {
```

would be never chosen when called in this context

```
using namespace boost::conversion;
return convert_to(from, dummy::type_tag<Target>());
```

as the library defines

```
namespace conversion {
    template < typename To, typename From >
    To boost::convert_to(const From& val, dummy::type_tag<To> const&);
}
```

Trick to avoid the use of the tag on the user side

The tag type is there to avoid infinite recursion, but it is quite cumbersome at the user side.

```
a = convert_to(b, dummy::type_tag<A>());
```

To avoid to pass it as parameter the tag parameter has a default value boost::dummy::base_tag<Target>().

```
template <typename Target, typename Source>
Target boost::convert_to(Source const& from, boost::dummy::base_tag<Tar↵
get> const& p=boost::dummy::base_tag<Target>()) {
```

This default value needs however to give the Target template parameter

```
a= convert_to<A>(b);
```

Mathematical background

Let be

```
A a,a2;
B b;
C c;
```

- Reflexive: A is convertible to A if it is CopyConstructible or a specialization of convert_to is provided.
- Anti-Symetric : A convertible to B don't implies B convertible to A
- Loss of precision: Conversions can loss precision but not at infinitum

Two convertible types don't loss precision if

```
b = convert_to<B>(a);
a2 = convert_to<A>(b);
assert(a==a2);
```

If they can loss precision they satisfy

```
b = convert_to<B>(a)
a2 = convert_to<A>(b)
assert(a==a2 || ct(a2)==b
```

- Transitive: A convertible to B && B convertible to C implies A convertible to C

The implementation could use a intermediary B b to make the conversion or make the conversion directly.

```
template <>
convert_to<C,A>(const C& c) {
    return convert_to<A>(convert_to<B>(c));
}
```

The library provides a convert_to_via function which helps to implement that.

Ambiguity of multiple overloadins

Warning

Ambiguity of multiple overloadins

Appendix C: Implementation Notes

Why `convert_to` between tuples is not implemented using `boost::fusion::transform`?

`convert_to<T>` is a kind of transformation, so the natural implementation of `convert_to` for homogeneous containers could be to use the transform function.

This can not be applied to heterogeneous containers as tuples because the function change with the type.

Appendix D: Acknowledgements

Thanks to Vladimir Batov proposing Boost.StringConversion which was the source of inspiration of this generic library. Thanks to Edward Diener to showing me indirectly that multiple assignments should be taken in account. Thanks to Jeffrey Hellrung to showing me that Boost.Conversion should use also ADL.

Appendix E: Tests

Builtins

Name	kind	Description	Result	Ticket
convert_to_with_builtin_types	run	check `convert_to` works for builting types	Pass	#
assign_to_with_builtin_types	run	check `assign_to` works for builtin types	Pass	#
assign_to_transitive	run	Use of `assign_to` transitively	Pass	#
mca_assign_to_with_builtin_types	run	check `mca works` for builtin types	Pass	#
mca_assign_to_transitive	run	use of `mca` to multiple assignments	Pass	#

Intrinsic Conversions

Name	kind	Description	Result	Ticket
convert_to_with_implicit_constructor	run	check `convert_to` works when there is an implicit constructor	Pass	#
convert_to_with_explicit_constructor	run	check `convert_to` works when there is an explicit constructor	Pass	#
convert_to_with_conversion_operator	run	check `assign_to` works when there is an conversion operator	Pass	#
assign_to_with_assignment_operator	run	check `assign_to` works when there is an assignment operator	Pass	#
assign_to_with_assignment_operator_and_implicit_constructor	run	check `assign_to` works when there is an assignment operator and implicit constructor	Pass	#
assign_to_with_assignment_operator_and_conversion_operator	run	check `convert_to` works when there is an assignment operator and a conversion operator	Pass	#
mca_with_assignment_operator	run	check `mca` works when there is an assignment operator	Pass	#
mca_with_assignment_operator_and_implicit_constructor	run	check `mca` works when there is an assignment operator and implicit constructor	Pass	#
mca_with_assignment_operator_and_conversion_operator	run	check `mca` works when there is an assignment operator and a conversion operator	Pass	#

Extrinsic Conversions

Name	kind	Description	Result	Ticket
explicit_convert_to	run	check `convert_to` works when `convert_to` is overloaded	Pass	#
explicit_assign_to	run	check `assign_to` works when `assign_to` is overloaded	Pass	#
explicit_mca	run	check `mca` works when `assign_to` is overloaded	Pass	#

Std

Name	kind	Description	Result	Ticket
convert_to_pair	run	check `convert_to` `std::pair` works when the parameters are convertible	Pass	#
convert_to_complex	run	check `convert_to` `std::complex` works when the parameters are convertible	Pass	#
convert_to_vector	run	check `convert_to` `std::vector` works when the parameters are convertible	Pass	#
convert_to_string	run	check `convert_to` `std::string` works when the parameter defines the `operator<<`	Pass	#
convert_from_string	run	check `convert_to` from `std::string` works when the parameter defines the `operator>>`	Pass	#

Boost

Name	kind	Description	Result	Ticket
convert_to_rational	run	check `convert_to` `boost::rational` works when the parameters are convertible	Pass	#
convert_to_interval	run	check `convert_to` `boost::interval` works when the parameters are convertible	Pass	#
convert_to_optional	run	check `convert_to` `boost::optional` works when the parameters are convertible	Pass	#
convert_to_time_point	run	check `convert_to` `boost::chrono::system_clock::time_point` from boost::posix_time::ptime works	Pass	#
convert_to_ptime	run	check `convert_to` `boost::posix_time::ptime` from boost::chrono::system_clock::time_point works	Pass	#
convert_to_duration	run	check `convert_to` `boost::chrono::duration` from `boost::posix_time::time_duration` works	Pass	#
convert_to_time_duration	run	check `convert_to` `boost::posix_time::time_duration` from `boost::chrono::duration` works	Pass	#
convert_to_array	run	check `convert_to` `boost::array` works when the parameters are convertible	Pass	#
convert_to_tuple	run	check `convert_to` `boost::tuple` works when the parameters are convertible	Pass	#

Appendix F: Tickets

Appendix F: Future plans

Tasks to do before review

For later releases

- conversion between types for which lexical_cast works.
- conversion between types for which numeric_cast works.

Make a proposal to the C++ standard

C++1x has added explicit conversion operators, but they must always be defined in the source class. The same applies to the assignment operator, it must be defined on the Target class.

What it will interesting is to be able to add constructors and assignments operators to the class std::pair, so we can say that two pairs are convertible if the parameters are explicitly convertible using a convert_to function

```
template<class U , class V>
//requires HasConvertTo<T1, const U&> && HasConvertTo<T2, const V&>
std::pair& operator=(const std::pair<U , V>& p) {
    return std::make_pair(convert_to<T1>(p.first), convert_to<T2>(p.second));
}
```

But this is not possible. We can not add operations to a class.

Another possibility could be to make an evolution to the standard, so the convertible concept takes car of extrinsic conversions. We could be able to implicitly or explicitly add extrinsic conversion operators between unrelated types. Non-member assignment operators could also be specialized.

```
template < typename To, typename From >
operator To(const From& val);

template < typename To, typename From >
To& operator=(To& to, const From& from);
```

For example we could define the explicit conversion from as chrono::time_point<Clock, Duration> to posix_time::ptime follows

```
template < class Clock, class Duration>
explicit operator boost::posix_time::ptime(const boost::chrono::time_point<Clock, Dura↵
tion>& from) {
    typedef boost::chrono::time_point<Clock, Duration> time_point_t;
    typedef boost::chrono::nanoseconds duration_t;
    typedef boost::duration_t::rep rep_t;
    rep_t d = boost::chrono::duration_cast<duration_t>(from.time_since_epoch()).count();
    rep_t sec = d/1000000000;
    rep_t nsec = d%1000000000;
    return  boost::posix_time::from_time_t(0)+
            boost::posix_time::seconds(static_cast<long>(sec))+
            boost::posix_time::nanoseconds(nsec);
}
```

With this explicit conversion and the equivalen for duration, the actual definition of std::pair will allows to

```
std::pair<chrono::time_point<Clock, Duration>, boost::chrono::duration<Rep, Period> > tp_dur_pair;
std::pair<boost::posix_time::ptime, boost::posix_time::time_duration> ppt;
ppt = tp_dur_pair;
```