Toward Boost.Integer.Endian 0.2.0

Beman Dawes
Vicente J. Botet Escriba

Table of Contents

Overview 1
- Motivation 2
- Description 5

Users'Guide 5
- Getting Started 5
- Tutorial 8
- Examples 15
- References 17

Reference 17
- Header <boost/binary_stream.hpp> 17
- Header <boost/integer/endian/endiannes.hpp> 18
- Header <boost/integer/endian/domain_map.hpp> 19
- Header <boost/integer/endian/endian_pack.hpp> 20
- Header <boost/integer/endian/endian.hpp> 24
- Header <boost/integer/endian/endian_binary_stream.hpp> 28
- Header <boost/integer/endian/endian_type.hpp> 28
- Header <boost/integer/endian/endian_view.hpp> 29
- Header <boost/integer/endian/endian_conversion.hpp> 31

Appendices 32
- Appendix A: History 32
- Appendix B: Rationale 33
- Appendix C: Implementation Notes 34
- Appendix D: Acknowledgements 35
- Appendix E: Tests 35
- Appendix F: Tickets 36
- Appendix G: Future plans 36

Warning

Boost.Integer.Endian.Ext 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`*.
- Free functions are rendered in the code font followed by (), as in `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.

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 endian files
#include <boost/integer/endian.hpp>
using namespace boost::integer;
```

Motivation

The original Beman's version provided through the boost::integer::endian<> class an integer-like class that providing arithmetics operations on

- Big endian | little endian | native endian byte ordering.
- Signed | unsigned
- Unaligned | aligned
- 1-8 byte (unaligned) | 2, 4, 8 byte (aligned)
- Choice of integer value type

This endian aware design seems to be unappropiated to work with endian unaware types, but as we will see this is not completly true, as we can construct on top of the Beman's design endian unaware operation. Next follows some of the main critics that have been done recently.

Intrusive versus non-intrusive

Endian conversion must be possible for any type even if you can't modify the original source code. Many people think that originel Beman's library requires the data types which need to be converted to be coded using his endian types as members.

We will see that this is not true, as the user can always use casting to reinterpret a given structure on another one, as the proposed extension does with the endian_views.

```
struct endian_unaware_structure {
    int32_t a;
    int16_t b;
};
endian_unaware_structure st_a;

endian_unaware_structure st_b;
as_endian<native>(st_b) = as_endian<big>(st_a);
```

Physical versus Logical data

The typical use case is to use native endianness until ready to read/write using wire format. It is only when ready to do I/O that there is a need to account for endianness.

Beman's original library requires two types one to specify the physical data and the other representing the logical data. Many people think that this means that the user has to maintain two separate structs/classes when only one is needed. This could be sometimes the case, but in other cases, the user will store the information elements included in the message on local variables and specific contexts that don't follow the external format.

Applications that don't separate the physical and the logical data structures don't scale well as any change on the external format will mean a refactoring of the whole application. Also, the wire structures cannot maintain invariants, so having a separate class is often useful even if its contents are essentially identical to the wire version.

In addition a single and efficient view is only possible when the pysical view can be represented by native aligned integers. But when the physical view contains unaliged integers with sizes not supported by the platform, the need to separate both view becomes anavoidable. Here the Beman's design is the best adapted.

Inherent inefficient arithmetic operations

The fact that the endian types provide operators gives the impression that it's ok to operate on them, as these operations can potentially require two endian conversions when the endianness is different.

Some people don't agree with that design choice, as they think the operation of endian conversion and operating on the data should be divorced from each other. A less experienced user that you may end up not realizing what the hidden costs are and use the endian aware integer-like types throughout the application, paying unnecessary overheads. They prefer a library that makes hard to misuse in that way; borrowing a phrase from python, "explicit is better than implicit". They believe that forcing the user to do the explicit endian conversion leads to better separation of concerns for the application, but they don't propose any mean to force this conversion.

The extension proposal separates the endian aware byte-holder type from the one able to make arithmetic operations so the user will be forced to convert for one to/from the native type, and avoid unfortunate costly atithmetic operations.

```
struct endian_aware_structure {
    endian_pack<big,int32_t> a;
    endian_pack<big,int16_t> b;
};
```

Structure `endian_aware_structure` can not be used without converting to native types, as `endian_pack` doesn't provide other operations than conversion.

```
endian_aware_structure st;
// ...
st.a=i;
st.b=j;
```

But don't removes the inefficient and endian safe class, as less demanding applications would benefit from the safe and transparent approach.

In-place conversion

Other contexts force the user to make in place conversions. While this could be dangerous, there are context on which a funtional conversion is not possible, mainly because the duplicated space could not be an option. These in-place conversions are only possible for aligned endian unaware types.

```
struct endian_unaware_structure {
    int32_t a;
    int16_t b;
};
endian_unaware_structure st;

as_endian<native>(st) = as_endian<big>(st);
```

The fact that some contexts need this in-place conversion doesn't mean that every application should use this approach.

UDT endianness

`boost::integer::endian` class didn't accept a UDT. We have updated the library so now UDT can be used for endian aligned types.

```
struct UserMessage {
  endian_pack<little, system_clock::time_point > timestamp;
  ulittle32_pt aircraft_id;
  struct Position {
     endian_pack<little, quantity<si::length, boost::int_least32_t> > x;
     endian_pack<little, quantity<si::length, boost::int_least32_t> > y;
     endian_pack<little, quantity<si::length, boost::int_least32_t> > z;
  } position;
  struct Attitude {
      endian_pack<little, quantity<si::plane_angle, boost::int_least8_t> > heading;
      endian_pack<little, quantity<si::plane_angle, boost::int_least8_t> > pitch;
      endian_pack<little, quantity<si::plane_angle, boost::int_least8_t> > roll;
  } attitude;
}; // UserMessage
```

Mixed endianness

Sometimes we need to deal with messages with mixed endianness. A functional approach means the programmer has to "know" which endianess the data he wants to convert has.

```
struct UdpHeader {
  ubig16_pt source_port;
  ubig16_pt destination_port;
  ubig16_pt length;
  ubig16_pt checksum;
}; // UdpHeader

struct Packet {
  internet::UdpHeader udpHeader;
  UserMessage userMessage;
}; // Packet
```

Floating point

From my understanding floating point types are not only concerned by endianness, but also by a multitude of standard on not standard formats. The library doent't manage with these standards.

Conclusion

The library must support applications needing to work safely with endian aware types and efficiently with in place endian conversions.

Conversions between native and big/little endian formats must be simple.

Description

This is an extension of the Beman's Boost.Integer.Endian, which was able to work with endian aware types, used to convert between types with different endian or even to carry on with arithmetic operations, by spliting the endian part from integer part and by adding support for aligned endian unaware types.

Boost.Integer.Endian.Ext provides:

- Endian packs
  - Big endian | little endian | native endian byte ordering.
  - Signed | unsigned
  - Unaligned | aligned
  - 1-8 byte (unaligned) | 2, 4, 8 byte (aligned)
  - Choice of integer value type
- Endian integers with the whole set of arithmetics operators.
- Operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of built-in, std::string types and of endian types.
- Views of aligned endian unaware integer types as endian packs or endian integers so we can make endian conversion.
- Generic in place conversion between different endian formats.
  - Very simple interface: convert_to/from<endiannes domain>(),
  - Support for built-in and user-defined data types view as fusion sequences.

Users'Guide

Getting Started

Installing Boost.Integer.Endian.Ext

Getting Boost.Integer.Endian.Ext

You can get the last stable release of **Boost.Integer.Endian.Ext** by downloading `integer_endian.zip` from the Boost Vault

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

Building Boost.Integer.Endian.Ext

Boost.Integer.Endian.Ext is implemented entirely within headers, with no need to link to any Boost object libraries.

Several macros allow user control over features:

- BOOST_ENDIAN_NO_CTORS causes class endian to have no constructors. The intended use is for compiling user code that must be portable between compilers regardless of C++0x Defaulted Functions support. Use of constructors will always fail,

- BOOST_ENDIAN_FORCE_PODNESS causes BOOST_ENDIAN_NO_CTORS to be defined if the compiler does not support C++0x Defaulted Functions. This is ensures that , and so can be used in unions. In C++0x, class endian objects are POD's even though they have constructors.

Requirements

Boost.Integer.Endian.Ext depends on some Boost library. The library has been tested on trunk but an older version should works also.

In particular, **Boost.Integer.Endian.Ext** depends on:

Boost.Config for configuration purposes, ...

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

The implementation will eventually work with most C++03 conforming compilers. Current version has been tested on:

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

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 Endian World!

```
#include <boost/integer/endian/endian.hpp>
#include <boost/integer/endian/endian_binary_stream.hpp>
#include <boost/binary_stream.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;

int main()
{
  int_least32_t v = 0x31323334L;  // = ASCII { '1', '2', '3', '4' }
                                  // value chosen to work on text stream
  big32_t    b(v);
  little32_t l(v);

  std::cout << "Hello, endian world!\n\n";

  std::cout << v << ' ' << b << ' ' << l << '\n';
  std::cout <= v <= ' ' <= b <= ' ' <= l <= '\n';
}
```

On a little-endian CPU, this program outputs:

```
Hello, endian world!

825373492 825373492 825373492
4321 1234 4321
```

Limitations

Requires `<climits>` `CHAR_BIT == 8`. If `CHAR_BIT` is some other value, compilation will result in an `#error`. This restriction is in place because the design, implementation, testing, and documentation has only considered issues related to 8-bit bytes, and there have been no real-world use cases presented for other sizes.

In C++03, endian does not meet the requirements for POD types because it has constructors, private data members, and a base class. This means that common use cases are relying on unspecified behavior in that the C++ Standard does not guarantee memory layout for non-POD types. This has not been a problem in practice since all known C++ compilers do layout memory as if endian were a POD type. In C++0x, it will be possible to specify the default constructor as trivial, and private data members and base classes will no longer disqualify a type from being a POD. Thus under C++0x, endian will no longer be relying on unspecified behavior.

Binary I/O warnings and cautions

Warning

Use only on streams opened with filemode `std::ios_base::binary`. Thus unformatted binary I/O should not be with the standard streams (`cout`, `cin`, etc.) since they are opened in text mode. Use on text streams may produce incorrect results, such as insertion of unwanted characters or premature end-of-file. For example, on Windows `0x0D` would become `0x0D`, `0x0A`.

Warning

Caution: When mixing formatted (i.e. operator << or >>) and unformatted (i.e. operator <= or >=) stream I/O, be aware that << and >> take precedence over <= and >=. Use parentheses to force correct order of evaluation. For example:

```
my_stream << foo <= bar;    // no parentheses needed
(my_stream <= foo) << bar;  // parentheses required
```

As a practical matter, it may be easier and safer to never mix the character and binary insertion or extraction operators in the same statement.

Tutorial

Endian aware types

basic endian holders

Boost.Integer.Endian.Ext provides an endian aware byte-holder class template:

```
template <endianness::enum_t E, typename T, std::size_t n_bytes=8*sizeof(T),
  alignment::enum_t A = alignment::aligned>
class endian_pack;
```

This class provide portable byte-holders for data, independent of particular computer architectures. Use cases almost always involve I/O, either via files or network connections. Although data portability is the primary motivation, these byte-holders may also be used to reduce memory use, file size, or network activity since they provide binary integer sizes not otherwise available.

Such byte-holder types are traditionally called endian types. See the Wikipedia for a full exploration of endianness, including definitions of big endian and little endian.

This class doesn't provides arithmetic operators, but of course automatic conversion to and assignment from the underlying integer value type are provided.

```
#include <boost/integer/endian/endian_pack.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;
using namespace boost::integer::endianness;


int main()
{
    int_least32_t v = 0x31323334L;  // = ASCII { '1', '2', '3', '4' }
                                    // value chosen to work on text stream

    endian_pack<big, int_least32_t>    b(v);
    endian_pack<little, int_least32_t> l(v);

    std::cout << v << ' ' << b << ' ' << l << '\n';
    return 0;
}
```

On a little-endian CPU, this program outputs:

```
825373492 825373492 825373492
```

Binary streams

Header <boost/binary_stream.hpp> provides operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of built-in and std::string types.

Header <boost/integer/endian/endian_binary_stream.hpp> provides operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of endian types.

```
#include <boost/integer/endian/endian_pack.hpp>
#include <boost/integer/endian/endian_binary_stream.hpp>
#include <boost/binary_stream.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;
using namespace boost::integer::endianness;


int main()
{
    int_least32_t n = 0x31323334L;  // = ASCII { '1', '2', '3', '4' }
                                    // value chosen to work on text stream

    endian_pack<big, int_least32_t>    b(n);
    endian_pack<little, int_least32_t> l(n);

    std::cout << n << ' ' << b << ' ' << l << '\n';
    std::cout <= n <= ' ' <= b <= ' ' <= l <= '\n';
    return 0;
}
```

On a little-endian CPU, this program outputs:

```
Hello, endian world!

825373492 825373492 825373492
4321 1234 4321
```

Endian aware unaligned byte holders

```
#include <boost/integer/endian/endian_pack.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;
using namespace boost::integer::endianness;


int main()
{
    int_least32_t v = 0x313233L;  // = ASCII { '1', '2', '3' }
                                  // value chosen to work on text stream

    endian_pack<big, int_least32_t, 24, unaligned>    b(v);
    endian_pack<little, int_least32_t, 24, unaligned> l(v);

    std::cout << v << ' ' << b << ' ' << l << '\n';
    std::cout << b << ' ' << l << '\n';
    return 0;
}
```

On a little-endian CPU, this program outputs:

```
3224115 3224115 3224115
123 321
```

Endian holders common typedefs

Sixty typedefs, such as big32_t, provide convenient naming conventions for common use cases:

Name	Endianness	Sign	Sizes in bits (n)	Alignment
big**n**_t	big	signed	8,16,24,32,40,48,56,64	unaligned
ubig**n**_t	big	unsigned	8,16,24,32,40,48,56,64	unaligned
little**n**_t	little	signed	8,16,24,32,40,48,56,64	unaligned
ulittle**n**_t	little	unsigned	8,16,24,32,40,48,56,64	unaligned
ulittle**n**_t	native	signed	8,16,24,32,40,48,56,64	unaligned
unative**n**_t	native	unsigned	16,32,64	aligned
aligned_big**n**_t	big	signed	16,32,64	aligned
aligned_ubig**n**_t	big	unsigned	16,32,64	aligned
aligned_little**n**_t	big	signed	16,32,64	aligned
aligned_ulittle**n**_t	big	unsigned	16,32,64	aligned

The unaligned types do not cause compilers to insert padding bytes in classes and structs. This is an important characteristic that can be exploited to minimize wasted space in memory, files, and network transmissions.

Warning

Code that uses aligned types is inherently non-portable because alignment requirements vary between hardware architectures and because alignment may be affected by compiler switches or pragmas. Furthermore, aligned types are only available on architectures with 16, 32, and 64-bit integer types.

Note

One-byte big-endian, little-endian, and native-endian types provide identical functionality. All three names are provided to improve code readability and searchability.

Comment on naming

When first exposed to endian types, programmers often fit them into a mental model based on the <cstdint> types. Using that model, it is natural to expect a 56-bit big-endian signed integer to be named `int_big56_t`. But these byte-hodlers are not realy integers.

That seems to lead to formation of a new mental model specific to endian byte-holder types. In that model, the endianness is the key feature, and the integer aspect is downplayed. Once that mental transition is made, a name like `big56_t` is a good reflection of the mental model.

```
#include <boost/integer/endian/endian_pack.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;


int main()
{
    int_least32_t v = 0x31323334L;  // = ASCII { '1', '2', '3', '4' }
                                    // value chosen to work on text stream
    big32_t    b(v);
    little32_t l(v);

    std::cout << v << ' ' << b << ' ' << l << '\n';
    return 0;
}
```

On a little-endian CPU, this program outputs:

```
825373492 825373492 825373492
```

Endian holders of UDT

The user can define wrappers to integer types that behave as integers, as it is the case for quantity<> class. This UDT can be packaged on an endian holder to take care of endian issues, as show the following example.

```
#include <boost/integer/endian/endian_pack.hpp>
#include <boost/integer/endian/endian_binary_stream.hpp>
#include <boost/binary_stream.hpp>
#include <iostream>
#include <boost/units/io.hpp>
#include <boost/units/quantity.hpp>
#include <boost/units/systems/si/length.hpp>
#include <boost/units/systems/si/plane_angle.hpp>

using namespace boost;
using namespace boost::integer;
using namespace boost::integer::endianness;
using namespace boost::units;

int main()
{
    quantity<si::length, boost::int_least32_t> q = 825373492 * si::meter;
    endian_pack<big, quantity<si::length, boost::int_least32_t> > b;
    endian_pack<little, quantity<si::length, boost::int_least32_t> > l;
    b=q;
    l=q;
    std::cout << q << ' ' << b << ' ' << l << '\n';
    std::cout <= b <= ' ' <= l <= '\n';
    return 0;
}
```

On a little-endian CPU, this program outputs:

```
825373492 m 825373492 m 825373492 m
1234 4321
```

Arrays of endian holders

Up to now we have see how to work with endian aware byte holders variables. In this section we will see how to work with arrays of endian aware byte holders.

```
#include <boost/integer/endian/endian_pack.hpp>
#include <boost/integer/endian/endian_binary_stream.hpp>
#include <boost/binary_stream.hpp>
#include <iostream>

using namespace boost;
using namespace boost::integer;

int main()
{
    int_least32_t an[100] = { 0x31323334L };  // = ASCII { '1', '2', '3', '4' }
                                  // value chosen to work on text stream
    big32_t    ab[100];
    little32_t al[100];

    for_each

    std::cout << an[3] << ' ' << ab[3] << ' ' << al[3] << '\n';
    std::cout <= an[3] <= ' ' <= ab[3] <= ' ' <= al[3] <= '\n';
    return 0;
}
```

Structures of endian holders

```
struct UdpHeader {
    ubig16_pt source_port;
    ubig16_pt destination_port;
    ubig16_pt length;
    ubig16_pt checksum;
}; // UdpHeader

UdpHeader header;
header.source_port = 1111;
header.destination_port = 1234;
header.length = get_length(p);
header.checksum = checksum_of(p);
```

Endian aware integers

Boost endian integers are based on the byte_holder endian_pack class template and provide in addition the same full set of C++ assignment, arithmetic, and relational operators as C++ standard integral types, with the standard semantics.

One class template is provided:

```
template <typename E, typename T, std::size_t n_bytes=8*sizeof(T),
  alignment::enum_t A = alignment::aligned>
class endian;
```

Unary arithmetic operators are +, -, ~, !, prefix and postfix -- and ++. Binary arithmetic operators are +, +=, -, -=, *, *=, /, /=, %, %=, &, &=, |, |=, ^, ^=, <<, <<=, >>, >>=. Binary relational operators are ==, !=, <, <=, >, >=.

Automatic conversion is provided to the underlying integer value type.

Typedefs

Sixty typedefs, such as big32_t, provide convenient naming conventions for common use cases:

Name	Endianness	Sign	Sizes in bits (n)	Alignment
int_big**n**_t	big	signed	8,16,24,32,40,48,56,64	unaligned
uint_big**n**_t	big	unsigned	8,16,24,32,40,48,56,64	unaligned
int_little**n**_t	little	signed	8,16,24,32,40,48,56,64	unaligned
uint_little**n**_t	little	unsigned	8,16,24,32,40,48,56,64	unaligned
uint_little**n**_t	native	signed	8,16,24,32,40,48,56,64	unaligned
uint_native**n**_t	native	unsigned	16,32,64	aligned
aligned_int_big**n**_t	big	signed	16,32,64	aligned
aligned_uint_big**n**_t	big	unsigned	16,32,64	aligned
aligned_int_little**n**_t	little	signed	16,32,64	aligned
aligned_uint_little**n**_t	little	unsigned	16,32,64	aligned

Comment on naming

When first exposed to endian types, programmers often fit them into a mental model based on the <cstdint> types. Using that model, it is natural to expect a 56-bit big-endian signed integer to be named `int_big56_t`.

As experience using these types grows, the realization creeps in that they are lousy arithmetic integers - they are really byte holders that for convenience support arithmetic operations - and that for use in internal interfaces or anything more than trivial arithmetic computations it is far better to convert values of these endian types to traditional integer types.

Endian unaware types

Endian unaware types don't track the endiannes of the stored data. For example

```
namespace X {

struct big_c {
    uint32_t a;
    uint16_t b;
};


struct little_c {
    int32_t a;
    int16_t b;
};

struct mixed_c {
    big_c a;
    little_c b;
};

}
```

Working with third party structures

Domain map

When we need to send the mixed_c structure through the network we need to state the endiannes of each one of the fields, so the library is able to make the adequated conversions. This is done using the domain_map metafunction, as follows:

```
struct network {};

namespace boost {
namespace integer {
namespace endianness {

    template <>
    struct domain_map <network, X::big_c> {
    typedef mpl::vector<big,big> type;
    };
    template <>
    struct domain_map <network, X::little_c> {
    typedef mpl::vector<little,little> type;
    };

}
}
}
```

In addition we need to see the types as fusion sequence so we can visit them in a homogeneous way.

```
BOOST_FUSION_ADAPT_STRUCT(
  X::big_c,
  (uint32_t, a)
  (uint16_t, b)
)

BOOST_FUSION_ADAPT_STRUCT(
  X::little_c,
  (int32_t, a)
  (int16_t, b)
)

BOOST_FUSION_ADAPT_STRUCT(
  X::mixed_c,
  (X::big_c, a)
  (X::little_c, b)
)
```

Now we are able to make an in-place conversion as follows:

```
int main( )
{
  X::mixed_c m;
  m.b.a=0x01020304;
  m.b.b=0x0A0B;
  m.a.a=0x04030201;
  m.a.b=0x0B0A;

  convert_to<network>(m);
  return 0;
} // main
```

In-place conversions

Examples

Endian Aware

The endian_example.cpp program writes a binary file containing four byte big-endian and little-endian integers.

This is an extract from a very widely used GIS file format. I have no idea why a designer would mix big and little endians in the same file - but this is a real-world format and users wishing to write low level code manipulating these files have to deal with the mixed endianness.

Low-level I/O such as POSIX read/write or <cstdio> fread/fwrite is sometimes used for binary file operations when ultimate efficiency is important. Such I/O is often performed in some C++ wrapper class, but to drive home the point that endian integers are often used in fairly low-level code that does bulk I/O operations, <cstdio> fopen/fwrite is used for I/O in this example.

```
#include <iostream>
#include <cassert>
#include <cstdio>
#include <boost/integer/endian.hpp>

using namespace boost::integer;

namespace
{

  struct header
  {
    big32_t     file_code;
    big32_t     file_length;
    little32_t  version;
    little32_t  shape_type;
  };

  const char * filename = "test.dat";
}

int main()
{
  assert( sizeof( header ) == 16 );  // requirement for interoperability

  header h;

  h.file_code   = 0x04030201;
  h.file_length = sizeof( header );
  h.version     = -1;
  h.shape_type  = 0x04030201;

  std::FILE * fi;

  if ( !(fi = std::fopen( filename, "wb" )) )  // MUST BE BINARY
  {
    std::cout << "could not open " << filename << '\n';
    return 1;
  }

  if ( std::fwrite( &h, sizeof( header ), 1, fi ) != 1 )
  {
    std::cout << "write failure for " << filename << '\n';
    return 1;
  }

  std::fclose( fi );

  std::cout << "created file " << filename << '\n';
  return 0;
}
```

After compiling and executing endian_example.cpp, a hex dump of test.dat shows:

```
0403 0201 0000 0010 ffff ffff 0102 0304
```

"receiver makes right"

References

Reference

Header <boost/binary_stream.hpp>

Header <boost/binary_stream.hpp> provides operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of built-in and std::string types.

Synopsis

```
namespace boost
{

  //  built-in types  ------------------------------------------------------------//

  std::ostream& operator<=(std::ostream& os, short v);
  std::istream& operator>=(std::istream& is, short& v);

  std::ostream& operator<=(std::ostream& os, unsigned short v);
  std::istream& operator>=(std::istream& is, unsigned short& v);

  std::ostream& operator<=(std::ostream& os, int v);
  std::istream& operator>=(std::istream& is, int& v);

  std::ostream& operator<=(std::ostream& os, unsigned int v);
  std::istream& operator>=(std::istream& is, unsigned int& v);

  std::ostream& operator<=(std::ostream& os, long v);
  std::istream& operator>=(std::istream& is, long& v);

  std::ostream& operator<=(std::ostream& os, unsigned long v);
  std::istream& operator>=(std::istream& is, unsigned long& v);

  std::ostream& operator<=(std::ostream& os, long long v);
  std::istream& operator>=(std::istream& is, long long& v);

  std::ostream& operator<=(std::ostream& os, unsigned long long v);
  std::istream& operator>=(std::istream& is, unsigned long long& v);

  std::ostream& operator<=(std::ostream& os, float v);
  std::istream& operator>=(std::istream& is, float& v);

  std::ostream& operator<=(std::ostream& os, double v);
  std::istream& operator>=(std::istream& is, double& v);

  std::ostream& operator<=(std::ostream& os, long double v);
  std::istream& operator>=(std::istream& is, long double& v);

  std::ostream& operator<=(std::ostream& os, char c);
  std::istream& operator>=(std::istream& is, char& c);

  std::ostream& operator<=(std::ostream& os, signed char c);
  std::istream& operator>=(std::istream& is, signed char& c);

  std::ostream& operator<=(std::ostream& os, unsigned char c);
  std::istream& operator>=(std::istream& is, unsigned char& c);
```

```
  std::ostream& operator<=(std::ostream& os, wchar_t v);
  std::istream& operator>=(std::istream& is, wchar_t& v);

  //  strings  ------------------------------------------------------------------//

  std::ostream& operator<=(std::ostream& os, const char* p);

  std::ostream& operator<=(std::ostream& os, const signed char* p);

  std::ostream& operator<=(std::ostream& os, const unsigned char* p);

#ifndef BOOST_NO_CWCHAR
  std::ostream& operator<=(std::ostream& os, const wchar_t* p);
#endif

  std::ostream& operator<=(std::ostream& os, const std::string& s);
  std::istream& operator>=(std::istream& is, std::string& s);

#ifndef BOOST_NO_STD_WSTRING
  std::ostream& operator<=(std::ostream& os, const std::wstring& s);
  std::istream& operator>=(std::istream& is, std::wstring& s);
#endif

}  // namespace boost
```

Note

Omission of bool and void* is deliberate; any semantics would be questionable

Warning

Note the asymmetry between output and input; a string with embedded nulls will be output with the embedded nulls, but input will stop at the first null. So it probably isn't a good idea to use these functions for strings with nulls.

Header <boost/integer/endian/endiannes.hpp>

Header <boost/endian/endian/endianness.hpp> provides the endianness tags.

Synopsis

```
namespace boost {
namespace integer {
    namespace endianness {
        struct big;
        struct little;
        struct middle;
        struct mixed;
        typedef <platform dependent> native ;
    }
}
}
```

Class big

Big endian tag.

```
struct big {};
```

Class `little`

Little endian tag.

```
struct little {};
```

Class `middle`

Middle endian tag.

```
struct middle {};
```

Class `mixed`

Mixed endian tag.

```
struct mixed {};
```

Class `native`

Native endian tag.

```
typedef <platform dependent> native ;
```

Header `<boost/integer/endian/domain_map.hpp>`

This file contains the default implementation ofr the `domain_map` metafunction.

Synopsis

```
namespace boost {
namespace endian {
    template <typename Domain, typename T>
    struct domain_map;
}}
```

Meta Function `domain_map<>`

```
template <typename Domain, typename T>
struct domain_map {
    typedef <see below> type;
};
```

Requires:

- `Domain` any class. Can be also `endianness::big` or `endianness::little`.

Result: The member typedef `type` names a mpl tree of sequence of `endianness` types as view from the point of view of the `Domain`. The default definition is a mpl tree having as leaves the `Domain` class for `T` fundamental types, and fusion sequences.

Example:

```
is_same<domain_map<endianness::big, int>::type, endianness::big>::value == true

struct ifA {};
```

The user needs to specialize this metafunction for specific domains.

Header <boost/integer/endian/endian_pack.hpp>

This file contains the core class template of **Boost.Integer.Endian.Ext**. Provides byte-holder binary types with explicit control over byte order, value type, size, and alignment. Typedefs provide easy-to-use names for common configurations.

Synopsis

```
namespace boost {
namespace integer {
    BOOST_SCOPED_ENUM_START(alignment) { unaligned, aligned }; BOOST_SCOPED_ENUM_END

    template <typename E,
        typename T,
        std::size_t n_bits=sizeof(T)*8,
        BOOST_SCOPED_ENUM(alignment) A = alignment::aligned
    > class endian_pack;

    // unaligned big endian_pack signed integer types
    typedef endian_pack< endianness::big, int_least8_t, 8, alignment::unaligned >          ↵
 big8_pt;
    typedef endian_pack< endianness::big, int_least16_t, 16, alignment::unaligned >        ↵
 big16_pt;
    typedef endian_pack< endianness::big, int_least32_t, 24, alignment::unaligned >        ↵
 big24_pt;
    typedef endian_pack< endianness::big, int_least32_t, 32, alignment::unaligned >        ↵
 big32_pt;
    typedef endian_pack< endianness::big, int_least64_t, 40, alignment::unaligned >        ↵
 big40_pt;
    typedef endian_pack< endianness::big, int_least64_t, 48, alignment::unaligned >        ↵
 big48_pt;
    typedef endian_pack< endianness::big, int_least64_t, 56, alignment::unaligned >        ↵
 big56_pt;
    typedef endian_pack< endianness::big, int_least64_t, 64, alignment::unaligned >        ↵
 big64_pt;

    // unaligned big endian_pack unsigned integer types
    typedef endian_pack< endianness::big, uint_least8_t, 8, alignment::unaligned >         ↵
 ubig8_pt;
    typedef endian_pack< endianness::big, uint_least16_t, 16, alignment::unaligned >       ↵
 ubig16_pt;
    typedef endian_pack< endianness::big, uint_least32_t, 24, alignment::unaligned >       ↵
 ubig24_pt;
    typedef endian_pack< endianness::big, uint_least32_t, 32, alignment::unaligned >       ↵
 ubig32_pt;
    typedef endian_pack< endianness::big, uint_least64_t, 40, alignment::unaligned >       ↵
 ubig40_pt;
    typedef endian_pack< endianness::big, uint_least64_t, 48, alignment::unaligned >       ↵
 ubig48_pt;
    typedef endian_pack< endianness::big, uint_least64_t, 56, alignment::unaligned >       ↵
 ubig56_pt;
    typedef endian_pack< endianness::big, uint_least64_t, 64, alignment::unaligned >       ↵
 ubig64_pt;

    // unaligned little endian_pack signed integer types
    typedef endian_pack< endianness::little, int_least8_t, 8, alignment::unaligned >       ↵
```

```
 little8_pt;
    typedef endian_pack< endianness::little, int_least16_t, 16, alignment::unaligned >     ↵
 little16_pt;
    typedef endian_pack< endianness::little, int_least32_t, 24, alignment::unaligned >     ↵
 little24_pt;
    typedef endian_pack< endianness::little, int_least32_t, 32, alignment::unaligned >     ↵
 little32_pt;
    typedef endian_pack< endianness::little, int_least64_t, 40, alignment::unaligned >     ↵
 little40_pt;
    typedef endian_pack< endianness::little, int_least64_t, 48, alignment::unaligned >     ↵
 little48_pt;
    typedef endian_pack< endianness::little, int_least64_t, 56, alignment::unaligned >     ↵
 little56_pt;
    typedef endian_pack< endianness::little, int_least64_t, 64, alignment::unaligned >     ↵
 little64_pt;

    // unaligned little endian_pack unsigned integer types
    typedef endian_pack< endianness::little, uint_least8_t, 8, alignment::unaligned >      ↵
 ulittle8_pt;
    typedef endian_pack< endianness::little, uint_least16_t, 16, alignment::unaligned >    ↵
 ulittle16_pt;
    typedef endian_pack< endianness::little, uint_least32_t, 24, alignment::unaligned >    ↵
 ulittle24_pt;
    typedef endian_pack< endianness::little, uint_least32_t, 32, alignment::unaligned >    ↵
 ulittle32_pt;
    typedef endian_pack< endianness::little, uint_least64_t, 40, alignment::unaligned >    ↵
 ulittle40_pt;
    typedef endian_pack< endianness::little, uint_least64_t, 48, alignment::unaligned >    ↵
 ulittle48_pt;
    typedef endian_pack< endianness::little, uint_least64_t, 56, alignment::unaligned >    ↵
 ulittle56_pt;
    typedef endian_pack< endianness::little, uint_least64_t, 64, alignment::unaligned >    ↵
 ulittle64_pt;

    // unaligned native endian_pack signed integer types
    typedef endian_pack< endianness::native, int_least8_t, 8, alignment::unaligned >       nat↵
ive8_pt;
    typedef endian_pack< endianness::native, int_least16_t, 16, alignment::unaligned >     nat↵
ive16_pt;
    typedef endian_pack< endianness::native, int_least32_t, 24, alignment::unaligned >     nat↵
ive24_pt;
    typedef endian_pack< endianness::native, int_least32_t, 32, alignment::unaligned >     nat↵
ive32_pt;
    typedef endian_pack< endianness::native, int_least64_t, 40, alignment::unaligned >     nat↵
ive40_pt;
    typedef endian_pack< endianness::native, int_least64_t, 48, alignment::unaligned >     nat↵
ive48_pt;
    typedef endian_pack< endianness::native, int_least64_t, 56, alignment::unaligned >     nat↵
ive56_pt;
    typedef endian_pack< endianness::native, int_least64_t, 64, alignment::unaligned >     nat↵
ive64_pt;

    // unaligned native endian_pack unsigned integer types
    typedef endian_pack< endianness::native, uint_least8_t, 8, alignment::unaligned >      un↵
ative8_pt;
    typedef endian_pack< endianness::native, uint_least16_t, 16, alignment::unaligned >    un↵
ative16_pt;
    typedef endian_pack< endianness::native, uint_least32_t, 24, alignment::unaligned >    un↵
ative24_pt;
    typedef endian_pack< endianness::native, uint_least32_t, 32, alignment::unaligned >    un↵
ative32_pt;
    typedef endian_pack< endianness::native, uint_least64_t, 40, alignment::unaligned >    un↵
ative40_pt;
```

```
    typedef endian_pack< endianness::native, uint_least64_t, 48, alignment::unaligned >     un↵
ative48_pt;
    typedef endian_pack< endianness::native, uint_least64_t, 56, alignment::unaligned >     un↵
ative56_pt;
    typedef endian_pack< endianness::native, uint_least64_t, 64, alignment::unaligned >     un↵
ative64_pt;


    //  These types only present if platform has exact size integers:
    //     aligned big endian_pack signed integer types
    //     aligned big endian_pack unsigned integer types
    //     aligned little endian_pack signed integer types
    //     aligned little endian_pack unsigned integer types


    # if defined(BOOST_HAS_INT16_T)
    typedef endian_pack< endianness::big, int16_t, 16, alignment::aligned >     aligned_big16_pt;
    typedef endian_pack< endianness::big, uint16_t, 16, alignment::aligned >    aligned_ubig16_pt;
    typedef endian_pack< endianness::little, int16_t, 16, alignment::aligned >  ↵
 aligned_little16_pt;
    typedef endian_pack< endianness::little, uint16_t, 16, alignment::aligned > ↵
 aligned_ulittle16_pt;
    # endif

    # if defined(BOOST_HAS_INT32_T)
    typedef endian_pack< endianness::big, int32_t, 32, alignment::aligned >     aligned_big32_pt;
    typedef endian_pack< endianness::big, uint32_t, 32, alignment::aligned >    aligned_ubig32_pt;
    typedef endian_pack< endianness::little, int32_t, 32, alignment::aligned >  ↵
 aligned_little32_pt;
    typedef endian_pack< endianness::little, uint32_t, 32, alignment::aligned > ↵
 aligned_ulittle32_pt;
    # endif

    # if defined(BOOST_HAS_INT64_T)
    typedef endian_pack< endianness::big, int64_t, 64, alignment::aligned >     aligned_big64_pt;
    typedef endian_pack< endianness::big, uint64_t, 64, alignment::aligned >    aligned_ubig64_pt;
    typedef endian_pack< endianness::little, int64_t, 64, alignment::aligned >  ↵
 aligned_little64_pt;
    typedef endian_pack< endianness::little, uint64_t, 64, alignment::aligned > ↵
 aligned_ulittle64_pt;
    # endif

    //     aligned native endian_pack typedefs are not provided because
    //     <cstdint> types are superior for this use case

}
}
```

Template class `endian_pack<>`

An `endian_pack` is a byte-holder with user-specified endianness, value type, size, and alignment.

```
    template <
        typename E,
        typename T,
        std::size_t n_bits=8*sizeof(T),
        alignment A = alignment::aligned>
    class endian_pack {
    public:
  typedef E endian_type;
      typedef T value_type;
      static const std::size_t width = n_bits;
      static const alignment alignment_value = A;

#ifndef BOOST_ENDIAN_NO_CTORS
  endian_pack() = default;       // = default replaced by {} on C++03
      explicit endian_pack(T v);
#endif
      endian_pack & operator=(T v);
      operator T() const;
      const char* data() const;
    };
```

Requires:

- `E` is one of `endianness::big` or `endianness::little`.
- `T` must be a POD with value semantics.
- `nbits` is a multiple of `8`
- If `A` is `alignment::aligned` then `nbits` must be equal to `8*sizeof(T)`

Default Contructor `endian_pack()`

Note

if BOOST_ENDIAN_FORCE_PODNESS is defined && C++0x POD's are not available then this constructor will not be present

```
endian_pack() = default;  // C++03: endian(){}
```

Effects: Constructs an object of type `endian_pack<E, T, n_bits, A>`.

Contructor from value_type `endian_pack(T)`

Note

if BOOST_ENDIAN_FORCE_PODNESS is defined && C++0x POD's are not available then this constructor will not be present.

```
explicit endian_pack(T v);
```

Effects: Constructs an object of type `endian_pack<E, T, n_bits, A>`.

Postcondition: `x == v`, where `x` is the constructed object.

Assignment Operator from value_type `operator=(T)`

```
endian & operator=(T v);
```

Postcondition: `x == v`, where `x` is the constructed object. **Returns:** `*this`.

Conversion Operator `operator T()`

```
operator T() const;
```

Returns: The current value stored in `*this`, converted to `value_type`.

Member Function `data()`

```
const char* data() const;
```

Returns: The current value stored in `*this`, converted to `value_type`.

Common typedefs

Header `<boost/integer/endian/endian.hpp>`

Header `<boost/integer/endian/endian.hpp>` provides integer-like byte-holder binary types with explicit control over byte order, value type, size, and alignment. Typedefs provide easy-to-use names for common configurations.

Synopsis

```
namespace boost {
namespace integer {

    template <typename E, typename T, std::size_t n_bits,
      alignment A = alignment::unaligned>
    class endian;

    // unaligned big endian signed integer types
    typedef endian< endianness::big, int_least8_t, 8, alignment::unaligned >   big8_t;
    typedef endian< endianness::big, int_least16_t, 16, alignment::unaligned > big16_t;
    typedef endian< endianness::big, int_least32_t, 24, alignment::unaligned > big24_t;
    typedef endian< endianness::big, int_least32_t, 32, alignment::unaligned > big32_t;
    typedef endian< endianness::big, int_least64_t, 40, alignment::unaligned > big40_t;
    typedef endian< endianness::big, int_least64_t, 48, alignment::unaligned > big48_t;
    typedef endian< endianness::big, int_least64_t, 56, alignment::unaligned > big56_t;
    typedef endian< endianness::big, int_least64_t, 64, alignment::unaligned > big64_t;

    // unaligned big endian unsigned integer types
    typedef endian< endianness::big, uint_least8_t, 8, alignment::unaligned >   ubig8_t;
    typedef endian< endianness::big, uint_least16_t, 16, alignment::unaligned > ubig16_t;
    typedef endian< endianness::big, uint_least32_t, 24, alignment::unaligned > ubig24_t;
    typedef endian< endianness::big, uint_least32_t, 32, alignment::unaligned > ubig32_t;
    typedef endian< endianness::big, uint_least64_t, 40, alignment::unaligned > ubig40_t;
    typedef endian< endianness::big, uint_least64_t, 48, alignment::unaligned > ubig48_t;
    typedef endian< endianness::big, uint_least64_t, 56, alignment::unaligned > ubig56_t;
    typedef endian< endianness::big, uint_least64_t, 64, alignment::unaligned > ubig64_t;

    // unaligned little endian signed integer types
    typedef endian< endianness::little, int_least8_t, 8, alignment::unaligned >   little8_t;
    typedef endian< endianness::little, int_least16_t, 16, alignment::unaligned > little16_t;
    typedef endian< endianness::little, int_least32_t, 24, alignment::unaligned > little24_t;
    typedef endian< endianness::little, int_least32_t, 32, alignment::unaligned > little32_t;
    typedef endian< endianness::little, int_least64_t, 40, alignment::unaligned > little40_t;
    typedef endian< endianness::little, int_least64_t, 48, alignment::unaligned > little48_t;
    typedef endian< endianness::little, int_least64_t, 56, alignment::unaligned > little56_t;
    typedef endian< endianness::little, int_least64_t, 64, alignment::unaligned > little64_t;

    // unaligned little endian unsigned integer types
    typedef endian< endianness::little, uint_least8_t, 8, alignment::unaligned >   ulittle8_t;
    typedef endian< endianness::little, uint_least16_t, 16, alignment::unaligned > ulittle16_t;
    typedef endian< endianness::little, uint_least32_t, 24, alignment::unaligned > ulittle24_t;
    typedef endian< endianness::little, uint_least32_t, 32, alignment::unaligned > ulittle32_t;
    typedef endian< endianness::little, uint_least64_t, 40, alignment::unaligned > ulittle40_t;
    typedef endian< endianness::little, uint_least64_t, 48, alignment::unaligned > ulittle48_t;
    typedef endian< endianness::little, uint_least64_t, 56, alignment::unaligned > ulittle56_t;
    typedef endian< endianness::little, uint_least64_t, 64, alignment::unaligned > ulittle64_t;

    // unaligned native endian signed integer types
    typedef endian< endianness::native, int_least8_t, 8, alignment::unaligned >   native8_t;
    typedef endian< endianness::native, int_least16_t, 16, alignment::unaligned > native16_t;
    typedef endian< endianness::native, int_least32_t, 24, alignment::unaligned > native24_t;
    typedef endian< endianness::native, int_least32_t, 32, alignment::unaligned > native32_t;
    typedef endian< endianness::native, int_least64_t, 40, alignment::unaligned > native40_t;
    typedef endian< endianness::native, int_least64_t, 48, alignment::unaligned > native48_t;
    typedef endian< endianness::native, int_least64_t, 56, alignment::unaligned > native56_t;
    typedef endian< endianness::native, int_least64_t, 64, alignment::unaligned > native64_t;

    // unaligned native endian unsigned integer types
    typedef endian< endianness::native, uint_least8_t, 8, alignment::unaligned >   unative8_t;
    typedef endian< endianness::native, uint_least16_t, 16, alignment::unaligned > unative16_t;
    typedef endian< endianness::native, uint_least32_t, 24, alignment::unaligned > unative24_t;
```

```
    typedef endian< endianness::native, uint_least32_t, 32, alignment::unaligned > unative32_t;
    typedef endian< endianness::native, uint_least64_t, 40, alignment::unaligned > unative40_t;
    typedef endian< endianness::native, uint_least64_t, 48, alignment::unaligned > unative48_t;
    typedef endian< endianness::native, uint_least64_t, 56, alignment::unaligned > unative56_t;
    typedef endian< endianness::native, uint_least64_t, 64, alignment::unaligned > unative64_t;

    // These types only present if platform has exact size integers:

    // aligned big endian signed integer types
    typedef endian< endianness::big, int16_t, 16, alignment::aligned >  aligned_big16_t;
    typedef endian< endianness::big, int32_t, 32, alignment::aligned >  aligned_big32_t;
    typedef endian< endianness::big, int64_t, 64, alignment::aligned >  aligned_big64_t;

    // aligned big endian unsigned integer types
    typedef endian< endianness::big, uint16_t, 16, alignment::aligned > aligned_ubig16_t;
    typedef endian< endianness::big, uint32_t, 32, alignment::aligned > aligned_ubig32_t;
    typedef endian< endianness::big, uint64_t, 64, alignment::aligned > aligned_ubig64_t;

    // aligned little endian signed integer types
    typedef endian< endianness::little, int16_t, 16, alignment::aligned > aligned_little2_t;
    typedef endian< endianness::little, int32_t, 32, alignment::aligned > aligned_little4_t;
    typedef endian< endianness::little, int64_t, 64, alignment::aligned > aligned_little8_t;

    // aligned little endian unsigned integer types
    typedef endian< endianness::little, uint16_t, 16, alignment::aligned > aligned_ulittle2_t;
    typedef endian< endianness::little, uint32_t, 32, alignment::aligned > aligned_ulittle4_t;
    typedef endian< endianness::little, uint64_t, 64, alignment::aligned > aligned_ulittle8_t;

    // aligned native endian typedefs are not provided because
    // <cstdint> types are superior for this use case

} // namespace integer
} // namespace boost
```

Template class `endian<>`

An endian integer is an integer byte-holder with user-specified endianness, value type, size, and alignment. The usual operations on integers are supplied.

```
template <
    typename E,
    typename T,
    std::size_t n_bits=8*sizeof(T),
    alignment A = alignment::aligned>
class endian : cover_operators< endian<E, T, n_bits, A>, T > {
public:
  typedef E endian_type;
  typedef T value_type;
  static const std::size_t width = n_bits;
  static const alignment alignment_value = A;

  #ifndef BOOST_ENDIAN_NO_CTORS
  endian() = default;       // = default replaced by {} on C++03
  explicit endian(T v);
  #endif
  endian & operator=(T v);
  operator T() const;
  const char* data() const;
};
```

Requires:

- `E` is one of `endianness::big` or `endianness::little`.
- `T` must be a POD with value semantics.
- `nbits` is a multiple of `8`
- If `A` is `alignment::aligned` then `nbits` must be equal to `8*sizeof(T)`

Default Contructor `endian()`

Note

if BOOST_ENDIAN_FORCE_PODNESS is defined && C++0x POD's are not available then this constructor will not be present

```
endian() = default;  // C++03: endian(){}
```

Effects: Constructs an object of type `endian<E, T, n_bits, A>`.

Contructor from value_type `endian(T)`

Note

if BOOST_ENDIAN_FORCE_PODNESS is defined && C++0x POD's are not available then this constructor will not be present

```
explicit endian(T v);
```

Effects: Constructs an object of type `endian<E, T, n_bits, A>`. **Postcondition:** `x == v`, where `x` is the constructed object.

Assignment Operator from value_type `operator=(T)`

```
endian & operator=(T v);
```

Postcondition: `value_type(*this) == v`. **Returns:** `*this`.

Conversion Operator `operator T()`

```
operator T() const;
```

Returns: The current value stored in `*this`, converted to `value_type`.

Member Function `data()`

```
const char* data() const;
```

Returns: The current value stored in `*this`, converted to `value_type`.

Other operators

Other operators on endian objects are forwarded to the equivalent operator on `value_type`.

Common typedefs

Header <boost/integer/endian/endian_binary_stream.hpp>

Header <boost/integer/endian/endian_binary_stream.hpp> provides operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of endian types.

Synopsis

```
namespace boost
{
  namespace integer
  {
    template< class T >
    struct is_endian { static const bool value = false; };
    template< typename E, typename T, std::size_t n_bits, alignment A >
    struct is_endian<endian_pack<E,T,n_bits,A> { static const bool value = true; };
    template< typename E, typename T, std::size_t n_bits, alignment A >
    struct is_endian<endian<E,T,n_bits,A> { static const bool value = true; };

    template < class Endian >
      typename boost::enable_if< is_endian<Endian>, std::ostream & >::type
         operator<=( std::ostream & os, const Endian & e );

    template < class Endian >
      typename boost::enable_if< is_endian<Endian>, std::istream & >::type
         operator>=( std::istream & is, Endian & e );
  }
}
```

Header <boost/integer/endian/endian_type.hpp>

Synopsis

```
namespace boost {
namespace integer {
    template <typename T>
    struct endian_type;
}
}
```

Meta Function endian_type<>

```
template <typename Domain, typename T>
struct endian_type {
    typedef type;
}
```

The member typedef `type` names one of the `endianness` types `big`, `little` or `mixed`. If all the leaves of the type T are of the same endianness type is this endiannes, otherwise it is `mixed`.

The default behavior works for all the endian aware types, fundamental types and any type that is a fusion sequence.

The user can specialize this metafunction for specific clases.

Example

```
is_same<endian_type<endian<endianness::big, int> >::type, endianness::big>::value == true
is_same<endian_type<int>::type, endianness::native>::value == true
```

Header <boost/integer/endian/endian_view.hpp>

This file provides the `endian_view<>` class template as well as some factory helper functions.

Synopsis

```
namespace boost {
namespace integer {

    template <typename Endian>
    class endian_view;

    template <typename E, typename T>
    endian_view<endian_pack<E,T> > as_endian(T& v);
    template <typename T>
    endian_view<endian_pack<endianness::native, T> > as(T& v);
    template <typename T>
    endian_view<endian_pack<endianness::little,T> > as_little(T& v);
    template <typename T>
    endian_view<endian_pack<endianness::big,T> > as_big(T& v);


} // namespace integer
} // namespace boost
```

Template class endian_view<>

```
template <typename Endian, typename T>
class endian_view {
public:
    typedef T value_type;
    endian_view()=delete;
    endian_view(value_type& ref);
    operator value_type() const;
    endian_view& operator=(value_type val);
    endian_view& operator=(endian_view const& rhs);
    template <typename Endian2 >
    endian_view& operator=(endian_view<Endian2,T> const& rhs) {
};
```

Constructor endian_view

```
endian_view(value_type& ref);
```

Effects: Constructs an object of type `endian_view<E, T, n_bits, A>`.

Conversion operator value_type()

```
operator value_type() const;
```

Returns: The converted `value_type` of the referenced type as it was seen as an endian aware type.

Assignement `operator =(endian_view const& )`

```
endian_view& operator=(endian_view const& rhs);
```

Postcondition: `this->ref_ == rhs.ref_`. **Returns:** `*this`.

Assignement from different endianness `operator =(endian_view<Endian2,T> const& )`

```
template <typename Endian2 >
endian_view& operator=(endian_view<Endian2,T> const& rhs) {
```

Postcondition: `value_type(this->ref_) == value_type(rhs.ref_)`. **Returns:** `*this`.

Assignement from value_type `operator =(value_type)`

```
endian_view& operator=(value_type val);
```

Postcondition: `value_type(this->ref_) == v`. **Returns:** `*this`.

Non-Member Function Template `as_endian<>`

```
template <typename E, typename T>
endian_view<endian<E,T> > as_endian(T& v);
```

Returns: An `endian_view<>` that depend on the endianness parameter `E` referencing the pararameter `v`.

Non-Member Function Template `as<>`

```
template <typename T>
endian_view<endian<endian::native, T> > as(T& v);
```

Returns: A native endian `endian_view<>` referencing the pararameter `v`.

Non-Member Function Template `as_little<>`

```
template <typename T>
endian_view<endian<endian::little, T> > as_little(T& v);
```

Returns: A little endian `endian_view<>` referencing the pararameter `v`.

Non-Member Function Template `as_big<>`

```
template <typename T>
endian_view<endian<endian::big, T> > as_big(T& v);
```

Returns: A big endian `endian_view<>` referencing the pararameter `v`.

Header <boost/integer/endian/endian_conversion.hpp>

Synopsis

```
namespace boost {
namespace integer {
namespace endianness {

    template <typename TargetDomain, typename SourceDomain, typename T>
    void convert_to_from(T& r);

    template <typename SourceDomain, typename T>
    void convert_from(T& r);

    template <typename TargetDomain, typename T>
    void convert_to(T& r);

    template <typename Endian> struct to;
    template <typename Endian> struct from;
    template <typename Endian> struct is_target;
    template <typename Endian> struct is_source;

    template <typename Endian1, typename Endian2, typename T>
    void convert(T& r);


} // namespace endianness
} // namespace integer
} // namespace boost
```

Non-Member Function Template convert_to_from<>

```
template <typename TargetDomain, typename SourceDomain, typename T>
void convert_to_from(T& r);
```

Returns: the conversion of v viewed from `SourceDomain` to `TargetDomain'.

Non-Member Function Template convert_from<>

```
template <typename SourceDomain, typename T>
void convert_from(T& r);
```

Returns: the conversion of v viewed from `SourceDomain` to the native doamin.

Non-Member Function Template convert_to<>

```
template <typename TargetDomain, typename T>
void convert_to(T& r);
```

Returns: the conversion of v viewed from the native to `TargetDomain'.

Class Template to<>

Tag class used to signal target conversion.

```
template <typename Endian>
struct to {
    typedef Endian type;
};
```

Class Template `from<>`

Tag class used to signal source conversion.

```
template <typename Endian>
struct from {
    typedef Endian type;
};
```

Class Template `is_target<>`

Template metafunction stating if the tag is a target domain.

```
template <typename Endian>
struct is_target : mpl::false_ {};
template <typename Endian>
struct is_target< to<Endian> > : mpl::true_ {};
```

Class Template `is_source<>`

Template metafunction stating if the tag is a source domain.

```
template <typename Endian>
struct is_source : mpl::false_ {};
template <typename Endian>
struct is_source< from<Endian> > : mpl::true_ {};
```

Non-Member Function Template `convert<>`

Coversion from source to target depending in the nature of the `Endian1` and `Endian2` template parameters. One of them must be a target domain (`to<>`) and the other a source domain (`from<>`).

```
template <typename Endian1, typename Endian2, typename T>
void convert(T& r);
```

Returns: the conversion of `v' viewed from source domain to the target domain'.

Appendices

Appendix A: History

Version 0.2.0, Febraury 15, 2011

Moved to boost/integer/endian directory.

Version 0.1.0, June 15, 2010

Split of Boost.Integer.Endian + Added Endian views.

Features:

- Endian packs
  - Big endian | little endian | native endian byte ordering.
  - Signed | unsigned
  - Unaligned | aligned
  - 1-8 byte (unaligned) | 2, 4, 8 byte (aligned)
  - Choice of integer value type
- Endian integers with the whole set of arithmetics operators based on endian pack.
- Operators <= and => for unformatted binary (as opposed to formatted character) stream insertion and extraction of built-in, std::string types and of endian types.
- Views of aligned endian unaware integer types as endian packs or endian integers so we can make endian conversion.
- Generic in place conversion between different endian formats.
  - Very simple interface: convert_to/from<endiannes domain>(),
  - Support for built-in and user-defined data types view as fusion sequences.

Appendix B: Rationale

Design considerations for Boost.Integer.Endian.Ext

- Must be suitable for I/O - in other words, must be memcpyable.
- Must provide exactly the size and internal byte ordering specified.
- Must work correctly when the internal integer representation has more bits that the sum of the bits in the external byte representation. Sign extension must work correctly when the internal integer representation type has more bits than the sum of the bits in the external bytes. For example, using a 64-bit integer internally to represent 40-bit (5 byte) numbers must work for both positive and negative values.
- Must work correctly (including using the same defined external representation) regardless of whether a compiler treats char as signed or unsigned.
- Unaligned types must not cause compilers to insert padding bytes.
- The implementation should supply optimizations only in very limited circumstances. Experience has shown that optimizations of endian integers often become pessimizations. While this may be obvious when changing machines or compilers, it also happens when changing compiler switches, compiler versions, or CPU models of the same architecture.
- It is better software engineering if the same implementation works regardless of the CPU endianness. In other words, #ifdefs should be avoided where possible.

Experience

Classes with similar functionality have been independently developed by several Boost programmers and used very successful in high-value, high-use applications for many years. These independently developed endian libraries often evolved from C libraries that were also widely used. Endian integers have proven widely useful across a wide range of computer architectures and applications.

Motivating use cases

Neil Mayhew writes: "I can also provide a meaningful use-case for this library: reading TrueType font files from disk and processing the contents. The data format has fixed endianness (big) and has unaligned values in various places. Using **Boost.Integer.Endian.Ext** simplifies and cleans the code wonderfully."

C++0x

The availability of the C++0x Defaulted Functions feature is detected automatically, and will be used if present to ensure that objects of class endian are trivial, and thus POD's.

Appendix C: Implementation Notes

FAQ

- **Why bother with endian types?** External data portability and both speed and space efficiency. Availability of additional binary integer sizes and alignments is important in some applications.
- **Why not just use Boost.Serialization?** Serialization involves a conversion for every object involved in I/O. Endian objects require no conversion or copying. They are already in the desired format for binary I/O. Thus they can be read or written in bulk.
- **Why bother with binary I/O? Why not just use C++ Standard Library stream inserters and extractors?** Using binary rather than character representations can be more space efficient, with a side benefit of faster I/O. CPU time is minimized because conversions to and from string are eliminated. Furthermore, binary integers are fixed size, and so fixed-size disk records are possible, easing sorting and allowing direct access. Disadvantages, such as the inability to use text utilities on the resulting files, limit usefulness to applications where the binary I/O advantages are paramount.
- **Do these types have any uses outside of I/O?** Probably not, except for native endianness which can be used for fine grained control over size and alignment.
- **Is there a performance hit when doing arithmetic using integer endian types?** Yes, for sure, compared to arithmetic operations on native integer types. However, these types are usually be faster, and sometimes much faster, for I/O compared to stream inserters and extractors, or to serialization.
- **Are endian types POD's?** Yes for C++0x. No for C++03, although several macros are available to force PODness in all cases.
- **What are the implications endian types not being POD's with C++03 compilers?** They can't be used in unions. Also, compilers aren't required to align or lay out storage in portable ways, although this potential problem hasn't prevented use of **Boost.Integer.Endian.Ext** with real compilers.
- **Which is better, big-endian or little-endian?** Big-endian tends to be a bit more of an industry standard, but little-endian may be preferred for applications that run primarily on x86 (Intel/AMD) and other little-endian CPU's. The Wikipedia article gives more pros and cons.
- **What good is native endianness?** It provides alignment and size guarantees not available from the built-in types. It eases generic programming.
- **Why bother with the aligned endian types?** Aligned integer operations may be faster (20 times, in one measurement) if the endianness and alignment of the type matches the endianness and alignment requirements of the machine. On common CPU architectures, that optimization is only available for aligned types. That allows I/O of maximally efficient types on an application's primary platform, yet produces data files are portable to all platforms. The code, however, is likely to be more fragile and less portable than with the unaligned types.
- **Endian types are really just byte-holders. Why provide the arithmetic operations at all?** Providing a full set of operations reduces program clutter and makes code both easier to write and to read. Consider incrementing a variable in a record. It is very convenient to write:

```
++record.foo;
```

Rather than:

```
int temp( record.foo);
++temp;
record.foo = temp;
```

- **Why do binary stream insertion and extraction use operators <= and >= rather than <<= and >>=?** <<= and >>= associate right-to-left, which is the opposite of << and >>, so would be very confusing and error prone. <= and >= associate left-to-right.

Appendix D: Acknowledgements

Original design developed by Darin Adler based on classes developed by Mark Borgerding. Four original class templates combined into a single endian class template by Beman Dawes, who put the library together, provided documentation, and added the typedefs. He also added the unrolled_byte_loops sign partial specialization to correctly extend the sign when cover integer size differs from endian representation size.

Comments and suggestions were received from Benaka Moorthi, Christopher Kohlhoff, Cliff Green, Gennaro Proto, Giovanni Piero Deretta, dizzy, Jeff Flinn, John Maddock, Kim Barrett, Marsh Ray, Martin Bonner, Matias Capeletto, Neil Mayhew, Phil Endecott, Rene Rivera, Roland Schwarz, Scott McMurray, Sebastian Redl, Tomas Puverle and Yuval Ronen.

Appendix E: Tests

binary_stream_test

Name	kind	Description	Result	Ticket
check_op	run	checl binary streams operations on builtin types	Pass	#

integer_endian_pack_test

Name	kind	Description	Result	Ticket
detect_endianness	run	detect endianness	Pass	#
check_size	run	check size for different endian types	Pass	#
check_alignment	run	check alignement for different endian types	Pass	#
check_representation_and_range_and_ops	run	check representation and range and operations	Pass	#

integer_endian_test

Name	kind	Description	Result	Ticket
detect_endianness	run	detect endianness	Pass	#
check_size	run	check size for different endian types	Pass	#
check_alignment	run	check alignement for different endian types	Pass	#
check_representation_and_range_and_ops	run	check representation and range and operations	Pass	#
check_data	run	check data starts at the same address	Pass	#

integer_endian_arithmetic_operation_test

Name	kind	Description	Result	Ticket
check_op	run	check arithmethic operations on integer endian types	Pass	#

integer_endian_view_test

Name	kind	Description	Result	Ticket
check_read	run	check read access	Pass	#
check_write	run	check write access	Pass	#

integer_endian_convert_test

Name	kind	Description	Result	Ticket
check_in_place_conversion	run	check in place conversion	Pass	#

Appendix F: Tickets

Appendix G: Future plans

Tasks to do before review

- Support for 'pure' endian conversion for endian unaware UDT.
- Support for 'in place' endian conversion for ranges of endian unaware types.
- endian iterator, which will iterate across a range endian converting values as necessary. It works with any type supporting the convert_to/from functions.

For later releases

- The library doesn't take advantage of special instructions on architectures which natively support endian conversion. This functionality could, however, be part of future releases.