Boost.Align

Glen Fernandes

Table of Contents

Introduction

This library provides function **align** for implementations which do not have the C++11 standard library `std::align` function available.

It provides allocation and deallocation functions, **aligned_alloc** and **aligned_free**, as their functionality is not yet available in the C++ standard library. They use platform specific functions, if available, or use standard library functions in conjunction with `align`.

It provides C++ allocators, class templates **aligned_allocator** and **aligned_allocator_adaptor**, which respect alignment. The first uses `aligned_alloc` and `aligned_free` while the second uses the allocator in conjunction with `align`.

It provides a deleter, class **aligned_delete**, which makes use of `aligned_free`. It is suitable for use with constructed objects allocated with `aligned_alloc`.

It provides type trait **alignment_of** for implementations without a conforming C++11 standard library `std::alignment_of` type trait.

It also provides function **is_aligned** to test the alignment of a pointer.

Rationale

C++11 added the ability to specify increased alignment (over-alignment) for class types. Unfortunately, `::operator new` allocation functions, `new` expressions, and the default allocator, `std::allocator`, do not support dynamic memory allocation of over-aligned data.

The **align** function can be used to align a pointer and is provided for implementations which do not yet provide `std::align`.

The **aligned_alloc** function can be used in place of `::operator new` to specify the alignment of the memory allocated.

The **aligned_allocator** class template can be used in place of `std::allocator` as an alignment-aware default allocator.

The **aligned_allocator_adaptor** class template can be used to adapt any allocator into an alignment-aware allocator.

The **aligned_delete** class can be used in place of `std::default_delete` to destroy and free objects allocated with `aligned_alloc`.

The **alignment_of** type trait gets the alignment of a type, and is provided for implementations without `std::alignment_of`.

The **is_aligned** function can be used to compare the alignment value of a pointer.

Tutorial

Using align

We want to default-construct an object of a potentially over-aligned generic type, `T`, in storage of size `n`, whose address is stored in a pointer, `p`.

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

Use the `align` function to adjust the pointer so that it is suitably aligned.

```
auto result = boost::alignment::
    align(alignof(T), sizeof(T), p, n);
```

If successful, `n` is decreased by the byte count that `p` was advanced to be suitably aligned, and the adjusted value of `p` is returned. It now points to an address at which to construct an object of type `T`.

```
if (result) {
    ::new(result) T();
}
```

If unsuccessful, because `n` has insufficient space to fit an object of the requested size after adjusting `p` to have the requested alignment, a null pointer is returned and `p` and `n` are not changed.

```
else {
    throw std::exception();
}
```

Using aligned_alloc

We want to dynamically allocate storage for, and default-construct within that storage, an object of a generic type `T` that is potentially over-aligned.

```
#include <boost/align/aligned_alloc.hpp>
```

Allocate an object of a desired alignment and size using the `aligned_alloc` allocation function.

```
auto p = boost::alignment::
    aligned_alloc(alignof(T), sizeof(T));
```

If successful, a non-null pointer is returned. To free this storage the `aligned_free` function is used.

```
if (p) {
    try {
        ::new(p) T();
    } catch (...) {
        boost::alignment::aligned_free(p);
        throw;
    }
}
```

If unsuccessful, a null pointer is returned.

```
else {
    throw std::bad_alloc();
}
```

Free this storage, via the `aligned_free` function, when it is no longer required.

```
boost::alignment::aligned_free(p);
```

Using aligned_allocator

We want to use standard library allocator-aware containers, such as `vector`, with a generic type, `T`, that is potentially over-aligned.

```
#include <boost/align/aligned_allocator.hpp>
```

Specify the `aligned_allocator` allocator via the container's allocator template parameter.

```
std::vector<T, boost::alignment::
    aligned_allocator<T> > v;
```

If we wanted a vector of a different type, such as `int`, but desired that each integer object had the alignment of `T`, this is possible by specifying the minimum alignment with the allocator.

```
std::vector<int, boost::alignment::
    aligned_allocator<int, alignof(T)> > v;
```

Using aligned_allocator_adaptor

We want to make an existing allocator type, `A`, alignment-aware and use it with a standard library container, such as `vector`, with a type, `T`, that is potentially over-aligned.

```
#include <boost/align/aligned_allocator_adaptor.hpp>
```

We use class template `aligned_allocator_adaptor` as the vector's allocator type.

```
std::vector<T, boost::alignment::
    aligned_allocator_adaptor<A> > v(a);
```

If we wanted a vector of a different type, such as `int`, but desired that each integer object had the alignment of `T`, this is possible by specifying the minimum alignment with the allocator adaptor.

```
std::vector<int, boost::alignment::
    aligned_allocator_adaptor<A, alignof(T)> > v(a);
```

Using aligned_delete

We want a default deleter for use with `unique_ptr` that can be used to deallocate and destroy objects which were constructed in storage that was allocated with `aligned_alloc`.

```
#include <boost/align/aligned_delete.hpp>
```

Storage is allocated for the object using `aligned_alloc`.

```
auto p = boost::alignment::
    aligned_alloc(alignof(T), sizeof(T));
if (!p) {
    throw std::bad_alloc();
}
```

An object is constructed in that storage using placement new.

```
try {
    q = ::new(p) T();
} catch (...) {
    boost::alignment::aligned_free(p);
    throw;
}
```

Use the `aligned_delete` class as the deleter template parameter.

```
std::unique_ptr<T,
    boost::alignment::aligned_delete> u(q);
```

Using alignment_of

We want to assert at compile time that the alignment of a type, `T`, is at least as large as the alignment of a type, `U`.

```
#include <boost/align/alignment_of.hpp>
```

Use the `alignment_of` class template to determine the alignment of the type.

```
static_assert(boost::alignment::alignment_of<T>::
    value >= boost::alignment::alignment_of<U>::
    value, "");
```

Using is_aligned

We want to assert that the alignment of a pointer, `p`, is at least the alignment of a generic type, `T`, that is potentially over-aligned.

```
#include <boost/align/is_aligned.hpp>
```

Use the `is_aligned` function to test the alignment of the pointer.

```
assert(boost::alignment::is_aligned(alignof(T), p));
```

Examples

aligned_ptr and make_aligned

The `aligned_ptr` alias template is a `unique_ptr` that uses `aligned_delete` as the deleter, for destruction and deallocation. This smart pointer type is suitable for managing objects that are allocated with `aligned_alloc`.

```
#include <boost/align/aligned_delete.hpp>
#include <memory>

template<class T>
using aligned_ptr = std::unique_ptr<T,
    boost::alignment::aligned_delete>;
```

The `make_aligned` function template creates an `aligned_ptr` for an object allocated with `aligned_alloc` and constructed with placement `new`. If allocation fails, it throws an object of `std::bad_alloc`. If an exception is thrown by the constructor, it uses `aligned_free` to free allocated memory and will rethrow the exception.

```
template<class T, class... Args>
inline aligned_ptr<T> make_aligned(Args&&... args)
{
    auto p = boost::alignment::aligned_alloc(alignof(T),
        sizeof(T));
    if (!p) {
        throw std::bad_alloc();
    }
    try {
        auto q = ::new(p) T(std::forward<Args>(args)...);
        return aligned_ptr<T>(q);
    } catch (...) {
        boost::alignment::aligned_free(p);
        throw;
    }
}
```

Here `make_aligned` is used to create an `aligned_ptr` object for a type which has extended alignment specified.

```
struct alignas(16) type {
    float data[4];
};

int main()
{
    auto p = make_aligned<type>();
    p->data[0] = 1.0f;
}
```

aligned_vector

The `aligned_vector` alias template is a `vector` that uses `aligned_allocator` as the allocator type, with a configurable minimum alignment. It can be used with types that have an extended alignment or to specify an minimum extended alignment when used with any type.

```
#include <boost/align/aligned_allocator.hpp>
#include <vector>

template<class T, std::size_t Alignment = 1>
using aligned_vector = std::vector<T,
    boost::alignment::aligned_allocator<T, Alignment> >;
```

Here `aligned_vector` is used to create a `vector` of integers where each integer object has extended cache alignment.

```
enum {
    cache_line = 64
};

int main()
{
    aligned_vector<int, cache_line> v(32);
    v[0] = 1;
}
```

Reference

Header <boost/align.hpp>

Boost.Align all headers.

Note

This header includes all public headers of the Boost.Align library.

Glen Fernandes

Header <boost/align/align.hpp>

Function align.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    void * align(std::size_t, std::size_t, void *&, std::size_t &);
  }
}
```

Function align

boost::alignment::align

Synopsis

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


void * align(std::size_t alignment, std::size_t size, void *& ptr,
             std::size_t & space);
```

Description

If it is possible to fit `size` bytes of storage aligned by `alignment` into the buffer pointed to by `ptr` with length `space`, the function updates `ptr` to point to the first possible address of such storage and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

Note: The function updates its `ptr` and space arguments so that it can be called repeatedly with possibly different `alignment` and `size` arguments for the same buffer.

Parameters:	`alignment`	Shall be a fundamental alignment value or an extended alignment value, and shall be a power of two.
	`ptr`	Shall point to contiguous storage of at least `space` bytes.
	`size`	The size in bytes of storage to fit into the buffer.
	`space`	The length of the buffer.
Returns:	A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of `ptr`.	

Header <boost/align/aligned_alloc.hpp>

Functions aligned_alloc and aligned_free.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    void * aligned_alloc(std::size_t, std::size_t);
    void aligned_free(void *);
  }
}
```

Function aligned_alloc

boost::alignment::aligned_alloc

Synopsis

```
// In header: <boost/align/aligned_alloc.hpp>


void * aligned_alloc(std::size_t alignment, std::size_t size);
```

Description

Allocates space for an object whose alignment is specified by `alignment`, whose size is specified by `size`, and whose value is indeterminate.

Note: On certain platforms, the alignment may be rounded up to `alignof(void*)` and the space allocated may be slightly larger than `size` bytes, by an additional `sizeof(void*)` and `alignment - 1` bytes.

Parameters:	`alignment`	Shall be a power of two.
	`size`	Size of space to allocate.
Returns:	A null pointer or a pointer to the allocated space.	

Function aligned_free

boost::alignment::aligned_free

Synopsis

```
// In header: <boost/align/aligned_alloc.hpp>


void aligned_free(void * ptr);
```

Description

Causes the space pointed to by `ptr` to be deallocated, that is, made available for further allocation. If `ptr` is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `aligned_alloc` function, or if the space has been deallocated by a call to `aligned_free`, the behavior is undefined.

Header <boost/align/aligned_allocator.hpp>

Class template aligned_allocator.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    template<typename T, std::size_t Alignment> class aligned_allocator;

    template<std::size_t Alignment> class aligned_allocator<void, Alignment>;
    template<typename T1, typename T2, std::size_t Alignment>
      bool operator==(const aligned_allocator< T1, Alignment > &,
                      const aligned_allocator< T2, Alignment > &);
    template<typename T1, typename T2, std::size_t Alignment>
      bool operator!=(const aligned_allocator< T1, Alignment > &,
                      const aligned_allocator< T2, Alignment > &);
  }
}
```

Class template aligned_allocator

boost::alignment::aligned_allocator

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>

template<typename T, std::size_t Alignment>
class aligned_allocator {
public:
  // types
  typedef T              value_type;
  typedef T *            pointer;
  typedef const T *      const_pointer;
  typedef void *         void_pointer;
  typedef const void *   const_void_pointer;
  typedef std::size_t    size_type;
  typedef std::ptrdiff_t difference_type;
  typedef T &            reference;
  typedef const T &      const_reference;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef aligned_allocator< U, Alignment > other;
  };

  // construct/copy/destruct
  aligned_allocator();
  template<typename U>
    aligned_allocator(const aligned_allocator< U, Alignment > &) noexcept;

  // public member functions
  pointer address(reference) const noexcept;
  const_pointer address(const_reference) const noexcept;
  pointer allocate(size_type, const_void_pointer = 0);
  void deallocate(pointer, size_type);
  BOOST_CONSTEXPR size_type max_size() const noexcept;
  template<typename U, class... Args> void construct(U *, Args &&...);
  template<typename U> void construct(U *);
  template<typename U> void destroy(U *);
};
```

Description

Class template aligned_allocator.

Note: Except for the destructor, member functions of the aligned allocator shall not introduce data races as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

Note

Specifying minimum alignment is generally only suitable for containers such as vector and undesirable with other, node-based, containers. For node-based containers, such as list, the node object would have the minimum alignment specified instead of the value type object.

Template Parameters

1. `typename T`

2.
```
std::size_t Alignment
```

Is the minimum alignment to specify for allocations, if it is larger than the alignment of the value type. It shall be a power of two.

`aligned_allocator` public construct/copy/destruct

1.
```
aligned_allocator();
```

2.
```
template<typename U>
  aligned_allocator(const aligned_allocator< U, Alignment > &) noexcept;
```

`aligned_allocator` public member functions

1.
```
pointer address(reference value) const noexcept;
```

Returns: The actual address of the object referenced by `value`, even in the presence of an overloaded operator&.

2.
```
const_pointer address(const_reference value) const noexcept;
```

Returns: The actual address of the object referenced by `value`, even in the presence of an overloaded operator&.

3.
```
pointer allocate(size_type size, const_void_pointer = 0);
```

Throw: Throws `std::bad_alloc` if the storage cannot be obtained.

Note: The storage is obtained by calling `aligned_alloc(std::size_t, std::size_t)`.
Returns: A pointer to the initial element of an array of storage of size `n * sizeof(T)`, aligned on the maximum of the minimum alignment specified and the alignment of objects of type `T`.

4.
```
void deallocate(pointer ptr, size_type);
```

Deallocates the storage referenced by `ptr`.

Note: Uses `alignment::aligned_free(void*)`.
Parameters: `ptr` Shall be a pointer value obtained from `allocate()`.

5.
```
BOOST_CONSTEXPR size_type max_size() const noexcept;
```

Returns: The largest value `N` for which the call `allocate(N)` might succeed.

6.
```
template<typename U, class... Args> void construct(U * ptr, Args &&... args);
```

Calls global `new((void*)ptr) U(std::forward<Args>(args)...)`.

7.
```
template<typename U> void construct(U * ptr);
```

Calls global `new((void*)ptr) U()`.

8.
```
template<typename U> void destroy(U * ptr);
```

Calls `ptr->~U()`.

Struct template rebind

boost::alignment::aligned_allocator::rebind

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>


template<typename U>
struct rebind {
  // types
  typedef aligned_allocator< U, Alignment > other;
};
```

Description

Rebind allocator.

Specializations

- Class template aligned_allocator<void, Alignment>

Class template aligned_allocator<void, Alignment>

boost::alignment::aligned_allocator<void, Alignment>

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>

template<std::size_t Alignment>
class aligned_allocator<void, Alignment> {
public:
  // types
  typedef void         value_type;
  typedef void *       pointer;
  typedef const void * const_pointer;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef aligned_allocator< U, Alignment > other;
  };
};
```

Description

Class template aligned_allocator specialization.

Struct template rebind

boost::alignment::aligned_allocator<void, Alignment>::rebind

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>


template<typename U>
struct rebind {
  // types
  typedef aligned_allocator< U, Alignment > other;
};
```

Description

Rebind allocator.

Function template operator==

boost::alignment::operator==

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>


template<typename T1, typename T2, std::size_t Alignment>
  bool operator==(const aligned_allocator< T1, Alignment > &,
                  const aligned_allocator< T2, Alignment > &);
```

Description

Returns: `true`.

Function template operator!=

boost::alignment::operator!=

Synopsis

```
// In header: <boost/align/aligned_allocator.hpp>


template<typename T1, typename T2, std::size_t Alignment>
  bool operator!=(const aligned_allocator< T1, Alignment > &,
                  const aligned_allocator< T2, Alignment > &);
```

Description

Returns: `false`.

Header <boost/align/aligned_allocator_adaptor.hpp>

Class template aligned_allocator_adaptor.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    template<typename Allocator, std::size_t Alignment>
      class aligned_allocator_adaptor;
    template<typename A1, typename A2, std::size_t Alignment>
      bool operator==(const aligned_allocator_adaptor< A1, Alignment > &,
                      const aligned_allocator_adaptor< A2, Alignment > &);
    template<typename A1, typename A2, std::size_t Alignment>
      bool operator!=(const aligned_allocator_adaptor< A1, Alignment > &,
                      const aligned_allocator_adaptor< A2, Alignment > &);
  }
}
```

Class template aligned_allocator_adaptor

boost::alignment::aligned_allocator_adaptor

Synopsis

```
// In header: <boost/align/aligned_allocator_adaptor.hpp>

template<typename Allocator, std::size_t Alignment>
class aligned_allocator_adaptor : public Allocator {
public:
  // types
  typedef Traits::value_type value_type;
  typedef Traits::size_type  size_type;
  typedef value_type *       pointer;
  typedef const value_type * const_pointer;
  typedef void *             void_pointer;
  typedef const void *       const_void_pointer;
  typedef std::ptrdiff_t     difference_type;

  // member classes/structs/unions
  template<typename U>
  struct rebind {
    // types
    typedef aligned_allocator_adaptor< typename Traits::template rebind_alloc< U >, Alignment > oth↵
er;
  };

  // construct/copy/destruct
  aligned_allocator_adaptor() = default;
  template<typename A> explicit aligned_allocator_adaptor(A &&) noexcept;
  template<typename U>
    aligned_allocator_adaptor(const aligned_allocator_adaptor< U, Alignment > &) noexcept;

  // public member functions
  Allocator & base() noexcept;
  const Allocator & base() const noexcept;
  pointer allocate(size_type);
  pointer allocate(size_type, const_void_pointer);
  void deallocate(pointer, size_type);
};
```

Description

Class template aligned_allocator_adaptor.

Note

This adaptor can be used with a C++11 allocator whose pointer type is a smart pointer but the adaptor will expose only raw pointers.

Template Parameters

1.
```
typename Allocator
```

2.
```
std::size_t Alignment
```

Is the minimum alignment to specify for allocations, if it is larger than the alignment of the value type. The value of `Alignment` shall be a fundamental alignment value or an extended alignment value, and shall be a power of two.

`aligned_allocator_adaptor` public construct/copy/destruct

1.
```
aligned_allocator_adaptor() = default;
```

Value-initializes the `Allocator` base class.

2.
```
template<typename A> explicit aligned_allocator_adaptor(A && alloc) noexcept;
```

Initializes the `Allocator` base class with `std::forward<A>(alloc)`.

Require: `Allocator` shall be constructible from `A`.

3.
```
template<typename U>
  aligned_allocator_adaptor(const aligned_allocator_adaptor< U, Alignment > & other) noexcept;
```

Initializes the `Allocator` base class with the base from other.

`aligned_allocator_adaptor` public member functions

1.
```
Allocator & base() noexcept;
```

Returns: `static_cast<Allocator&>(*this)`.

2.
```
const Allocator & base() const noexcept;
```

Returns: `static_cast<const Allocator&>(*this)`.

3.
```
pointer allocate(size_type size);
```

Throw: Throws an exception thrown from `A2::allocate` if the storage cannot be obtained.

Note: The storage is obtained by calling `A2::allocate` on an object `a2`, where `a2` of type `A2` is a rebound copy of `base()` where its `value_type` is unspecified.

Parameters: `size` The size of the value type object to allocate.

Returns: A pointer to the initial element of an array of storage of size `n * sizeof(value_type)`, aligned on the maximum of the minimum alignment specified and the alignment of objects of type `value_type`.

4.
```
pointer allocate(size_type size, const_void_pointer hint);
```

Throw: Throws an exception thrown from `A2::allocate` if the storage cannot be obtained.

Note: The storage is obtained by calling `A2::allocate` on an object `a2`, where `a2` of type `A2` is a rebound copy of `base()` where its `value_type` is unspecified.

Parameters: `hint` is a value obtained by calling `allocate()` on any equivalent aligned allocator adaptor object, or else `nullptr`.
`size` The size of the value type object to allocate.

Returns: A pointer to the initial element of an array of storage of size `n * sizeof(value_type)`, aligned on the maximum of the minimum alignment specified and the alignment of objects of type `value_type`.

5.
```
void deallocate(pointer ptr, size_type size);
```

Deallocates the storage referenced by `ptr`.

Note: Uses `A2::deallocate` on an object `a2`, where `a2` of type `A2` is a rebound copy of `base()` where its `value_type` is unspecified.

Parameters: `ptr` Shall be a pointer value obtained from `allocate()`.
`size` Shall equal the value passed as the first argument to the invocation of `allocate` which returned `ptr`.

Struct template rebind

boost::alignment::aligned_allocator_adaptor::rebind

Synopsis

```
// In header: <boost/align/aligned_allocator_adaptor.hpp>


template<typename U>
struct rebind {
  // types
  typedef aligned_allocator_adaptor< typename Traits::template rebind_alloc< U >, Alignment > oth↵
er;
};
```

Description

Rebind allocator.

Function template operator==

boost::alignment::operator==

Synopsis

```
// In header: <boost/align/aligned_allocator_adaptor.hpp>


template<typename A1, typename A2, std::size_t Alignment>
  bool operator==(const aligned_allocator_adaptor< A1, Alignment > & a,
                  const aligned_allocator_adaptor< A2, Alignment > & b);
```

Description

Returns: `a.base() == b.base()`.

Function template operator!=

boost::alignment::operator!=

Synopsis

```
// In header: <boost/align/aligned_allocator_adaptor.hpp>


template<typename A1, typename A2, std::size_t Alignment>
  bool operator!=(const aligned_allocator_adaptor< A1, Alignment > & a,
                  const aligned_allocator_adaptor< A2, Alignment > & b);
```

Description

Returns: `!(a == b)`.

Header <boost/align/aligned_allocator_adaptor_forward.hpp>

Class template aligned_allocator_adaptor forward declaration.

Note

This header provides a forward declaration for the `aligned_allocator_adaptor` class template.

Glen Fernandes

Header <boost/align/aligned_allocator_forward.hpp>

Class template aligned_allocator forward declaration.

Note

This header provides a forward declaration for the `aligned_allocator` class template.

Glen Fernandes

Header <boost/align/aligned_delete.hpp>

Class aligned_delete.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    class aligned_delete;
  }
}
```

Class aligned_delete

boost::alignment::aligned_delete

Synopsis

```
// In header: <boost/align/aligned_delete.hpp>


class aligned_delete {
public:

  // public member functions
  template<typename T>
    void operator()(T *) const noexcept(BOOST_NOEXCEPT_EXPR(ptr->~T()));
};
```

Description

Class aligned_delete.

`aligned_delete` public member functions

1. ```
   template<typename T>
     void operator()(T * ptr) const noexcept(BOOST_NOEXCEPT_EXPR(ptr->~T()));
   ```

   Calls `~T()` on `ptr` to destroy the object and then calls `aligned_free` on `ptr` to free the allocated memory.

   Note: If `T` is an incomplete type, the program is ill-formed.

Header <boost/align/aligned_delete_forward.hpp>

Class aligned_delete forward declaration.

Note

This header provides a forward declaration for the `aligned_delete` class.

Glen Fernandes

Header <boost/align/alignment_of.hpp>

Class template alignment_of.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    template<typename T> struct alignment_of;
  }
}
```
```

Struct template alignment_of

boost::alignment::alignment_of

Synopsis

```
// In header: <boost/align/alignment_of.hpp>

template<typename T>
struct alignment_of {

  enum @2 {  };
};
```

Description

Class template alignment_of.

Value: `alignof(T)`.

Header <boost/align/alignment_of_forward.hpp>

Class template alignment_of forward declaration.

Note

This header provides a forward declaration for the `alignment_of` class template.

Glen Fernandes

Header <boost/align/is_aligned.hpp>

Function is_aligned.

Glen Fernandes

```
namespace boost {
  namespace alignment {
    bool is_aligned(std::size_t, const void *);
  }
}
```

Function is_aligned

boost::alignment::is_aligned

Synopsis

```
// In header: <boost/align/is_aligned.hpp>


bool is_aligned(std::size_t alignment, const void * ptr);
```

Description

Determines whether the space pointed to by `ptr` has alignment specified by `alignment`.

Parameters:	`alignment`	Shall be a power of two.
	`ptr`	Pointer to test for alignment.
Returns:	`true` if and only if `ptr` points to space that has alignment specified by `alignment`.	

Vocabulary

3.11 [basic.align]

Object types have **alignment requirements** which place restrictions on the addresses at which an object of that type may be allocated. An **alignment** is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment specifier.

A **fundamental alignment** is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)`. The alignment required for a type might be different when it is used as the type of a complete object and when it is used as the type of a subobject.

Tip

```
struct B { long double d; };
struct D : virtual B { char c; };
```

When `D` is the type of a complete object, it will have a subobject of type `B`, so it must be aligned appropriately for a `long double`. If `D` appears as a subobject of another object that also has `B` as a virtual base class, the `B` subobject might be part of a different subobject, reducing the alignment requirementson the `D` subobject.

The result of the `alignof` operator reflects the alignment requirement of the type in the complete-object case.

An **extended alignment** is represented by an alignment greater than `alignof(std::max_align_t)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported. A type having an extended alignment requirement is an **over-aligned type**.

Note

Every over-aligned type is or contains a class type to which extended alignment applies (possibly through a non-static data member).

Alignments are represented as values of the type `std::size_t`. Valid alignments include only those values returned by an `alignof` expression for the fundamental types plus an additional implementation-defined set of values, which may be empty. Every alignment value shall be a non-negative integral power of two.

Alignments have an order from **weaker** to **stronger** or **stricter** alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

The alignment requirement of a complete type can be queried using an `alignof` expression. Furthermore, the types `char`, `signed char`, and `unsigned char` shall have the weakest alignment requirement.

Note

This enables the character types to be used as the underlying type for an aligned memory area.

Comparing alignments is meaningful and provides the obvious results:

- Two alignments are equal when their numeric values are equal.
- Two alignments are different when their numeric values are not equal.
- When an alignment is larger than another it represents a stricter alignment.

Note

The runtime pointer alignment function can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library can be used to obtain aligned storage.

If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed. Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.

Compatibility

The following compilers and platforms have been tested.

- clang 3.0, 3.1, 3.2, 3.4, 3.5, linux
- gcc 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, linux
- icc 11.1, 12.1, 13.0, linux
- msvc 9.0, 10.0, 11.0, 12.0, windows

Any conforming C++11 or C++03 compiler is supported.

Acknowledgments

Thank you to everyone who reviewed this library and helped me improve the design, implementation, tests, or documentation.

- Peter Dimov
- Andrey Semashev
- Bjorn Reese
- Steven Watanabe
- Antony Polukhin
- Lars Viklund
- Michael Spencer
- Paul A. Bristow

Thank you to Ahmed Charles for managing the review.

History

Version 1.0

- Glen Fernandes implemented the Boost.Align library.