# `std::assume_aligned`

Timur Doumler (papers@timur.audio)
Chandler Carruth (chandlerc@google.com)

|  |  |
|---|---|
| Document #: | P1007R0 |
| Date: | 2018-05-04 |
| Project: | Programming Language C++ |
| Audience: | Library Evolution Working Group, Library Working Group |

**Abstract**

We propose a new function template `std::assume_aligned`. This utility takes a pointer and hints to the compiler that the value of this pointer is a memory address aligned to a certain number of bytes. This will allow the compiler to generate better-optimised code. Currently, compiler vendors offer various built-ins to achieve the same effect. We propose to standardise this existing practice. This paper is a follow-up to [P0886R0]. We propose to add this functionality via a library function instead of a core language attribute.

## 1 Motivation

Consider a pointer to data allocated at an over-aligned memory address. Such data can be obtained, for example, from the `std::align_val_t` version of `operator new`, or from `std::align`. If such a pointer is passed on to another function, or returned from one, it is often desirable to make the over-alignment property transparent to the compiler. This will allow the compiler to generate better optimised code. Use cases include:

— Often, the compiler will auto-vectorise a loop over a buffer with contiguous data, generating SIMD instructions. We can make such a buffer to be over-aligned with the SIMD register size. If we could make this property visible to the compiler, it could skip the loop prologue and epilogue, resulting in fewer branches and instructions, and generate aligned SIMD move instructions instead of unaligned ones. This results in a significant performance improvement on some platforms.

— To implement low-latency file I/O, it is useful to have pointers to data known to be aligned to the cache line size, the page size, the huge page size and so on. This way the algorithm can make the correct assumptions for data that can be accessed via DMA (direct memory access).

See [P0886R0] for a more detailed discussion of these use cases and motivating code examples.

## 2 The problem

Consider a function modifying a range of contiguously stored numerical data:

```
void mult(float* x, int size, float factor)
{
    for (int i = 0; i < size; ++i)
        x[i] *= factor;
}
```

Assume further that a static invariant of the program is that `float* x` will always point to a buffer aligned with the SIMD register size.

How can we make the over-alignment property transparent to the compiler? In case `size` is known at compile time, we could wrap the buffer into a class with an alignment specification:

```
template <typename T, std::size_t size, std::size_t alignment>
struct alignas(alignment) aligned_pack
{
   T data[size];
};
```

But what if the data is dynamically allocated and `size` is not known at compile time?

In this case, even if there is some class wrapping the data, access will ultimately be through a pointer of some type `T*`. Given such a pointer, the compiler will always assume the natural alignment requirement of type `T`, because any such pointer value is considered valid. We are missing out on optimisations that would otherwise be easy for the compiler to perform.

How can we fix this? Any attempt to create a wrapper class that works like `aligned_pack` above, but with a dynamic buffer size, will not work using current standard C++. Existing tools such as `alignof` and `alignas` are useless: they only operate on *types*, but here, alignment is the property of a *value*. This property is not static, but dynamic — it will cease to be true after the pointer is incremented or assigned to. There is currently no way in C++ to express such a property, and this problem cannot be solved via the type system.

## 3 Status quo: platform-specific compiler built-ins

Several popular compiler vendors already offer custom built-ins that exist specifically to solve this problem. Unfortunately, the syntax and semantics vary between vendors.

GCC and Clang offer a built-in function

```
void* __builtin_assume_aligned(const void* ptr, size_t N);
```

which returns its first argument, and allows the compiler to assume that the returned pointer is aligned with least `N` bytes. ICC offers a similar built-in,

```
__assume_aligned(ptr, N);
```

but it is a statement rather than a function call, and does not have a return value. MSVC has a different built-in,

```
__assume(expression)
```

This is more generic, but can be used to achieve the same effect. Other languages offer similar tools: OpenMP has `#pragma omp simd aligned`, and Fortran has a compiler directive `ASSUME_ALIGNED`.

The main goal of the paper is to standardise existing practice: we propose to add a standard C++ version of the above built-ins to enable cross-platform use of this feature.

## 4 Previous work and EWG guidance

A previous paper [P0886R0] proposed to add this functionality to C++ via a new standard attribute, `[[assume_aligned(N)]]`. It was discussed at the 2018 Jacksonville WG21 meeting. The feedback given by EWG can be summarised as follows:

— We want to have this functionality in C++.

— We do not want a new attribute that appertains to *values*, even though syntactically it appears to appertain to *objects*.

— We do not want to add a core language feature (such as an attribute) for this, if it can be done by adding a library feature.

— We would prefer a "magic" library function (somewhat similar to `std::launder`).

The present proposal satisfies all of the above requirements.

## 5 Proposed solution

We propose the following new standard library function template:

```
template<size_t N, class T>
  constexpr T* assume_aligned(T* ptr) noexcept;
```

It takes a pointer and returns it unchanged, but allows the compiler to assume that the returned pointer is aligned with `N` bytes. If the pointer passed in is *not* aligned with `N` bytes, the behaviour is undefined.

These semantics are equivalent to GCC and Clang's `__builtin_assume_aligned`. The two main differences are: the alignment `N` is a non-type template parameter (rather than a function parameter), and the function operates on pointers of some concrete type `T*` instead of `void*` pointers. Both modifications facilitate usage of `assume_aligned` in modern, generic C++ code. We further added `constexpr` and `noexcept`.

A call to this function in client code (re-using the example from section 2) could look like this:

```
void mult(float* x, int size, float factor)
{
    float* ax = std::assume_aligned<64>(x);    // we promise that x is aligned with 64 bytes
    for (int i = 0; i < size; ++i)             // loop will be optimised accordingly
        ax[i] *= factor;
}
```

A function that returns a pointer `T*`, and guarantees that it will point to over-aligned memory, could return like this:

```
T* get_overaligned_ptr()
{
    // code...
    return std::assume_aligned<N>(_data);
}
```

This technique can be used e.g. in the `begin()` and `end()` implementations of a class wrapping an over-aligned range of data. As long as such functions are inline, the over-alignment will be transparent to the compiler at the call-site, enabling it to perform the appropriate optimisations without any extra work by the caller.

## 6 Possible implementations

`std::assume_aligned` is implementable today on all major C++ compilers by exploiting the aforementioned compiler built-ins. The following example implementation works for `N` $\leq 128$ on current versions of Clang, GCC, MSVC, and ICC:

```
#include <cstddef>
#include <cstdint>

template <std::size_t N, typename T>
#if defined(__clang__) || defined(__GNUC__)
__attribute__((always_inline))
#elif defined(_MSC_VER)
__forceinline
#endif
constexpr T* assume_aligned(T* ptr) noexcept
{
#if defined(__clang__) || (defined(__GNUC__) && !defined(__ICC))
    return reinterpret_cast<T*>(__builtin_assume_aligned(ptr, N));
#elif defined(_MSC_VER)
    if ((reinterpret_cast<std::uintptr_t>(ptr) & ((1 << N) - 1)) == 0)
        return ptr;
    else
        __assume(0);
#elif defined(__ICC)
    switch (N) {
        case 2:   __assume_aligned(ptr, 2);   break;
        case 4:   __assume_aligned(ptr, 4);   break;
        case 8:   __assume_aligned(ptr, 8);   break;
        case 16:  __assume_aligned(ptr, 16);  break;
        case 32:  __assume_aligned(ptr, 32);  break;
        case 64:  __assume_aligned(ptr, 64);  break;
        case 128: __assume_aligned(ptr, 128); break;
    }
    return ptr;
#else
    // Unknown compiler — do nothing
    return ptr;
#endif
}
```

For compilers that do not support this optimisation, we can provide a trivial implementation that does nothing and just returns its argument unchanged — calling the function would have no effect. Such behaviour is conforming to the wording proposed here.

There is a striking similarity between the over-alignment requirement and a contract. Indeed, if [P0542R4] were adopted, we could consider implementing `std::assume_aligned` using a contract. The following might work on some compilers:

```
#include <cstddef>
#include <cstdint>

template <std::size_t N, typename T>
constexpr T* assume_aligned(T* ptr) noexcept
    [[expects: reinterpret_cast<std::uintptr_t>(ptr) & ((1 << N) - 1)) == 0]]
{
    return ptr;
}
```

However, the expression inside the contract precondition above is not portable. The current definition of [basic.align] does not allow for the conclusion that this expression (or, in fact, *any* expression in standard C++) is equivalent to the statement "the value of `ptr` is an address aligned with `N` bytes". Further, such an implementation has to rely on the compiler being able to correctly interpret the meaning of such an expression and derive the desired optimisation opportunity from it.

Even though the *implementation* of `std::assume_aligned` relies on the compiler to work, it would provide a platform-independent *interface*, thus freeing the end user from using compiler-specific built-ins or incantations like the contract expression above.

## 7 Proposed wording

The proposed changes are relative to the C++ working paper [Smith2018].

Add to **Header `memory` synopsis [memory.syn]**:

```
// [ptr.aligned], pointer alignment hint
template<size_t N, class T>
  constexpr T* assume_aligned(T* ptr) noexcept;
```

Add to **Header `memory` [memory]**:

### Assume aligned [ptr.aligned]

```
template<size_t N, class T> constexpr T* assume_aligned(T* ptr) noexcept;
```

*Requires:*

- `N` shall be a power of two
- `ptr` shall represent the address of an object or array of objects of type `T`
- This address shall be aligned to at least `N` bytes

*Returns:*

The value of `ptr`.

*Remarks:*

This function returns its argument unchanged, but allows the compiler to assume that the returned pointer is aligned to at least `N` bytes. If the value of the pointer passed in is not aligned with at least `N` bytes, the effect of calling this function is undefined. [ *Note:* The alignment assumption expressed by a call to `assume_aligned` may result in generation of more efficient code. It is up to the program to ensure that the assumption actually holds. The call does not cause the compiler to verify or enforce this. — *end note* ]

## Acknowledgements

Many thanks to Fabian Renn-Giles for providing the idea and motivation for this work, and to Gašper Ažman, Arthur O'Dwyer, Niall Douglas, Mathias Gaunard, Peter Dimov, Jason McGuiness, and Antony Peacock for their very helpful comments.

## References

[P0542R4] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, and B. Stroustrup. Support for contract based programming in C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r4.html, 2017 (accessed 2018-05-04).

[P0886R0] Timur Doumler. The assume aligned attribute. https://wg21.link/p0886, 2018 (accessed 2018-05-04).

[Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2018 (accessed 2018-05-04).