Memory alignment

Donnerstag, 7. Juli 2022 12:43

Author: Damir Ljubic
email: damirlj@yahoo.com

This is short introduction to a **memory alignment**.
Be aware that, when we talk about memory alignment, we talk about the virtual address space (abstraction), and not about the physical memory itself.
Some key-notes, that will be demonstrated with attached code snippet:

- Alignment must be power of 2

```
/*
 *   The number a is power of two ( a = 2^n == 1 << n ), if
 *   it has only one bit (nth) set to 1.
 *   The most convenient way to test it is: "a & (a-1) == 0"
*/
template <typename T>
constexpr bool is_power_of_2(const T n) noexcept
{
    static_assert(is_unsigned_integral_v<T>, "<Error> Works only with unsigned integral types!");

    if (n < 1) return false;
    return (n & (n-1)) == 0;
}
```

  More on the bit operations: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/Bits.pdf

  The scalar types alignment is determined by their memory footprint - *sizeof*(T).
  The stricter alignment on a type can be generally enforced with *alignas*(N), where N is integral constant expression that is power of 2, and can be inspect with *alignof*(T).
  The maximal natural alignment for embedded types is *std::max_align_t* (basically, *alignof*(*long double*)).
  This is also known as *fundamental alignment*. The alignments that are beyond this value are labeled as *extended alignments*.

- The sizeof(T), for user defined (aggregate) types, will be multiple of alignment.
  The alignment of a type itself is determined with the most strictness alignment of a containing non-static field.
  The alignas specifier can be applied on the type definition as well: *struct **alignas**(N) T{…};* and will be therefore participate in resolving the alignment of a T.
- The address will be multiple of alignment (the *log_2(alignment)* LSB bits of address will be zeroed), for the solely purpose:
  to fetch the physical memory location in one instruction.
- For the arrays of T, the alignment will be alignof(T): setting explicitly *alignment* will have no direct effect on alignment of elements, except on the address of array itself - and therefore, the address of all elements in array - as multiple of *alignment*.
  The reason for that is specified by the standard - no padding zeros between successive elements in array are prohibited.
  But, there is a way to overcome this limitation (@see example)

For other subjects related with this topic, like "*false sharing*" I would recommend "***Embracing modern C++ safely***".
The story behind: the term refers on the fact that two (or more) threads hosted on the same core, which fetch the logically independent values, as result of the locality - the fact that these values may be part of the same cache line,
they access will be synchronized by the underlying system (hardware lock), which degrades their performance.
One solution would be to enforce these values to be in different cache lines - through the alignment

```
alignas(64) int val1 = ...
alignas(64) int val2 = ...
```

Starting with C++17, there is portable way to determine the size of the L1 cache line: std::hardware_destructive_interference_size.
The obvious issue of this approach is memory consumption.
The better approach is to use per-thread (*thread_local*) local allocators (*std::pmr::monotonic_buffer_resource*).

More on that: modern_cpp_tutorials/Local allocator.pdf

The code to play with
https://godbolt.org/z/M4c4W7n5r
(Compiler Explorer)

```
#include <iostream>
#include <iterator>
#include <cstddef>
#include <numeric>

struct test // inefficient alignment: sizeof(test) = 24
{
   std::uint8_t a; // 1 | 7 padding 0
   long b;         // 8 bytes
   int c;          // 4 | 4 padding 0

   friend std::ostream& operator << (std::ostream& out, const test& t)
   {
```

```
        out << "a=" << static_cast<std::uint16_t>(t.a);
        out << ", addr(a)=" << (void*)(&t.a) << ",\n";
        out << "b=" << t.b;
        out << ", addr(b)=" << (void*)(&t.b) << ",\n";
        out << "c=" << t.c;
        out << ", addr(c)=" << (void*)(&t.c) << '\n';
        out << '\n';

        return out;
    }
};

struct test2 // efficient alignment: sizeof(test2) = 16
{
    long a;           // 8 bytes
    int b;            // 4 bytes
    std::uint8_t c;   // 1 | 3 padding 0

    friend std::ostream& operator << (std::ostream& out, const test2& t)
    {
        out << "a=" << t.a;
        out << ", addr(a)=" << (void*)(&t.a) << ",\n";
        out << "b=" << t.b;
        out << ", addr(b)=" << (void*)(&t.b) << ",\n";
        out << "c=" << static_cast<uint16_t>(t.c);
        out << ", addr(c)=" << (void*)(&t.c) << '\n';
        out << '\n';

        return out;
    }
};


template <typename T, std::size_t N>
void test_array_aligned_to_t(const T& val)
{
    constexpr auto size = sizeof(T);
    // alignas can't be applied to the function parameter: it's the same as alignof(T)
    constexpr auto alignment = alignof(val);
    static_assert(0 == size % alignment, "This is guaranteed by the standard");

    std::cout << "\n<< " << __func__ << " >>: alignment(" << alignment << ")\n\n";
    std::cout << "sizeof(T): " << size << "\n\n";

    alignas(alignment) std::byte mem[N * size]; // "memory" like representation

    // As if we calling 'std::iota()' for array of T

    for (std::size_t i = 0; i < N; ++i)
    {
        auto * addr = reinterpret_cast<T*>(mem) + i; // pointer arithmetic: &mem[i *sizeof(T)]

        // Construct the object T into "memory" - placement new
        new (addr)T(val); // T must be - in this case, copy-constructible
        std::cout << "&mem[" << i << "]=" << addr << '\n';
        std::cout << *addr;
    }
}


template <typename T, std::size_t N, std::size_t alignment = alignof(T)>
void test_aligned_array_of_t(const T& val)
{
    constexpr auto size = std::max(alignment, sizeof(val)); // in case: alignment > sizeof(T)

    std::cout << "\n<< " << __func__ << " >>: alignment(" << alignment << ")\n\n";
    std::cout << "sizeof(T): " << sizeof(T) << "\n\n";

    // "memory" like representation: make sure that &mem[0] is multiple of alignment
    alignas(alignment) std::byte mem[N * size];

    // As if we calling 'std::iota()' for array of T

    for (std::size_t i = 0; i < N; ++i)
    {
        auto* addr = &mem[i * size]; // forcing the given alignment as address offset (for alignment > sizeof(T))

        // Construct the object T into "memory" - placement new
        new (addr)T(val); // T must be - in this case, copy-constructible
        std::cout << "&mem[" << i << "]=" << addr << '\n'; // addressof(T)
        std::cout << *reinterpret_cast<T*>(addr); // T::operator <<
    }
}
```

```cpp
/*
* For arrays of T, the padding zeros between element is
* explicitly prohibited by the standard.
* Therefore, specifying the alignment bigger than alignof(T) has no direct
* effect on alignment of elements in array- it affects the address boundary of the array itself (multiple of alignment),
* as with any other data type
*/
template <typename T, std::size_t N, std::size_t alignment = alignof(T)>
std::enable_if_t<std::is_arithmetic_v<T>>
test_array_of_aligned_arithmetics(T init)
{
    std::cout << "\n<< " << __func__ << " >>: alignment(" << alignment << ")\n\n";

    // alignment specifier affects here the address of 'arr' - not the alignment of elements in array.
    // @see https://godbolt.org/z/v83W85zGz
    alignas(alignment) T arr[N];

    std::iota(std::begin(arr), std::end(arr), init);

    std::cout << "sizeof(arr): " << sizeof(arr) << "[bytes]\n";
    std::copy(std::begin(arr), std::end(arr), std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';

    for (std::size_t i = 0; i < N; ++i)
    {
        auto* addr = arr + i; // pointer arithmetic

        std::cout << "&arr[" << i << "]=";
        std::cout << addr << '\n';
        std::cout << *addr << '\n';
    }
}

int main()
{
     // User-defined types

    test_array_aligned_to_t<test, 10>(test{.a = 1, .b = 2, .c = 3});
    test_array_aligned_to_t<test2, 10>(test2{.a = 4, .b = 5, .c = 6});

     /*
     * Doesn't affect the size, only alignment (address).
     * Only by definition of type
     * struct alignas(32) T{...};
     * if the 32 is the strictness alignment (bigger than any
     * alignment of containing non-static field), the
     * sizeof(T) == alignof(T) == 32
     */


    alignas(32) auto t = test{.a = 1, .b = 2, .c = 3};
    std::cout << "alignof(t)= " << alignof(t) << '\n';
    std::cout << "sizeof(t)= " << sizeof(t) << '\n';
    // "alignas" can't be applied to the function parameter - it will be stripped away
    test_array_aligned_to_t<test, 10>(t);

    // Forcing other alignment than alignof(T) on array of T elements
    alignas(32) auto t1 = test{.a = 1, .b = 2, .c = 3};
    test_aligned_array_of_t<decltype(t1), 10, alignof(t1)>(t1);

     // Scalar - arithmetic types

    test_array_of_aligned_arithmetics<int, 10>(255);
    // has no effect on T alignment, but rather affects the address of array
    test_array_of_aligned_arithmetics<int, 10, 8>(1);

    test_array_of_aligned_arithmetics<double, 10>(1);
}
```

Bonus chapter: **std::align**

Unlike the *memaling()* and *posix_memalign()* which do allocate memory on the heap, returning the address aligned to a given boundary - which must be power of two, std::align doesn't allocate memory at all - it only aligns the given pointer to a preallocated memory, assuming there is enough space remained in memory storage for a required size in bytes, or returns the *nullptr* otherwise.
As you can guess, it's useful when you write your own (local) allocator, to maintain the preallocated (on stack) memory alig ned.
If you doubt how the std::align is internally implemented, check the link below

https://godbolt.org/z/xP1dd9sG5

```cpp
namespace details
{
    constexpr void* align(std::size_t alignment, std::size_t size, void*& ptr, std::size_t& space) noexcept
    {
        // Switch to integral arithmetic
        const auto addr = reinterpret_cast<std::uintptr_t>(ptr);

        // Round pointer - address up to given alignment
        const auto alignedAddr = (addr + alignment -1u) & ~(alignment - 1u);

        // Cost of alignment to be considered
        const auto diff = alignedAddr - addr;

        if ( (size + diff) > space) { return nullptr; }
        else
        {
            // Update remaining space with alignment
            space -= diff;
            // Aligned pointer
            return ptr = reinterpret_cast<void*>(alignedAddr);
        }
    }
}
```