Dynamic Memory Allocation for Over-Aligned Data

This lesson explains memory alignment in C++ 17 and how it fixes the holes in C++11/14 memory alignment.

WE'LL COVER THE FOLLOWING

Embedded environments, kernel, drivers, game development and other areas might require a non-default alignment for memory allocations. Complying those requirements might improve the performance or satisfy some hardware interface.

Memory Alignment

For example, to perform geometric data processing using SIMD[^simd] instructions, you might need 16-byte or 32-byte alignment for a structure that holds 3D coordinates:

> **[^simd]:** Single Instruction, Multiple Data, for example, SSE2, AVX, see https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

```
struct alignas(32) Vec3d { // alignas is available since C++11
    double x, y, z;
};
auto pVectors = new Vec3d[1000];
```

`Vec3d` holds `double` fields, and usually, its natural alignment should be 8 bytes. Now, with `alignas` keyword, we change this alignment to 32. This approach allows the compiler to fit the objects into SIMD registers like AVX (256-bit-wide registers).

Aligning Memory in C++ 17

Unfortunately, in C++11/14, you have no guarantee how the memory will be aligned after `new[]`. Often, you have to use routines like `std::aligned_alloc()` or MSVC's `_aligned_malloc()` to be sure the alignment is preserved. That's not ideal as it's not working easily with smart pointers and also makes memory management visible in the code.

C++17 fixes that hole by introducing new memory allocation function overloads for `new()` and `delete()` with the `align_val_t` parameter. Example function signatures below[^allnewfunctions]:

> **[^allnewfunctions]:** See all 22 `new()` overloads at https://en.cppreference.com/w/cpp/memory/new/operator_new

```
void* operator new(size_t, align_val_t);
void operator delete(void*, size_t, align_val_t);
```

The Standard also defines `__STDCPP_DEFAULT_NEW_ALIGNMENT__` macro that specifies the default alignment for dynamic memory allocations. On common platforms, Clang, GCC and MSVC specify it as 16 bytes.

Now, in C++17, when you allocate:

```
auto pVectors = new Vec3d[1000];
```

The alignment for `Vec3d` is larger than `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, and thus the compiler will select the overloads with the `align_val_t` parameter.

> In Clang and GCC you can control the default alignment by using the `fnew-alignment` switch (see Clang's documentation). The MSVC compiler exposes the `/Zc:alignedNew` flag that turns the feature on or off.

We can also provide custom implementation, have a look:

```cpp
void* operator new(std::size_t size, std::align_val_t align) {
#if defined(_WIN32) || defined(__CYGWIN__)
```

```
#if defined(_WIN32) || defined(__CYGWIN__)
    auto ptr = _aligned_malloc(size, static_cast<std::size_t>(align));
#else
    auto ptr = std::aligned_alloc(static_cast<std::size_t>(align), size);
#endif

    if (!ptr) throw std::bad_alloc{};

    std::cout << "new: " << size << ", align: "
              << static_cast<std::size_t>(align) << ", ptr: " << ptr << '\n';

    return ptr;
}

void operator delete(void* ptr, std::size_t size, std::align_val_t algn) noexcept {
    std::cout << "delete: " << size << ", align: "
              << static_cast<std::size_t>(algn) << ", ptr : " << ptr << '\n';
#if defined(_WIN32) || defined(__CYGWIN__)
    _aligned_free(ptr);
#else
    std::free(ptr);
#endif
}

void operator delete(void* ptr, std::align_val_t algn) noexcept { ... } // hidden
```

The code uses `_aligned_malloc()` and `_aligned_free()` for the Windows version[^wincompileralgn]. It's because the Windows platform uses different allocation mechanisms for over-aligned data, and that's why `std::free()` wouldn't release the memory correctly. On other platforms that conform with C11 you can try using `std::aligned_alloc()`, as since C++17 the Standard is based on the C11 specification. In that context `free()` can delete the aligned memory.

> **[^wincompileralgn]:** This applies to MSVC, MinGW, Clang on Windows or Cygwin.

The new functionality can significantly improve the code, and you can now hold the aligned objects in standard containers without writing custom allocators, or custom deleters for smart pointers.

For example:

```
std::vector<Vec3d> vec;
vec.push_back({});
vec.push_back({});
vec.push_back({});
```

```
assert(reinterpret_cast<uintptr_t>(vec.data()) % alignof(Vec3d) == 0);
```

When executed, our replaced allocation functions might log the following output:

```
new: 32, align: 32, ptr: 000001F866625960
new: 64, align: 32, ptr: 000001F866625680
delete: 32, align: 32, ptr : 000001F866625960
new: 96, align: 32, ptr: 000001F866623EA0
delete: 64, align: 32, ptr : 000001F866625680
delete: 96, align: 32, ptr : 000001F866623EA0
```

The example allocates memory for a single entry, then deletes it and increases the size for the vector twice to make space for all three elements. At the end we check the pointer alignment to be sure it's aligned to 32 bytes.

You can read more about the experiments with the new functionality at New new() - The C++17's Alignment Parameter for Operator new(). This blog post also shows the dangerous side when you try to ask for a non-standard alignment with placement new.

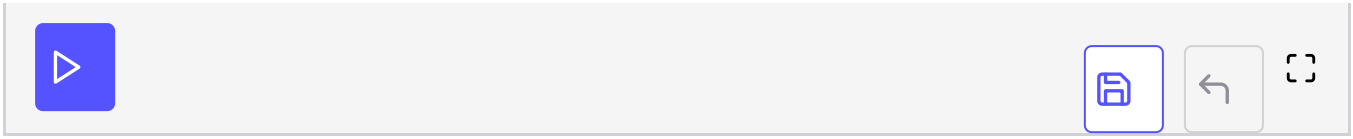
> *Extra Info:* The change was proposed in: P0035.

Here is the full code for you to execute (discussed code has been highlighted):

```
#include <cassert>
#include <iostream>
#include <memory>
#include <new>
#include <vector>

#if defined(_WIN32) || defined(__CYGWIN__)
#    include <malloc.h> // _aligned_malloc/_aligned_free

    void* custom_aligned_alloc(size_t size, size_t alignment) {
        return _aligned_malloc(size, alignment);
    }
    void custom_aligned_free(void* ptr) {
        _aligned_free(ptr);
    }
#else
#    include <cstdint>  // aligned_alloc from C11

    void* custom_aligned_alloc(size_t size, size_t alignment) {
        return std::aligned_alloc(alignment, size);
    }
```

```
    void custom_aligned_free(void* ptr) {
        std::free(ptr);
    }

#endif

void* operator new(std::size_t size, std::align_val_t align) {
    auto ptr = custom_aligned_alloc(size, static_cast<std::size_t>(align));

    if (!ptr)
        throw std::bad_alloc{};

    std::cout << "new: " << size << ", align: " << static_cast<std::size_t>(align) << ", ptr:

    return ptr;
}

void operator delete(void* ptr, std::size_t size, std::align_val_t align) noexcept {
    std::cout << "delete: " << size << ", align: " << static_cast<std::size_t>(align) << ", p
    custom_aligned_free(ptr);
}

void operator delete(void* ptr, std::align_val_t align) noexcept {
    std::cout << "delete: align: " << static_cast<std::size_t>(align) << ", ptr : " << ptr <<
    custom_aligned_free(ptr);
}

struct alignas(32) Vec3dAVX {
    double x, y, z;
};

int main() {
    std::cout << "__STDCPP_DEFAULT_NEW_ALIGNMENT__ is " << __STDCPP_DEFAULT_NEW_ALIGNMENT__ <

    std::cout << "sizeof(Vec3dAVX) is " << sizeof(Vec3dAVX) << '\n';
    std::cout << "alignof(Vec3dAVX) is " << alignof(Vec3dAVX) << '\n';

    {
        std::cout << "---- new Vec3dAVX[10]\n";
        auto pVec = new Vec3dAVX[10];
        assert(reinterpret_cast<uintptr_t>(pVec) % alignof(Vec3dAVX) == 0);
        delete[] pVec;
    }

    {
        std::cout << "---- new int[10]\n";
        auto p2 = new int[10];
        delete[] p2;
    }

    {
        std::cout << "---- vector<Vec3dAVX>\n";
        std::vector<Vec3dAVX> vec;
        vec.push_back({});
        vec.push_back({});
        vec.push_back({});
        assert(reinterpret_cast<uintptr_t>(vec.data()) % alignof(Vec3dAVX) == 0);
    }

    {
        std::cout << "---- unique_ptr<Vec3dAVX[]>\n";
        auto pUnique = std::make_unique<Vec3dAVX[]>(10);
    }
```

```
}
```



Now let’s take a look at Exception Specification features in C++ 17 in the next lesson.