4 January 2024 (Draft No. 4)

`std::unaligned<T>` and `typedef<packed>`

by TPK Healy

This paper describes a template class to be added to the `std` namespace inside the standard header `<memory>`:

```
namespace std {
    template<typename T>
    class unaligned {
        array<byte, __datasizeof(T)> _m_data;
    public:
        using value_type = T;
        static constexpr size_t alignment = alignof(T);
        class scoped_manipulator { . . . };

        template<typename D, typename S>
        static void unaligned_relocate(D *const d, S *const s) noexcept { . . . }

        . . .
        . . .
    };
}
```

The '`std::unaligned`' class can be used to store a contained object of type **T** in unaligned memory, and can be used to manipulate the contained object.

A '*packed struct*' is a struct defined using '`typedef<packed>`'. It prises open the curly braces of a pre-existing struct and changes every member variable into an '`std::unaligned`'. For example the following two pieces of code are equivalent:

```
struct Monkey {
    int a;
    double b;
    char c;
public:
    std::string d;
};

struct PackedMonkey {
    std::unaligned<int> a;
    std::unaligned<double> b;
    std::unaligned<char> c;
public:
    std::unaligned<std::string> d;
};
```

```
struct Monkey {
    int a;
    double b;
    char c;
public:
    std::string d;
};

typedef<packed> Monkey PackedMonkey;
```

The '`std::unaligned`' class is designed so that every operation that is trivial for **T** will also be trivial for '`std::unaligned<T>`' (e.g. trivial destructor, trivial assignment, trivial copy-constructor, etc.). The '`std::unaligned`' class can also be used with types that have nontrivial operations (e.g. nontrivial destructor), such as '`std::unaligned< std::vector<int> >`'.

Before the '`std::unaligned`' class can perform an operation on the contained object, it must relocate the bytes of the contained object from unaligned memory into aligned memory. This relocation operation shall be performed by the first matching rule **A** or **B** or **C**.

A. If **T** contains a static member function named '`_Relocate`', invoke the '`T::_Relocate`' static member function. (A program is illformed if a class contains a nonstatic member function named '`_Relocate`', and the compiler shall issue a diagnostic). There are four overloads of the static member function '`T::_Relocate`' in order to facilitate the work of micro-optimisers:

```
void _Relocate(void*,void*) noexcept; (1)
void _Relocate(void*,T   *) requires (!std::is_void_v<T>) noexcept; (2)
void _Relocate(T   *,void*) requires (!std::is_void_v<T>) noexcept; (3)
void _Relocate(T   *,T   *) requires (!std::is_void_v<T>) noexcept; (4)
```

(1) relocates from unaligned to unaligned – (implementation is mandatory)
(2) relocates from aligned to unaligned – (will fall back to $^{(1)}$ if missing)
(3) relocates from unaligned to aligned – (will fall back to $^{(1)}$ if missing)
(4) relocates from aligned to aligned – (will fall back to $^{(3)}$ or $^{(2)}$ or $^{(1)}$ if missing)

B. If there exists in the global namespace a specialisation of the template class, '`_Relocator_Custom`', which accepts **T** as a template parameter, invoke:

```
::_Relocator_Custom<T>::R::relocate( destination , source );
```

(*Micro-optimisers may provide four overloads of this static member function*)

C. Use '`std::memcpy`' to relocate the bytes.

Here is a possible implementation of the static member function which performs the relocation:

```
template<typename D, typename S>
requires    ((is_void_v<D> || is_same_v<D,T>)
          &&  is_void_v<S> || is_same_v<S,T>))
static void unaligned_relocate(D *const d, S *const s) noexcept
{
    if constexpr ( requires { T::_Relocate(d,s); } )
        { T::_Relocate(d,s); }
    else if constexpr ( requires { T::_Relocate(static_cast<void*>(d),s); } )
        { T::_Relocate(static_cast<void*>(d),s); }
    else if constexpr ( requires { T::_Relocate(d,static_cast<void*>(s)); } )
        { T::_Relocate(d,static_cast<void*>(s)); }
    else if constexpr(requires{T::_Relocate(static_cast<void*>(d),static_cast<void*>(s));})
        { T::_Relocate(static_cast<void*>(d),static_cast<void*>(s)); }
    else if constexpr ( !is_void_v< typename _Relocator_Custom<T>::R > )
    {
        using R = typename _Relocator_Custom<T>::R;
        if constexpr ( requires { R::relocate(d,s); } )
            { R::relocate(d,s); }
        else if constexpr ( requires { R::relocate(static_cast<void*>(d),s); } )
            { R::relocate(static_cast<void*>(d),s); }
        else if constexpr ( requires { R::relocate(d,static_cast<void*>(s)); } )
            { R::relocate(d,static_cast<void*>(s)); }
        else
            { R::relocate(static_cast<void*>(d),static_cast<void*>(s)); }
    }
    else
    {
        memcpy(d,s,__datasizeof(T));
    }
}
```

The '`std::unaligned`' class is similar in functionality to the '`std::synchronized_value`' class in that a proxy object is used as an intermediary, as follows:

```
std::unaligned<double> mydub;
std::unaligned<double>::scoped_manipulator m = mydub.align();
  or more simply:    auto m = mydub.align();
double &d = m.value();
d = 78.2;
```

Upon construction of the manipulator object, the contained object is relocated ***from*** unaligned memory.
Upon destruction of the manipulator object, the contained object is relocated ***back into*** unaligned memory.
For convenience, the '`std::unaligned`' class has overloaded operators and forwarding constructors, allowing for:

```
std::unaligned<double> mydub(78.2);
mydub = 56.3;

std::unaligned< std::vector<double> > v(10u, 123.45);
v->push_back(6);
```

The '`std::unaligned`' class has a specialisation for `alignof(T) == 1`, as the relocation operation into aligned memory is unnecessary and is therefore elided.

Optionally, you can provide your own relocator class to use with a specific type. Your custom relocator class must implement the static member function ‘relocate’ that accepts two 'void*' pointers (the other three overloads are not mandatory and are for use by micro-optimisers). Here is an example of a custom relocator class for the **libstdc++** implementation of 'std::basic_string' (which is not trivially relocatable):

```
#include <memory>         // unaligned<T>

template<typename T>
struct MyRelocatorForBasicString {
    typedef int Tag_Relocator;  // so we know that this is a Relocator class
    using value_type = T;
    static void relocate(void *const dst_v, void *const src_v) noexcept // both unaligned
    {
        using std::byte, std::memcpy;

        byte *const dst = static_cast<byte*>(dst_v),
             *const src = static_cast<byte*>(src_v);

        // Step 1: Copy the bytes
        memcpy(dst, src, __datasizeof(T));

        // Step 2: Pluck out the pointer (which might be unaligned)
        //         (The pointer is located at offset 0 from 'this')
        byte *p;
        memcpy(&p, src, sizeof p);

        // Step 3: Check if the pointer points to within the object (and if not, return)
        if ( p <   src ) return;
        if ( p >= (src + __datasizeof(T)) ) return;

        // Step 4: Adjust the pointer
        p += dst - src;

        // Step 5: Store in (possibly unaligned) destination
        memcpy(dst,&p,sizeof p);
    }
};

template<typename T>
struct _Relocator_Custom< std::basic_string<T> > {
    typedef MyRelocatorForBasicString< std::basic_string<T> > R;
};

int main(void)
{
    std::unaligned<std::string> monkey('a', 5);
    monkey->resize(4u);
}
```

It is permissible to treat a pre-existing block of memory as an object of '`std::unaligned`', as follows:

```
int main(void)
{
   struct Monkey {
       int a;
       long b;
       char c;
   };

   char buf[32u] = {};

   unaligned<Monkey> &m = *std::start_lifetime_as< unaligned<Monkey> >( buf + 3u );

   m->a = 7;
}
```

For a sample implementation of '`std::unaligned`', adapted from Connor Horman's original code, along with example usages, see GodBolt:

https://godbolt.org/z/W4se6j3dG

Please respond to this paper on the C++ Standard Proposals Mailing List:

https://lists.isocpp.org/mailman/listinfo.cgi/std-proposals

You can view the original discussion here:

https://lists.isocpp.org/std-proposals/2023/12/8585.php

Change Log:

- **Draft No. 2 - GodBolt implementation was missing '`return *this;`'.**
- **Draft No. 3 - Treat pre-existing block of memory as '`std::unaligned`'.**
- **Draft No. 4 - Custom relocators are now implemented with '`_Relocator_Custom`'.**

FIN