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

by TPK Healy

This paper describes a template class to be added to the `std` namespace:

```
template<typename T, typename Relocator = std::relocator<T> >
class std::unaligned {
protected:
    std::byte buf[ __datasizeof(T) ];
public:
    . . .
};
```

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:
    float f;
};

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

```
struct Monkey {
    int a;
    double b;
    char c;
public:
    float f;
};

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. To perform this relocation, the standard header `<memory>` contains the definition of the standard relocator, '`std::relocator<T>`', which shall do one of two things:

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 ***non***static 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 **T** does not contain a static member function named '`_Relocate`', use '`std::memcpy`' to relocate the bytes.

Here is a possible implementation of '`std::relocator<T>`':

```
template<typename T>
requires std::is_same_v< std::remove_cvref_t<T>, T >
struct std::relocator {
    using value_type = T;
    template<typename D, typename S>
    requires   ((std::is_void_v<D> || std::is_same_v<D,T>)
             && (std::is_void_v<S> || std::is_same_v<S,T>))
    static void 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
        {
            std::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;
```

The contained object is relocated ***from*** unaligned memory upon construction of the manipulator object, and the contained object is relocated ***back into*** unaligned memory upon destruction of the manipulator object. 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 when `alignof(T) == 1`, as the relocation operation into aligned memory is unnecessary and is therefore elided.

see next page

Optionally, you can provide your own `Relocator` class to use with '`std::unaligned`' . Your custom relocator must implement the static member function '`relocate`' that takes 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 use with the **libstdc++** implementation of '`std::basic_string`' which is not trivially relocatable:

```
template<typename T>
struct MyBasicStringRelocator {
    using value_type = T;
    static void relocate(void *const dst, void *const src) noexcept /* both unaligned */
    {
        using std::byte, std::memcpy;

        byte *const dstb = static_cast<byte*>(dst),
             *const srcb = static_cast<byte*>(src);

        // 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')
        typename T::value_type *pval;
        memcpy(&pval, src, sizeof pval);
        byte *p = static_cast<byte*>(static_cast<void*>(pval));

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

        // Step 4: Adjust the pointer
        p += dstb - srcb;

        // Step 5: Convert to pointer type and store in (possibly unaligned) destination
        pval = static_cast<typename T::value_type*>(static_cast<void*>(p));
        std::memcpy(dst,&pval,sizeof pval);
    }
};

int main(void)
{
    std::unaligned< std::string, MyBasicStringRelocator<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] = {};

#if __cpp_lib_start_lifetime_as
   unaligned<Monkey> &m = *std::start_lifetime_as< unaligned<Monkey> >(buf+3u);
#else
   unaligned<Monkey> &m = *static_cast< unaligned<Monkey>* >(static_cast<void*>(buf+3u));
#endif

   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/q4c14aPox

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`'.**

FIN