
Subject: Re: Impressive improvement in `std::vector` when dealing with raw memory.
Posted by [Lance](#) on Mon, 14 Nov 2022 14:28:52 GMT

[View Forum Message](#) <> [Reply to Message](#)

Extract of <CoreExt/relocatable.hpp>

```
// a value of -1 indicate that we don't know if the class('s object) is trivially
// relocatable and if not, how to do adjustment after relocation
//
template <class T> struct relocate_traits { static constexpr int value = -1; };
```

```
// a value of 0 indicates object of T can be move around like raw bytes.
```

```
template <class T> requires std::is_trivial_v<T>
struct relocate_traits<T> { static constexpr int value = 0; };
```

```
template <class T>
concept UppMoveable = requires{
    typename T::MoveableBase; // Note, this require to add a typedef in Topt.h/Moveable definition
    requires std::derived_from<T, Upp::Moveable<T, typename T::MoveableBase> >;
};
```

```
// the following will teach compiler to treat all Upp::Moveable derivatives as
```

```
// trivially relocatable
```

```
template <UppMoveable T>
struct relocate_traits<T> { static constexpr int value = 0; };
```

```
template <class T>
inline constexpr bool is_trivially_relocatable_v = relocate_traits<T>::value == 0;
```

With that, the `AssertMoveable` part can be done without. This involves C++20 language features unfortunately (concept/requires). Maybe a extra `#if cplusplusversion>2020`(in the spirit) is needed to make both worlds happy.

And there are more opportunities opened with this approach.

The rest of my <CoreExt/relocatable.hpp>

```
// facility to get the class name from a member function pointer
```

```
//
template<class T> struct get_class;
template<class T, class R>
struct get_class<R T::*> { using type = T; };
```

```
template <class T>
requires requires(T t){
    { t.DoPostRelocationAdjustment() }noexcept;
```

```

requires std::same_as<
  T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
>;
}
struct relocate_traits<T>{
  static constexpr int value = 1; // simple
  static void Do(T* obj)noexcept{ obj->DoPostRelocationAdjustment(); }
};

template <class T>
requires requires(T * p){// need to fix, pointer to derived class should be rejected
  {DoPostRelocationAdjustment(p) }noexcept;
}
struct relocate_traits<T>{
  static constexpr int value = 1; // simple
  static void Do(T* obj)noexcept{ DoPostRelocationAdjustment(obj); }
};

template <class T>
requires requires(T t, const T* old){
  {t.DoPostRelocationAdjustment(old)}noexcept;
  requires std::same_as<
    T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
  >;
}
struct relocate_traits<T>{
  static constexpr int value = 2; // old address is supplied.
  static void Do(T* obj, const T* old)noexcept{ obj->DoPostRelocationAdjustment(old); }
};

template <class T>
requires requires(T *obj, const T * old){
  {DoPostRelocationAdjustment(obj, old)}noexcept;
}
struct relocate_traits<T>{
  static constexpr int value = 2; // old address is supplied.
  void Do(T* obj, const T* old)noexcept{ DoPostRelocationAdjustment(obj, old); }
};

template <class T>
requires requires(T t, const T* old, const T * from, const T * to){
  {t.DoPostRelocationAdjustment(old, from, to)}noexcept;
  requires std::same_as<
    T, typename get_class<decltype(&T::DoPostRelocationAdjustment)>::type
  >;
}
struct relocate_traits<T>{
  static constexpr int value = 4; // 4 address version

```

```

void Do(T* obj, const T* old, const T* from, const T* to)noexcept{
    obj->DoPostRelocationAdjustment(old, from, to);
}
};

```

```

template <class T>
requires requires(T *p, const T * o, const T * from, const T* to){
    {DoPostRelocationAdjustment(p, o, from, to)}noexcept;
}
struct relocate_traits<T>{
    static constexpr int value = 4; // 4 address version.
    void Do(T* obj, const T* old, const T* from, const T* to)noexcept{
        DoPostRelocationAdjustment(obj, old, from, to);
    }
};

```

```

template <class T>
concept relocatable = relocate_traits<T>::value != -1;

```

```

#define DECLARE_TRIVIALY_RELOCATABLE( T ) namespace lz{\
    template <> struct relocate_traits<T>{ const static int value = 0; };\
}

```

```

#define RELOCATE_ADJUSTMENT_1(T, func) namespace lz{\
    struct relocate_traits<T>{\
        static constexpr int value = 1;\
        static void Do(T* obj)noexcept{ func(obj); }\
    };\
}

```

```

#define RELOCATE_ADJUSTMENT_2(T, func) namespace lz{\
    struct relocate_traits<T>{\
        static constexpr int value = 2;\
        static void Do(T* obj, const T* old)noexcept{ func(obj, old); }\
    };\
}

```

```

#define RELOCATE_ADJUSTMENT_4(T, func) namespace lz{\
    struct relocate_traits<T>{\
        static constexpr int value = 4;\
        static void Do(T* obj, const T* old, const T* start, const T* end)noexcept{\
            { func(obj, old, start, end); }\
        };\
}

```

With this, we can teach a vector or Vector to handle objects that's not trivially relocatable, for example, Upp::Ctrl, or some one like this

```

class SomeClassWithBackpointer{
    struct Node{
        SomeClassWithBackpointer * owner;

        void SomeFunction(){}

        char buff[1024];
    }

    SomeClassWithBackpointer()=default;
    SomeClassWithBackpointer(const SomeClassWithBackpointer& ){...}

    void DoPostRelocationAdjustment()noexcept{
        if(node) node->owner = this;
    }

    Node * node = nullptr;
    char buff[1024]; // to make the class heavier
                    // otherwise, it's cheaper
                    // by simply move construct it.
};

```

With this definition, SomeClassWithBackpointer objects can be housed in a (revised)vector/Vector quite efficiently. I have done a trial implementation(incomplete) of such a vector.
