D:\MAL23-P7-G5-Main\Engine\Source\Utilities\CommonUtilities\include\CommonUtilities\Structures\FreeVector.hpp

```cpp
#pragma once

#include <vector>
#include <variant>
#include <functional>
#include <cassert>

#include <CommonUtilities/Utility/JsonUtils.hpp>

#include <CommonUtilities/Config.h>

namespace CommonUtilities
{
    /// Structure that enables for quick insertion and removal from anywhere in the container.
    ///
    template<class T>
    class FreeVector
    {
    public:
        using value_type        = T;
        using reference         = T&;
        using const_reference   = const T&;
        using pointer           = T*;
        using const_pointer     = const T*;
        using size_type         = std::size_t;

        constexpr FreeVector() = default;
        constexpr ~FreeVector() = default;

        constexpr FreeVector(const FreeVector&) = default;
        constexpr FreeVector(FreeVector&&) noexcept = default;

        constexpr auto operator=(const FreeVector&) -> FreeVector& = default;
        constexpr auto operator=(FreeVector&&) noexcept -> FreeVector& = default;
```

```
35
36          NODISC constexpr auto operator[](size_type anIndex) -> reference;
37          NODISC constexpr auto operator[](size_type anIndex) const -> const_reference;
38
39          NODISC constexpr auto at(size_type anIndex) -> reference;
40          NODISC constexpr auto at(size_type anIndex) const -> const_reference;
41
42          NODISC constexpr bool valid(size_type anIndex) const;
43
44          NODISC constexpr bool empty() const noexcept;
45          NODISC constexpr auto size() const noexcept -> size_type;
46          NODISC constexpr auto count() const noexcept -> size_type;
47
48          NODISC constexpr auto max_size() const noexcept -> size_type;
49
50          template<typename... Args> requires std::constructible_from<T, Args...>
51          constexpr auto emplace(Args&&... someArgs) -> size_type;
52
53          constexpr auto insert(const T& anElement) -> size_type;
54          constexpr auto insert(T&& anElement) -> size_type;
55
56          constexpr void erase(size_type anIndex);
57
58          constexpr void clear();
59
60          constexpr void reserve(size_type aCapacity);
61
62          template<typename Func>
63          constexpr void for_each(const Func& func) const;
64
65          template<typename Func>
66          constexpr void for_each(const Func& func);
67
68      private:
69          using container_type = std::vector<std::variant<T, std::int64_t>>;
70
71          container_type  myData;
```

```
72          std::int64_t    myFirstFree {-1};
73          size_type       myCount     {0};
74
75          NLOHMANN_DEFINE_TYPE_INTRUSIVE(FreeVector, myData, myFirstFree, myCount);
76      };
77
78      template<class T>
79      constexpr auto FreeVector<T>::operator[](size_type anIndex) -> reference
80      {
81          return std::get<T>(myData[anIndex]);
82      }
83
84      template<class T>
85      constexpr auto FreeVector<T>::operator[](size_type anIndex) const -> const_reference
86      {
87          return std::get<T>(myData[anIndex]);
88      }
89
90      template<class T>
91      constexpr auto FreeVector<T>::at(size_type anIndex) -> reference
92      {
93          return std::get<T>(myData.at(anIndex));
94      }
95
96      template<class T>
97      constexpr auto FreeVector<T>::at(size_type anIndex) const -> const_reference
98      {
99          return std::get<T>(myData.at(anIndex));
100     }
101
102     template<class T>
103     constexpr bool FreeVector<T>::valid(size_type anIndex) const
104     {
105         return myData.at(anIndex).index() == 0;
106     }
107
108     template<class T>
```

```
109     constexpr bool FreeVector<T>::empty() const noexcept
110     {
111         return myCount == 0;
112     }
113 
114     template<class T>
115     constexpr auto FreeVector<T>::size() const noexcept -> size_type
116     {
117         return static_cast<size_type>(myData.size());
118     }
119 
120     template<class T>
121     constexpr auto FreeVector<T>::count() const noexcept -> size_type
122     {
123         return myCount;
124     }
125 
126     template<class T>
127     constexpr auto FreeVector<T>::max_size() const noexcept -> size_type
128     {
129         return static_cast<size_type>(myData.max_size());
130     }
131 
132     template<class T>
133     template<typename... Args> requires std::constructible_from<T, Args...>
134     constexpr auto FreeVector<T>::emplace(Args&&... someArgs) -> size_type
135     {
136         size_type index = 0;
137 
138         if (myFirstFree != -1)
139         {
140             assert(!valid(myFirstFree));
141 
142             index = myFirstFree;
143 
144             myFirstFree     = std::get<std::int64_t>(myData[myFirstFree]);
145             myData[index]   = std::variant<T, std::int64_t>(std::in_place_index<0>, std::forward<Args>(someArgs)...);
```

```
        }
        else
        {
            assert(myData.size() == myCount); // should be the same if no more available space

            index = myData.size();
            myData.emplace_back(std::in_place_index<0>, std::forward<Args>(someArgs)...);
        }

        ++myCount;

        return index;
    }

    template<class T>
    constexpr auto FreeVector<T>::insert(const T& anElement) -> size_type
    {
        return emplace(anElement);
    }

    template<class T>
    constexpr auto FreeVector<T>::insert(T&& anElement) -> size_type
    {
        return emplace(std::move(anElement));
    }

    template<class T>
    constexpr void FreeVector<T>::erase(size_type anIndex)
    {
        assert(anIndex >= 0 && anIndex < myData.size() && valid(anIndex));

        myData[anIndex] = myFirstFree;
        myFirstFree = anIndex;

        --myCount;
    }

```

```
    template<class T>
    constexpr void FreeVector<T>::clear()
    {
        myData.clear();
        myFirstFree = -1;
        myCount = 0;
    }

    template<class T>
    constexpr void FreeVector<T>::reserve(size_type aCapacity)
    {
        myData.reserve(aCapacity);
    }

    template<class T>
    template<typename Func>
    constexpr void FreeVector<T>::for_each(const Func& func) const
    {
        for (const auto& elt : myData)
        {
            if (elt.index() == 0)
                func(std::get<T>(elt));
        }
    }

    template<class T>
    template<typename Func>
    constexpr void FreeVector<T>::for_each(const Func& func)
    {
        for (auto& elt : myData)
        {
            if (elt.index() == 0)
                func(std::get<T>(elt));
        }
    }
}
```