9

C++ Exercises
Set 2

Author(s): Pau Lopez, Peter Versluis

FB

13:24

February 20, 2025

9 8
10 9
11 8
13 8
14 0
16 0

9

A class template with a unique_ptr data type is implemented.

Listing 1: unique.hh

```
#ifndef UNIQUE_HH_
#define UNIQUE_HH_

#include <memory>

template <typename Type>
class Unique
{
    std::unique_ptr<Type> d_ptr;

    public:
        // default, copy and move constructor
        Unique() = default; // uses default constructor of unique_ptr
        Unique(Unique const &other);
        Unique(Unique &&tmp) = default; // uses move constructor of unique_ptr

        // constructors accepting Types
        Unique(Type const &tp);
        Unique(Type *ptr);

        // copy and move assignment
        Unique &operator=(Unique const &other);
        Unique &operator=(Unique &&other) = default; // uses unique_ptr default
                                                     // move assignment
        // get members
        Type &get();
        Type const &get() const;
};

#include "unique.ii"
#endif
```

the exercise states: 'a Type value': a Type && tmp prevents copying

These are not source files: No includes here and we're already in unique.hh's scope

Listing 2: unique.i

```
#include "unique.hh"
#include "assignment.ii"

template <typename Type>
inline Unique<Type>::Unique(Unique const &other)
:
    d_ptr(new Type{*(other.d_ptr)})
{}

template <typename Type>
inline Unique<Type>::Unique(Type const &tp)
:
    d_ptr(new Type{tp})
{}
```

what's this?

```
template <typename Type>
inline Unique<Type>::Unique(Type *ptr)
:
    d_ptr(ptr)
{}

template <typename Type>
inline Type &Unique<Type>::get()
{
    return *d_ptr;
}

template <typename Type>
inline Type const &Unique<Type>::get() const
{
    return *d_ptr;
}
```

see also #10.

Listing 3: assignment.i

```
#include "unique.hh"

template <typename Type>
Unique<Type> &Unique<Type>::operator=(Unique<Type> const &other)
{
    std::unique_ptr<Type> tmp{new Type{*(other.d_ptr)}};
    d_ptr.swap(tmp);
    return *this;
}
```

10

s_count and s_actual static data members are added to Unique class.

Listing 4: unique.hh

```
#ifndef UNIQUE_HH_
#define UNIQUE_HH_

#include <memory>

template <typename Type>
class Unique
{
    std::unique_ptr<Type> d_ptr;
    static size_t s_count;
    static size_t s_actual;

    public:
        // destructor
        ~Unique();

        // default, copy and move constructor
        Unique();
        Unique(Unique const &other);
        Unique(Unique &&tmp);

        // constructors accepting Types
        Unique(Type const &tp);
        Unique(Type *ptr);

        // copy and move assignment
        Unique &operator=(Unique const &other);
        Unique &operator=(Unique &&other) = default; // uses unique_ptr default
                                                     // move assignment
        // get members
        Type &get();
        Type const &get() const;
```

```
    size_t actual() const;
    size_t count() const;
};

#include "unique.ii"
#endif
```

not ordinary members: they retrieve the static data

Listing 5: unique.i

```
#include "unique.hh"
#include "assignment.ii"


template <typename Type>
inline size_t Unique<Type>::s_count = 0;

template <typename Type>
inline size_t Unique<Type>::s_actual = 0;


template <typename Type>
inline Unique<Type>::~Unique()
{
    --s_actual;
}

template <typename Type>
inline Unique<Type>::Unique()
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Unique const &other)
:
    d_ptr(new Type{*(other.d_ptr)})
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Unique &&tmp)
:
    d_ptr(std::move(tmp.d_ptr))
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Type const &tp)
:
    d_ptr(new Type{tp})
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Type *ptr)
:
    d_ptr(ptr)
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Type &Unique<Type>::get()
{
```

NSC

Especially with templates: ~~may~~ add semantic comment

4

```
    return *d_ptr;
}

template <typename Type>
inline Type const &Unique<Type>::get() const
{
    return *d_ptr;
}

template <typename Type>
inline size_t Unique<Type>::actual() const
{
    return s_actual;
}

template <typename Type>
inline size_t Unique<Type>::count() const
{
    return s_count;
}
```

Listing 6: assignment.i

```
template <typename Type>
Unique<Type> &Unique<Type>::operator=(Unique<Type> const &other)
{
    std::unique_ptr<Type> tmp{new Type{*(other.d_ptr)}};
    d_ptr.swap(tmp);
    return *this;
}
```

11

Modifying the previous class so count and actual are independent of the template type arguments.

Listing 7: globalcounter/counter.hh

```
#ifndef COUNTER_HH_
#define COUNTER_HH_
#include <iosfwd>

class GlobalCounter
{
    static size_t s_actual;
    static size_t s_count;

    public:
        ~GlobalCounter();
        GlobalCounter();

        size_t count() const;
        size_t actual() const;
};

inline GlobalCounter::~GlobalCounter()
{
    --s_actual;
}

inline size_t GlobalCounter::count() const
{
    return s_count;
}

inline size_t GlobalCounter::actual() const
{
    return s_actual;
}
```

OK, but not required by the exercise

```
#endif
```

Listing 8: globalcounter/constr.cc

```
#include "counter.hh"

size_t GlobalCounter::s_actual = 0;
size_t GlobalCounter::s_count = 0;

GlobalCounter::GlobalCounter()
{
    ++s_actual;
    ++s_count;
}
```

Listing 9: unique.hh

```
#ifndef UNIQUE_HH_
#define UNIQUE_HH_

#include "globalcounter/counter.hh"
#include <memory>

template <typename Type>
class Unique
{
    std::unique_ptr<Type> d_ptr;
    static size_t s_count;
    static size_t s_actual;
    GlobalCounter d_counter;     // independent of template arguments

    public:
        // destructor
        ~Unique();

        // default, copy and move constructor
        Unique();
        Unique(Unique const &other);
        Unique(Unique &&tmp);

        // constructors accepting Types
        Unique(Type const &tp);
        Unique(Type *ptr);

        // copy and move assignment
        Unique &operator=(Unique const &other);
        Unique &operator=(Unique &&other) = default; // uses unique_ptr default
                                                     // move assignment

        // get members
        Type &get();
        Type const &get() const;

        size_t actual() const;
        size_t count() const;
        size_t globalActual() const;
        size_t globalCount() const;
};

#include "unique.ii"
#endif
```

8

OK, but now you're modifying Unique. You should also be able to do the global count given Unique.

Hint: use inheritance and you don't need the extra data member (well, OK, that also modifies Unique...

Listing 10: unique.i

```
#include "unique.hh"
#include "assignment.ii"


template <typename Type>
inline size_t Unique<Type>::s_count = 0;
```

```
template <typename Type>
inline size_t Unique<Type>::s_actual = 0;


template <typename Type>
inline Unique<Type>::~Unique()
{
    --s_actual;
}

template <typename Type>
inline Unique<Type>::Unique()
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Unique const &other)
:
    d_ptr(new Type{*(other.d_ptr)})
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Unique &&tmp)
:
    d_ptr(std::move(tmp.d_ptr))
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Type const &tp)
:
    d_ptr(new Type{tp})
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Unique<Type>::Unique(Type *ptr)
:
    d_ptr(ptr)
{
    ++s_count;
    ++s_actual;
}

template <typename Type>
inline Type &Unique<Type>::get()
{
    return *d_ptr;
}

template <typename Type>
inline Type const &Unique<Type>::get() const
{
    return *d_ptr;
}

template <typename Type>
inline size_t Unique<Type>::actual() const
{
    return s_actual;
}

template <typename Type>
```

(NSC)

...

(

most of this is has already been shown before...

```
inline size_t Unique<Type>::count() const
{
    return s_count;
}

template <typename Type>
inline size_t Unique<Type>::globalActual() const
{
    return d_counter.actual();
}

template <typename Type>
inline size_t Unique<Type>::globalCount() const
{
    return d_counter.actual();
}
```

No need for these when using inheritance

Listing 11: assignment.i

```
template <typename Type>
Unique<Type> &Unique<Type>::operator=(Unique<Type> const &other)
{
    std::unique_ptr<Type> tmp{new Type{*(other.d_ptr)}};
    d_ptr.swap(tmp);
    return *this;
}
```

13

Developing a class that can be used with back_inserter.

8

Listing 12: data.hh

```
#ifndef DATA_HH
#define DATA_HH

#include <string>
#include <vector>
#include <memory>

class Data
{
    using DataVector = std::vector<std::shared_ptr<std::string>>;

    DataVector d_data;

    public:
        using value_type = std::string;     // for back_inserter
        void push_back(std::string const &str);
};

inline void Data::push_back(std::string const &str)
{
    d_data.push_back(std::make_shared<std::string>(str));
}

#endif
```

Listing 13: main.cc

```
#include "data.hh"
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
    // use copy generic algorithm to fill a Data object with lines from cin.
```

```
    Data obj;
    copy(istream_iterator<string>(cin), istream_iterator<string>(),
                                                      back_inserter(obj));
}
```

8

14

Providing begin, end, rbegin and rend public members to class Storage.

Listing 14: Storage.hh

```
#ifndef STORAGE_HH
#define STORAGE_HH

#include <vector>

template <typename Data>
class Storage
{
    std::vector<Data *> d_storage;

    public:
        using iterator = std::vector<Data *>::iterator;
        using reverse_iterator = std::vector<Data *>::reverse_iterator;

        iterator begin();
        iterator end();
        reverse_iterator rbegin();
        reverse_iterator rend();
};

template<typename Data>
inline typename Storage<Data>::iterator Storage<Data>::begin()
{
    return d_storage.begin();
}

template<typename Data>
inline typename Storage<Data>::iterator Storage<Data>::end()
{
    return d_storage.end();
}

template<typename Data>
inline typename Storage<Data>::reverse_iterator Storage<Data>::rbegin()
{
    return d_storage.rbegin();
}

template<typename Data>
inline typename Storage<Data>::reverse_iterator Storage<Data>::rend()
{
    return d_storage.rend();
}

#endif
```

these return pointers, sorting pointers is not the same as sorting the values they point to. Your iterators must return the objects, not their pointers.

16

Constructing a class to access an unordered_map in an ordered way.

Listing 15: unorderedsort.hh

```
#ifndef UNORDERED_SORT_HH
#define UNORDERED_SORT_HH


#include <unordered_map>
#include <algorithm>
```

```
#include <map>

// Class declaration
template <typename KeyT, typename ValT>
class UnorderedSort
{
    std::unordered_map<KeyT, ValT> d_map;

    public:
        UnorderedSort(std::unordered_map<KeyT, ValT> const &other);

        std::unordered_map<KeyT, ValT> const &map() const;
        std::map<KeyT, ValT> sort() const;

        template<class Compare>
        std::map<KeyT, ValT, Compare> sort(Compare comp) const;
};

// Copy constructor
template <typename KeyT, typename ValT>
inline UnorderedSort<KeyT, ValT>::UnorderedSort(
                                std::unordered_map<KeyT, ValT> const &other)
:
    d_map(other)
{}

// accessor
template <typename KeyT, typename ValT>
inline std::unordered_map<KeyT, ValT> const &
UnorderedSort<KeyT, ValT>::map() const
{
    return d_map;
}

template <typename KeyT, typename ValT>
std::map<KeyT, ValT> UnorderedSort<KeyT, ValT>::sort() const
{
    return std::map<KeyT, ValT> {d_map.begin(), d_map.end()};
}

template <typename KeyT, typename ValT>
template<class Compare>
std::map<KeyT, ValT, Compare> UnorderedSort<KeyT, ValT>::sort(Compare comp)
   const
{
    return std::map<KeyT, ValT, Compare> {d_map.begin(), d_map.end()};
}



#endif
```

This is a nice(?) example showing the no semantic comment results in hard to understand code. But apart from that: you don't want to copy unordered maps...

In the case that the unordered_map may be altered during the lifetime of our class's object, we can turn d_map into a reference to std::unordered_map<KeyT, ValT>, so whenever we modify the original unordered_map that was used to construct the class, d_map is also modified.

However, if we want to access the new data in an ordered way, we'll need to re-run the sort member functions (except for the constructor).