1. What is iterator invalidation? Why does it occur?

   Iterator invalidation occurs when a mutable operation changes the state of a container in such a way that any previous iterators for the container are no longer valid. This may occur simply because the object no longer exists (eg. std::set::erase), because the iterator was stored internally as a pointer and the object has been moved (eg. std::vector), or for some other reason.

2. Assignment 3 involves abstracting a container of containers as a single container. Consider a simple example of a vector<vector<int>>. Abstracting the container requires storing the outer iterator and the inner iterator. Discuss potential issues with this concept, and once you think you've discovered all of them, discuss how you might solve these issues.

   - Comparing end iterators (no inner iterator exists, and trying to access one is undefined behaviour)
   - ++ operator on an empty vector in the middle
   - begin() with an empty vector at the start

   Solution: Store a sentinel value (you need to know when it's the end, so you can know when not to check the inner iterator).

3. We have defined a rope class as a bunch of strings tied together, and have defined a basic class and some starter code using it. Write an iterator for this class so that main.cpp works. Look at **rope_user.cpp**

```
#include <iostream>

#include "tutorials/week8/rope.h"

int main() {
  std::vector<Rope> ropes{
      Rope{{"a"}},
      Rope{{"abc"}},
      Rope{{"abc"}},
      Rope{{"abc", "def"}},
      Rope{{"abc", "", "def"}},
      Rope{{""}},
      Rope{{"", "abc", "def", ""}}
  };

  // TODO(tutorial): make this support const auto& rope, if you have time.
  for (auto& rope : ropes) {
    std::cout << "Rope: \"";
    for (char letter : rope) {
      std::cout << letter;
    }
    std::cout << "\"\n";
  }
}
```

   And **rope.h>**

```
#ifndef TUTORIALS_WEEK8_ROPE_H_
#define TUTORIALS_WEEK8_ROPE_H_

#include <string>
#include <utility>
#include <vector>

class Rope {
 public:
  explicit Rope(std::vector<std::string> rope): rope_{std::move(rope)} {}

  class iterator {
   public:
    // TODO(tutorial): fill this in.
    using iterator_category = ?;
    using value_type = ?;
    using reference = ?;
    using pointer = ?;
    using difference_type = int;

    reference operator*() const;
    iterator& operator++();
    iterator operator++(int) {
      auto copy{*this};
      ++(*this);
      return copy;
    }
    // This one isn't strictly required, but it's nice to have.
    pointer operator->() const { return &(operator*()); }

    friend bool operator==(const iterator& lhs, const iterator& rhs);
    friend bool operator!=(const iterator& lhs, const iterator& rhs) { return !(lhs == rhs); }

   private:
    // TODO(tutorial): What data members should we put here?
  };

 private:
  std::vector<std::string> rope_;
};


#endif  // TUTORIALS_WEEK8_ROPE_H_
```

Header file:

```
#ifndef TUTORIALS_WEEK8_ROPE_H_
#define TUTORIALS_WEEK8_ROPE_H_

#include <optional>
#include <string>
#include <utility>
#include <vector>

class Rope {
 public:
  explicit Rope(std::vector<std::string> rope): rope_{std::move(rope)} {}

  class iterator {
   public:
    using iterator_category = std::forward_iterator_tag;
    using value_type = char;
    using reference = char&;
    using pointer = char*;
    using difference_type = int;

    reference operator*() const;
    iterator& operator++();
    iterator operator++(int) {
      auto copy{*this};
      ++(*this);
      return copy;
    }

    // This one isn't strictly required, but it's nice to have.
    pointer operator->() const { return &(operator*()); }

    friend bool operator==(const iterator& lhs, const iterator& rhs) {
      // We need to check the sentinel because comparison of default constructed iterators is unde
      return lhs.outer_ == rhs.outer_ && (lhs.outer_ == lhs.sentinel_ || lhs.inner_ == rhs.inner_)
    }

    friend bool operator!=(const iterator& lhs, const iterator& rhs) { return !(lhs == rhs); }


   private:
    std::vector<std::string>::iterator outer_;
    const std::vector<std::string>::iterator sentinel_;
    std::string::iterator inner_;

    friend class Rope;
    iterator(const decltype(outer_)& outer, const decltype(sentinel_)& sentinel, const decltype(in
  };

  iterator begin();
  iterator end();

 private:
  std::vector<std::string> rope_;
};


#endif  // TUTORIALS_WEEK8_ROPE_H_
```

Cpp file:

```
#include "tutorials/week8/rope.h"

#include <algorithm>

Rope::iterator::reference Rope::iterator::operator*() const {
  return *inner_;
}

Rope::iterator& Rope::iterator::operator++() {
  ++inner_;
  if (inner_ == outer_->end()) {
    do {
      ++outer_;
    } while (outer_ != sentinel_ && outer_->begin() == outer_->end());
    if (outer_ != sentinel_) {
      inner_ = outer_->begin();
    }
  }
  return *this;
}

Rope::iterator Rope::begin() {
  // What if the first element is empty?
  if (auto first = std::find_if(rope_.begin(), rope_.end(), [] (const std::string& s) { return !s.
    return {first, rope_.end(), first->begin()};
  }
  return end();
}

Rope::iterator Rope::end() {
  return {rope_.end(), rope_.end(), {}};
}
```

4. Discuss two ways we might modify our code to allow both a const and a non-const iterator. What are the advantages and disadvantages for each?

   The simple way is to copy-and paste the code, and modify as required to make both of the iterators. A cleaner way would be to use templates:

   ```
   class Rope {
     template <typename T>
     class BaseIterator {
       // When returning values, you're returning copies. The copy shouldn't be const.
       using value_type = std::remove_const<T>::type;
       using reference = T&;
       using pointer = T*;

       ...
     };

    public:
     using iterator = BaseIterator<char>;
     using const_iterator = BaseIterator<const char>;
   };
   ```

   Note that if we were to use standard library iterators, then we would either need to template over both the inner and outer iterator type, or use a type trait (eg. std::conditional) to change the type we use based on template parameters.

5. If you have time, try modifying rope_user.cpp so it iterates through the rope in reverse. What changes will you need to the class in order to ensure this works?

6. Complete this Set wrapper class (yes, it's a bit redundant).
   - Implement the class methods
   - Template the container type and set set to be the default container

```
#include <iostream>
#include <set>

template <typename T>
class SillySet {
 public:
  void insert(T t);
  bool erase(T t);
 private:
  std::set<T> set_;
};

int main() {

  SillySet<float> s;
  SillySet<int> us;

  s.insert(5.4);
  s.insert(6.2);
  s.erase(5.4);
  std::cout << s << "\n";

  us.insert(5);
  us.insert(6);
  us.erase(5);

  std::cout << us << "\n";

}
```

```
#include <iostream>
#include <set>
#include <unordered_set>

template <typename T, typename CONT = std::set<T>>
class SillySet {
 public:
  void insert(T t);
  bool erase(T t);
  friend std::ostream& operator<<(std::ostream& os, SillySet s) {
        for (const auto& i : s.set_) {
          os << i << " ";
        }
        return os;
  }
 private:
  CONT set_;
};

template <typename T, typename CONT>
void SillySet<T, CONT>::insert(T t) {
  set_.insert(t);
}

template <typename T, typename CONT>
bool SillySet<T, CONT>::erase(T t) {
  auto it = set_.find(t);
  if (it != set_.end()) {
        set_.erase(it);
        return true;
  }
  return false;
}

int main() {

  SillySet<float, std::set<float>> s;
  SillySet<int, std::unordered_set<int>> us;

  s.insert(5.4);
  s.insert(6.2);
  s.erase(5.4);
  std::cout << s << "\n";

  us.insert(5);
  us.insert(6);
  us.erase(5);

  std::cout << us << "\n";

}
```

- Use template template parameters to avoid the case where we can accidentally define

```
SillySet<int, std::vector<float>>
```

```
#include <iostream>
#include <set>
#include <unordered_set>
template <typename T, template<class, class...> class CONT>
class SillySet {
 public:
  void insert(T t);
  bool erase(T t);
  friend std::ostream& operator<<(std::ostream& os, SillySet s) {
    for (const auto& i : s.set_) {
      os << i << " ";
    }
    return os;
  }
 private:
  CONT<T> set_;
};
template <typename T, template<class, class...> class CONT>
void SillySet<T, CONT>::insert(T t) {
  set_.insert(t);
}
template <typename T, template<class, class...> class CONT>
bool SillySet<T, CONT>::erase(T t) {
  auto it = set_.find(t);
  if (it != set_.end()) {
    set_.erase(it);
    return true;
  }
  return false;
}
int main() {
  SillySet<float, std::set> s;
  SillySet<int, std::unordered_set> us;
  s.insert(5.4);
  s.insert(6.2);
  s.erase(5.4);
  std::cout << s << "\n";
  us.insert(5);
  us.insert(6);
  us.erase(5);
  std::cout << us << "\n";
}
```