Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем

Расчетно-графическая работа
по дисциплине «Современные технологии программирования»
на тему «Реализация шаблонного типа данных SmallVector»

Выполнил:
ст. гр. ИВ-121
Ермаков А. В.

Проверил:
доц. Пименов Е. С.

Новосибирск 2024

Содержание

1. Постановка задачи

Спроектировать шаблонный тип данных SmallVector. Использовать стандарт языка C++17. Реализовать итераторы, совместимые с алгоритмами стандартной библиотеки. Покрыть модульными тестами. При реализации не пользоваться контейнерами стандартной библиотеки, реализовать управление ресурсами в идиоме RAII.

В качестве системы сборки использовать CMake. Структурировать проект в соответствии с соглашениями Pitchfork Separate Headers + Separate Test [1]. Всю разработку вести в системе контроля версий git. Настроить автоматическое форматирование средствами clang-format.

Проверить код анализаторами Valgrind Memcheck, undefined sanitizer, address sanitizer, clang-tidy.

2. Мотивация

SmallVector – это структура данных, которая представляет собой гибридный контейнер, сочетающий в себе преимущества статического и динамического массивов, и подходит для тех случаев, когда размер контейнер заранее неизвестен, но ожидается, что он будет относительно небольшим. В таком случае SmallVector позволяет избежать лишних выделений памяти, конструирований, деконструирований и копирований элементов, что нередко происходит при использовании обычного вектора.

Ключевые свойства данного контейнера:
1) Эффективное использование памяти и высокая производительность за счет хранения элементов внутри самого объекта при небольшом размере SmallVector.
2) Быстрое добавление и удаление элементов в конце контейнера за счет использования динамического массива при большом размере контейнера.
3) Сохранение локальности данных при доступе к элементам контейнера за счет хранения их внутри объекта при небольшом размере контейнера.

4) При достижении предельного размера внутреннего статического массива происходит автоматический переход к использованию динамического массива на куче, что позволяет хранить любое количество элементов.

5) Быстрый доступ к элементам как по индексу, так использованием итераторов, что делает данный контейнер удобным для многих операций, таких как доступ, вставка и удаление элементов.
6) Реализация итераторов с произвольным доступом: iterator, const iterator, reverse iterator, const reverse iterator, которые соответствуют требованиям концепции Random Access Iterator, что позволяет эффективно перемещаться по элементам контейнера в любом направлении, выполнять разные операции (в том числе и арифметические) между итераторами, работать с различными стандартными алгоритмами из библиотеки STL.

7) Эффективное использование памяти осуществляется путем разделения выделения памяти при помощи operator new (при удалении оператор delete) и конструирование элементов (а также их деконструирование) с использованием std::move и без него, что позволяет не вызывать лишние дефолтные конструкторы, а также деструкторы при использовании std::move.

8) Предоставление интерфейса, аналогичного std::vector, что делает SmallVector понятным и удобным в использовании разработчиками, знакомыми со стандартной библиотекой языка C++.

3. Реализация

.3.1 Внутреннее устройство

SmallVector состоит из следующих полей:

- Size_ - размер вектор или количество элементов в нем.
- Capacity_ - емкость вектора.
- Buffer_ - статический массив, который используется через указатель data_ по мере его заполнения.
- Data_ - динамический массив, который используется, когда места в статическом массиве становится недостаточно, также через указатель data_. С этого момента статический буфер больше не будет использоваться.

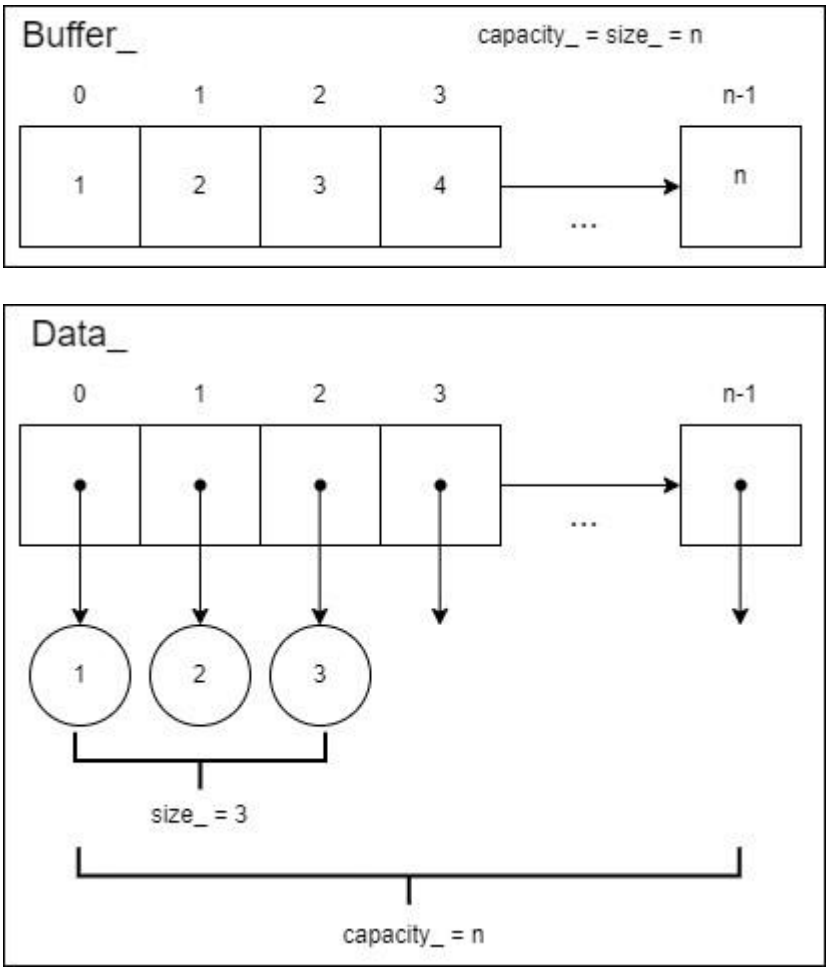

Рисунок 3.1: Структура SmallVector

.3.2 Интерфейс

SmallVector содержит следующие операции, доступные через публичный интерфейс:

1) Конструктор по умолчанию, который создает пустой вектор. Data_ указывает на статический массив buffer_. Сложность: O(1).

```
SmallVector()
    : size_(0),
      capacity_(N),
      buffer_(),
      data_(reinterpret_cast<T*>(buffer_.data())) {
}
```

Рисунок 3.2.1: Дефолтный конструктор

2) Конструктор копирования, который создает копию SmallVector. Сложность: O(n), где n – это размер копируемого вектора.

```
SmallVector(const SmallVector& other)
    : size_(other.size_),
      capacity_(other.capacity_),
      buffer_(),
      data_(nullptr) {
  if (other.data_ == reinterpret_cast<const T*>(other.buffer_.data())) {
    data_ = reinterpret_cast<T*>(buffer_.data());
  } else {
    data_ = static_cast<T*>(operator new((other.size_) * sizeof(T)));
  }

  for (size_t i = 0; i < size_; ++i) {
    new (data_ + i) T(other.data_[i]);
  }
}
```

Рисунок 3.2.2: Конструктор копирования

3) Конструктор перемещения, который создает новый вектор на основе перемещаемого вектора. Сложность: O(1).

```
SmallVector(SmallVector&& other) noexcept
    : size_(other.size_),
      capacity_(other.capacity_),
      buffer_(),
      data_(nullptr) {
  if (other.data_ == reinterpret_cast<const T*>(other.buffer_.data())) {
    buffer_ = std::move(other.buffer_);
    data_ = reinterpret_cast<T*>(buffer_.data());
  } else {
    data_ = other.data_;
  }
  other.data_ = nullptr;
  other.size_ = 0;
  other.capacity_ = 0;
}
```

Рисунок 3.2.3: Конструктор перемещения

4) Конструктор с инициализацией списком. Создает вектор и заполняет его при помощи метода **insert**. Сложность: O(k), где k – это количество элементов в списке инициализации.

```
1    SmallVector(std::initializer_list<T> initList)
2        : size_(0),
3          capacity_(N),
4          buffer_(),
5          data_(reinterpret_cast<T*>(buffer_.data())) {
6      for (const auto& elem : initList) {
7        insert(cend(), elem);
8      }
9    }
```

Рисунок 3.2.4: Конструктор с инициализацией списком

5) Конструктор из диапазона итераторов, который создает вектор и заполняет элементами из указанного диапазона при помощи метода **insert**. Сложность: O(n), где n – это количество элементов в диапазоне.

```
 1    template <typename Iterator>
 2    SmallVector(Iterator begin, Iterator end)
 3        : size_(0),
 4          capacity_(N),
 5          buffer_(),
 6          data_(reinterpret_cast<T*>(buffer_.data())) {
 7      for (auto it = begin; it != end; ++it) {
 8        insert(cend(), *it);
 9      }
10    }
```

Рисунок 3.2.5: Конструктор из диапазона итераторов

6) Деструктор, который освобождает память, занимаемую вектором, при помощи метода **clear**. Сложность: O(n), где n – это количество элементов в векторе.

```
1    ~SmallVector() {
2      clear();
3    }
```

Рисунок 3.2.6: Деструктор

7) Оператор копирования, который копирует элементы из другого вектора. Реализован на основе идиомы “Copy and Swap”. Сложность: O(n), где n – это количество элементов в копируемом векторе.

```
1    SmallVector& operator=(const SmallVector& other) {
2      if (this != &other) {
```

```
3         SmallVector tmp(other);
4         tmp.swap(*this);
5       }
6       return *this;
7     }
```

Рисунок 3.2.7: Оператор копирования

8) Оператор перемещения, перемещает элементы из другого вектора. Реализован аналогично оператору копирования. Сложность: O(1).

```
1     SmallVector& operator=(SmallVector&& other) noexcept {
2       if (this != &other) {
3         SmallVector tmp(std::move(other));
4         tmp.swap(*this);
5       }
6       return *this;
7     }
```

Рисунок 3.2.8: Оператор перемещения

9) Метод **data**, который имеет обычную ми константную версию, возвращает обычный или константный указатель соответственно на начало массива (статического или динамического в зависимости на что указывает data_). Сложность: O(1).

```
1     T* data() {
2       return data_;
3     }
```

Рисунок 3.2.9.1: Метод data

```
1     const T* data() const {
2       return data_;
3     }
```

Рисунок 3.2.9.2: Константный метод data

10) Метод buffer, который возвращает указатель на начало статического массива. Сложность O(1).

```
1     T* buffer() {
2       return reinterpret_cast<T*>(buffer_.data());
3     }
```

Рисунок 3.2.10: Метод buffer

11) Оператор [], который, в зависимости от реализации, возвращает обычную или константную ссылку на элемент по указанному индексу. Сложность O(1).

```
1     T& operator[](std::size_t index) {
```

```
    assert(index < size_ && "Index out of range");
    return data_[index];
  }
```

Рисунок 3.2.11.1: Оператор []

```
  const T& operator[](std::size_t index) const {
    assert(index < size_ && "Index out of range");
    return data_[index];
  }
```

Рисунок 3.2.11.2: Константный оператор []

12) Метод at, который, в зависимости от реализации, возвращает обычную или константную ссылку на элемент по указанному индексу с проверкой границ. Сложность O(1).

```
  T& at(std::size_t index) {
    if (index >= size_) {
      throw std::out_of_range("Index out of range");
    }
    return data_[index];
  }
```

Рисунок 3.2.12.1: Метод at

```
  const T& at(std::size_t index) const {
    if (index >= size_) {
      throw std::out_of_range("Index out of range");
    }
    return data_[index];
  }
```

Рисунок 3.2.12.2: Константный метод at

13) Метод clear, который освобождает память, занятую вектором и устанавливающие size_ и capacity_ в нулевые значения. Сложность O(n), где n - это количество элементов в векторе.

```
  void clear() {
    for (std::size_t i = 0; i < size_; ++i) {
      data_[i].~T();
    }
    if ((data_ != reinterpret_cast<T*>(buffer_.data()))) {
      operator delete(data_);
    }

    size_ = 0;
    capacity_ = 0;
    data_ = nullptr;
  }
```

Рисунок 3.2.13: Метод clear

14) Метод empty, который проверят размер вектора на нулевое значение. Сложность: O(1).

```
1    bool empty() const {
2      return size_ == 0;
3    }
```

Рисунок 3.2.14: Метод empty

15) Метод size, который возвращает размер вектора. Сложность: O(1).

```
1    std::size_t size() const {
2      return size_;
3    }
```

Рисунок 3.2.15: Метод size

16) Метод capacity, который возвращает емкость вектора. Сложность: O(1).

```
1    std::size_t capacity() const {
2      return capacity_;
3    }
```

Рисунок 3.2.16: Метод capacity

17) Метод push_back, который записывает в конец вектора новый элемент со значением value. Если емкости вектора хватает, то конструируется новый элемент и инкрементируется size_, в противном случае выделяется новая память размером с две емкости, в ней конструируются элементы из старого массива, в конец записывается новый элемент, после чего старая память очищается, обновляются size_, capacity и data_. Сложность: O(1), если не надо перевыделять память, в противном случае O(n).

```
 1    void push_back(const T& value) {
 2      if (size_ < capacity_) {
 3        new (data_ + size_) T(value);
 4        ++size_;
 5        return;
 6      }
 7
 8      T* new_data = static_cast<T*>(operator new((capacity_ * 2) *
 9  sizeof(T)));
10
11      for (std::size_t i = 0; i < size_; ++i) {
12        new (new_data + i) T(std::move(data_[i]));
13      }
14
15      new (new_data + size_) T(value);
16
17      if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr) {
18        operator delete(data_);
19      }
20      data_ = new_data;
```

```
21      capacity_ *= 2;
22      ++size_;
23    }
```

Рисунок 3.2.17: Метод push_back

18) Метод resize, который изменяет размер вектора. Если новый размер равен старому, что ничего не изменяется; если новый размер меньше чем старый, то вызываются деструкторы для элементов из диапазона [new_size; size_); если же новый размер больше старого, но меньше или равен емкости, то вызывается дефолтный конструктор для элементов из диапазона [size_, new_size); в противном случае выделяется новый участок памяти для new_size элементов, в котором они конструируются из старого массива, для всех элементов из диапазона [size, new_size) вызываются дефолтные конструкторы, память, выделенная под старый массив, очищается, обновляются size_, capacity и data_. Сложность: в лучшем случае O(1), когда нужно сконструировать/деконструировать небольшое число элементов, в худшем случае – O(n), если приходится выделять память.

```
 1    void resize(size_t n) {
 2      if (size_ == n) {
 3        return;
 4      }
 5
 6      if (n < size_) {
 7        for (size_t i = n; i < size_; ++i) {
 8          data_[i].~T();
 9        }
10        size_ = n;
11        return;
12      }
13
14      if (n <= capacity_) {
15        for (std::size_t i = size_; i < n; ++i) {
16          new (data_ + i) T();
17        }
18        size_ = n;
19        return;
20      }
21
22      T* new_data = static_cast<T*>(operator new(n * sizeof(T)));
23      for (size_t i = 0; i < size_; ++i) {
24        new (new_data + i) T(std::move(data_[i]));
25      }
26
27      for (size_t i = size_; i < n; ++i) {
28        new (new_data + i) T();
29      }
30      capacity_ = n;
31      size_ = n;
32
33      if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr) {
34        operator delete(data_);
35      }
```

```
36      data_ = new_data;
37    }
```

Рисунок 3.2.18: Метод resize

19) Метод reserve, который увеличивает емкость вектора. Если новая емкость меньше или равна старой, то ничего не происходит. В противном случае выделяется память размером с новый capacity_, в котором первые size_ элементов конструируются из старого массива. Память, выделенная под старый массив, освобождается, обновляется data_ и capacity_. Сложность: O(n), если требуется выделение новой памяти, в противном случае – O(1).

```
 1    void reserve(size_t n) {
 2      if (n <= capacity_) {
 3        return;
 4      }
 5
 6      T* new_data = static_cast<T*>(operator new(n * sizeof(T)));
 7
 8      for (size_t i = 0; i < size_; ++i) {
 9        new (new_data + i) T(std::move(data_[i]));
10      }
11
12      if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr) {
13        operator delete(data_);
14      }
15
16      data_ = new_data;
17      capacity_ = n;
18    }
```

Рисунок 3.2.19: Метод reserve

20) Метод shrink_to_fit, который уменьшает емкость вектора до его размера. Если размер равен емкости или массив статический, то ничего не происходит. В противном случае выделяется память для size_ элементов, в которой конструируются элементы из старого массива, обновляются data_ и capacity_, старая память освобождается. Сложность: O(n), если требуется выделение памяти, в противном случае – O(1).

```
 1    void shrink_to_fit() {
 2      if (size_ == capacity_ || data_ == reinterpret_cast<T*>(buffer_.data()))
 3  {
 4        return;
 5      }
 6
 7      T* new_data = static_cast<T*>(operator new(size_ * sizeof(T)));
 8
 9      for (size_t i = 0; i < size_; ++i) {
10        new (new_data + i) T(std::move(data_[i]));
11      }
12
13      operator delete(data_);
```

```
14
15      data_ = new_data;
16      capacity_ = size_;
17    }
```

Рисунок 3.2.20: Метод shrink_to_fit

21) Метод erase (для удаления одного элемента), который удаляет один элемент из вектора. Для начала вычисляется индекс удаляемого элемента, проверяется существование элемента по индексу. Если элемент существует, то он перезаписывается следующим справа значением элемента, следующий справа – значением следующего от него справа элемента и т.д., размера вектора уменьшается на единицу, вызывается деструктор для элемента с индексом size_ - 1. Сложность: в лучшем случае O(1), если элемент находится в самом конце вектора, в худшем случае – O(k), где k – это число перезаписываемых элементов.

```
 1    iterator erase(const_iterator pos) {
 2      const std::size_t index = pos - cbegin();
 3      if (index >= size_ || index < 0) {
 4        throw std::out_of_range("Iterator pos is not valid for this contain-
 5  er");
 6      }
 7
 8      for (std::size_t i = index + 1; i < size_; ++i) {
 9        data_[i - 1] = std::move(data_[i]);
10      }
11
12      data_[size_ - 1].~T();
13
14      --size_;
15      return begin() + index;
16    }
```

Рисунок 3.2.21: Метод erase (для удаления одного элемента)

22) Метод erase (для удаления диапазона элементов), который удаляет элементы из соответствующего диапазона. Работает аналогично предыдущему, с учетом того, что размер массива уменьшится на размер диапазона элементов размером remove_count, а самый левый удаляемый элемент будет перезаписан следующим справа от самого правого удаляемого элемента элементом и т.д., деструктор вызовется для последних remove_count элементов. Сложность: O(n), где n – максимальное из количеств удаляемых и перезаписываемых элементов.

```
 1   iterator erase(const_iterator first, const_iterator last) {
 2     const std::size_t index_first = first - cbegin();
 3     if (index_first >= size_ || index_first < 0) {
 4       throw std::out_of_range("Iterator pos is not valid for this contain-
 5  er");
 6     }
 7
 8     const std::size_t index_last = last - cbegin();
```

```
 9     if (index_last >= size_ || index_last < 0) {
10       throw std::out_of_range("Iterator pos is not valid for this contain-
11 er");
12     }
13
14     const std::size_t remove_count = last - first;
15
16     for (std::size_t i = index_last; i < size_; ++i) {
17       data_[i - remove_count] = std::move(data_[i]);
18     }
19
20     for (std::size_t i = size_ - remove_count; i < size_; ++i) {
21       data_[i].~T();
22     }
23
24     size_ -= remove_count;
25
26     return begin() + index_first + remove_count;
27   }
```

Рисунок 3.2.22: Метод erase (для удаления диапазона элементов),

23) Метод insert (для вставки одного элемента), который вставляет новый элемент со значением value в вектор. Вычисляется индекс вставляемого элемента. В случае предельного размера вектора, выделяется новый участок памяти длиной в две емкости. Если индекс равен size_ (справа от самого последнего элемента), то он просто записывается, в противном случае все элементы, имеющие индексы больше или равными индексу вставляемого элемента смещаются вправо, начиная с крайнего правого элемента, после чего записывается новый элемент, размер вектора увеличивается на единицу. В случае выделения нового участка памяти указатель data_ обновляется, старая память освобождается. Сложность: O(n), если индекс вставляемого элемента приближен к нулю, что требует сдвигать вправо практически все элементы вектора, а также если требуется выделять новую память, и O(1) в случае, когда новый элемент необходимо вставить в самый конец вектора без выделения новой памяти.

```
 1   iterator insert(const_iterator pos, const T& value) {
 2     const std::size_t index = pos - cbegin();
 3     if (index > size_ || index < 0) {
 4       throw std::out_of_range("Iterator pos is not valid for this contain-
 5 er");
 6     }
 7
 8     if (size_ == capacity_) {
 9       reserve((capacity_ + 1) * 2);
10     }
11
12     if (pos == cbegin() + size_) {
13       new (data_ + size_) T(value);
14       ++size_;
15       return begin() + index;
16     }
```

```
17
18      new (data_ + size_) T(std::move(data_[size_ - 1]));
19
20      for (std::size_t i = size_ - 1; i > index; --i) {
21        data_[i] = std::move(data_[i - 1]);
22      }
23
24      data_[index] = value;
25      ++size_;
26
27      return begin() + index;
28    }
```

Рисунок 3.2.23: Метод insert (для вставки одного элемента),

24) Метод insert (для диапазона элементов), который вставляет k элементов из диапазона. Аналогичен предыдущему за исключением того, что размер увеличивается на k, память выделяется под size + k элементов, элементы сдвигаются на k ячеек вправо. Сложность: O(n), где n – это максимальное из количеств вставляемых элементов и перезаписываемых.

```
 1    template <class InputIt>
 2    iterator insert(const_iterator pos, InputIt first, InputIt last) {
 3      const std::size_t index = pos - cbegin();
 4
 5      if (index > size_ || index < 0) {
 6        throw std::out_of_range("Iterator pos is not valid for this contain-
 7  er");
 8      }
 9      const std::size_t insert_count = last - first;
10
11      if (size_ + insert_count > capacity_) {
12        reserve(size_ + insert_count);
13      }
14
15      for (std::size_t i = size_ + insert_count - 1; i > index + insert_count
16  - 1;
17           --i) {
18        new (data_ + i) T(std::move(data_[i - insert_count]));
19        data_[i - insert_count].~T();
20      }
21
22      std::size_t i = index;
23
24      for (auto it = first; it != last; ++it) {
25        new (data_ + i) T(*it);
26        ++i;
27      }
28
29      size_ += insert_count;
30
31      return begin() + index;
32    }
```

Рисунок 3.2.24: Метод insert (для вставки диапазона элементов),

25) Метод begin, который возвращает итератор, указывающий на начало вектора (первый элемент). Сложность: O(n).

```
1    iterator begin() {
2      return iterator(data_);
3    }
```

Рисунок 3.2.25: Метод begin

26) Метод end, который возвращает итератор, указывающий на элемент, следующий за последний элементом вектора. Сложность: O(n).

```
1    iterator end() {
2      return iterator(data_ + size_);
3    }
```

Рисунок 3.2.26: Метод end

27) Метод cbegin, который аналогичен методу begin, но возвращает константный итератор, не позволяющий изменять значения элементов вектора.

```
1    const_iterator cbegin() const {
2      return const_iterator(data_);
3    }
```

Рисунок 3.2.27: Метод cbegin

28) Метод cend, который аналогичен методу end, но возвращает константный итератор.

```
1    const_iterator cend() const {
2      return const_iterator(data_ + size_);
3    }
```

Рисунок 3.2.28: Метод cend

29) Метод rbegin, который возвращает обратный итератор, указывающий на последний элемент в векторе.

```
1    reverse_iterator rbegin() {
2      return reverse_iterator(data_ + size_ - 1);
3    }
```

Рисунок 3.2.29: Метод rbegin

30) Метод rend, который возвращает обратный итератор, указывающий на элемент, после которого следует первый элемент вектора.

```
1    reverse_iterator rend() {
2      return reverse_iterator(data_ - 1);
3    }
```

Рисунок 3.2.30: Метод rend

31) Метод crbegin, который аналогичен методу rbegin, но возвращает константный обратный итератор.

```
1    const_reverse_iterator crbegin() const {
2      return const_reverse_iterator(data_ + size_ - 1);
3    }
```

Рисунок 3.2.31: Метод crbegin

32) Метод crend, который аналогичен методу rend, за исключением того, что он возвращает константный обратный итератор.

```
1   const_reverse_iterator crend() const {
2     return const_reverse_iterator(data_ - 1);
3   }
```

Рисунок 3.2.32: Метод crend

.3.3 Итераторы

SmallVector предоставляет следующие типы итераторов:

1. Iterator
2. Const_iterator
3. Reverse_iterator
4. Const_reverse_iterator

Каждый из которых имеет следующие свойства:

```
1       using iterator_category = std::random_access_iterator_tag;
2       using value_type = T;
3       using difference_type = std::ptrdiff_t;
4       using pointer = T*;
5       using reference = T&;
```

Рисунок 3.3: Свойства итераторов

Все реализованные итераторы являются Random Access Iterator, поддерживающими следующие операции:

Операция	Действие	Описание
iterator(T ptr)	постфиксный инкремент	Этот конструктор принимает указатель на элемент вектора и инициализирует итератор этим указателем.

operator++()	префиксный инкремент	Увеличивает указатель на следующий элемент вектора и возвращает ссылку на сам итератор.
operator++(int)	постфиксный инкремент	Создает копию текущего итератора, затем инкрементирует текущий итератор, и возвращает созданную копию.
operator-- ()	префиксный декремент	Уменьшает указатель на предыдущий элемент вектора и возвращает ссылку на сам итератор.
operator--(int)	постфиксный декремент	Создает копию текущего итератора, затем декрементирует текущий итератор, и возвращает созданную копию.
operator+(difference_type n)	сложение с числом	Возвращает итератор, указывающий на элемент вектора, который находится на n позиций впереди относительно текущего итератора.
operator-(difference_type n)	вычитание числа	Возвращает итератор, указывающий на элемент вектора, который находится на n позиций назад относительно текущего итератора.
operator-(const iterator& other)	вычитание итераторов	Возвращает разницу в позициях между текущим итератором и другим итератором.
operator+=(difference_type n)	сложение с присваиванием	Перемещает итератор на n позиций вперед.
operator-=(difference_type n)	вычитание с присваиванием	Перемещает итератор на n позиций назад.
operator()	разыменование	Возвращает ссылку на элемент, на который указывает текущий итератор.
operator->()	доступ к члену по указателю	Возвращает указатель на элемент, на который указывает текущий итератор.
operator[](difference_type n)	доступ по индексу	Возвращает ссылку на элемент в позиции n от текущего элемента.
swap(iterator& other)	обмен значениями	Меняет местами текущий итератор с другим итератором.
operator==(const iterator&	оператор сравнения на	Сравнивает текущий итератор с

other)	равенство	другим итератором и возвращает true, если они указывают на один и тот же элемент.
operator!=(const iterator& other)	оператор сравнения на неравенство	Сравнивает текущий итератор с другим итератором и возвращает true, если они указывают на разные элементы.
operator<(const iterator& other)	оператор "меньше"	Сравнивает текущий итератор с другим итератором и возвращает true, если текущий указывает на элемент, предшествующий элементу, на который указывает другой итератор.
operator<=(const iterator& other)	оператор "меньше или равно"	Сравнивает текущий итератор с другим итератором и возвращает true, если текущий указывает на элемент, предшествующий или равный элементу, на который указывает другой итератор.
operator>(const iterator& other)	оператор "больше"	Сравнивает текущий итератор с другим итератором и возвращает true, если текущий указывает на элемент, следующий за элементом, на который указывает другой итератор.
operator>=(const iterator& other)	оператор "больше или равно"	Сравнивает текущий итератор с другим итератором и возвращает true, если текущий указывает на элемент, следующий или равный элементу, на который указывает другой итератор.

.3.4 Тестирование

SmallVector был полностью протестирован: всего было написано более ста тестов которые по папкам были распределены на следующие категории:

1) Тесты для конструкторов, операторов и деструктора.
2) Тесты для небольших методов, которые что-либо возвращают и не изменяют вектор (at(), data(), buffer(), size(), empty() и т.д.).
3) Тесты для больших методов, которые работают с памятью и могут изменять вектор (push_back(), insert(), erase(), resize() и т.д.)

4) Тестирование итераторов было разбито на 4 категории по их типам (iterator, const_iterator, reverse_iterator, const_reverse_iterator). Каждый файл с тестами полностью тестировал весь итератор.
5) Тестирование SmallVector и его итераторов на стандартных методах (в том числе требующих Random Access Iterator), таких как std::binary_search, std::accumulate, std::reverse, std::sort и т.д., также было разбито на 4 категории по типам итераторов.

4. Список источников

1. Mizux. Pitchfork [Электронный ресурс]. URL: https://github.com/Mizux/pitchfork (дата обращения: 01.04.2024).
2. Neacsu, C. Pitchfork [Электронный ресурс]. URL: https://neacsu.net/docs/programming/code_layout/pitchfork/ (дата обращения: 01.04.2024).
3. OpenAI Chat [Электронный ресурс]. URL: https://chat.openai.com/ (дата обращения: 01.04.2024).
4. Mistral Chat [Электронный ресурс]. URL: https://chat.mistral.ai/ (дата обращения: 01.04.2024).
5. Internal Pointers. Writing Custom Iterators in Modern C++ [Электронный ресурс]. URL: https://www.internalpointers.com/post/writing-custom-iterators-modern-cpp (дата обращения: 01.04.2024).
6. Metanit. Итераторы в C++ [Электронный ресурс]. URL: https://metanit.com/cpp/tutorial/7.3.php (дата обращения: 01.04.2024).
7. Metanit. Класс std::vector в C++ [Электронный ресурс]. URL: https://metanit.com/cpp/tutorial/13.1.php (дата обращения: 01.04.2024).
8. Study Plan. Understanding C++ Iterators and Iterator Concepts [Электронный ресурс]. URL: https://www.studyplan.dev/pro-cpp/iterator-concepts (дата обращения: 01.04.2024).
9. Computing on Plains. Custom C++ Container Classes with Iterators [Электронный ресурс]. URL: https://computingonplains.wordpress.com/custom-c-container-classes-with-iterators/ (дата обращения: 01.04.2024).
10. cppreference.com. C++ Iterator Library [Электронный ресурс]. URL: https://en.cppreference.com/w/cpp/iterator (дата обращения: 01.04.2024).
11. cppreference.com. C++ Vector Container [Электронный ресурс]. URL: https://en.cppreference.com/w/cpp/container/vector (дата обращения: 01.04.2024).

5. Приложение

// file libsvector/libsvector/svector/svector.hpp

```
#pragma once
#include <array>

#include <cstddef>

#include <initializer_list>

#include <iostream>

#include <cassert>

namespace svector {
template <typename T, std::size_t N>
class SmallVector {
 private:
  std::size_t size_;
  std::size_t capacity_;
  std::array<char, N * sizeof(T)> buffer_;
  T* data_;

 public:
  SmallVector()
      : size_(0),
        capacity_(N),
        buffer_(),
        data_(reinterpret_cast<T*>(buffer_.data())) {
  }

  class iterator {
   private:
    T* ptr_;

   public:
    using iterator_category = std::random_access_iterator_tag;
    using value_type = T;
    using difference_type = std::ptrdiff_t;
    using pointer = T*;
    using reference = T&;

    explicit iterator(T* ptr) : ptr_(ptr) {
    }

    iterator& operator++() {
      ++ptr_;
      return *this;
    }

    iterator operator++(int) {
      iterator temp = *this;
      ++(*this);
      return temp;
```

```
52      }
53
54      iterator& operator--() {
55        --ptr_;
56        return *this;
57      }
58
59      iterator operator--(int) {
60        iterator temp = *this;
61        --(*this);
62        return temp;
63      }
64
65      iterator operator+(difference_type n) const {
66        return iterator(ptr_ + n);
67      }
68
69      iterator operator-(difference_type n) const {
70        return iterator(ptr_ - n);
71      }
72
73      difference_type operator-(const iterator& other) const {
74        return ptr_ - other.ptr_;
75      }
76
77      iterator& operator+=(difference_type n) {
78        ptr_ += n;
79        return *this;
80      }
81
82      iterator& operator-=(difference_type n) {
83        ptr_ -= n;
84        return *this;
85      }
86
87      reference operator*() const {
88        return *ptr_;
89      }
90
91      pointer operator->() const {
92        return ptr_;
93      }
94
95      reference operator[](difference_type n) const {
96        return *(ptr_ + n);
97      }
98
99      void swap(iterator& other) noexcept {
100       T* temp = ptr_;
101       ptr_ = other.ptr_;
102       other.ptr_ = temp;
103     }
104
105     bool operator==(const iterator& other) const {
106       return ptr_ == other.ptr_;
107     }
108
```

```
109     bool operator!=(const iterator& other) const {
110       return ptr_ != other.ptr_;
111     }
112
113     bool operator<(const iterator& other) const {
114       return ptr_ < other.ptr_;
115     }
116
117     bool operator<=(const iterator& other) const {
118       return ptr_ <= other.ptr_;
119     }
120
121     bool operator>(const iterator& other) const {
122       return ptr_ > other.ptr_;
123     }
124
125     bool operator>=(const iterator& other) const {
126       return ptr_ >= other.ptr_;
127     }
128   };
129
130   class const_iterator {
131    private:
132     const T* ptr_;
133
134    public:
135     using iterator_category = std::random_access_iterator_tag;
136     using value_type = T;
137     using difference_type = std::ptrdiff_t;
138     using pointer = const T*;
139     using reference = const T&;
140
141     explicit const_iterator(const T* ptr) : ptr_(ptr) {
142     }
143
144     const_iterator& operator++() {
145       ++ptr_;
146       return *this;
147     }
148
149     const_iterator operator++(int) {
150       const_iterator temp = *this;
151       ++(*this);
152       return temp;
153     }
154
155     const_iterator& operator--() {
156       --ptr_;
157       return *this;
158     }
159
160     const_iterator operator--(int) {
161       const_iterator temp = *this;
162       --(*this);
163       return temp;
164     }
165
```

```
166     const_iterator operator+(difference_type n) const {
167       return const_iterator(ptr_ + n);
168     }
169
170     const_iterator operator-(difference_type n) const {
171       return const_iterator(ptr_ - n);
172     }
173
174     difference_type operator-(const const_iterator& other) const {
175       return ptr_ - other.ptr_;
176     }
177
178     const_iterator& operator+=(difference_type n) {
179       ptr_ += n;
180       return *this;
181     }
182
183     const_iterator& operator-=(difference_type n) {
184       ptr_ -= n;
185       return *this;
186     }
187
188     reference operator*() const {
189       return *ptr_;
190     }
191
192     pointer operator->() const {
193       return ptr_;
194     }
195
196     reference operator[](difference_type n) const {
197       return *(ptr_ + n);
198     }
199
200     void swap(const_iterator& other) noexcept {
201       const T* temp = ptr_;
202       ptr_ = other.ptr_;
203       other.ptr_ = temp;
204     }
205
206     bool operator==(const const_iterator& other) const {
207       return ptr_ == other.ptr_;
208     }
209
210     bool operator!=(const const_iterator& other) const {
211       return ptr_ != other.ptr_;
212     }
213
214     bool operator<(const const_iterator& other) const {
215       return ptr_ < other.ptr_;
216     }
217
218     bool operator<=(const const_iterator& other) const {
219       return ptr_ <= other.ptr_;
220     }
221
222     bool operator>(const const_iterator& other) const {
```

```
223       return ptr_ > other.ptr_;
224     }
225
226     bool operator>=(const const_iterator& other) const {
227       return ptr_ >= other.ptr_;
228     }
229   };
230
231   class reverse_iterator {
232    private:
233     T* ptr_;
234
235    public:
236     using iterator_category = std::random_access_iterator_tag;
237     using value_type = T;
238     using difference_type = std::ptrdiff_t;
239     using pointer = T*;
240     using reference = T&;
241
242     explicit reverse_iterator(T* ptr) : ptr_(ptr) {
243     }
244
245     reverse_iterator& operator++() {
246       --ptr_;
247       return *this;
248     }
249
250     reverse_iterator operator++(int) {
251       reverse_iterator temp = *this;
252       --ptr_;
253       return temp;
254     }
255
256     reverse_iterator& operator--() {
257       ++ptr_;
258       return *this;
259     }
260
261     reverse_iterator operator--(int) {
262       reverse_iterator temp = *this;
263       ++ptr_;
264       return temp;
265     }
266
267     reverse_iterator operator+(difference_type n) const {
268       return reverse_iterator(ptr_ - n);
269     }
270
271     reverse_iterator operator-(difference_type n) const {
272       return reverse_iterator(ptr_ + n);
273     }
274
275     difference_type operator-(const reverse_iterator& other) const {
276       return other.ptr_ - ptr_;
277     }
278
279     reverse_iterator& operator+=(difference_type n) {
```

```
280       ptr_ -= n;
281       return *this;
282     }
283
284     reverse_iterator& operator-=(difference_type n) {
285       ptr_ += n;
286       return *this;
287     }
288
289     reference operator*() const {
290       return *ptr_;
291     }
292
293     pointer operator->() const {
294       return ptr_;
295     }
296
297     reference operator[](difference_type n) const {
298       return *(ptr_ - n);
299     }
300
301     void swap(reverse_iterator& other) noexcept {
302       T* temp = ptr_;
303       ptr_ = other.ptr_;
304       other.ptr_ = temp;
305     }
306
307     bool operator==(const reverse_iterator& other) const {
308       return ptr_ == other.ptr_;
309     }
310
311     bool operator!=(const reverse_iterator& other) const {
312       return ptr_ != other.ptr_;
313     }
314
315     bool operator<(const reverse_iterator& other) const {
316       return ptr_ > other.ptr_;
317     }
318
319     bool operator<=(const reverse_iterator& other) const {
320       return ptr_ >= other.ptr_;
321     }
322
323     bool operator>(const reverse_iterator& other) const {
324       return ptr_ < other.ptr_;
325     }
326
327     bool operator>=(const reverse_iterator& other) const {
328       return ptr_ <= other.ptr_;
329     }
330   };
331
332   class const_reverse_iterator {
333    private:
334     const T* ptr_;
335
336    public:
```

```
337     using iterator_category = std::random_access_iterator_tag;
338     using value_type = const T;
339     using difference_type = std::ptrdiff_t;
340     using pointer = const T*;
341     using reference = const T&;
342 
343     explicit const_reverse_iterator(const T* ptr) : ptr_(ptr) {
344     }
345 
346     const_reverse_iterator& operator++() {
347       --ptr_;
348       return *this;
349     }
350 
351     const_reverse_iterator operator++(int) {
352       const_reverse_iterator temp = *this;
353       --ptr_;
354       return temp;
355     }
356 
357     const_reverse_iterator& operator--() {
358       ++ptr_;
359       return *this;
360     }
361 
362     const_reverse_iterator operator--(int) {
363       const_reverse_iterator temp = *this;
364       ++ptr_;
365       return temp;
366     }
367 
368     const_reverse_iterator operator+(difference_type n) const {
369       return const_reverse_iterator(ptr_ - n);
370     }
371 
372     const_reverse_iterator operator-(difference_type n) const {
373       return const_reverse_iterator(ptr_ + n);
374     }
375 
376     difference_type operator-(const const_reverse_iterator& other) const {
377       return other.ptr_ - ptr_;
378     }
379 
380     const_reverse_iterator& operator+=(difference_type n) {
381       ptr_ -= n;
382       return *this;
383     }
384 
385     const_reverse_iterator& operator-=(difference_type n) {
386       ptr_ += n;
387       return *this;
388     }
389 
390     reference operator*() const {
391       return *ptr_;
392     }
393 
```

```
394       pointer operator->() const {
395         return ptr_;
396       }
397
398       reference operator[](difference_type n) const {
399         return *(ptr_ - n);
400       }
401
402       void swap(const_reverse_iterator& other) noexcept {
403         const T* temp = ptr_;
404         ptr_ = other.ptr_;
405         other.ptr_ = temp;
406       }
407
408       bool operator==(const const_reverse_iterator& other) const {
409         return ptr_ == other.ptr_;
410       }
411
412       bool operator!=(const const_reverse_iterator& other) const {
413         return ptr_ != other.ptr_;
414       }
415
416       bool operator<(const const_reverse_iterator& other) const {
417         return ptr_ > other.ptr_;
418       }
419
420       bool operator<=(const const_reverse_iterator& other) const {
421         return ptr_ >= other.ptr_;
422       }
423
424       bool operator>(const const_reverse_iterator& other) const {
425         return ptr_ < other.ptr_;
426       }
427
428       bool operator>=(const const_reverse_iterator& other) const {
429         return ptr_ <= other.ptr_;
430       }
431     };
432
433     SmallVector(const SmallVector& other)
434         : size_(other.size_),
435           capacity_(other.capacity_),
436           buffer_(),
437           data_(nullptr) {
438       if (other.data_ == reinterpret_cast<const T*>(other.buffer_.data())) {
439         data_ = reinterpret_cast<T*>(buffer_.data());
440       } else {
441         data_ = static_cast<T*>(operator new((other.size_) * sizeof(T)));
442       }
443       for (size_t i = 0; i < size_; ++i) {
444         new (data_ + i) T(other.data_[i]);
445       }
446     }
447
448     SmallVector(SmallVector&& other) noexcept
449         : size_(other.size_),
450           capacity_(other.capacity_),
```

```
451           buffer_(),
452           data_(nullptr) {
453       if (other.data_ == reinterpret_cast<const T*>(other.buffer_.data())) {
454         buffer_ = std::move(other.buffer_);
455         data_ = reinterpret_cast<T*>(buffer_.data());
456       } else {
457         data_ = other.data_;
458       }
459       other.data_ = nullptr;
460       other.size_ = 0;
461       other.capacity_ = 0;
462     }
463
464     ~SmallVector() {
465       clear();
466     }
467
468     SmallVector(std::initializer_list<T> initList)
469         : size_(0),
470           capacity_(N),
471           buffer_(),
472           data_(reinterpret_cast<T*>(buffer_.data())) {
473       for (const auto& elem : initList) {
474         insert(cend(), elem);
475       }
476     }
477
478     template <typename Iterator>
479     SmallVector(Iterator begin, Iterator end)
480         : size_(0),
481           capacity_(N),
482           buffer_(),
483           data_(reinterpret_cast<T*>(buffer_.data())) {
484       for (auto it = begin; it != end; ++it) {
485         insert(cend(), *it);
486       }
487     }
488
489     SmallVector& operator=(const SmallVector& other) {
490       if (this != &other) {
491         SmallVector tmp(other);
492         tmp.swap(*this);
493       }
494       return *this;
495     }
496
497     void swap(SmallVector& other) noexcept {
498       const std::size_t tmp_size = size_;
499       size_ = other.size_;
500       other.size_ = tmp_size;
501
502       const std::size_t tmp_capacity = capacity_;
503       capacity_ = other.capacity_;
504       other.capacity_ = tmp_capacity;
505
506       if (data_ != reinterpret_cast<T*>(buffer_.data()) &&
507           other.data_ != reinterpret_cast<T*>(other.buffer_.data())) {
```

```
508         T* tmp_data = data_;
509         data_ = other.data_;
510         other.data_ = tmp_data;
511       }
512
513       else {
514         auto tmp_buffer = buffer_;
515         buffer_ = other.buffer_;
516         other.buffer_ = tmp_buffer;
517
518         if (data_ != reinterpret_cast<T*>(buffer_.data()) &&
519             other.data_ == reinterpret_cast<T*>(other.buffer_.data())) {
520           other.data_ = data_;
521           data_ = reinterpret_cast<T*>(buffer_.data());
522           return;
523         }
524         if (data_ == reinterpret_cast<T*>(buffer_.data()) &&
525             other.data_ != reinterpret_cast<T*>(other.buffer_.data())) {
526           data_ = other.data_;
527           other.data_ = reinterpret_cast<T*>(other.buffer_.data());
528         }
529       }
530     }
531
532     SmallVector& operator=(SmallVector&& other) noexcept {
533       if (this != &other) {
534         SmallVector tmp(std::move(other));
535         tmp.swap(*this);
536       }
537       return *this;
538     }
539
540     T* data() {
541       return data_;
542     }
543
544     const T* data() const {
545       return data_;
546     }
547
548     /// \brief For testing purposes only.
549     /// buffer.
550     T* buffer() {
551       return reinterpret_cast<T*>(buffer_.data());
552     }
553
554     T& operator[](std::size_t index) {
555       assert(index < size_ && "Index out of range");
556       return data_[index];
557     }
558
559     const T& operator[](std::size_t index) const {
560       assert(index < size_ && "Index out of range");
561       return data_[index];
562     }
563
564     T& at(std::size_t index) {
```

```
565       if (index >= size_) {
566         throw std::out_of_range("Index out of range");
567       }
568       return data_[index];
569     }
570
571     const T& at(std::size_t index) const {
572       if (index >= size_) {
573         throw std::out_of_range("Index out of range");
574       }
575       return data_[index];
576     }
577
578     void clear() {
579       for (std::size_t i = 0; i < size_; ++i) {
580         data_[i].~T();
581       }
582       if ((data_ != reinterpret_cast<T*>(buffer_.data()))) {
583         operator delete(data_);
584       }
585
586       size_ = 0;
587       capacity_ = 0;
588       data_ = nullptr;
589     }
590
591     bool empty() const {
592       return size_ == 0;
593     }
594
595     std::size_t size() const {
596       return size_;
597     }
598
599     std::size_t capacity() const {
600       return capacity_;
601     }
602
603     void push_back(const T& value) {
604       if (size_ < capacity_) {
605         new (data_ + size_) T(value);
606         ++size_;
607         return;
608       }
609
610       T* new_data = static_cast<T*>(operator new((capacity_ * 2) *
611 sizeof(T)));
612
613       for (std::size_t i = 0; i < size_; ++i) {
614         new (new_data + i) T(std::move(data_[i]));
615       }
616
617       new (new_data + size_) T(value);
618
619       if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr)
620 {
621         operator delete(data_);
```

```
622       }
623       data_ = new_data;
624       capacity_ *= 2;
625       ++size_;
626     }
627
628     void resize(size_t n) {
629       if (size_ == n) {
630         return;
631       }
632
633       if (n < size_) {
634         for (size_t i = n; i < size_; ++i) {
635           data_[i].~T();
636         }
637         size_ = n;
638         return;
639       }
640
641       if (n <= capacity_) {
642         for (std::size_t i = size_; i < n; ++i) {
643           new (data_ + i) T();
644         }
645         size_ = n;
646         return;
647       }
648
649       T* new_data = static_cast<T*>(operator new(n * sizeof(T)));
650       for (size_t i = 0; i < size_; ++i) {
651         new (new_data + i) T(std::move(data_[i]));
652       }
653
654       for (size_t i = size_; i < n; ++i) {
655         new (new_data + i) T();
656       }
657       capacity_ = n;
658       size_ = n;
659
660       if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr)
661 {
662         operator delete(data_);
663       }
664       data_ = new_data;
665     }
666
667     void reserve(size_t n) {
668       if (n <= capacity_) {
669         return;
670       }
671
672       T* new_data = static_cast<T*>(operator new(n * sizeof(T)));
673
674       for (size_t i = 0; i < size_; ++i) {
675         new (new_data + i) T(std::move(data_[i]));
676       }
677
678       if (data_ != reinterpret_cast<T*>(buffer_.data()) && data_ != nullptr)
```

```
679 {
680       operator delete(data_);
681     }
682 
683     data_ = new_data;
684     capacity_ = n;
685   }
686 
687   void shrink_to_fit() {
688     if (size_ == capacity_ || data_ == reinter-
689 pret_cast<T*>(buffer_.data())) {
690       return;
691     }
692 
693     T* new_data = static_cast<T*>(operator new(size_ * sizeof(T)));
694 
695     for (size_t i = 0; i < size_; ++i) {
696       new (new_data + i) T(std::move(data_[i]));
697     }
698 
699     operator delete(data_);
700 
701     data_ = new_data;
702     capacity_ = size_;
703   }
704 
705   iterator erase(const_iterator pos) {
706     const std::size_t index = pos - cbegin();
707     if (index >= size_ || index < 0) {
708       throw std::out_of_range("Iterator pos is not valid for this contain-
709 er");
710     }
711 
712     for (std::size_t i = index + 1; i < size_; ++i) {
713       data_[i - 1] = std::move(data_[i]);
714     }
715 
716     data_[size_ - 1].~T();
717 
718     --size_;
719     return begin() + index;
720   }
721 
722   iterator erase(const_iterator first, const_iterator last) {
723     const std::size_t index_first = first - cbegin();
724     if (index_first >= size_ || index_first < 0) {
725       throw std::out_of_range("Iterator pos is not valid for this contain-
726 er");
727     }
728 
729     const std::size_t index_last = last - cbegin();
730     if (index_last >= size_ || index_last < 0) {
731       throw std::out_of_range("Iterator pos is not valid for this contain-
732 er");
733     }
734 
735     const std::size_t remove_count = last - first;
```

```
736
737     for (std::size_t i = index_last; i < size_; ++i) {
738       data_[i - remove_count] = std::move(data_[i]);
739     }
740
741     for (std::size_t i = size_ - remove_count; i < size_; ++i) {
742       data_[i].~T();
743     }
744
745     size_ -= remove_count;
746
747     return begin() + index_first + remove_count;
748   }
749
750   iterator insert(const_iterator pos, const T& value) {
751     const std::size_t index = pos - cbegin();
752     if (index > size_ || index < 0) {
753       throw std::out_of_range("Iterator pos is not valid for this contain-
754 er");
755     }
756
757     if (size_ == capacity_) {
758       reserve((capacity_ + 1) * 2);
759     }
760
761     if (pos == cbegin() + size_) {
762       new (data_ + size_) T(value);
763       ++size_;
764       return begin() + index;
765     }
766
767     new (data_ + size_) T(std::move(data_[size_ - 1]));
768
769     for (std::size_t i = size_ - 1; i > index; --i) {
770       data_[i] = std::move(data_[i - 1]);
771     }
772
773     data_[index] = value;
774     ++size_;
775
776     return begin() + index;
777   }
778
779   template <class InputIt>
780   iterator insert(const_iterator pos, InputIt first, InputIt last) {
781     const std::size_t index = pos - cbegin();
782     if (index > size_ || index < 0) {
783       throw std::out_of_range("Iterator pos is not valid for this contain-
784 er");
785     }
786
787     const std::size_t insert_count = last - first;
788
789     if (size_ + insert_count > capacity_) {
790       reserve(size_ + insert_count);
791     }
792
```

```
793       for (std::size_t i = size_ + insert_count - 1; i > index + insert_count
794   - 1;
795            --i) {
796         new (data_ + i) T(std::move(data_[i - insert_count]));
797         data_[i - insert_count].~T();
798       }
799
800       std::size_t i = index;
801       for (auto it = first; it != last; ++it) {
802         new (data_ + i) T(*it);
803         ++i;
804       }
805
806       size_ += insert_count;
807
808       return begin() + index;
809     }
810
811     iterator begin() {
812       return iterator(data_);
813     }
814
815     iterator end() {
816       return iterator(data_ + size_);
817     }
818
819     const_iterator cbegin() const {
820       return const_iterator(data_);
821     }
822
823     const_iterator cend() const {
824       return const_iterator(data_ + size_);
825     }
826
827     reverse_iterator rbegin() {
828       return reverse_iterator(data_ + size_ - 1);
829     }
830
831     reverse_iterator rend() {
832       return reverse_iterator(data_ - 1);
833     }
834
835     const_reverse_iterator crbegin() const {
836       return const_reverse_iterator(data_ + size_ - 1);
837     }
839
840     const_reverse_iterator crend() const {
841       return const_reverse_iterator(data_ - 1);
842     }
843   };
844   }  // namespace svector
```

// file libsvector/libsvector/iterators/iterator_test.cpp

```
1   #include <gtest/gtest.h>
2
```

```cpp
#include <libsvector/svector/svector.hpp>

TEST(IteratorTest, ConstructorTest) {
  svector::SmallVector<int, 5> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);

  const svector::SmallVector<int, 5>::iterator it = vec.begin();

  EXPECT_EQ(*it, 10);
}

TEST(IteratorTest, OperatorIncrementPrefixTest) {
  svector::SmallVector<int, 5> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);

  svector::SmallVector<int, 5>::iterator it = vec.begin();

  ++it;

  EXPECT_EQ(*it, 20);
}

TEST(IteratorTest, OperatorIncrementPostfixTest) {
  svector::SmallVector<int, 5> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);

  svector::SmallVector<int, 5>::iterator it = vec.begin();

  const svector::SmallVector<int, 5>::iterator old_it = it++;

  EXPECT_EQ(*old_it, 10);

  EXPECT_EQ(*it, 20);
}

TEST(IteratorTest, OperatorDecrementPrefixTest) {
  svector::SmallVector<int, 5> vec;
  vec.push_back(10);
  vec.push_back(20);
  vec.push_back(30);

  svector::SmallVector<int, 5>::iterator it = vec.end();
  --it;

  EXPECT_EQ(*it, 30);
}

TEST(IteratorTest, OperatorDecrementPostfixTest) {
  svector::SmallVector<int, 5> vec;
  vec.push_back(10);
  vec.push_back(20);
```

```
60    vec.push_back(30);
61
62    svector::SmallVector<int, 5>::iterator it = vec.end();
63    --it;
64    const svector::SmallVector<int, 5>::iterator old_it = it--;
65
66    EXPECT_EQ(*old_it, 30);
67
68    EXPECT_EQ(*it, 20);
69  }
70
71  TEST(IteratorTest, AdditionOperatorTest) {
72    svector::SmallVector<int, 5> vec;
73    vec.push_back(10);
74    vec.push_back(20);
75    vec.push_back(30);
76
77    const svector::SmallVector<int, 5>::iterator it = vec.begin();
78
79    const svector::SmallVector<int, 5>::iterator new_it = it + 2;
80
81    EXPECT_EQ(*new_it, 30);
82  }
83
84  TEST(IteratorTest, SubtractionOperatorTest) {
85    svector::SmallVector<int, 5> vec;
86    vec.push_back(10);
87    vec.push_back(20);
88    vec.push_back(30);
89
90    const svector::SmallVector<int, 5>::iterator it = vec.begin() + 2;
91
92    const svector::SmallVector<int, 5>::iterator new_it = it - 2;
93
94    EXPECT_EQ(*new_it, 10);
95  }
96
97  TEST(IteratorTest, SubtractionBetweenOperatorsTest) {
98    svector::SmallVector<int, 5> vec;
99    vec.push_back(10);
100   vec.push_back(20);
101   vec.push_back(30);
102
103   const svector::SmallVector<int, 5>::iterator it_begin = vec.begin();
104   const svector::SmallVector<int, 5>::iterator it_end = vec.end();
105   const svector::SmallVector<int, 5>::iterator it_third = vec.begin() + 2;
106
107   auto diff = it_end - it_begin;
108
109   EXPECT_EQ(diff, 3);
110
111   diff = it_third - it_begin;
112
113   EXPECT_EQ(diff, 2);
114 }
115
116 TEST(IteratorTest, AdditionAssignmentOperatorTest) {
```

```
117   svector::SmallVector<int, 5> vec;
118   vec.push_back(10);
119   vec.push_back(20);
120   vec.push_back(30);
121
122   svector::SmallVector<int, 5>::iterator it = vec.begin();
123
124   it += 2;
125
126   EXPECT_EQ(*it, 30);
127 }
128
129 TEST(IteratorTest, SubtractionAssignmentOperatorTest) {
130   svector::SmallVector<int, 5> vec;
131   vec.push_back(10);
132   vec.push_back(20);
133   vec.push_back(30);
134
135   svector::SmallVector<int, 5>::iterator it = vec.end();
136
137   it -= 2;
138
139   EXPECT_EQ(*it, 20);
140 }
141
142 TEST(IteratorTest, DereferenceOperatorTest) {
143   svector::SmallVector<int, 5> vec;
144   vec.push_back(10);
145
146   const svector::SmallVector<int, 5>::iterator it = vec.begin();
147
148   EXPECT_EQ(*it, 10);
149 }
150
151 TEST(IteratorTest, MemberAccessOperatorTest) {
152   svector::SmallVector<std::string, 5> vec;
153   vec.push_back("Hello");
154
155   const svector::SmallVector<std::string, 5>::iterator it = vec.begin();
156
157   EXPECT_EQ(it->size(), 5);
158   EXPECT_EQ(*it, "Hello");
159 }
160
161 TEST(IteratorTest, BracketOperatorTest) {
162   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
163
164   const svector::SmallVector<int, 5>::iterator it = vec.begin();
165
166   EXPECT_EQ(it[0], 10);
167
168   EXPECT_EQ(it[2], 30);
169 }
170
171 TEST(IteratorTest, OperatorEqualTest) {
172   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
173
```

```
174   auto it1 = vec.begin();
175   auto it2 = vec.begin();
176
177   EXPECT_EQ(it1 == it2, true);
178
179   ++it1;
180
181   EXPECT_NE(it1 == it2, true);
182 }
183
184 TEST(IteratorTest, OperatorNotEqualTest) {
185   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
186
187   auto it1 = vec.begin();
188   auto it2 = vec.begin();
189
190   EXPECT_NE(it1 != it2, true);
191
192   ++it1;
193
194   EXPECT_EQ(it1 != it2, true);
195 }
196
197 TEST(IteratorTest, OperatorLessThanTest) {
198   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
199
200   auto it1 = vec.begin();
201   auto it2 = vec.begin();
202
203   EXPECT_FALSE(it1 < it2);
204
205   ++it2;
206
207   EXPECT_TRUE(it1 < it2);
208 }
209
210 TEST(IteratorTest, OperatorLessThanOrEqualTest) {
211   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
212
213   auto it1 = vec.begin();
214   auto it2 = vec.begin();
215
216   EXPECT_TRUE(it1 <= it2);
217
218   ++it2;
219
220   EXPECT_TRUE(it1 <= it2);
221
222   ++it1;
223   ++it1;
224
225   EXPECT_FALSE(it1 <= it2);
226
227   it2 = vec.end();
228
229   EXPECT_TRUE(it1 <= it2);
230 }
```

```
231
232 TEST(IteratorTest, OperatorGreaterThanTest) {
233   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
234
235   auto it1 = vec.begin();
236   auto it2 = vec.begin();
237
238   EXPECT_FALSE(it1 > it2);
239
240   ++it2;
241
242   EXPECT_FALSE(it1 > it2);
243
244   ++it1;
245   ++it1;
246
247   EXPECT_TRUE(it1 > it2);
248
249   it2 = vec.end();
250
251   EXPECT_FALSE(it1 > it2);
252 }
253
254 TEST(IteratorTest, OperatorGreaterThanOrEqualTest) {
255   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
256
257   auto it1 = vec.begin();
258   auto it2 = vec.begin();
259
260   EXPECT_TRUE(it1 >= it2);
261
262   ++it2;
263
264   EXPECT_FALSE(it1 >= it2);
265
266   ++it1;
267   ++it1;
268
269   EXPECT_TRUE(it1 > it2);
270
271   it2 = vec.end();
272
273   EXPECT_FALSE(it1 >= it2);
274 }
275
276 TEST(IteratorTest, SwapTest) {
277   svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};
278
279   auto it1 = vec.begin();
280   auto it2 = vec.end() - 1;
281
282   auto value1 = *it1;
283   auto value2 = *it2;
284
285   it1.swap(it2);
286
287   EXPECT_EQ(*it1, value2);
```

```
288   EXPECT_EQ(*it2, value1);
289 }
290
291 int main(int argc, char** argv) {
292   ::testing::InitGoogleTest(&argc, argv);
293   return RUN_ALL_TESTS();
294 }
```

// file libsvector/libsvector/iterators/const_iterator_test.cpp

```
  1 #include <gtest/gtest.h>
  2
  3 #include <libsvector/svector/svector.hpp>
  4
  5 TEST(ConstIteratorTest, ConstructorTest) {
  6   svector::SmallVector<int, 5> vec;
  7   vec.push_back(10);
  8   vec.push_back(20);
  9   vec.push_back(30);
 10
 11   const svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
 12
 13   EXPECT_EQ(*it, 10);
 14 }
 15
 16 TEST(ConstIteratorTest, OperatorIncrementPrefixTest) {
 17   svector::SmallVector<int, 5> vec;
 18   vec.push_back(10);
 19   vec.push_back(20);
 20   vec.push_back(30);
 21
 22   svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
 23
 24   ++it;
 25
 26   EXPECT_EQ(*it, 20);
 27 }
 28
 29 TEST(ConstIteratorTest, OperatorIncrementPostfixTest) {
 30   svector::SmallVector<int, 5> vec;
 31   vec.push_back(10);
 32   vec.push_back(20);
 33   vec.push_back(30);
 34
 35   svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
 36
 37   const svector::SmallVector<int, 5>::const_iterator old_it = it++;
 38
 39   EXPECT_EQ(*old_it, 10);
 40
 41   EXPECT_EQ(*it, 20);
 42 }
 43
 44 TEST(ConstIteratorTest, OperatorDecrementPrefixTest) {
 45   svector::SmallVector<int, 5> vec;
 46   vec.push_back(10);
```

```
 47    vec.push_back(20);
 48    vec.push_back(30);
 49
 50    svector::SmallVector<int, 5>::const_iterator it = vec.cend();
 51    --it;
 52
 53    EXPECT_EQ(*it, 30);
 54  }
 55
 56  TEST(ConstIteratorTest, OperatorDecrementPostfixTest) {
 57    svector::SmallVector<int, 5> vec;
 58    vec.push_back(10);
 59    vec.push_back(20);
 60    vec.push_back(30);
 61
 62    svector::SmallVector<int, 5>::const_iterator it = vec.cend();
 63    --it;
 64    const svector::SmallVector<int, 5>::const_iterator old_it = it--;
 65
 66    EXPECT_EQ(*old_it, 30);
 67
 68    EXPECT_EQ(*it, 20);
 69  }
 70
 71  TEST(ConstIteratorTest, AdditionOperatorTest) {
 72    svector::SmallVector<int, 5> vec;
 73    vec.push_back(10);
 74    vec.push_back(20);
 75    vec.push_back(30);
 76
 77    const svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
 78
 79    const svector::SmallVector<int, 5>::const_iterator new_it = it + 2;
 80
 81    EXPECT_EQ(*new_it, 30);
 82  }
 83
 84  TEST(ConstIteratorTest, SubtractionOperatorTest) {
 85    svector::SmallVector<int, 5> vec;
 86    vec.push_back(10);
 87    vec.push_back(20);
 88    vec.push_back(30);
 89
 90    const svector::SmallVector<int, 5>::const_iterator it = vec.cbegin() + 2;
 91
 92    const svector::SmallVector<int, 5>::const_iterator new_it = it - 2;
 93
 94    EXPECT_EQ(*new_it, 10);
 95  }
 96
 97  TEST(ConstIteratorTest, SubtractionBetweenOperatorsTest) {
 98    svector::SmallVector<int, 5> vec;
 99    vec.push_back(10);
100    vec.push_back(20);
101    vec.push_back(30);
102
103    const svector::SmallVector<int, 5>::const_iterator it_begin =
```

```
104 vec.cbegin();
105   const svector::SmallVector<int, 5>::const_iterator it_end = vec.cend();
106   const svector::SmallVector<int, 5>::const_iterator it_third =
107       vec.cbegin() + 2;
108 
109   auto diff = it_end - it_begin;
110 
111   EXPECT_EQ(diff, 3);
112 
113   diff = it_third - it_begin;
114 
115   EXPECT_EQ(diff, 2);
116 }
117 
118 TEST(ConstIteratorTest, AdditionAssignmentOperatorTest) {
119   svector::SmallVector<int, 5> vec;
120   vec.push_back(10);
121   vec.push_back(20);
122   vec.push_back(30);
123 
124   svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
125 
126   it += 2;
127 
128   EXPECT_EQ(*it, 30);
129 }
130 
131 TEST(ConstIteratorTest, SubtractionAssignmentOperatorTest) {
132   svector::SmallVector<int, 5> vec;
133   vec.push_back(10);
134   vec.push_back(20);
135   vec.push_back(30);
136 
137   svector::SmallVector<int, 5>::const_iterator it = vec.cend();
138 
139   it -= 2;
140 
141   EXPECT_EQ(*it, 20);
142 }
143 
144 TEST(ConstIteratorTest, DereferenceOperatorTest) {
145   svector::SmallVector<int, 5> vec;
146   vec.push_back(10);
147 
148   const svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
149 
150   EXPECT_EQ(*it, 10);
151 }
152 
153 TEST(ConstIteratorTest, MemberAccessOperatorTest) {
154   svector::SmallVector<std::string, 5> vec;
155   vec.push_back("Hello");
156 
157   const svector::SmallVector<std::string, 5>::const_iterator it =
158 vec.cbegin();
159 
160   EXPECT_EQ(it->size(), 5);
```

```
161   EXPECT_EQ(*it, "Hello");
162 }
163
164 TEST(ConstIteratorTest, BracketOperatorTest) {
165   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
166
167   const svector::SmallVector<int, 5>::const_iterator it = vec.cbegin();
168
169   EXPECT_EQ(it[0], 10);
170
171   EXPECT_EQ(it[2], 30);
172 }
173
174 TEST(ConstIteratorTest, OperatorEqualTest) {
175   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
176
177   auto it1 = vec.cbegin();
178   auto it2 = vec.cbegin();
179
180   EXPECT_EQ(it1 == it2, true);
181
182   ++it1;
183
184   EXPECT_NE(it1 == it2, true);
185 }
186
187 TEST(ConstIteratorTest, OperatorNotEqualTest) {
188   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
189
190   auto it1 = vec.cbegin();
191   auto it2 = vec.cbegin();
192
193   EXPECT_NE(it1 != it2, true);
194
195   ++it1;
196
197   EXPECT_EQ(it1 != it2, true);
198 }
199
200 TEST(ConstIteratorTest, OperatorLessThanTest) {
201   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
202
203   auto it1 = vec.cbegin();
204   auto it2 = vec.cbegin();
205
206   EXPECT_FALSE(it1 < it2);
207
208   ++it2;
209
210   EXPECT_TRUE(it1 < it2);
211 }
212
213 TEST(ConstIteratorTest, OperatorLessThanOrEqualTest) {
214   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
215
216   auto it1 = vec.cbegin();
217   auto it2 = vec.cbegin();
```

```
218
219   EXPECT_TRUE(it1 <= it2);
220
221   ++it2;
222
223   EXPECT_TRUE(it1 <= it2);
224
225   ++it1;
226   ++it1;
227
228   EXPECT_FALSE(it1 <= it2);
229
230   it2 = vec.cend();
231
232   EXPECT_TRUE(it1 <= it2);
233 }
234
235 TEST(ConstIteratorTest, OperatorGreaterThanTest) {
236   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
237
238   auto it1 = vec.cbegin();
239   auto it2 = vec.cbegin();
240
241   EXPECT_FALSE(it1 > it2);
242
243   ++it2;
244
245   EXPECT_FALSE(it1 > it2);
246
247   ++it1;
248   ++it1;
249
250   EXPECT_TRUE(it1 > it2);
251
252   it2 = vec.cend();
253
254   EXPECT_FALSE(it1 > it2);
255 }
256
257 TEST(ConstIteratorTest, OperatorGreaterThanOrEqualTest) {
258   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
259
260   auto it1 = vec.cbegin();
261   auto it2 = vec.cbegin();
262
263   EXPECT_TRUE(it1 >= it2);
264
265   ++it2;
266
267   EXPECT_FALSE(it1 >= it2);
268
269   ++it1;
270   ++it1;
271
272   EXPECT_TRUE(it1 > it2);
273
274   it2 = vec.cend();
```

```
275
276   EXPECT_FALSE(it1 >= it2);
277 }
278
279 TEST(ConstIteratorTest, SwapTest) {
280   const svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};
281
282   auto it1 = vec.cbegin();
283   auto it2 = vec.cend() - 1;
284
285   auto value1 = *it1;
286   auto value2 = *it2;
287
288   it1.swap(it2);
289
290   EXPECT_EQ(*it1, value2);
291   EXPECT_EQ(*it2, value1);
292 }
293
294 int main(int argc, char** argv) {
295   ::testing::InitGoogleTest(&argc, argv);
296   return RUN_ALL_TESTS();
297 }
```

// file libsvector/libsvector/iterators/reverse_iterator_test.cpp

```
  1 #include <gtest/gtest.h>
  2
  3 #include <libsvector/svector/svector.hpp>
  4
  5 TEST(ReverseIteratorTest, ConstructorTest) {
  6   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
  7
  8   int* ptr = &vec[vec.size() - 1];
  9
 10   const svector::SmallVector<int, 5>::reverse_iterator it(ptr);
 11
 12   EXPECT_EQ(*it, 5);
 13 }
 14
 15 TEST(ReverseIteratorTest, OperatorIncrementPrefixTest) {
 16   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
 17
 18   auto it = vec.rbegin();
 19
 20   ++it;
 21
 22   EXPECT_EQ(*it, 4);
 23 }
 24
 25 TEST(ReverseIteratorTest, OperatorIncrementPostfixTest) {
 26   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
 27
 28   auto it = vec.rbegin();
 29
 30   auto prev_it = it;
```

```
31
32    auto result = it++;
33
34    EXPECT_EQ(*prev_it, *result);
35
36    EXPECT_EQ(*it, 4);
37  }
38
39  TEST(ReverseIteratorTest, OperatorDecrementPrefixTest) {
40    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
41
42    auto it = vec.rend();
43
44    --it;
45
46    EXPECT_EQ(*it, 1);
47
48    auto result = --it;
49
50    EXPECT_EQ(*result, 2);
51  }
52
53  TEST(ReverseIteratorTest, OperatorDecrementPostfixTest) {
54    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
55
56    auto it = vec.rend();
57
58    --it;
59
60    EXPECT_EQ(*it, 1);
61
62    auto result = it--;
63
64    EXPECT_EQ(*result, 1);
65
66    EXPECT_EQ(*it, 2);
67  }
68
69  TEST(ReverseIteratorTest, AdditionOperatorTest) {
70    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
71
72    auto it = vec.rbegin();
73
74    auto result = it + 2;
75
76    EXPECT_EQ(*result, 3);
77  }
78
79  TEST(ReverseIteratorTest, SubtractionOperatorTest) {
80    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
81
82    auto it = vec.rend() - 3;
83
84    EXPECT_EQ(*it, 3);
85  }
86
87  TEST(ReverseIteratorTest, SubtractionBetweenOperatorsTest) {
```

```
 88    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
 89
 90    auto it1 = vec.rend() - 2;
 91    auto it2 = vec.rend() - 4;
 92
 93    EXPECT_EQ(it1 - it2, 2);
 94    EXPECT_EQ(it2 - it1, -2);
 95  }
 96
 97  TEST(ReverseIteratorTest, AdditionAssignmentOperatorTest) {
 98    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
 99
100    auto it = vec.rbegin();
101    EXPECT_EQ(*it, 5);
102    it += 3;
103
104    EXPECT_EQ(*it, 2);
105  }
106
107  TEST(ReverseIteratorTest, SubtractionAssignmentOperatorTest) {
108    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
109
110    auto it = vec.rend();
111    it -= 2;
112
113    EXPECT_EQ(*it, 2);
114  }
115
116  TEST(ReverseIteratorTest, DereferenceOperatorTest) {
117    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
118
119    auto it = vec.rend();
120    --it;
121
122    EXPECT_EQ(*it, 1);
123  }
124
125  TEST(ReverseIteratorTest, MemberAccessOperatorTest) {
126    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
127
128    auto it = vec.rend();
129    --it;
130
131    EXPECT_EQ(it.operator->(), vec.data());
132  }
133
134  TEST(ReverseIteratorTest, BracketOperatorTest) {
135    svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
136
137    auto it = vec.rbegin();
138
139    EXPECT_EQ(it[0], 5);
140    EXPECT_EQ(it[1], 4);
141    EXPECT_EQ(it[2], 3);
142  }
143
144  TEST(ReverseIteratorTest, OperatorEqualTest) {
```

```
145   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
146
147   auto it1 = vec.rend();
148   auto it2 = vec.rend();
149   EXPECT_TRUE(it1 == it2);
150
151   --it1;
152   EXPECT_FALSE(it1 == it2);
153 }
154
155 TEST(ReverseIteratorTest, OperatorNotEqualTest) {
156   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
157
158   auto it1 = vec.rend();
159   auto it2 = vec.rbegin();
160
161   EXPECT_TRUE(it1 != it2);
162   EXPECT_FALSE(it2 != vec.rbegin());
163 }
164
165 TEST(ReverseIteratorTest, OperatorLessThanTest) {
166   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
167
168   auto it1 = vec.rend();
169   auto it2 = vec.rbegin();
170
171   EXPECT_FALSE(it1 < it2);
172 }
173
174 TEST(ReverseIteratorTest, OperatorLessThanOrEqualTest) {
175   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
176
177   auto it1 = vec.rend();
178   auto it2 = vec.rbegin();
179
180   EXPECT_FALSE(it1 <= it2);
181   EXPECT_TRUE(it2 <= vec.rbegin());
182   EXPECT_TRUE(vec.rbegin() <= it2);
183 }
184
185 TEST(ReverseIteratorTest, OperatorGreaterThanTest) {
186   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
187
188   auto it1 = vec.rbegin();
189   auto it2 = vec.rbegin();
190
191   EXPECT_FALSE(it1 > it2);
192
193   ++it2;
194
195   EXPECT_FALSE(it1 > it2);
196
197   ++it1;
198   ++it1;
199
200   EXPECT_TRUE(it1 > it2);
201
```

```
202   it2 = vec.rend();
203
204   EXPECT_FALSE(it1 > it2);
205 }
206
207 TEST(ReverseIteratorTest, OperatorGreaterThanOrEqualTest) {
208   svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
209
210   auto it1 = vec.rbegin();
211   auto it2 = vec.rbegin();
212
213   EXPECT_TRUE(it1 >= it2);
214
215   ++it2;
216
217   EXPECT_FALSE(it1 >= it2);
218
219   ++it1;
220   ++it1;
221
222   EXPECT_TRUE(it1 > it2);
223
224   it2 = vec.rend();
225
226   EXPECT_FALSE(it1 >= it2);
227 }
228
229 TEST(ReverseIteratorTest, SwapTest) {
230   svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};
231
232   auto it1 = vec.rbegin();
233   auto it2 = vec.rend() - 1;
234
235   auto value1 = *it1;
236   auto value2 = *it2;
237
238   it1.swap(it2);
239
240   EXPECT_EQ(*it1, value2);
241   EXPECT_EQ(*it2, value1);
242 }
243
244 int main(int argc, char** argv) {
245   ::testing::InitGoogleTest(&argc, argv);
246   return RUN_ALL_TESTS();
247 }
```

// file libsvector/libsvector/iterators/const_reverse_iterator_test.cpp

```
  1 #include <gtest/gtest.h>
  2
  3 #include <libsvector/svector/svector.hpp>
  4
  5 TEST(ConstReverseIteratorTest, ConstructorTest) {
  6   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
  7
```

```
  int* ptr = &vec[vec.size() - 1];

  const svector::SmallVector<int, 5>::const_reverse_iterator it(ptr);

  EXPECT_EQ(*it, 5);
}

TEST(ConstReverseIteratorTest, OperatorIncrementPrefixTest) {
  const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};

  auto it = vec.crbegin();

  ++it;

  EXPECT_EQ(*it, 4);
}

TEST(ConstReverseIteratorTest, OperatorIncrementPostfixTest) {
  const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};

  auto it = vec.crbegin();

  auto prev_it = it;

  auto result = it++;

  EXPECT_EQ(*prev_it, *result);

  EXPECT_EQ(*it, 4);
}

TEST(ConstReverseIteratorTest, OperatorDecrementPrefixTest) {
  const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};

  auto it = vec.crend();

  --it;

  EXPECT_EQ(*it, 1);

  auto result = --it;

  EXPECT_EQ(*result, 2);
}

TEST(ConstReverseIteratorTest, OperatorDecrementPostfixTest) {
  const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};

  auto it = vec.crend();

  --it;

  EXPECT_EQ(*it, 1);

  auto result = it--;

  EXPECT_EQ(*result, 1);
```

```
65
66    EXPECT_EQ(*it, 2);
67  }
68
69  TEST(ConstReverseIteratorTest, AdditionOperatorTest) {
70    const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
71
72    auto it = vec.crbegin();
73
74    auto result = it + 2;
75
76    EXPECT_EQ(*result, 3);
77  }
78
79  TEST(ConstReverseIteratorTest, SubtractionOperatorTest) {
80    const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
81
82    auto it = vec.crend() - 3;
83
84    EXPECT_EQ(*it, 3);
85  }
86
87  TEST(ConstReverseIteratorTest, SubtractionBetweenOperatorsTest) {
88    const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
89
90    auto it1 = vec.crend() - 2;
91    auto it2 = vec.crend() - 4;
92
93    EXPECT_EQ(it1 - it2, 2);
94    EXPECT_EQ(it2 - it1, -2);
95  }
96
97  TEST(ConstReverseIteratorTest, AdditionAssignmentOperatorTest) {
98    const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
99
100   auto it = vec.crbegin();
101   EXPECT_EQ(*it, 5);
102   it += 3;
103
104   EXPECT_EQ(*it, 2);
105 }
106
107 TEST(ConstReverseIteratorTest, SubtractionAssignmentOperatorTest) {
108   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
109
110   auto it = vec.crend();
111   it -= 2;
112
113   EXPECT_EQ(*it, 2);
114 }
115
116 TEST(ConstReverseIteratorTest, DereferenceOperatorTest) {
117   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
118
119   auto it = vec.crend();
120   --it;
121
```

```
122   EXPECT_EQ(*it, 1);
123 }
124
125 TEST(ConstReverseIteratorTest, MemberAccessOperatorTest) {
126   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
127
128   auto it = vec.rend();
129   --it;
130
131   EXPECT_EQ(it.operator->(), vec.data());
132 }
133
134 TEST(ConstReverseIteratorTest, BracketOperatorTest) {
135   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
136
137   auto it = vec.rbegin();
138
139   EXPECT_EQ(it[0], 5);
140   EXPECT_EQ(it[1], 4);
141   EXPECT_EQ(it[2], 3);
142 }
143
144 TEST(ConstReverseIteratorTest, OperatorEqualTest) {
145   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
146
147   auto it1 = vec.crend();
148   auto it2 = vec.crend();
149   EXPECT_TRUE(it1 == it2);
150
151   --it1;
152   EXPECT_FALSE(it1 == it2);
153 }
154
155 TEST(ConstReverseIteratorTest, OperatorNotEqualTest) {
156   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
157
158   auto it1 = vec.crend();
159   auto it2 = vec.crbegin();
160
161   EXPECT_TRUE(it1 != it2);
162   EXPECT_FALSE(it2 != vec.crbegin());
163 }
164
165 TEST(ConstReverseIteratorTest, OperatorLessThanTest) {
166   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
167
168   auto it1 = vec.crend();
169   auto it2 = vec.crbegin();
170
171   EXPECT_FALSE(it1 < it2);
172   EXPECT_FALSE(it2 < vec.crbegin());
173 }
174
175 TEST(ConstReverseIteratorTest, OperatorLessThanOrEqualTest) {
176   const svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
177
178   auto it1 = vec.crend();
```

```
179   auto it2 = vec.crbegin();
180
181   EXPECT_FALSE(it1 <= it2);
182   EXPECT_TRUE(it2 <= vec.crbegin());
183   EXPECT_TRUE(vec.crbegin() <= it2);
184 }
185
186 TEST(ConstReverseIteratorTest, OperatorGreaterThanTest) {
187   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
188
189   auto it1 = vec.crbegin();
190   auto it2 = vec.crbegin();
191
192   EXPECT_FALSE(it1 > it2);
193
194   ++it2;
195
196   EXPECT_FALSE(it1 > it2);
197
198   ++it1;
199   ++it1;
200
201   EXPECT_TRUE(it1 > it2);
202
203   it2 = vec.crend();
204
205   EXPECT_FALSE(it1 > it2);
206 }
207
208 TEST(ConstReverseIteratorTest, OperatorGreaterThanOrEqualTest) {
209   const svector::SmallVector<int, 5> vec = {10, 20, 30, 40, 50};
210
211   auto it1 = vec.crbegin();
212   auto it2 = vec.crbegin();
213
214   EXPECT_TRUE(it1 >= it2);
215
216   ++it2;
217
218   EXPECT_FALSE(it1 >= it2);
219
220   ++it1;
221   ++it1;
222
223   EXPECT_TRUE(it1 > it2);
224
225   it2 = vec.crend();
226
227   EXPECT_FALSE(it1 >= it2);
228 }
229
230 TEST(ConstReverseIteratorTest, SwapTest) {
231   const svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};
232
233   auto it1 = vec.crbegin();
234   auto it2 = vec.crend() - 1;
235
```

```
236   auto value1 = *it1;
237   auto value2 = *it2;
238
239   it1.swap(it2);
240
241   EXPECT_EQ(*it1, value2);
242   EXPECT_EQ(*it2, value1);
243 }
244
245 int main(int argc, char** argv) {
246   ::testing::InitGoogleTest(&argc, argv);
247   return RUN_ALL_TESTS();
248 }
```

// file libsvector/libsvector/smallvector/basic_methods_test.cpp

```
  1 #include <gtest/gtest.h>
  2
  3 #include <libsvector/svector/svector.hpp>
  4
  5 TEST(SmallVectorTest, SwapTestOnlyBuffer) {
  6   svector::SmallVector<int, 4> vec1 = {1, 2, 3};
  7   svector::SmallVector<int, 4> vec2 = {4, 5, 6};
  8
  9   const std::size_t size1 = vec1.size();
 10   const std::size_t size2 = vec2.size();
 11   const std::size_t capacity1 = vec1.capacity();
 12   const std::size_t capacity2 = vec2.capacity();
 13
 14   vec1.swap(vec2);
 15
 16   EXPECT_EQ(vec1.size(), size2);
 17   EXPECT_EQ(vec1.capacity(), capacity2);
 18   EXPECT_EQ(vec2.size(), size1);
 19   EXPECT_EQ(vec2.capacity(), capacity1);
 20   EXPECT_EQ(vec2[0], 1);
 21   EXPECT_EQ(vec2[1], 2);
 22   EXPECT_EQ(vec2[2], 3);
 23   EXPECT_EQ(vec1[0], 4);
 24   EXPECT_EQ(vec1[1], 5);
 25   EXPECT_EQ(vec1[2], 6);
 26 }
 27
 28 TEST(SmallVectorTest, SwapTestOnlyData) {
 29   svector::SmallVector<int, 2> vec1 = {1, 2};
 30   svector::SmallVector<int, 2> vec2 = {4, 5};
 31
 32   vec1.push_back(3);
 33   vec2.push_back(6);
 34
 35   const std::size_t size1 = vec1.size();
 36   const std::size_t size2 = vec2.size();
 37   const std::size_t capacity1 = vec1.capacity();
 38   const std::size_t capacity2 = vec2.capacity();
 39
 40   vec1.swap(vec2);
```

```
41
42    EXPECT_EQ(vec1.size(), size2);
43    EXPECT_EQ(vec1.capacity(), capacity2);
44    EXPECT_EQ(vec2.size(), size1);
45    EXPECT_EQ(vec2.capacity(), capacity1);
46    EXPECT_EQ(vec2[0], 1);
47    EXPECT_EQ(vec2[1], 2);
48    EXPECT_EQ(vec2[2], 3);
49    EXPECT_EQ(vec1[0], 4);
50    EXPECT_EQ(vec1[1], 5);
51    EXPECT_EQ(vec1[2], 6);
52  }
53
54  TEST(SmallVectorTest, SwapTestDataAndBuffer) {
55    svector::SmallVector<int, 2> vec1 = {1, 2};
56    svector::SmallVector<int, 2> vec2 = {4, 5};
57    vec1.push_back(3);
58
59    const std::size_t size1 = vec1.size();
60    const std::size_t size2 = vec2.size();
61    const std::size_t capacity1 = vec1.capacity();
62    const std::size_t capacity2 = vec2.capacity();
63
64    vec1.swap(vec2);
65
66    EXPECT_EQ(vec1.size(), size2);
67    EXPECT_EQ(vec1.capacity(), capacity2);
68    EXPECT_EQ(vec2.size(), size1);
69    EXPECT_EQ(vec2.capacity(), capacity1);
70    EXPECT_EQ(vec2[0], 1);
71    EXPECT_EQ(vec2[1], 2);
72    EXPECT_EQ(vec2[2], 3);
73    EXPECT_EQ(vec1[0], 4);
74    EXPECT_EQ(vec1[1], 5);
75  }
76
77  TEST(SmallVectorTest, SwapTestBufferAndData) {
78    svector::SmallVector<int, 2> vec2 = {1, 2};
79    svector::SmallVector<int, 2> vec1 = {4, 5};
80    vec1.push_back(6);
81
82    const std::size_t size1 = vec1.size();
83    const std::size_t size2 = vec2.size();
84    const std::size_t capacity1 = vec1.capacity();
85    const std::size_t capacity2 = vec2.capacity();
86
87    vec1.swap(vec2);
88
89    EXPECT_EQ(vec1.size(), size2);
90    EXPECT_EQ(vec1.capacity(), capacity2);
91    EXPECT_EQ(vec2.size(), size1);
92    EXPECT_EQ(vec2.capacity(), capacity1);
93    EXPECT_EQ(vec2[0], 4);
94    EXPECT_EQ(vec2[1], 5);
95    EXPECT_EQ(vec2[2], 6);
96    EXPECT_EQ(vec1[0], 1);
97    EXPECT_EQ(vec1[1], 2);
```

```
 98 }
 99
100 TEST(SmallVectorTest, Data) {
101   svector::SmallVector<int, 3> vec;
102   vec.push_back(1);
103   vec.push_back(2);
104   vec.push_back(3);
105
106   int* data_ptr = vec.data();
107
108   EXPECT_EQ(*data_ptr, 1);
109   EXPECT_EQ(*(data_ptr + 1), 2);
110   EXPECT_EQ(*(data_ptr + 2), 3);
111
112   vec.push_back(4);
113   vec.push_back(5);
114
115   EXPECT_EQ(*data_ptr, 1);
116   EXPECT_EQ(*(data_ptr + 1), 2);
117   EXPECT_EQ(*(data_ptr + 2), 3);
118
119   data_ptr = vec.data();
120
121   EXPECT_EQ(*data_ptr, 1);
122   EXPECT_EQ(*(data_ptr + 1), 2);
123   EXPECT_EQ(*(data_ptr + 2), 3);
124   EXPECT_EQ(*(data_ptr + 3), 4);
125   EXPECT_EQ(*(data_ptr + 4), 5);
126 }
127
128 TEST(SmallVectorTest, Buffer) {
129   svector::SmallVector<int, 3> vec;
130   int* buffer_ptr = vec.buffer();
131
132   ASSERT_NE(buffer_ptr, nullptr);
133
134   vec.push_back(1);
135   vec.push_back(2);
136   vec.push_back(3);
137
138   EXPECT_EQ(*buffer_ptr, 1);
139   EXPECT_EQ(*(buffer_ptr + 1), 2);
140   EXPECT_EQ(*(buffer_ptr + 2), 3);
141 }
142
143 TEST(SmallVectorTest, Empty) {
144   svector::SmallVector<int, 5> vec;
145   EXPECT_TRUE(vec.empty());
146
147   vec.push_back(42);
148   EXPECT_FALSE(vec.empty());
149
150   const auto it = vec.cbegin();
151
152   vec.erase(it);
153
154   EXPECT_TRUE(vec.empty());
```

```
155 }
156
157 TEST(SmallVectorTest, At) {
158   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
159
160   for (std::size_t i = 0; i < vec.size(); ++i) {
161     EXPECT_EQ(vec.at(i), i + 1);
162   }
163
164   vec.push_back(6);
165
166   for (std::size_t i = 0; i < vec.size(); ++i) {
167     EXPECT_EQ(vec.at(i), i + 1);
168   }
169 }
170
171 TEST(SmallVectorTest, Clear) {
172   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
173
174   vec.clear();
175   EXPECT_TRUE(vec.empty());
176   EXPECT_EQ(vec.data(), nullptr);
177
178   svector::SmallVector<int, 3> vec1;
179
180   vec1.push_back(6);
181   vec1.push_back(7);
182   vec1.push_back(8);
183   vec1.push_back(9);
184   vec1.push_back(10);
185
186   EXPECT_EQ(vec1.size(), 5);
187   EXPECT_EQ(vec1.capacity(), 6);
188
189   vec1.push_back(11);
190
191   vec1.clear();
192   EXPECT_TRUE(vec1.empty());
193   EXPECT_EQ(vec1.data(), nullptr);
194 }
195
196 TEST(SmallVectorTest, ClearString) {
197   svector::SmallVector<std::string, 5> vec = {
198       "aaaaaaaaaaaaaaaaaaaa",
199       "bbbbbbbbbbbbbbbbbbbb",
200       "ccccccccccccccccccc",
201       "ddddddddddddddddddddd",
202       "eeeeeeeeeeeeeeeeeeee"};
203
204   vec.clear();
205   EXPECT_TRUE(vec.empty());
206   EXPECT_EQ(vec.data(), nullptr);
207
208   svector::SmallVector<std::string, 3> vec1;
209
210   vec1.push_back("aaaaaaaaaaaaaaaaaaaa");
211   vec1.push_back("bbbbbbbbbbbbbbbbbbbb");
```

```
212   vec1.push_back("ccccccccccccccccccc");
213   vec1.push_back("ddddddddddddddddddddd");
214   vec1.push_back("eeeeeeeeeeeeeeeeeeee");
215
216   EXPECT_EQ(vec1.size(), 5);
217   EXPECT_EQ(vec1.capacity(), 6);
218
219   vec1.push_back("fffffffffffffffffff");
220
221   vec1.clear();
222   EXPECT_TRUE(vec1.empty());
223   EXPECT_EQ(vec1.data(), nullptr);
224 }
225
226 TEST(SmallVectorTest, Size) {
227   svector::SmallVector<int, 5> vec;
228
229   EXPECT_EQ(vec.size(), 0);
230
231   vec.push_back(1);
232
233   EXPECT_EQ(vec.size(), 1);
234
235   const svector::SmallVector<int, 3> vec_i = {1, 2};
236
237   EXPECT_EQ(vec_i.size(), 2);
238 }
239
240 TEST(SmallVectorTest, Capacity) {
241   svector::SmallVector<int, 3> vec = {1, 2};
242
243   EXPECT_EQ(vec.capacity(), 3);
244
245   vec.push_back(1);
246
247   EXPECT_EQ(vec.capacity(), 3);
248
249   vec.push_back(1);
250
251   EXPECT_EQ(vec.capacity(), 6);
252 }
253
254 TEST(SmallVectorTest, BeginEndVector) {
255   svector::SmallVector<int, 5> vec;
256
257   EXPECT_EQ(vec.begin(), vec.end());
258
259   vec.push_back(1);
260
261   EXPECT_NE(vec.begin(), vec.end());
262   EXPECT_EQ(*vec.begin(), 1);
263   EXPECT_EQ(vec.begin() + 1, vec.end());
264 }
265
266 TEST(SmallVectorTest, CBeginCEndVector) {
267   svector::SmallVector<int, 5> vec;
268
```

```
269   EXPECT_EQ(vec.cbegin(), vec.cend());
270
271   vec.push_back(1);
272
273   EXPECT_NE(vec.cbegin(), vec.cend());
274   EXPECT_EQ(*vec.cbegin(), 1);
275   EXPECT_EQ(vec.cbegin() + 1, vec.cend());
276 }
277
278 TEST(SmallVectorTest, RBeginREndVector) {
279   svector::SmallVector<int, 5> vec;
280
281   EXPECT_EQ(vec.rbegin(), vec.rend());
282
283   vec.push_back(1);
284
285   EXPECT_NE(vec.rbegin(), vec.rend());
286   EXPECT_EQ(*vec.rbegin(), 1);
287
288   vec.push_back(2);
289   EXPECT_EQ(*vec.rbegin(), 2);
290
291   EXPECT_EQ(vec.rbegin() + 2, vec.rend());
292 }
293
294 TEST(SmallVectorTest, CRBeginCREndVector) {
295   svector::SmallVector<int, 5> vec;
296
297   EXPECT_EQ(vec.crbegin(), vec.crend());
298
299   vec.push_back(1);
300
301   EXPECT_NE(vec.crbegin(), vec.crend());
302   EXPECT_EQ(*vec.crbegin(), 1);
303
304   vec.push_back(2);
305   EXPECT_EQ(*vec.crbegin(), 2);
306
307   EXPECT_EQ(vec.crbegin() + 2, vec.crend());
308 }
309
310 int main(int argc, char** argv) {
311   ::testing::InitGoogleTest(&argc, argv);
312   return RUN_ALL_TESTS();
313 }
```

// file libsvector/libsvector/smallvector/constructor_operator_test.cpp

```
1  #include <gtest/gtest.h>
2
3  #include <libsvector/svector/svector.hpp>
4
5  TEST(SmallVectorTest, CopyConstructorInt) {
6    svector::SmallVector<int, 4> original;
7    original.push_back(1);
8    original.push_back(2);
```

```
  original.push_back(3);

  svector::SmallVector<int, 4> copy1(original);

  EXPECT_EQ(copy1.size(), original.size());
  for (std::size_t i = 0; i < original.size(); ++i) {
    EXPECT_EQ(copy1[i], original[i]);
  }

  original.push_back(4);
  original.push_back(5);
  original.push_back(6);

  svector::SmallVector<int, 4> copy2(original);

  EXPECT_EQ(copy2.size(), original.size());
  for (std::size_t i = 0; i < original.size(); ++i) {
    EXPECT_EQ(copy2[i], original[i]);
  }
}

TEST(SmallVectorTest, CopyConstructorStringMore16) {
  svector::SmallVector<std::string, 4> original;
  original.push_back("aaaaaaaaaaaaaaaaaaaa");
  original.push_back("bbbbbbbbbbbbbbbbbbbb");
  original.push_back("cccccccccccccccccccc");

  svector::SmallVector<std::string, 4> copy1(original);

  EXPECT_EQ(copy1.size(), original.size());
  for (std::size_t i = 0; i < original.size(); ++i) {
    EXPECT_EQ(copy1[i], original[i]);
  }

  original.push_back("ddddddddddddddddddddd");
  original.push_back("fffffffffffffffffff");

  svector::SmallVector<std::string, 4> copy2(original);

  EXPECT_EQ(copy2.size(), original.size());
  for (std::size_t i = 0; i < original.size(); ++i) {
    EXPECT_EQ(copy2[i], original[i]);
  }
}

TEST(SmallVectorTest, InitializerListConstructorTestInt) {
  svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};

  EXPECT_EQ(vec.size(), 5);

  EXPECT_EQ(vec[0], 1);
  EXPECT_EQ(vec[1], 2);
  EXPECT_EQ(vec[2], 3);
  EXPECT_EQ(vec[3], 4);
  EXPECT_EQ(vec[4], 5);
}

```

```
 66 TEST(SmallVectorTest, InitializerListConstructorTestStringMore16) {
 67   svector::SmallVector<std::string, 5> vec = {
 68       "aaaaaaaaaaaaaaaaaaa",
 69       "bbbbbbbbbbbbbbbbbbb",
 70       "ccccccccccccccccccc",
 71       "eeeeeeeeeeeeeeeeeee",
 72       "fffffffffffffffffff"};
 73
 74   EXPECT_EQ(vec.size(), 5);
 75
 76   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaa");
 77   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbb");
 78   EXPECT_EQ(vec[2], "ccccccccccccccccccc");
 79   EXPECT_EQ(vec[3], "eeeeeeeeeeeeeeeeeee");
 80   EXPECT_EQ(vec[4], "fffffffffffffffffff");
 81 }
 82
 83 TEST(SmallVectorTest, IteratorRangeConstructorTestInt) {
 84   std::vector<int> input = {1, 2, 3, 4, 5};
 85
 86   svector::SmallVector<int, 5> vec(input.begin(), input.end());
 87
 88   EXPECT_EQ(vec.size(), input.size());
 89
 90   for (size_t i = 0; i < input.size(); ++i) {
 91     EXPECT_EQ(vec[i], input[i]);
 92   }
 93 }
 94
 95 TEST(SmallVectorTest, IteratorRangeConstructorTestStringMore16) {
 96   std::vector<std::string> input = {
 97       "aaaaaaaaaaaaaaaaaaa",
 98       "bbbbbbbbbbbbbbbbbbb",
 99       "ccccccccccccccccccc",
100       "eeeeeeeeeeeeeeeeeee",
101       "fffffffffffffffffff"};
102
103   svector::SmallVector<std::string, 5> vec(input.begin(), input.end());
104
105   EXPECT_EQ(vec.size(), input.size());
106
107   for (size_t i = 0; i < input.size(); ++i) {
108     EXPECT_EQ(vec[i], input[i]);
109   }
110 }
111
112 TEST(SmallVectorTest, MoveConstructorStaticBufferInt) {
113   svector::SmallVector<int, 5> vec;
114   vec.push_back(1);
115   vec.push_back(2);
116   vec.push_back(3);
117
118   auto old_size = vec.size();
119   auto old_capacity = vec.capacity();
120   int* old_data = vec.data();
121
122   svector::SmallVector<int, 5> moved_vec(std::move(vec));
```

```
123
124   EXPECT_EQ(moved_vec.size(), old_size);
125   EXPECT_EQ(moved_vec.capacity(), old_capacity);
126   EXPECT_NE(moved_vec.data(), old_data);
127
128   EXPECT_EQ(moved_vec[0], 1);
129   EXPECT_EQ(moved_vec[1], 2);
130   EXPECT_EQ(moved_vec[2], 3);
131 }
132
133 TEST(SmallVectorTest, MoveConstructorStaticBufferStringMore16) {
134   svector::SmallVector<std::string, 5> vec;
135   vec.push_back("aaaaaaaaaaaaaaaaaaaa");
136   vec.push_back("bbbbbbbbbbbbbbbbbbbb");
137   vec.push_back("cccccccccccccccccccc");
138
139   auto old_size = vec.size();
140   auto old_capacity = vec.capacity();
141   std::string* old_data = vec.data();
142
143   svector::SmallVector<std::string, 5> moved_vec(std::move(vec));
144
145   EXPECT_EQ(moved_vec.size(), old_size);
146   EXPECT_EQ(moved_vec.capacity(), old_capacity);
147   EXPECT_NE(moved_vec.data(), old_data);
148
149   EXPECT_EQ(moved_vec[0], "aaaaaaaaaaaaaaaaaaaa");
150   EXPECT_EQ(moved_vec[1], "bbbbbbbbbbbbbbbbbbbb");
151   EXPECT_EQ(moved_vec[2], "cccccccccccccccccccc");
152 }
153
154 TEST(SmallVectorTest, MoveConstructorDynamicBufferInt) {
155   svector::SmallVector<int, 3> vec;
156   vec.push_back(1);
157   vec.push_back(2);
158   vec.push_back(3);
159   vec.push_back(4);
160   vec.push_back(5);
161
162   auto old_size = vec.size();
163   auto old_capacity = vec.capacity();
164   int* old_data = vec.data();
165
166   svector::SmallVector<int, 3> moved_vec(std::move(vec));
167
168   EXPECT_EQ(moved_vec.size(), old_size);
169   EXPECT_EQ(moved_vec.capacity(), old_capacity);
170   EXPECT_EQ(moved_vec.data(), old_data);
171
172   EXPECT_EQ(moved_vec[0], 1);
173   EXPECT_EQ(moved_vec[1], 2);
174   EXPECT_EQ(moved_vec[2], 3);
175   EXPECT_EQ(moved_vec[3], 4);
176   EXPECT_EQ(moved_vec[4], 5);
177 }
178
179 TEST(SmallVectorTest, MoveConstructorDynamicBufferStringMore16) {
```

```
180   svector::SmallVector<std::string, 3> vec;
181   vec.push_back("aaaaaaaaaaaaaaaaaaaa");
182   vec.push_back("bbbbbbbbbbbbbbbbbbbb");
183   vec.push_back("cccccccccccccccccccc");
184   vec.push_back("ddddddddddddddddddddd");
185   vec.push_back("eeeeeeeeeeeeeeeeeeee");
186
187   auto old_size = vec.size();
188   auto old_capacity = vec.capacity();
189   std::string* old_data = vec.data();
190
191   svector::SmallVector<std::string, 3> moved_vec(std::move(vec));
192
193   EXPECT_EQ(moved_vec.size(), old_size);
194   EXPECT_EQ(moved_vec.capacity(), old_capacity);
195   EXPECT_EQ(moved_vec.data(), old_data);
196
197   EXPECT_EQ(moved_vec[0], "aaaaaaaaaaaaaaaaaaaa");
198   EXPECT_EQ(moved_vec[1], "bbbbbbbbbbbbbbbbbbbb");
199   EXPECT_EQ(moved_vec[2], "cccccccccccccccccccc");
200   EXPECT_EQ(moved_vec[3], "ddddddddddddddddddddd");
201   EXPECT_EQ(moved_vec[4], "eeeeeeeeeeeeeeeeeeee");
202 }
203
204 TEST(SmallVectorTest, AssignmentOperatorTestInt) {
205   svector::SmallVector<int, 4> vec1 = {1, 2, 3};
206   svector::SmallVector<int, 4> vec2;
207   svector::SmallVector<int, 4> vec3;
208   vec2 = vec1;
209   vec3 = vec1;
210
211   EXPECT_EQ(vec2.size(), vec1.size());
212   EXPECT_EQ(vec2.capacity(), vec1.capacity());
213
214   for (size_t i = 0; i < vec1.size(); ++i) {
215     EXPECT_EQ(vec2[i], vec1[i]);
216   }
217
218   vec1.push_back(4);
219   vec1.push_back(5);
220
221   vec2 = vec1;
222
223   EXPECT_EQ(vec2.size(), vec1.size());
224   EXPECT_EQ(vec2.capacity(), vec1.capacity());
225
226   for (size_t i = 0; i < vec1.size(); ++i) {
227     EXPECT_EQ(vec2[i], vec1[i]);
228   }
229
230   vec2 = vec1;
231
232   EXPECT_EQ(vec2.size(), vec1.size());
233   EXPECT_EQ(vec2.capacity(), vec1.capacity());
234
235   for (size_t i = 0; i < vec1.size(); ++i) {
236     EXPECT_EQ(vec2[i], vec1[i]);
```

```
237   }
238
239   vec2 = vec3;
240
241   EXPECT_EQ(vec2.size(), vec3.size());
242   EXPECT_EQ(vec2.capacity(), vec3.capacity());
243
244   for (size_t i = 0; i < vec3.size(); ++i) {
245     EXPECT_EQ(vec2[i], vec3[i]);
246   }
247 }
248
249 TEST(SmallVectorTest, AssignmentOperatorTestStringMore16) {
250   svector::SmallVector<std::string, 4> vec1 = {
251       "aaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
252 "ccccccccccccccccccc"};
253   svector::SmallVector<std::string, 4> vec2;
254   svector::SmallVector<std::string, 4> vec3;
255   vec2 = vec1;
256   vec3 = vec1;
257
258   EXPECT_EQ(vec2.size(), vec1.size());
259   EXPECT_EQ(vec2.capacity(), vec1.capacity());
260
261   for (size_t i = 0; i < vec1.size(); ++i) {
262     EXPECT_EQ(vec2[i], vec1[i]);
263   }
264
265   vec1.push_back("ddddddddddddddddddddd");
266   vec1.push_back("eeeeeeeeeeeeeeeeeeee");
267
268   vec2 = vec1;
269
270   EXPECT_EQ(vec2.size(), vec1.size());
271   EXPECT_EQ(vec2.capacity(), vec1.capacity());
272
273   for (size_t i = 0; i < vec1.size(); ++i) {
274     EXPECT_EQ(vec2[i], vec1[i]);
275   }
276
277   vec2 = vec1;
278
279   EXPECT_EQ(vec2.size(), vec1.size());
280   EXPECT_EQ(vec2.capacity(), vec1.capacity());
281
282   for (size_t i = 0; i < vec1.size(); ++i) {
283     EXPECT_EQ(vec2[i], vec1[i]);
284   }
285
286   vec2 = vec3;
287
288   EXPECT_EQ(vec2.size(), vec3.size());
289   EXPECT_EQ(vec2.capacity(), vec3.capacity());
290
291   for (size_t i = 0; i < vec3.size(); ++i) {
292     EXPECT_EQ(vec2[i], vec3[i]);
293   }
```

```
294 }
295
296 TEST(SmallVectorTest, MoveAssignmentOperatorTestInt) {
297   svector::SmallVector<int, 5> vec1 = {1, 2, 3};
298
299   svector::SmallVector<int, 5> vec2;
300   vec2 = std::move(vec1);
301
302   EXPECT_EQ(vec2.size(), 3);
303   EXPECT_EQ(vec2.capacity(), 5);
304
305   for (size_t i = 0; i < vec2.size(); ++i) {
306     EXPECT_EQ(vec2[i], i + 1);
307   }
308
309   svector::SmallVector<int, 3> vec3 = {1, 2, 3};
310
311   vec3.push_back(4);
312   vec3.push_back(5);
313
314   svector::SmallVector<int, 3> vec4;
315   vec4 = std::move(vec3);
316
317   EXPECT_EQ(vec4.size(), 5);
318   EXPECT_EQ(vec4.capacity(), 6);
319
320   for (size_t i = 0; i < vec4.size(); ++i) {
321     EXPECT_EQ(vec4[i], i + 1);
322   }
323 }
324
325 TEST(SmallVectorTest, MoveAssignmentOperatorTestStringMore16) {
326   svector::SmallVector<std::string, 5> vec1 = {
327       "aaaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
328 "ccccccccccccccccccc"};
329
330   svector::SmallVector<std::string, 5> rez1 = {
331       "aaaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
332 "ccccccccccccccccccc"};
333
334   svector::SmallVector<std::string, 5> vec2;
335   vec2 = std::move(vec1);
336
337   EXPECT_EQ(vec2.size(), 3);
338   EXPECT_EQ(vec2.capacity(), 5);
339
340   for (size_t i = 0; i < vec2.size(); ++i) {
341     EXPECT_EQ(vec2[i], rez1[i]);
342   }
343
344   svector::SmallVector<std::string, 3> vec3 = {
345       "aaaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
346 "ccccccccccccccccccc"};
347
348   svector::SmallVector<std::string, 5> rez2 = {
349       "aaaaaaaaaaaaaaaaaaaa",
350       "bbbbbbbbbbbbbbbbbbbb",
```

```
351         "ccccccccccccccccccc",
352         "ddddddddddddddddddddd",
353         "eeeeeeeeeeeeeeeeeee"};
354
355   vec3.push_back("ddddddddddddddddddddd");
356   vec3.push_back("eeeeeeeeeeeeeeeeeee");
357
358   svector::SmallVector<std::string, 3> vec4;
359   vec4 = std::move(vec3);
360
361   EXPECT_EQ(vec4.size(), 5);
362   EXPECT_EQ(vec4.capacity(), 6);
363
364   for (size_t i = 0; i < vec4.size(); ++i) {
365     EXPECT_EQ(vec4[i], rez2[i]);
366   }
367 }
368
369 int main(int argc, char** argv) {
370   ::testing::InitGoogleTest(&argc, argv);
371   return RUN_ALL_TESTS();
372 }
```

// file libsvector/libsvector/smallvector/main_methods_test.cpp

```
  1 #include <gtest/gtest.h>
  2
  3 #include <libsvector/svector/svector.hpp>
  4
  5 TEST(SmallVectorTest, PushBackStaticBufferInt) {
  6   svector::SmallVector<int, 3> vec;
  7
  8   EXPECT_TRUE(vec.empty());
  9   EXPECT_EQ(vec.size(), 0);
 10   EXPECT_EQ(vec.capacity(), 3);
 11
 12   vec.push_back(10);
 13   vec.push_back(20);
 14   vec.push_back(30);
 15
 16   EXPECT_FALSE(vec.empty());
 17   EXPECT_EQ(vec.size(), 3);
 18   EXPECT_EQ(vec.capacity(), 3);
 19   EXPECT_EQ(vec[0], 10);
 20   EXPECT_EQ(vec[1], 20);
 21   EXPECT_EQ(vec[2], 30);
 22
 23   EXPECT_EQ(vec.data(), vec.buffer());
 24 }
 25
 26 TEST(SmallVectorTest, PushBackStaticBufferStringMore16) {
 27   svector::SmallVector<std::string, 3> vec;
 28
 29   EXPECT_TRUE(vec.empty());
 30   EXPECT_EQ(vec.size(), 0);
 31   EXPECT_EQ(vec.capacity(), 3);
```

```
32
33    vec.push_back("aaaaaaaaaaaaaaaaaaaa");
34    vec.push_back("bbbbbbbbbbbbbbbbbbbb");
35    vec.push_back("ccccccccccccccccccc");
36
37    EXPECT_FALSE(vec.empty());
38    EXPECT_EQ(vec.size(), 3);
39    EXPECT_EQ(vec.capacity(), 3);
40    EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
41    EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
42    EXPECT_EQ(vec[2], "ccccccccccccccccccc");
43
44    EXPECT_EQ(vec.data(), vec.buffer());
45  }
46
47  TEST(SmallVectorTest, PushBackDynamicBufferInt) {
48    svector::SmallVector<int, 2> vec;
49
50    vec.push_back(10);
51    vec.push_back(20);
52    vec.push_back(30);
53    vec.push_back(40);
54    vec.push_back(50);
55    vec.push_back(60);
56
57    EXPECT_FALSE(vec.empty());
58    EXPECT_EQ(vec.size(), 6);
59    EXPECT_EQ(vec.capacity(), 8);
60    EXPECT_EQ(vec[0], 10);
61    EXPECT_EQ(vec[1], 20);
62    EXPECT_EQ(vec[2], 30);
63    EXPECT_EQ(vec[3], 40);
64    EXPECT_EQ(vec[4], 50);
65    EXPECT_EQ(vec[5], 60);
66
67    EXPECT_NE(vec.data(), vec.buffer());
68  }
69
70  TEST(SmallVectorTest, PushBackDynamicBufferStringMore16) {
71    svector::SmallVector<std::string, 2> vec;
72
73    vec.push_back("aaaaaaaaaaaaaaaaaaaa");
74    vec.push_back("bbbbbbbbbbbbbbbbbbbb");
75    vec.push_back("ccccccccccccccccccc");
76    vec.push_back("ddddddddddddddddddddd");
77    vec.push_back("eeeeeeeeeeeeeeeeeeee");
78    vec.push_back("fffffffffffffffffff");
79
80    EXPECT_FALSE(vec.empty());
81    EXPECT_EQ(vec.size(), 6);
82    EXPECT_EQ(vec.capacity(), 8);
83    EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
84    EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
85    EXPECT_EQ(vec[2], "ccccccccccccccccccc");
86    EXPECT_EQ(vec[3], "ddddddddddddddddddddd");
87    EXPECT_EQ(vec[4], "eeeeeeeeeeeeeeeeeeee");
88    EXPECT_EQ(vec[5], "fffffffffffffffffff");
```

```
 89
 90    EXPECT_NE(vec.data(), vec.buffer());
 91  }
 92
 93  TEST(SmallVectorTest, ResizeInt) {
 94    svector::SmallVector<int, 5> vec = {1, 2, 3};
 95
 96    vec.resize(3);
 97
 98    EXPECT_EQ(vec.size(), 3);
 99    EXPECT_EQ(vec.capacity(), 5);
100
101    vec.resize(2);
102
103    EXPECT_EQ(vec.size(), 2);
104    EXPECT_EQ(vec.capacity(), 5);
105
106    vec.resize(4);
107
108    EXPECT_EQ(vec.size(), 4);
109    EXPECT_EQ(vec.capacity(), 5);
110
111    vec.resize(7);
112
113    EXPECT_EQ(vec.size(), 7);
114    EXPECT_EQ(vec.capacity(), 7);
115
116    EXPECT_EQ(vec[0], 1);
117    EXPECT_EQ(vec[1], 2);
118
119    EXPECT_NE(vec.data(), vec.buffer());
120  }
121
122  TEST(SmallVectorTest, ResizeStringMore16) {
123    svector::SmallVector<std::string, 5> vec = {
124        "aaaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
125  "ccccccccccccccccccc"};
126
127    vec.resize(3);
128
129    EXPECT_EQ(vec.size(), 3);
130    EXPECT_EQ(vec.capacity(), 5);
131
132    vec.resize(2);
133
134    EXPECT_EQ(vec.size(), 2);
135    EXPECT_EQ(vec.capacity(), 5);
136
137    vec.resize(4);
138
139    EXPECT_EQ(vec.size(), 4);
140    EXPECT_EQ(vec.capacity(), 5);
141
142    vec.resize(7);
143
144    EXPECT_EQ(vec.size(), 7);
145    EXPECT_EQ(vec.capacity(), 7);
```

```
146
147   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
148   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
149
150   EXPECT_NE(vec.data(), vec.buffer());
151 }
152
153 TEST(SmallVectorTest, ReserveInt) {
154   svector::SmallVector<int, 5> vec = {1, 2, 3};
155
156   vec.reserve(4);
157   vec.reserve(5);
158
159   EXPECT_EQ(vec.size(), 3);
160   EXPECT_EQ(vec.capacity(), 5);
161
162   vec.reserve(7);
163
164   EXPECT_EQ(vec.size(), 3);
165   EXPECT_EQ(vec.capacity(), 7);
166
167   EXPECT_EQ(vec[0], 1);
168   EXPECT_EQ(vec[1], 2);
169   EXPECT_EQ(vec[2], 3);
170
171   EXPECT_NE(vec.data(), vec.buffer());
172 }
173
174 TEST(SmallVectorTest, ReserveStringMore16) {
175   svector::SmallVector<std::string, 5> vec = {
176       "aaaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbbb",
177 "ccccccccccccccccccc"};
178
179   vec.reserve(4);
180   vec.reserve(5);
181
182   EXPECT_EQ(vec.size(), 3);
183   EXPECT_EQ(vec.capacity(), 5);
184
185   vec.reserve(7);
186
187   EXPECT_EQ(vec.size(), 3);
188   EXPECT_EQ(vec.capacity(), 7);
189
190   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
191   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
192   EXPECT_EQ(vec[2], "ccccccccccccccccccc");
193
194   EXPECT_NE(vec.data(), vec.buffer());
195 }
196
197 TEST(SmallVectorTest, ShrinkToFitInt) {
198   svector::SmallVector<int, 3> vec = {1, 2, 3};
199
200   vec.shrink_to_fit();
201
202   EXPECT_EQ(vec.size(), 3);
```

```
203   EXPECT_EQ(vec.capacity(), 3);
204
205   vec.push_back(4);
206   vec.push_back(5);
207
208   vec.reserve(7);
209
210   EXPECT_EQ(vec[0], 1);
211   EXPECT_EQ(vec[1], 2);
212   EXPECT_EQ(vec[2], 3);
213   EXPECT_EQ(vec[3], 4);
214   EXPECT_EQ(vec[4], 5);
215
216   EXPECT_EQ(vec.size(), 5);
217   EXPECT_EQ(vec.capacity(), 7);
218
219   vec.shrink_to_fit();
220
221   EXPECT_EQ(vec.size(), 5);
222   EXPECT_EQ(vec.capacity(), 5);
223
224   EXPECT_NE(vec.data(), vec.buffer());
225 }
226
227 TEST(SmallVectorTest, ShrinkToFitStringMore16) {
228   svector::SmallVector<std::string, 3> vec = {
229       "aaaaaaaaaaaaaaaaaaa", "bbbbbbbbbbbbbbbbbbb",
230 "ccccccccccccccccccc"};
231
232   vec.shrink_to_fit();
233
234   EXPECT_EQ(vec.size(), 3);
235   EXPECT_EQ(vec.capacity(), 3);
236
237   vec.push_back("ddddddddddddddddddddd");
238   vec.push_back("eeeeeeeeeeeeeeeeeeee");
239
240   vec.reserve(7);
241
242   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaa");
243   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbb");
244   EXPECT_EQ(vec[2], "ccccccccccccccccccc");
245   EXPECT_EQ(vec[3], "ddddddddddddddddddddd");
246   EXPECT_EQ(vec[4], "eeeeeeeeeeeeeeeeeeee");
247
248   EXPECT_EQ(vec.size(), 5);
249   EXPECT_EQ(vec.capacity(), 7);
250
251   vec.shrink_to_fit();
252
253   EXPECT_EQ(vec.size(), 5);
254   EXPECT_EQ(vec.capacity(), 5);
255
256   EXPECT_NE(vec.data(), vec.buffer());
257 }
258
259 TEST(SmallVectorTest, EraseInt) {
```

```
260   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
261
262   auto it = vec.cbegin() + 1;
263   vec.erase(it);
264
265   EXPECT_EQ(vec.size(), 4);
266
267   EXPECT_EQ(vec[0], 1);
268   EXPECT_EQ(vec[1], 3);
269   EXPECT_EQ(vec[2], 4);
270   EXPECT_EQ(vec[3], 5);
271 }
272
273 TEST(SmallVectorTest, EraseStringMore16) {
274   svector::SmallVector<std::string, 5> vec = {
275       "aaaaaaaaaaaaaaaaaaaa",
276       "bbbbbbbbbbbbbbbbbbbb",
277       "ccccccccccccccccccc",
278       "ddddddddddddddddddddd",
279       "eeeeeeeeeeeeeeeeeeee"};
280
281   auto it = vec.cbegin() + 1;
282   vec.erase(it);
283
284   EXPECT_EQ(vec.size(), 4);
285
286   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
287   EXPECT_EQ(vec[1], "ccccccccccccccccccc");
288   EXPECT_EQ(vec[2], "ddddddddddddddddddddd");
289   EXPECT_EQ(vec[3], "eeeeeeeeeeeeeeeeeeee");
290 }
291
292 TEST(SmallVectorTest, EraseRangeInt) {
293   svector::SmallVector<int, 5> vec = {1, 2, 3, 4, 5};
294
295   auto first = vec.cbegin() + 1;
296   auto last = vec.cbegin() + 4;
297   vec.erase(first, last);
298
299   EXPECT_EQ(vec.size(), 2);
300
301   EXPECT_EQ(vec[0], 1);
302   EXPECT_EQ(vec[1], 5);
303 }
304
305 TEST(SmallVectorTest, EraseRangeStringMore16) {
306   svector::SmallVector<std::string, 5> vec = {
307       "aaaaaaaaaaaaaaaaaaaa",
308       "bbbbbbbbbbbbbbbbbbbb",
309       "ccccccccccccccccccc",
310       "ddddddddddddddddddddd",
311       "eeeeeeeeeeeeeeeeeeee"};
312
313   auto first = vec.cbegin() + 1;
314   auto last = vec.cbegin() + 4;
315   vec.erase(first, last);
316
```

```
317   EXPECT_EQ(vec.size(), 2);
318
319   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaa");
320   EXPECT_EQ(vec[1], "eeeeeeeeeeeeeeeeeee");
321 }
322
323 TEST(SmallVectorTest, InsertTestInt) {
324   svector::SmallVector<int, 5> vec;
325   vec.push_back(1);
326   vec.push_back(2);
327   vec.push_back(3);
328   vec.push_back(4);
329
330   auto it = vec.insert(vec.cbegin() + 2, 100);
331
332   EXPECT_EQ(vec.size(), 5);
333   EXPECT_EQ(vec.capacity(), 5);
334
335   EXPECT_EQ(*it, 100);
336
337   EXPECT_EQ(vec[0], 1);
338   EXPECT_EQ(vec[1], 2);
339   EXPECT_EQ(vec[2], 100);
340   EXPECT_EQ(vec[3], 3);
341   EXPECT_EQ(vec[4], 4);
342
343   it = vec.insert(vec.cbegin() + 3, 200);
344
345   EXPECT_EQ(vec.size(), 6);
346   EXPECT_EQ(vec.capacity(), 12);
347
348   EXPECT_EQ(*it, 200);
349
350   EXPECT_EQ(vec[0], 1);
351   EXPECT_EQ(vec[1], 2);
352   EXPECT_EQ(vec[2], 100);
353   EXPECT_EQ(vec[3], 200);
354   EXPECT_EQ(vec[4], 3);
355   EXPECT_EQ(vec[5], 4);
356 }
357
358 TEST(SmallVectorTest, InsertTestStringMore16) {
359   svector::SmallVector<std::string, 5> vec;
360   vec.push_back("aaaaaaaaaaaaaaaaaaa");
361   vec.push_back("bbbbbbbbbbbbbbbbbbb");
362   vec.push_back("ccccccccccccccccccc");
363   vec.push_back("ddddddddddddddddddddd");
364
365   auto it = vec.insert(vec.cbegin() + 2, "eeeeeeeeeeeeeeeeeee");
366
367   EXPECT_EQ(vec.size(), 5);
368   EXPECT_EQ(vec.capacity(), 5);
369
370   EXPECT_EQ(*it, "eeeeeeeeeeeeeeeeeee");
371
372   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaa");
373   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbb");
```

```
374   EXPECT_EQ(vec[2], "eeeeeeeeeeeeeeeeeee");
375   EXPECT_EQ(vec[3], "ccccccccccccccccccc");
376   EXPECT_EQ(vec[4], "ddddddddddddddddddddd");
377
378   it = vec.insert(vec.cbegin() + 3, "fffffffffffffffffff");
379
380   EXPECT_EQ(vec.size(), 6);
381   EXPECT_EQ(vec.capacity(), 12);
382
383   EXPECT_EQ(*it, "fffffffffffffffffff");
384
385   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaa");
386   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbb");
387   EXPECT_EQ(vec[2], "eeeeeeeeeeeeeeeeeee");
388   EXPECT_EQ(vec[3], "fffffffffffffffffff");
389   EXPECT_EQ(vec[4], "ccccccccccccccccccc");
390   EXPECT_EQ(vec[5], "ddddddddddddddddddddd");
391 }
392
393 TEST(SmallVectorTest, InsertStaticBufferRangeInt) {
394   svector::SmallVector<int, 9> vec;
395   vec.push_back(1);
396   vec.push_back(2);
397   vec.push_back(3);
398   vec.push_back(4);
399
400   const std::vector<int> to_insert = {100, 200, 300};
401
402   vec.insert(vec.cbegin() + 2, to_insert.cbegin(), to_insert.cend());
403
404   EXPECT_EQ(vec.size(), 7);
405   EXPECT_EQ(vec.capacity(), 9);
406
407   EXPECT_EQ(vec[0], 1);
408   EXPECT_EQ(vec[1], 2);
409   EXPECT_EQ(vec[2], 100);
410   EXPECT_EQ(vec[3], 200);
411   EXPECT_EQ(vec[4], 300);
412   EXPECT_EQ(vec[5], 3);
413   EXPECT_EQ(vec[6], 4);
414 }
415
416 TEST(SmallVectorTest, InsertStaticBufferRangeStringMore16) {
417   svector::SmallVector<std::string, 9> vec;
418   vec.push_back("aaaaaaaaaaaaaaaaaaa");
419   vec.push_back("bbbbbbbbbbbbbbbbbbb");
420   vec.push_back("ccccccccccccccccccc");
421   vec.push_back("ddddddddddddddddddddd");
422
423   const std::vector<std::string> to_insert = {
424       "11111111111111111111", "22222222222222222222",
425 "33333333333333333333"};
426
427   vec.insert(vec.cbegin() + 2, to_insert.cbegin(), to_insert.cend());
428
429   EXPECT_EQ(vec.size(), 7);
430   EXPECT_EQ(vec.capacity(), 9);
```

```
431
432   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
433   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
434   EXPECT_EQ(vec[2], "11111111111111111111");
435   EXPECT_EQ(vec[3], "22222222222222222222");
436   EXPECT_EQ(vec[4], "33333333333333333333");
437   EXPECT_EQ(vec[5], "ccccccccccccccccccccc");
438   EXPECT_EQ(vec[6], "ddddddddddddddddddddd");
439 }
440
441 TEST(SmallVectorTest, InsertDynamicBufferRangeInt) {
442   svector::SmallVector<int, 5> vec;
443   vec.push_back(1);
444   vec.push_back(2);
445   vec.push_back(3);
446   vec.push_back(4);
447
448   const std::vector<int> to_insert = {100, 200, 300};
449
450   vec.insert(vec.cbegin() + 2, to_insert.cbegin(), to_insert.cend());
451
452   EXPECT_EQ(vec.size(), 7);
453   EXPECT_EQ(vec.capacity(), 7);
454
455   EXPECT_EQ(vec[0], 1);
456   EXPECT_EQ(vec[1], 2);
457   EXPECT_EQ(vec[2], 100);
458   EXPECT_EQ(vec[3], 200);
459   EXPECT_EQ(vec[4], 300);
460   EXPECT_EQ(vec[5], 3);
461   EXPECT_EQ(vec[6], 4);
462 }
463
464 TEST(SmallVectorTest, InsertDynamicBufferRangeStringMore16) {
465   svector::SmallVector<std::string, 5> vec;
466   vec.push_back("aaaaaaaaaaaaaaaaaaaa");
467   vec.push_back("bbbbbbbbbbbbbbbbbbbb");
468   vec.push_back("ccccccccccccccccccccc");
469   vec.push_back("ddddddddddddddddddddd");
470
471   const std::vector<std::string> to_insert = {
472       "11111111111111111111", "22222222222222222222",
473 "33333333333333333333"};
474
475   vec.insert(vec.cbegin() + 2, to_insert.cbegin(), to_insert.cend());
476
477   EXPECT_EQ(vec.size(), 7);
478   EXPECT_EQ(vec.capacity(), 7);
479
480   EXPECT_EQ(vec[0], "aaaaaaaaaaaaaaaaaaaa");
481   EXPECT_EQ(vec[1], "bbbbbbbbbbbbbbbbbbbb");
482   EXPECT_EQ(vec[2], "11111111111111111111");
483   EXPECT_EQ(vec[3], "22222222222222222222");
484   EXPECT_EQ(vec[4], "33333333333333333333");
485   EXPECT_EQ(vec[5], "ccccccccccccccccccccc");
486   EXPECT_EQ(vec[6], "ddddddddddddddddddddd");
487 }
```

```
488
489 int main(int argc, char** argv) {
490   ::testing::InitGoogleTest(&argc, argv);
491   return RUN_ALL_TESTS();
492 }
```

// file libsvector/libsvector/std_methods/std_iterator_test.cpp

```
1  #include <gtest/gtest.h>
2
3  #include <libsvector/svector/svector.hpp>
4
5  #include <algorithm>
6
7  #include <numeric>
8
9  TEST(SVectorTest, Sort) {
10   svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};
11
12   std::vector<int> rez = {1, 2, 3, 4, 5};
13
14   std::sort(vec.begin(), vec.end());
15
16   for (std::size_t i = 0; i < vec.size(); ++i) {
17     EXPECT_EQ(vec.at(i), rez[i]);
18   }
19 }
20
21 TEST(SVectorTest, Reverse) {
22   svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};
23
24   std::vector<int> rez = {3, 4, 5, 1, 2};
25
26   std::reverse(vec.begin(), vec.end());
27
28   for (std::size_t i = 0; i < vec.size(); ++i) {
29     EXPECT_EQ(vec.at(i), rez[i]);
30   }
31 }
32
33 TEST(SVectorTest, Accumulate) {
34   svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};
35
36   auto sum = std::accumulate(vec.begin(), vec.end(), 0);
37
38   ASSERT_EQ(sum, 15);
39 }
40
41 TEST(SVectorTest, Merge) {
42   svector::SmallVector<int, 5> vec1{1, 3, 5, 7, 9};
43   svector::SmallVector<int, 5> vec2{2, 4, 6, 8, 10};
44   svector::SmallVector<int, 10> merged;
45
46   std::merge(
47       vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), merged.begin());
48
```

```
  std::vector<int> expected = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  for (std::size_t i = 0; i < merged.size(); ++i) {
    EXPECT_EQ(merged.at(i), expected[i]);
  }
}

TEST(SVectorTest, MinMaxElement) {
  svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto min_it = std::min_element(vec.begin(), vec.end());
  auto max_it = std::max_element(vec.begin(), vec.end());

  EXPECT_EQ(*min_it, 1);
  EXPECT_EQ(*max_it, 5);
}

TEST(SVectorTest, BinarySearch) {
  svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};

  for (int i = 1; i <= 5; ++i) {
    EXPECT_TRUE(std::binary_search(vec.begin(), vec.end(), i));
  }

  EXPECT_FALSE(std::binary_search(vec.begin(), vec.end(), 0));
  EXPECT_FALSE(std::binary_search(vec.begin(), vec.end(), 6));
  EXPECT_FALSE(std::binary_search(vec.begin(), vec.end(), 100));
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

// file libsvector/libsvector/std_methods/std_const_iterator_test.cpp

```
#include <gtest/gtest.h>

#include <libsvector/svector/svector.hpp>

#include <algorithm>

#include <numeric>

TEST(SVectorTest, Accumulate) {
  const svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto sum = std::accumulate(vec.cbegin(), vec.cend(), 0);

  ASSERT_EQ(sum, 15);
}

TEST(SVectorTest, MinMaxElement) {
  const svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto min_it = std::min_element(vec.cbegin(), vec.cend());
  auto max_it = std::max_element(vec.cbegin(), vec.cend());
```

```
22
23    EXPECT_EQ(*min_it, 1);
24    EXPECT_EQ(*max_it, 5);
25  }
26
27  TEST(SVectorTest, BinarySearch) {
28    const svector::SmallVector<int, 5> vec{1, 2, 3, 4, 5};
29
30    for (int i = 1; i <= 5; ++i) {
31      EXPECT_TRUE(std::binary_search(vec.cbegin(), vec.cend(), i));
32    }
33
34    EXPECT_FALSE(std::binary_search(vec.cbegin(), vec.cend(), 0));
35    EXPECT_FALSE(std::binary_search(vec.cbegin(), vec.cend(), 6));
36    EXPECT_FALSE(std::binary_search(vec.cbegin(), vec.cend(), 100));
37  }
38
39  int main(int argc, char** argv) {
40    ::testing::InitGoogleTest(&argc, argv);
41    return RUN_ALL_TESTS();
42  }
```

// file libsvector/libsvector/std_methods/std_rev_iterator_test.cpp

```
1   #include <gtest/gtest.h>
2
3   #include <libsvector/svector/svector.hpp>
4
5   #include <algorithm>
6
7   #include <numeric>
8
9   TEST(SVectorTest, Sort) {
10    svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};
11
12    std::vector<int> rez = {5, 4, 3, 2, 1};
13
14    std::sort(vec.rbegin(), vec.rend());
15
16    for (std::size_t i = 0; i < vec.size(); ++i) {
17      EXPECT_EQ(vec.at(i), rez[i]);
18    }
19  }
20
21  TEST(SVectorTest, Reverse) {
22    svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};
23
24    std::vector<int> rez = {3, 4, 5, 1, 2};
25
26    std::reverse(vec.rbegin(), vec.rend());
27
28    for (std::size_t i = 0; i < vec.size(); ++i) {
29      EXPECT_EQ(vec.at(i), rez[i]);
30    }
31  }
32
```

```cpp
TEST(SVectorTest, Accumulate) {
  svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto sum = std::accumulate(vec.rbegin(), vec.rend(), 0);

  ASSERT_EQ(sum, 15);
}

TEST(SVectorTest, Merge) {
  svector::SmallVector<int, 5> vec1{9, 7, 5, 3, 1};
  svector::SmallVector<int, 5> vec2{10, 8, 6, 4, 2};
  svector::SmallVector<int, 10> merged;

  std::merge(
      vec1.rbegin(), vec1.rend(), vec2.rbegin(), vec2.rend(),
merged.begin());

  std::vector<int> expected = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
  for (std::size_t i = 0; i < merged.size(); ++i) {
    EXPECT_EQ(merged.at(i), expected[i]);
  }
}

TEST(SVectorTest, MinMaxElement) {
  svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto min_it = std::min_element(vec.rbegin(), vec.rend());
  auto max_it = std::max_element(vec.rbegin(), vec.rend());

  EXPECT_EQ(*min_it, 1);
  EXPECT_EQ(*max_it, 5);
}

TEST(SVectorTest, BinarySearchReverse) {
  svector::SmallVector<int, 5> vec{5, 4, 3, 2, 1};

  for (int i = 1; i <= 5; ++i) {
    EXPECT_TRUE(std::binary_search(vec.rbegin(), vec.rend(), i));
  }

  EXPECT_FALSE(std::binary_search(vec.rbegin(), vec.rend(), 0));
  EXPECT_FALSE(std::binary_search(vec.rbegin(), vec.rend(), 6));
  EXPECT_FALSE(std::binary_search(vec.rbegin(), vec.rend(), 100));
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

// file libsvector/libsvector/std_methods/std_const_rev_iterator_test.cpp

```cpp
#include <gtest/gtest.h>

#include <libsvector/svector/svector.hpp>

```

```cpp
#include <algorithm>

#include <numeric>

TEST(SVectorTest, Accumulate) {
  const svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto sum = std::accumulate(vec.crbegin(), vec.crend(), 0);

  ASSERT_EQ(sum, 15);
}

TEST(SVectorTest, MinMaxElement) {
  const svector::SmallVector<int, 5> vec{2, 1, 5, 4, 3};

  auto min_it = std::min_element(vec.crbegin(), vec.crend());
  auto max_it = std::max_element(vec.crbegin(), vec.crend());

  EXPECT_EQ(*min_it, 1);
  EXPECT_EQ(*max_it, 5);
}

TEST(SVectorTest, BinarySearchReverse) {
  const svector::SmallVector<int, 5> vec{5, 4, 3, 2, 1};

  for (int i = 1; i <= 5; ++i) {
    EXPECT_TRUE(std::binary_search(vec.crbegin(), vec.crend(), i));
  }

  EXPECT_FALSE(std::binary_search(vec.crbegin(), vec.crend(), 0));
  EXPECT_FALSE(std::binary_search(vec.crbegin(), vec.crend(), 6));
  EXPECT_FALSE(std::binary_search(vec.crbegin(), vec.crend(), 100));
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```