Double-Ended Queue (Deque) Implementation Documentation

Custom Deque Implementation in C++

May 14, 2025

Contents

1 Introduction

This document provides comprehensive documentation for a custom Double-Ended Queue (Deque) implementation in C++. The deque data structure supports efficient insertion and deletion of elements at both the beginning and the end of the container. The implementation is contained in the namespace `sml`.

2 Implementation Overview

The deque is implemented using a bucket-based approach, where elements are stored in fixed-size buckets. This allows for efficient memory management and constant-time complexity for most operations.

2.1 Key Features

- Efficient O(1) insertion and deletion at both ends
- Memory efficient with bucket-based storage
- Support for standard container operations
- Custom iterator implementation
- Memory-aligned element storage

3 Class Structure

The deque is implemented as a template class to support any data type.

3.1 Deque Class Declaration

```
namespace sml {
template <typename T>
class deque {
    // Implementation details
public:
    // Public interface
};
}
```

4 Internal Implementation

4.1 Bucket Structure

Elements are stored in fixed-size buckets to optimize memory allocation and access.

```
struct Bucket {
    alignas(T) uint8_t data[sizeof(T) * bucketSize];

    Bucket() {
        // Initialize memory to zero
        std::memset(data, 0, sizeof(data));
    }

    ~Bucket() {
        // Bucket doesn't own the objects, it just provides storage
    }
};
```

4.2 Memory Management

The implementation utilizes a dynamic array of buckets, with memory-aligned storage for elements. The bucket size is configurable using the `bucketSize` constant.

```
constexpr size_t bucketSize = 8; // Configurable bucket size
constexpr size_t growFactor = 2; // Growth factor when resizing
```

4.3 Position Tracking

The implementation tracks the position of the first and last elements using a custom structure:

```
struct deqIndx {
    size_t bucket;
    size_t index;
} first_, last_;
```

5 Public Interface

5.1 Constructors and Destructors

- **Default Constructor**: Creates an empty deque with initial capacity.
- **Copy Constructor**: Creates a deep copy of another deque.
- **Move Constructor**: Efficiently transfers ownership from another deque.
- **Destructor**: Properly cleans up all resources.

5.2 Element Access

- `operator[](size_t index)`: Access element at specified position.
- `front()`: Access the first element.
- `back()`: Access the last element.

5.3 Modifiers

- `push_front(const T& value)`: Insert element at the beginning.
- `push_back(const T& value)`: Insert element at the end.
- `pop_front()`: Remove element from the beginning.
- `pop_back()`: Remove element from the end.
- `clear()`: Remove all elements.

5.4 Capacity

- `empty()`: Check if the deque is empty.
- `size()`: Get the number of elements.

6 Implementation Challenges and Solutions

6.1 Memory Alignment Issues

One of the key challenges in implementing a custom deque was ensuring proper memory alignment for elements of different types. This was especially important for complex data types that have alignment requirements.

6.1.1 Problem

Initially, we observed memory corruption and undefined behavior when storing elements in the deque. The issue was traced to improper alignment of memory for elements within buckets.

6.1.2 Solution

We used the `alignas` specifier to ensure proper alignment of memory in the buckets:

```
struct Bucket {
    alignas(T) uint8_t data[sizeof(T) * bucketSize];
    // ...
};
```

This ensures that the memory allocated for elements is properly aligned according to the requirements of type `T`.

6.2 Element Construction and Destruction

Proper management of object lifetimes was another significant challenge.

6.2.1 Problem

The initial implementation directly accessed memory without properly constructing or destroying objects, leading to resource leaks and undefined behavior.

6.2.2 Solution

We implemented proper object construction using placement new and explicit destructor calls:

```
// Better approach - using element_at helper for type safety
T* element_at(size_t bucket, size_t index) {
    return reinterpret_cast<T*>(&buckets_[bucket]->data[index *
        sizeof(T)]);
}

void push_back(const T& value) {
    // ...
    new (element_at(last_.bucket, last_.index)) T(value);
    // ...
}

void pop_back() {
    // ...
    element_at(last_.bucket, last_.index)->~T();
    // ...
}
```

This approach correctly manages object lifetimes and ensures that destructors are called when elements are removed.

6.3 Index Calculation

Correctly calculating indices for element access across multiple buckets was a complex challenge.

6.3.1 Problem

The original implementation had bugs in calculating the correct bucket and index for elements, resulting in incorrect element access and iterator behavior.

6.3.2 Solution

We developed a more robust approach to calculate element positions:

```
T& operator[](size_t idx) {
    if (idx >= size_) throw std::out_of_range("Index out of range")
        ;

    // Calculate which bucket and position this element is in
    size_t relative_pos = idx;
    size_t curr_bucket = first_.bucket;
    size_t curr_index = first_.index;

    // Navigate to the right position
    while (relative_pos > 0) {
        if (curr_index + relative_pos < bucketSize) {
            curr_index += relative_pos;
```

```
            break;
        } else {
            relative_pos -= (bucketSize - curr_index);
            curr_index = 0;
            curr_bucket = (curr_bucket + 1) % capacity_;
        }
    }

    return *element_at(curr_bucket, curr_index);
}
```

This implementation carefully handles boundary conditions when traversing buckets.

7 Debugging Experience

During the development of this deque implementation, several bugs were encountered and fixed:

7.1 Uninitialized Memory Issues

7.1.1 Problem

When iterating through the deque, we observed random values being displayed instead of the expected values.

7.1.2 Root Cause

The memory in the buckets was not properly initialized, leading to garbage values being interpreted as valid elements.

7.1.3 Fix

Added memory initialization in the Bucket constructor:

```
Bucket() {
    // Initialize memory to zero
    std::memset(data, 0, sizeof(data));
}
```

7.2 Iterator Boundary Issues

7.2.1 Problem

Iterators would sometimes access invalid memory after the end of the container.

7.2.2 Root Cause

The iterator's end condition was incorrectly implemented, not properly accounting for the circular buffer structure.

7.2.3 Fix

Rewrote the iterator implementation to use a simpler, more robust approach based on element indices:

```
class iterator {
private:
    deque* container_;
    size_t position_;

public:
    // ...

    bool operator==(const iterator& other) const {
        return position_ == other.position_ ||
               (position_ >= container_->size_ && other.position_ >=
                   container_->size_);
    }
};
```

7.3 Memory Leaks

7.3.1 Problem

Memory profiling revealed leaks when elements were removed from the deque.

7.3.2 Root Cause

Elements' destructors were not being called when they were removed from the deque.

7.3.3 Fix

Implemented proper destructor calls and a clear method to clean up resources:

```
void clear() {
    // Destroy all objects
    while (!empty()) {
        pop_back();
    }
}

~deque() {
    clear();
    for (size_t i = 0; i < capacity_; ++i) {
        delete buckets_[i];
    }
    delete[] buckets_;
}
```

8 Detailed Implementation Examples

8.1 Resizing Strategy

The deque implementation uses a growth strategy that doubles the capacity when the container becomes full:

```
void increase_size() {
    size_t old_cap = capacity_;
    size_t used_buckets = (size_ + bucketSize - 1) / bucketSize;
    capacity_ *= growFactor;
    Bucket** new_buckets = new Bucket*[capacity_]();

    size_t mid = (capacity_ - used_buckets) / 2;

    for (size_t i = 0; i < used_buckets; ++i) {
        size_t old_idx = (first_.bucket + i) % old_cap;
        new_buckets[mid + i] = buckets_[old_idx];
    }
    for (size_t i = 0; i < capacity_; ++i) {
        if (!new_buckets[i]) new_buckets[i] = new Bucket();
    }

    delete[] buckets_;
    buckets_ = new_buckets;
    first_.bucket = mid;
    last_.bucket  = mid + used_buckets - 1;
}
```

This approach has several advantages:

- Amortized O(1) cost for insertion operations
- Centralizes elements in the new bucket array, allowing for growth in both directions
- Reuses existing buckets to avoid unnecessary copying of elements

8.2 Bucket Navigation

The implementation includes helper methods for navigating buckets, which simplify the logic for advancing and retreating through the circular buffer:

```
void advance_back() {
    if (++last_.index == bucketSize) {
        last_.index = 0;
        ++last_.bucket;

        if (last_.bucket >= capacity_) {
            increase_size();
        }
    }
}

```

```
void retreat_front() {
    if (first_.index == 0) {
        first_.index = bucketSize - 1;
        first_.bucket = (first_.bucket == 0 ? capacity_ - 1 :
           first_.bucket - 1);
    } else {
        --first_.index;
    }
}
```

These methods handle the complexities of bucket transitions, making the higher-level operations cleaner and less error-prone.

9 Performance Optimizations and Potential Improvements

9.1 Current Optimizations

- **Bucket-based Storage**: Reduces memory fragmentation and improves cache locality.
- **Memory Alignment**: Ensures optimal performance for different data types.
- **Placement New**: Minimizes unnecessary copying of elements.
- **Circular Buffer Design**: Enables constant-time operations at both ends.

9.2 Potential Improvements

- **Lazy Initialization**: Currently, all buckets are allocated upfront. We could optimize memory usage by allocating buckets only when needed.

```
    // Example of lazy bucket initialization
    T* element_at(size_t bucket, size_t index) {
        if (!buckets_[bucket]) {
            buckets_[bucket] = new Bucket();
        }
        return reinterpret_cast<T*>(&buckets_[bucket]->data[
           index * sizeof(T)]);
    }
```

- **Custom Allocator Support**: Add support for custom allocators to allow further optimization for specific use cases.

```
    template <typename T, typename Allocator = std::allocator<
       T>>
    class deque {
        // Implementation with allocator support
    };
```

- **SIMD Operations**: For numeric types, operations like bulk initialization could be optimized using SIMD instructions.
- **Shrinking Policy**: Implement a strategy to reduce capacity when a significant portion of the deque is unused, to reclaim memory.

```
    void shrink_to_fit() {
        // Reduce capacity if utilization is below a threshold
        if (size_ < capacity_ * bucketSize / 4) {
            // Implementation of shrinking logic
        }
    }
```

- **Thread Safety**: Add thread-safe operations or documented synchronization points for concurrent access.

9.3 Benchmarking Considerations

To further optimize the implementation, benchmarking against different scenarios would be valuable:

- Different bucket sizes for various element types
- Performance with small vs. large elements
- Comparison with std::deque and other container implementations
- Memory usage patterns for different operations

9.4 Common Pitfalls to Avoid

When working with this deque implementation, be aware of these potential issues:

- **Type Requirements**: Types with special alignment requirements might need additional handling.
- **Exception Safety**: The current implementation provides basic exception safety but could be improved for strong exception safety.
- **Iterator Invalidation**: Be aware that iterators may be invalidated after insertion operations that cause resizing.

10 Time Complexity Analysis

Operation	Time Complexity
push_front, push_back	O(1) amortized
pop_front, pop_back	O(1)
front, back	O(1)
operator[]	O(1)
size, empty	O(1)
iterator operations	O(1)
clear	O(n)

11 Memory Optimization

The implementation employs several memory optimization techniques:

- **Alignment**: Elements are properly aligned to minimize padding.
- **Bucket Allocation**: Allocating memory in buckets reduces fragmentation.
- **Placement New**: Used for in-place construction of elements without unnecessary copying.
- **Amortized Resizing**: Only grows the container when needed, using a growth factor.

12 Usage Examples

12.1 Basic Usage

```cpp
#include "deque.h"
#include <iostream>

int main() {
    sml::deque<int> deque;

    // Add elements
    deque.push_front(1);
    deque.push_back(2);

    // Access elements
    std::cout << "Front: " << deque.front() << std::endl;
    std::cout << "Back: " << deque.back() << std::endl;

    // Iterate through elements
    for (auto& value : deque) {
        std::cout << value << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

12.2 Advanced Usage

```cpp
sml::deque<std::string> strDeque;

// Add elements
strDeque.push_back("Hello");
strDeque.push_back("World");

// Iterate and modify
```

```cpp
for (auto& str : strDeque) {
    str += "!";
}

// Use with standard algorithms
std::for_each(strDeque.begin(), strDeque.end(),
              [](const std::string& s) {
                  std::cout << s << std::endl;
              });
```

12.3 Complex Object Storage

```cpp
struct ComplexObject {
    ComplexObject(int a, double b) : a_(a), b_(b) {}
    ComplexObject(const ComplexObject& other) : a_(other.a_), b_(other.b_) {
        std::cout << "Copy constructor called" << std::endl;
    }
    ~ComplexObject() {
        std::cout << "Destructor called for " << a_ << std::endl;
    }

    int a_;
    double b_;
};

int main() {
    sml::deque<ComplexObject> objDeque;

    // Emplace elements without unnecessary copies
    objDeque.push_back(ComplexObject(1, 2.5));
    objDeque.push_back(ComplexObject(3, 4.5));

    // Access properties
    for (const auto& obj : objDeque) {
        std::cout << "a: " << obj.a_ << ", b: " << obj.b_ << std::endl;
    }

    // Deque will properly destroy all objects when it goes out of scope
    return 0;
}
```

12.4 Using as a Queue or Stack

The deque can easily be used as either a queue or a stack:

```cpp
// As a queue (FIFO)
sml::deque<int> queue;
```

```cpp
queue.push_back(1);  // Enqueue
queue.push_back(2);
queue.push_back(3);

while (!queue.empty()) {
    std::cout << queue.front() << " ";  // Peek
    queue.pop_front();  // Dequeue
}
// Output: 1 2 3

// As a stack (LIFO)
sml::deque<int> stack;
stack.push_back(1);  // Push
stack.push_back(2);
stack.push_back(3);

while (!stack.empty()) {
    std::cout << stack.back() << " ";  // Peek
    stack.pop_back();  // Pop
}
// Output: 3 2 1
```

13 Implementation Details

13.1 Memory Layout

The deque maintains a dynamic array of bucket pointers. Each bucket contains fixed-size raw memory for storing elements. Elements are distributed across buckets with proper tracking of the first and last positions.

13.2 Growth Strategy

When the deque becomes full, it allocates a new array of buckets with a size scaled by the growth factor (default is 2). Elements are redistributed to maintain the logical order while optimizing the memory layout.

13.3 Element Construction and Destruction

The implementation uses placement new for constructing objects in pre-allocated memory:

```cpp
new (element_at(bucket, index)) T(value);
```

And explicit destructor calls for destroying objects without deallocating memory:

```cpp
element_at(bucket, index)->~T();
```

14 Best Practices

14.1 When to Use This Deque

This deque implementation is particularly useful when:

- You need efficient insertion/removal at both ends
- You require better cache locality than linked-list-based solutions
- You need a more memory-efficient alternative to std::deque

14.2 Performance Considerations

To optimize performance:

- Adjust the bucket size based on your use case
- Consider the element size when setting the bucket size
- Avoid unnecessary copying of large elements

15 Conclusion

This custom deque implementation provides an efficient and flexible double-ended queue with constant-time operations at both ends. It balances memory efficiency with performance through its bucket-based design. The implementation addresses common challenges in container design, such as memory management, object lifetime, and efficient element access. Through careful debugging and optimization, we've created a robust and performant data structure that can be used in a variety of applications.