C++ Programming Guide

A Comprehensive Journey Through Modern C++

STL, Exception Handling, and Real-Time Systems

Avantika Srivastava

November 5, 2025

Preface

This book is a comprehensive guide to C++ programming, covering fundamental concepts to advanced topics. It is designed to be both detailed and accessible, making complex topics understandable through clear explanations, practical examples, and concise summaries.

What You'll Learn:

- String manipulation in C++
- Exception handling best practices
- Polymorphism and runtime type information
- Standard Template Library (STL) containers
- FreeRTOS for embedded systems
- Multithreading and synchronization

Each chapter includes:

- **TL;DR** sections for quick reference
- **Detailed explanations** of concepts
- **Code examples** with annotations
- **Key points** highlighted in boxes
- **Summaries** at the end of each chapter
- **Time complexity** analysis where applicable

Contents

Part I

C++ Fundamentals

Chapter 1

String Manipulation in C++

TL;DR

C++ provides both C-style strings (character arrays) and `std::string` class. The `std::string` class is safer, more flexible, and recommended for modern C++ programming. It provides automatic memory management, bounds checking, and rich functionality.

1.1 C-Style String Functions

C-style strings are null-terminated character arrays inherited from C. While they are still used in legacy code and certain system-level programming, they require manual memory management and are prone to buffer overflow errors.

1.1.1 Common C-String Functions

Key Point

All C-style string functions are found in `<cstring>` header. They operate on `char*` or `const char*` pointers and assume null termination.

Function	Description
`strlen(str)`	Returns length of string (excluding null terminator)
`strcmp(s1, s2)`	Compares two strings lexicographically
`strncmp(s1, s2, n)`	Compares first n characters
`strchr(str, ch)`	Finds first occurrence of character
`strrchr(str, ch)`	Finds last occurrence of character
`strcpy(dest, src)`	Copies source string to destination
`strncpy(dest, src, n)`	Copies n characters from source
`strcat(dest, src)`	Concatenates source to destination
`strncat(dest, src, n)`	Concatenates n characters

Table 1.1: Common C-style string functions

Warning

C-style string functions like `strcpy` and `strcat` do not perform bounds checking. Always use safer alternatives like `strncpy` and `strncat`, or better yet, use `std::string`.

1.2 The std::string Class

The `std::string` class is part of the C++ Standard Library and provides a modern, safe way to work with strings.

1.2.1 Basic Operations

Creating and Converting Strings

```cpp
#include <string>
#include <iostream>

int main() {
    // Creating strings
    std::string str1 = "Hello";
    std::string str2("World");
    std::string str3(5, 'A');  // "AAAAA"

    // Converting numbers to strings
    std::string numStr = std::to_string(67);  // "67"

    // Append operation
    str1.append(str2);  // str1 is now "HelloWorld"

    std::cout << str1 << std::endl;
    return 0;
}
```

Listing 1.1: String creation and conversion

Accessing String Elements

```cpp
#include <string>
#include <iostream>

int main() {
    std::string str1 = "Hello";

    // Size of string
    std::cout << "Size: " << str1.size() << std::endl;  // 5

    // Access character at index
    char ch = str1.at(1);  // 'e' (with bounds checking)
    char ch2 = str1[1];    // 'e' (no bounds checking)

    // Range-based for loop
    for(char value : str1) {
```

```
        std::cout << " " << value;
    }
    std::cout << std::endl;

    // First and last character
    char first = str1.front();  // 'H'
    char last = str1.back();    // 'o'

    return 0;
}
```

Listing 1.2: String element access

Key Point

at() vs [] operator:

- `at(index)` performs bounds checking and throws `std::out_of_range` exception if index is invalid
- `operator[]` does not check bounds but is faster
- Use `at()` when safety is important, `[]` when performance matters and you're sure the index is valid

Modifying String Characters

```
#include <string>
#include <iostream>

int main() {
    std::string str2 = "hello";

    // Get reference to first character
    char& front_char_ref = str2.front();

    // Modify through reference
    str2.front() = 'H';  // str2 is now "Hello"

    std::cout << str2 << std::endl;
    return 0;
}
```

Listing 1.3: Modifying strings through references

C-String Interoperability

```
#include <string>
#include <iostream>
#include <cstring>

int main() {
    std::string str2 = "Hello";

    // c_str() method: returns const char*
```

```
    // Cannot modify the returned pointer
    const char* wrapped_c_string = str2.c_str();
    std::cout << wrapped_c_string << std::endl;

    // data() method: returns char* (C++17: returns const char*)
    // In older C++ standards, can be modified
    char* string = str2.data();

    // Note: In C++11 and later, data() also returns const char*
    // In C++17+, both c_str() and data() are identical

    return 0;
}
```

Listing 1.4: Converting between std::string and C-strings

Warning

The pointer returned by `c_str()` and `data()` is only valid until the next modification of the string. Store it immediately or copy it if you need to keep it longer.

1.3 String Properties and Capacity Management

Understanding string capacity and size is crucial for performance optimization.

1.3.1 Size and Capacity Functions

```
#include <string>
#include <iostream>

int main() {
    std::string str1 = "Hello";

    // Check if empty
    if (str1.empty()) {
        std::cout << "String is empty" << std::endl;
    }

    // Size and length (identical)
    std::cout << "Size: " << str1.size() << std::endl;
    std::cout << "Length: " << str1.length() << std::endl;

    // Maximum possible size on this system
    std::cout << "Max size: " << std::string::max_size() << std::
    endl;
    std::cout << "Max size (member): " << str1.max_size() << std::
    endl;

    // Current capacity (storage allocated)
    std::cout << "Capacity: " << str1.capacity() << std::endl;

    // Reserve capacity (prevent reallocations)
    str1.reserve(100);
```

```
25      std::cout << "After reserve(100): " << str1.capacity() << std
        ::endl;
26
27      // Shrink capacity to fit current size
28      str1.shrink_to_fit();
29      std::cout << "After shrink_to_fit: " << str1.capacity() << std
        ::endl;
30
31      return 0;
32  }
33  \</lstlisting>
34
35  \begin{keypoint}
36  \textbf{Capacity vs Size:}
37  \begin{itemize}
38      \item \textbf{Size/Length}: Actual number of characters in the
        string (excluding null terminator)
39      \item \textbf{Capacity}: Amount of storage allocated (may be
        larger than size)
40      \item \textbf{Default capacity}: Typically 15 characters for
        small string optimization (SSO)
41      \item When size exceeds capacity, string is reallocated (
        expensive operation)
42  \end{itemize}
43  \end{keypoint}
44
45  \subsection{Performance Tip: Reserve Memory}
46
47  \begin{example}
48  If you know you'll be building a large string, reserve space
        upfront:
49
50  \begin{lstlisting}
51  std::string result;
52  result.reserve(10000);  // Prevent multiple reallocations
53
54  for (int i = 0; i < 10000; ++i) {
55      result += 'A';  // No reallocation needed
56  }
```

Listing 1.5: String size and capacity operations

1.4 String Modification Operations

1.4.1 Clearing and Inserting

bĖnclude <string> include <iostream>
int main() std::string str1 = "Hello World";
// Clear all characters str1.clear(); std::cout « "After clear, size: " « str1.size() « std::endl; // 0
str1 = "Hello";
// Insert operations // insert(index, count, ch) - insert character 'ch' count times at index std::string str2 = "Helo"; str2.insert(2, 1, 'l'); // "Hello" std::cout

« str2 « std::endl;

// insert(index, string) - insert string at index std::string str3 = "HelloWorld"; std::string txt3 = " "; str3.insert(5, txt3); // "Hello World" std::cout « str3 « std::endl;

// insert(index, string, count) - insert first count chars std::string str4 = "HelloWorld"; std::string txt4 = "Beautiful"; str4.insert(5, txt4, 6); // Insert first 6 chars of "Beautiful" std::cout « str4 « std::endl;

// insert(index, $source_str, source_index, count$)$std :: string str7 = "HelloWorld"; std :: string str8 = "Thisisabeautifulday"; str7.insert(5, str8, 10, 10); //Insert "beautiful" from str8 std :: cout << str7 << std :: endl;$

return 0;

1.4.2 Erasing Characters

```
#include <string>
#include <iostream>

int main() {
    std::string str9 = "Hello World!!!!!";

    // erase(index, count) - remove count characters starting at index
    str9.erase(11, str9.size() - 11);  // Remove all '!' characters
    std::cout << str9 << std::endl;  // "Hello World"

    return 0;
}
```

Listing 1.6: Erasing from strings

1.4.3 Push and Pop Operations

```
#include <string>
#include <iostream>

int main() {
    std::string str10 = "Hello World";

    // Append character at end
    str10.push_back('!');
    std::cout << str10 << std::endl;  // "Hello World!"

    // Remove last character
    str10.pop_back();
    std::cout << str10 << std::endl;  // "Hello World"

    return 0;
}
```

Listing 1.7: Push and pop operations

1.5 String Comparison

C++ provides convenient operators for string comparison.

```cpp
#include <string>
#include <iostream>

int main() {
    std::string str1 = "Hello";
    std::string str2 = "World";
    std::string str3 = "Hello";

    // Comparison operators
    if (str1 == str3) {
        std::cout << "str1 equals str3" << std::endl;
    }

    if (str1 != str2) {
        std::cout << "str1 not equal to str2" << std::endl;
    }

    if (str1 < str2) {
        std::cout << "str1 is lexicographically less than str2" <<
    std::endl;
    }

    // Comparing std::string with C-string
    const char* cstr = "Hello";
    if (str1 == cstr) {
        std::cout << "str1 equals cstr" << std::endl;
    }

    return 0;
}
```

Listing 1.8: String comparison

Key Point

String Comparison Rules:

- Operators: `==`, `!=`, `<`, `<=`, `>`, `>=`
- Lexicographical comparison (dictionary order)
- Case-sensitive comparison
- Can compare `std::string` with C-strings directly
- Always ensure C-strings are null-terminated to avoid undefined behavior

Warning

Never compare C-strings without null terminators. This can lead to undefined behavior as the comparison functions read memory until they find a null character.

Summary

Chapter 1 Summary:

- **C-Style Strings**: Use `<cstring>` functions; require manual memory management
- **std::string**: Modern C++ string class with automatic memory management
- **Conversion**: Use `std::to_string()` for numbers to strings
- **Access**: `at()` for safe access, `[]` for fast access
- **Capacity**: `reserve()` to preallocate, `shrink_to_fit()` to reduce
- **Modification**: `insert()`, `erase()`, `push_back()`, `pop_back()`
- **Comparison**: Use standard operators (`==`, `!=`, `<`, etc.)
- **C Interop**: Use `c_str()` or `data()` to get C-style pointer

Best Practices:

1. Prefer `std::string` over C-style strings
2. Use `reserve()` when building large strings
3. Use `at()` for safety during development, `[]` for performance in production
4. Always ensure C-strings are null-terminated

Chapter 2

Exception Handling in C++

TL;DR

Exception handling in C++ uses `try`, `throw`, and `catch` blocks to manage errors gracefully. Key concepts: virtual destructors prevent memory leaks, `noexcept` specifiers control exception propagation, and standard exceptions provide consistent error handling. Always catch more specific exceptions before generic ones.

2.1 Basic Exception Handling

Exception handling provides a way to transfer control from one part of a program to another when an exceptional circumstance (error) occurs.

2.1.1 The throw and catch Mechanism

```cpp
#include <iostream>
#include <stdexcept>

double divide(double a, double b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero!");
    }
    return a / b;
}

int main() {
    try {
        double result = divide(10, 0);
        std::cout << "Result: " << result << std::endl;
    }
    catch (const std::runtime_error& ex) {
        std::cout << "Error: " << ex.what() << std::endl;
    }

    std::cout << "Program continues..." << std::endl;
    return 0;
```

```
}
```

Listing 2.1: Basic exception handling

Key Point

Exception Handling Flow:

1. Code in `try` block executes normally
2. If `throw` is encountered, execution immediately jumps to matching `catch`
3. Remaining code in `try` block is skipped
4. Appropriate `catch` block handles the exception
5. Program continues after the `catch` blocks

2.1.2 Catch Block Ordering

```
#include <iostream>
#include <exception>
#include <stdexcept>

class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "My custom exception";
    }
};

void riskyFunction(int code) {
    if (code == 1) {
        throw std::runtime_error("Runtime error");
    } else if (code == 2) {
        throw std::logic_error("Logic error");
    } else if (code == 3) {
        throw MyException();
    }
}

int main() {
    try {
        riskyFunction(2);
    }
    catch (const std::runtime_error& ex) {
        std::cout << "Caught runtime_error: " << ex.what() << std
    ::endl;
    }
    catch (const std::logic_error& ex) {
        std::cout << "Caught logic_error: " << ex.what() << std::
    endl;
    }
    catch (const std::exception& ex) {
        // More generic catch - should be at the end
```

```
34          std::cout << "Caught exception: " << ex.what() << std::
    endl;
35      }
36      catch (...) {
37          // Catch all - must be last
38          std::cout << "Caught unknown exception" << std::endl;
39      }
40
41      return 0;
42  }
```

Listing 2.2: Multiple catch blocks - order matters!

Warning

Always use the most generic catch block last!
If you place a base class catch block (like `std::exception`) before derived class blocks, the derived class catches will never be reached. The compiler may warn you, but it's a logical error.
Order from most specific to most generic:

1. Derived class exceptions
2. Base class exceptions
3. `catch(...)` – the ellipsis catch-all

2.2 Virtual Destructors and Exceptions

2.2.1 Why Destructors Should Be Virtual

```
1  #include <iostream>
2
3  class Base {
4  public:
5      Base() { std::cout << "Base constructor" << std::endl; }
6
7      // Without virtual: only Base destructor called when deleting
8      // through base pointer - MEMORY LEAK!
9      virtual ~Base() {
10         std::cout << "Base destructor" << std::endl;
11     }
12 };
13
14 class Derived : public Base {
15     int* data;
16 public:
17     Derived() : data(new int[100]) {
18         std::cout << "Derived constructor" << std::endl;
19     }
20
21     ~Derived() {
22         delete[] data;  // Won't be called if Base destructor isn'
    t virtual!
```

```
        std::cout << "Derived destructor" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();

    // If ~Base() is not virtual:
    // - Only Base destructor is called (static binding)
    // - Derived destructor is NOT called
    // - Memory leak! (data array not freed)
    delete ptr;

    return 0;
}
```

Listing 2.3: Virtual destructors prevent memory leaks

Key Point

Virtual Destructor Rule:
If a class is designed to be inherited from (has virtual functions or is a base class), **always make the destructor virtual**.
Why?

- Without virtual destructor: static binding determines which destructor to call
- With base pointer to derived object: only base destructor called = memory leak
- With virtual destructor: proper cleanup chain (derived first, then base)

2.2.2 Using Virtual Functions with Exceptions

```
#include <iostream>
#include <exception>

class BaseException : public std::exception {
public:
    virtual const char* what() const noexcept override {
        return "Base exception";
    }
    virtual ~BaseException() = default;
};

class DerivedException : public BaseException {
public:
    const char* what() const noexcept override {
        return "Derived exception";
    }
};

void throwException(int type) {
```

```
    if (type == 1) {
        throw BaseException();
    } else {
        throw DerivedException();
    }
}

int main() {
    try {
        throwException(2);
    }
    catch (const BaseException& ex) {
        // Single catch block handles both types polymorphically
        std::cout << "Caught: " << ex.what() << std::endl;
        // Prints "Derived exception" due to virtual function
    }

    return 0;
}
```

Listing 2.4: Polymorphic exception handling

2.3 Exception Objects and Copy Constructors

2.3.1 Requirements for Exception Objects

```
#include <iostream>
#include <string>

class MyError {
    std::string message;
public:
    // Copy constructor must be accessible (public)
    MyError(const std::string& msg) : message(msg) {}

    // Copy constructor (implicitly generated if not defined)
    MyError(const MyError& other) : message(other.message) {
        std::cout << "Copy constructor called" << std::endl;
    }

    const std::string& what() const { return message; }
};

void throwError() {
    MyError err("Something went wrong");
    throw err;  // Object is copied when thrown
}

int main() {
    try {
        throwError();
    }
    catch (const MyError& ex) {
        std::cout << "Caught: " << ex.what() << std::endl;
    }

```

```
    return 0;
}
```

Listing 2.5: Exception classes need copy constructors

Key Point

Exception Object Requirements:

- Must have a public copy constructor
- The exception object is copied when thrown
- Original object (if local) is destroyed when function exits
- Catch by reference to avoid additional copy and slicing

2.3.2 Never Throw Pointers to Local Variables

```
#include <iostream>

class MyError {
public:
    std::string message;
    MyError(const std::string& msg) : message(msg) {}
};

void badFunction() {
    MyError localError("Local error");

    // WRONG! localError will be destroyed when function returns
    // throw &localError;  // Dangling pointer!
}

void goodFunction() {
    // Throw by value (will be copied)
    throw MyError("Good error");

    // Or allocate on heap (but caller must manage memory)
    // throw new MyError("Heap error");  // Requires delete in
   catch
}

int main() {
    try {
        goodFunction();
    }
    catch (const MyError& ex) {
        std::cout << "Caught: " << ex.message << std::endl;
    }

    return 0;
}
```

Listing 2.6: Dangerous exception practice - DON'T DO THIS

Warning

Never throw the address of a local auto variable!
When a function exits:

- Local (automatic) variables are destroyed
- Their memory addresses become invalid
- Pointer to destroyed object = undefined behavior

Best practice: Throw exceptions by value, catch by const reference.

2.4 Exception Propagation

2.4.1 Unwinding the Call Stack

```cpp
#include <iostream>
#include <stdexcept>

void level3() {
    std::cout << "Level 3: About to throw" << std::endl;
    throw std::runtime_error("Error at level 3");
    std::cout << "Level 3: This won't execute" << std::endl;
}

void level2() {
    std::cout << "Level 2: Calling level 3" << std::endl;
    level3();
    std::cout << "Level 2: This won't execute" << std::endl;
}

void level1() {
    std::cout << "Level 1: Calling level 2" << std::endl;
    try {
        level2();
    }
    catch (const std::exception& ex) {
        std::cout << "Level 1: Caught exception: " << ex.what() << std::endl;
    }
    std::cout << "Level 1: Continues after catch" << std::endl;
}

int main() {
    level1();
    std::cout << "Main: Program continues" << std::endl;
    return 0;
}

/* Output:
Level 1: Calling level 2
Level 2: Calling level 3
Level 3: About to throw
Level 1: Caught exception: Error at level 3
```

```
Level 1: Continues after catch
Main: Program continues
*/
```

Listing 2.7: Exception propagation through call stack

Key Point

Stack Unwinding:

1. When exception is thrown, function immediately exits
2. Control passes to caller
3. Process repeats until matching catch block found
4. All local objects are properly destroyed during unwinding
5. Code after `throw` in same block never executes

2.5 Uncaught Exceptions and Terminate Handlers

2.5.1 Custom Terminate Function

```
#include <iostream>
#include <exception>
#include <thread>
#include <chrono>

void our_terminate_func() {
    std::cout << "Custom terminate function called!" << std::endl;
    std::cout << "Cleaning up..." << std::endl;

    // Optional: delay before termination
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    // Must terminate the program
    std::abort();
}

void problematicFunction() {
    throw std::runtime_error("Unhandled exception");
}

int main() {
    // Set custom terminate handler
    std::set_terminate(our_terminate_func);

    // Alternative with lambda
    std::set_terminate([]() {
        std::cout << "Lambda terminate handler" << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds
   (10000));
```

```
        std::abort();
    });

    // This will call terminate since there's no catch
    problematicFunction();

    return 0;
}
\</lstlisting>

\begin{keypoint}
\textbf{Terminate Handlers:}
\begin{itemize}
    \item Called when exception is thrown but not caught
    \item Set using \texttt{std::set\_terminate()}
    \item Must terminate the program (typically calls \texttt{std::abort()})
    \item If you don't call \texttt{std::abort()}, it's called automatically
    \item Useful for logging, cleanup, or graceful shutdown
\end{itemize}
\end{keypoint}

\section{Hierarchical Error Handling}

\subsection{Nested Try-Catch Blocks}

\begin{lstlisting}[caption={Nested exception handling}]
#include <iostream>
#include <stdexcept>

class OuterException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Outer exception";
    }
};

class InnerException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Inner exception";
    }
};

void innerFunction() {
    try {
        throw InnerException();
    }
    catch (const InnerException& ex) {
        std::cout << "Inner catch: " << ex.what() << std::endl;
        // Re-throw for outer handler
        throw OuterException();
    }
}

int main() {
    try {
```

```
85          innerFunction();
86      }
87      catch (const OuterException& ex) {
88          std::cout << "Outer catch: " << ex.what() << std::endl;
89      }
90
91      return 0;
92  }
93  \</lstlisting>
94
95  \begin{warning}
96  \textbf{Nested Exception Best Practices:}
97  \begin{itemize}
98      \item Only throw exceptions that the outer catch block can
        handle
99      \item Don't throw objects with values that may have been
        destroyed
100     \item Avoid creating complex exception hierarchies
101     \item Consider using exception chaining (storing original
        exception)
102 \end{itemize}
103 \end{warning}
104
105 \section{The Ellipsis Catch-All}
106
107 \begin{lstlisting}[caption={Catch-all handler}]
108 #include <iostream>
109
110 void unpredictableFunction(int code) {
111     if (code == 1) {
112         throw 42;                   // integer
113     } else if (code == 2) {
114         throw 3.14;                 // double
115     } else if (code == 3) {
116         throw "string error";       // const char*
117     }
118 }
119
120 int main() {
121     for (int i = 1; i <= 4; ++i) {
122         try {
123             unpredictableFunction(i);
124         }
125         catch (int ex) {
126             std::cout << "Caught int: " << ex << std::endl;
127         }
128         catch (double ex) {
129             std::cout << "Caught double: " << ex << std::endl;
130         }
131         catch (...) {
132             // Catches ANY type of exception
133             std::cout << "Caught unknown exception" << std::endl;
134         }
135     }
136
137     return 0;
```

```
138 }
```

Listing 2.8: Setting a custom terminate handler

Key Point

Ellipsis Catch Block `catch(...)`

- Catches exceptions of any type
- Cannot access the exception object
- Must be the last catch block
- Use for logging or cleanup when exception type is unknown
- Useful in generic library code

2.6 The noexcept Specifier

2.6.1 Basic noexcept Usage

```cpp
#include <iostream>

// Promise: this function will not throw exceptions
void safeFunction() noexcept {
    std::cout << "This function never throws" << std::endl;
}

// This is dangerous!
void dangerousNoexcept() noexcept {
    throw std::runtime_error("Exception in noexcept function");
    // Program will terminate!
}

int main() {
    safeFunction();

    // This will terminate the program
    // dangerousNoexcept();

    return 0;
}
```

Listing 2.9: noexcept functions

Warning

noexcept Behavior:

- `noexcept` is a promise that function won't throw
- If exception escapes a `noexcept` function, program terminates
- No stack unwinding occurs
- Compiler may issue warnings but won't prevent compilation
- `std::terminate()` is called immediately

2.6.2 Destructors and noexcept

```cpp
#include <iostream>

class Item {
public:
    // Destructors are implicitly noexcept
    ~Item() {
        // If this throws, program terminates!
        std::cout << "Destructor called" << std::endl;
    }
};

class ThrowingDestructor {
public:
    // Explicitly allow exceptions from destructor (not
    recommended!)
    ~ThrowingDestructor() noexcept(false) {
        throw std::runtime_error("Destructor threw");
    }
};

int main() {
    Item item;  // OK

    try {
        ThrowingDestructor td;
    }  // Destructor called here
    catch (const std::exception& ex) {
        std::cout << "Caught: " << ex.what() << std::endl;
    }

    return 0;
}
```

Listing 2.10: Destructors are noexcept by default

Key Point

Destructors and Exceptions:

- All destructors are `noexcept` by default
- Throwing from destructor during stack unwinding = double exception = terminate
- Use `noexcept(false)` to explicitly allow exceptions (dangerous!)
- **Best practice:** Never throw from destructors

2.7 Standard Exceptions

C++ provides a hierarchy of standard exception classes in `<stdexcept>`.

2.7.1 Exception Hierarchy

```
/*
std::exception (base class)
    |
    +-- std::logic_error
    |       +-- std::invalid_argument
    |       +-- std::domain_error
    |       +-- std::length_error
    |       +-- std::out_of_range
    |
    +-- std::runtime_error
    |       +-- std::range_error
    |       +-- std::overflow_error
    |       +-- std::underflow_error
    |
    +-- std::bad_alloc
    +-- std::bad_cast
    +-- std::bad_typeid
    +-- std::bad_exception
*/
```

Listing 2.11: Standard exception hierarchy

2.7.2 Catching Standard Exceptions

```
#include <iostream>
#include <stdexcept>
#include <vector>

void demonstrateExceptions() {
    std::vector<int> vec = {1, 2, 3};

    try {
        // This throws std::out_of_range
        int value = vec.at(10);
```

```
    }
    catch (const std::exception& ex) {
        // Catches all standard exceptions
        std::cout << "Caught exception: " << ex.what() << std::
    endl;
    }
}

int main() {
    demonstrateExceptions();
    return 0;
}
```

Listing 2.12: Using standard exceptions

2.7.3 Throwing Standard Exceptions

```
#include <iostream>
#include <stdexcept>
#include <string>

int getElement(const std::vector<int>& vec, size_t index) {
    if (index >= vec.size()) {
        std::string message = "Index " + std::to_string(index) +
                              " out of range (size: " +
                              std::to_string(vec.size()) + ")";
        throw std::out_of_range(message);
    }
    return vec[index];
}

double divide(double a, double b) {
    if (b == 0.0) {
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}

int main() {
    std::vector<int> numbers = {10, 20, 30};

    try {
        int val = getElement(numbers, 5);
    }
    catch (const std::out_of_range& ex) {
        std::cout << "Error: " << ex.what() << std::endl;
    }

    try {
        double result = divide(10, 0);
    }
    catch (const std::invalid_argument& ex) {
        std::cout << "Error: " << ex.what() << std::endl;
    }

    return 0;
```

```
40 }
```

Listing 2.13: Throwing standard exceptions with custom messages

Key Point

When to Use Each Standard Exception:
Logic Errors (programming mistakes):

- `std::invalid_argument`: Invalid function argument
- `std::out_of_range`: Index/key out of valid range
- `std::length_error`: Attempted size exceeds maximum
- `std::domain_error`: Mathematical domain error

Runtime Errors (external conditions):

- `std::runtime_error`: General runtime error
- `std::range_error`: Result outside representable range
- `std::overflow_error`: Arithmetic overflow
- `std::underflow_error`: Arithmetic underflow

Summary

Chapter 2 Summary:
Core Concepts:

- **try-throw-catch**: Standard exception handling mechanism
- **Stack unwinding**: Automatic cleanup during exception propagation
- **Catch ordering**: Specific to generic (derived before base)
- **Virtual destructors**: Necessary for proper polymorphic cleanup

Best Practices:

1. Always use virtual destructors in base classes
2. Throw by value, catch by const reference
3. Never throw from destructors
4. Order catch blocks from specific to generic
5. Use `noexcept` for functions that truly never throw
6. Prefer standard exceptions over custom types
7. Never throw pointers to local variables
8. Set terminate handler for cleanup in critical applications

Standard Exceptions:

- `std::exception`: Base class for all standard exceptions
- `std::logic_error`: For programming errors
- `std::runtime_error`: For runtime conditions
- All have `what()` method returning error description

Advanced Features:

- `noexcept` specifier and `noexcept(false)` for destructors
- `catch(...)` for catching all exception types
- `std::set_terminate()` for custom termination handling
- Nested exception handling for complex error scenarios

Chapter 3

Polymorphism and Runtime Type Information

TL;DR

Polymorphism allows objects of different types to be treated uniformly through base class interfaces. The `typeid` operator provides runtime type information. Pure virtual functions create abstract classes that define interfaces without implementation. Abstract classes cannot be instantiated but serve as blueprints for derived classes.

3.1 Runtime Type Information (RTTI)

3.1.1 The typeid Operator

The `typeid` operator returns type information about an expression at runtime.

```
#include <iostream>
#include <typeinfo>
#include <string>

class Base {
    virtual void dummy() {}  // Need virtual function for polymorphic RTTI
};

class Derived : public Base {};

int main() {
    // Built-in types
    std::cout << "Type of int: " << typeid(int).name() << std::endl;
    std::cout << "Type of double: " << typeid(double).name() << std::endl;
    std::cout << "Type of string: " << typeid(std::string).name() << std::endl;

    // Variables
    int x = 42;
    double y = 3.14;
```

```
20      std::cout << "Type of x: " << typeid(x).name() << std::endl;
21      std::cout << "Type of y: " << typeid(y).name() << std::endl;
22
23      // Polymorphic types
24      Base* basePtr = new Derived();
25      std::cout << "Type through pointer: " << typeid(*basePtr).name
        ()
26                << std::endl;
27
28      // Type comparison
29      if (typeid(int) == typeid(x)) {
30          std::cout << "x is of type int" << std::endl;
31      }
32
33      delete basePtr;
34      return 0;
35  }
```

Listing 3.1: Using typeid operator

Key Point

typeid Operator:

- Returns a `const std::type_info&` reference
- `name()` method returns implementation-defined string
- For polymorphic types, requires at least one virtual function
- Can compare types using `==` and `!=`
- Available in `<typeinfo>` header

Warning

The string returned by `typeid(type).name()` is implementation-defined and may be mangled. For example:

- GCC/Clang: "i" for int, "d" for double, "NSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE" for std::string
- MSVC: "int", "double", "class std::basic_string<...>"

For readable names, use compiler-specific demangling functions or C++11's type traits.

3.2 Pure Virtual Functions

A pure virtual function is a virtual function that has no implementation in the base class and must be overridden by derived classes.

3.2.1 Syntax and Purpose

```cpp
#include <iostream>
#include <string>

class Shape {
public:
    // Pure virtual function (note the = 0)
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual std::string name() const = 0;

    // Regular virtual function (has implementation)
    virtual void display() const {
        std::cout << "Shape: " << name()
                  << ", Area: " << area()
                  << ", Perimeter: " << perimeter() << std::endl;
    }

    // Virtual destructor
    virtual ~Shape() = default;
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}

    // Must override all pure virtual functions
    double area() const override {
        return 3.14159 * radius * radius;
    }

    double perimeter() const override {
        return 2 * 3.14159 * radius;
    }

    std::string name() const override {
        return "Circle";
    }
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const override {
        return width * height;
    }

    double perimeter() const override {
        return 2 * (width + height);
    }

    std::string name() const override {
        return "Rectangle";
    }
```

```
};

int main() {
    // Cannot instantiate abstract class:
    // Shape s;  // ERROR!

    // Can use pointers/references to abstract class
    Shape* shapes[2];
    shapes[0] = new Circle(5.0);
    shapes[1] = new Rectangle(4.0, 6.0);

    for (int i = 0; i < 2; ++i) {
        shapes[i]->display();
    }

    delete shapes[0];
    delete shapes[1];

    return 0;
}
```

Listing 3.2: Pure virtual function syntax

Key Point

Pure Virtual Functions:

- Declared with `= 0` at the end
- Make the class abstract (cannot be instantiated)
- Must be overridden in derived classes
- Derived class must implement all pure virtual functions to be concrete
- Can still have implementation (rarely used)

3.3 Abstract Classes

An abstract class is a class that contains at least one pure virtual function and cannot be instantiated.

3.3.1 Characteristics of Abstract Classes

```
#include <iostream>
#include <string>
#include <vector>

// Abstract base class (interface)
class Printable {
public:
    virtual void print() const = 0;
    virtual ~Printable() = default;
};
```

```
11
12 // Another abstract class
13 class Serializable {
14 public:
15     virtual std::string serialize() const = 0;
16     virtual void deserialize(const std::string& data) = 0;
17     virtual ~Serializable() = default;
18 };
19
20 // Concrete class implementing multiple interfaces
21 class Document : public Printable, public Serializable {
22     std::string title;
23     std::string content;
24
25 public:
26     Document(const std::string& t, const std::string& c)
27         : title(t), content(c) {}
28
29     void print() const override {
30         std::cout << "=== " << title << " ===" << std::endl;
31         std::cout << content << std::endl;
32     }
33
34     std::string serialize() const override {
35         return title + "|" + content;
36     }
37
38     void deserialize(const std::string& data) override {
39         size_t pos = data.find('|');
40         title = data.substr(0, pos);
41         content = data.substr(pos + 1);
42     }
43 };
44
45 int main() {
46     Document doc("My Document", "This is the content.");
47
48     // Use through interface pointers
49     Printable* printable = &doc;
50     printable->print();
51
52     Serializable* serializable = &doc;
53     std::string data = serializable->serialize();
54     std::cout << "Serialized: " << data << std::endl;
55
56     return 0;
57 }
```

Listing 3.3: Abstract class as interface

Key Point

Abstract Classes:

- Contain at least one pure virtual function
- Cannot be instantiated directly
- Can have constructors (used by derived classes)
- Can have data members
- Can have regular member functions
- Serve as interfaces or base classes
- Can have pointers/references to abstract class type

3.3.2 Abstract Class Constructors

```
#include <iostream>
#include <string>

class Animal {
protected:
    std::string name;
    int age;

public:
    // Constructor for abstract class
    Animal(const std::string& n, int a) : name(n), age(a) {
        std::cout << "Animal constructor called for " << name << std::endl;
    }

    // Pure virtual function
    virtual void makeSound() const = 0;

    // Regular member function
    void displayInfo() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }

    virtual ~Animal() {
        std::cout << "Animal destructor called for " << name << std::endl;
    }
};

class Dog : public Animal {
public:
    Dog(const std::string& n, int a) : Animal(n, a) {
        std::cout << "Dog constructor called" << std::endl;
    }

```

```
34      void makeSound() const override {
35          std::cout << name << " says: Woof!" << std::endl;
36      }
37
38      ~Dog() {
39          std::cout << "Dog destructor called" << std::endl;
40      }
41  };
42
43  class Cat : public Animal {
44  public:
45      Cat(const std::string& n, int a) : Animal(n, a) {
46          std::cout << "Cat constructor called" << std::endl;
47      }
48
49      void makeSound() const override {
50          std::cout << name << " says: Meow!" << std::endl;
51      }
52
53      ~Cat() {
54          std::cout << "Cat destructor called" << std::endl;
55      }
56  };
57
58  int main() {
59      std::cout << "Creating Dog:" << std::endl;
60      Animal* dog = new Dog("Buddy", 5);
61      dog->displayInfo();
62      dog->makeSound();
63
64      std::cout << "\nCreating Cat:" << std::endl;
65      Animal* cat = new Cat("Whiskers", 3);
66      cat->displayInfo();
67      cat->makeSound();
68
69      std::cout << "\nDeleting animals:" << std::endl;
70      delete dog;
71      delete cat;
72
73      return 0;
74  }
75
76  /* Output:
77  Creating Dog:
78  Animal constructor called for Buddy
79  Dog constructor called
80  Name: Buddy, Age: 5
81  Buddy says: Woof!
82
83  Creating Cat:
84  Animal constructor called for Whiskers
85  Cat constructor called
86  Name: Whiskers, Age: 3
87  Whiskers says: Meow!
88
89  Deleting animals:
90  Dog destructor called
91  Animal destructor called for Buddy
```

```
Cat destructor called
Animal destructor called for Whiskers
*/
```

Listing 3.4: Abstract class with constructor

Key Point

Purpose of Abstract Class Constructors:

- Initialize base class members
- Used by derived class constructors
- Cannot be called directly (class is abstract)
- Part of the derived class construction process
- Ensure proper initialization of inherited data

3.3.3 Practical Example: Plugin Architecture

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>

// Abstract plugin interface
class Plugin {
protected:
    std::string pluginName;
    std::string version;

public:
    Plugin(const std::string& name, const std::string& ver)
        : pluginName(name), version(ver) {}

    // Pure virtual functions
    virtual bool initialize() = 0;
    virtual void execute() = 0;
    virtual void shutdown() = 0;

    // Regular member functions
    std::string getName() const { return pluginName; }
    std::string getVersion() const { return version; }

    virtual ~Plugin() = default;
};

// Concrete plugin implementations
class LoggerPlugin : public Plugin {
public:
    LoggerPlugin() : Plugin("Logger", "1.0") {}

    bool initialize() override {
        std::cout << "Logger plugin initialized" << std::endl;
```

```
35          return true;
36      }
37
38      void execute() override {
39          std::cout << "[LOG] Plugin is running" << std::endl;
40      }
41
42      void shutdown() override {
43          std::cout << "Logger plugin shut down" << std::endl;
44      }
45  };
46
47  class DatabasePlugin : public Plugin {
48  public:
49      DatabasePlugin() : Plugin("Database", "2.1") {}
50
51      bool initialize() override {
52          std::cout << "Database plugin initialized" << std::endl;
53          return true;
54      }
55
56      void execute() override {
57          std::cout << "[DB] Executing database operations" << std::
        endl;
58      }
59
60      void shutdown() override {
61          std::cout << "Database plugin shut down" << std::endl;
62      }
63  };
64
65  // Plugin manager
66  class PluginManager {
67      std::vector<std::unique_ptr<Plugin>> plugins;
68
69  public:
70      void registerPlugin(std::unique_ptr<Plugin> plugin) {
71          std::cout << "Registering plugin: " << plugin->getName()
72                    << " v" << plugin->getVersion() << std::endl;
73          plugins.push_back(std::move(plugin));
74      }
75
76      void initializeAll() {
77          for (auto& plugin : plugins) {
78              plugin->initialize();
79          }
80      }
81
82      void executeAll() {
83          for (auto& plugin : plugins) {
84              plugin->execute();
85          }
86      }
87
88      void shutdownAll() {
89          for (auto& plugin : plugins) {
90              plugin->shutdown();
91          }
```

```
    }
};

int main() {
    PluginManager manager;

    manager.registerPlugin(std::make_unique<LoggerPlugin>());
    manager.registerPlugin(std::make_unique<DatabasePlugin>());

    std::cout << "\nInitializing plugins:" << std::endl;
    manager.initializeAll();

    std::cout << "\nExecuting plugins:" << std::endl;
    manager.executeAll();

    std::cout << "\nShutting down plugins:" << std::endl;
    manager.shutdownAll();

    return 0;
}
```

Listing 3.5: Abstract class for plugin system

Example

Real-World Uses of Abstract Classes:

- **GUI Frameworks**: Base `Widget` class with pure virtual `draw()`, `handleEvent()`
- **Game Engines**: Base `Entity` class with pure virtual `update()`, `render()`
- **Data Structures**: Base `Container` class with pure virtual `insert()`, `remove()`
- **Strategy Pattern**: Base `Strategy` class with pure virtual `execute()`
- **File Systems**: Base `FileSystem` class with pure virtual `read()`, `write()`

3.3.4 Partially Abstract Classes

```
#include <iostream>
#include <string>

class Vehicle {
protected:
    std::string brand;
    int year;

public:
    Vehicle(const std::string& b, int y) : brand(b), year(y) {}

    // Pure virtual - must override
```

```
13      virtual void start() = 0;
14      virtual void stop() = 0;
15      virtual double fuelEfficiency() const = 0;
16
17      // Concrete functions - can use as-is or override
18      virtual void displayInfo() const {
19          std::cout << year << " " << brand << std::endl;
20      }
21
22      virtual std::string getType() const {
23          return "Generic Vehicle";
24      }
25
26      virtual ~Vehicle() = default;
27  };
28
29  class Car : public Vehicle {
30      int doors;
31
32  public:
33      Car(const std::string& b, int y, int d)
34          : Vehicle(b, y), doors(d) {}
35
36      void start() override {
37          std::cout << "Car engine started with key" << std::endl;
38      }
39
40      void stop() override {
41          std::cout << "Car engine stopped" << std::endl;
42      }
43
44      double fuelEfficiency() const override {
45          return 25.5;  // mpg
46      }
47
48      // Override concrete function
49      std::string getType() const override {
50          return "Car";
51      }
52
53      // Use base class displayInfo() as-is
54  };
55
56  class ElectricCar : public Car {
57  public:
58      ElectricCar(const std::string& b, int y)
59          : Car(b, y, 4) {}
60
61      void start() override {
62          std::cout << "Electric car powered on silently" << std::
        endl;
63      }
64
65      double fuelEfficiency() const override {
66          return 100.0;  // MPGe (miles per gallon equivalent)
67      }
68
69      std::string getType() const override {
```

```
70          return "Electric Car";
71      }
72  };
73
74  int main() {
75      Vehicle* vehicles[2];
76      vehicles[0] = new Car("Toyota", 2020, 4);
77      vehicles[1] = new ElectricCar("Tesla", 2023);
78
79      for (int i = 0; i < 2; ++i) {
80          std::cout << "\nVehicle type: " << vehicles[i]->getType()
    << std::endl;
81          vehicles[i]->displayInfo();
82          vehicles[i]->start();
83          std::cout << "Fuel efficiency: " << vehicles[i]->
    fuelEfficiency()
84                    << " mpg" << std::endl;
85          vehicles[i]->stop();
86      }
87
88      delete vehicles[0];
89      delete vehicles[1];
90
91      return 0;
92  }
```

Listing 3.6: Class with some pure virtual and some concrete functions

Summary

Chapter 3 Summary:
Runtime Type Information (RTTI):

- `typeid` operator returns type information at runtime
- Returns `std::type_info` object
- `name()` method provides string representation (implementation-defined)
- Can compare types using `==` operator
- Requires `<typeinfo>` header

Pure Virtual Functions:

- Syntax: `virtual returnType functionName() = 0;`
- Force derived classes to provide implementation
- Make the containing class abstract
- Enable polymorphic behavior
- Can have implementation (rarely used)

Abstract Classes:

- Contain at least one pure virtual function
- Cannot be instantiated
- Can have constructors (for derived class use)
- Can have data members and regular functions
- Serve as interfaces or base classes
- Can have pointers/references to abstract type

Design Patterns Using Abstract Classes:

- **Interface Pattern**: Define pure interfaces
- **Template Method**: Mix pure virtual and concrete functions
- **Strategy Pattern**: Interchangeable algorithms
- **Plugin Architecture**: Extensible systems

Best Practices:

1. Always make destructors virtual in base classes
2. Use abstract classes to define interfaces
3. Initialize base class data in abstract class constructor
4. Keep interfaces simple and focused
5. Use `override` keyword for clarity

Part II

Standard Template Library (STL)

Chapter 4

STL forward_list Container

TL;DR

`forward_list` is a singly-linked list providing constant time O(1) insertion and deletion at the front. It's the most memory-efficient sequence container but only allows forward iteration. Use it when you need efficient insertion/removal and don't need bidirectional traversal or random access.

4.1 Introduction to forward_list

`forward_list` is a container that stores elements in non-contiguous memory locations, connected by single links.

4.1.1 Key Characteristics

Key Point

forward_list Properties:

- **Data Structure**: Singly-linked list
- **Memory**: Non-contiguous, dynamically allocated nodes
- **Iteration**: Forward only (no reverse iterators)
- **Size tracking**: No `size()` method (saves memory)
- **Header**: `<forward_list>`

4.2 Basic Operations

4.2.1 Creating and Initializing

```
1 #include <forward_list>
2 #include <iostream>
3
```

```cpp
int main() {
    // Default constructor
    std::forward_list<int> l;

    // Initialize with values
    std::forward_list<int> l2 = {10, 20, 30, 10};

    // Initialize with repeated value: assign(count, value)
    std::forward_list<int> l3;
    l3.assign(5, 10);  // {10, 10, 10, 10, 10}

    // Initialize from another forward_list
    std::forward_list<int> l4;
    l4.assign(l2.begin(), l2.end());

    // Print
    std::cout << "l3: ";
    for (int val : l3) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.1: forward_list initialization

4.2.2 Front Operations

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l;

    // Push at front: O(1)
    l.push_front(30);
    l.push_front(20);
    l.push_front(10);
    // List: 10 -> 20 -> 30

    // Pop from front: O(1)
    l.pop_front();
    // List: 20 -> 30

    std::cout << "After operations: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.2: Push and pop at front

Key Point

Time Complexities:

- `push_front()`: O(1)
- `pop_front()`: O(1)
- `assign()`: O(1) for single element, O(n) for n elements

4.3 Iterators and Traversal

4.3.1 Iterator-based Access

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l = {15, 20, 30};

    // Iterator declaration
    std::forward_list<int>::iterator it;

    // Range-based for loop (preferred)
    std::cout << "Range-based: ";
    for (auto value : l) {
        std::cout << value << " ";
    }
    std::cout << std::endl;

    // Iterator loop
    std::cout << "Iterator: ";
    for (it = l.begin(); it != l.end(); ++it) {
        std::cout << (*it) << " ";
    }
    std::cout << std::endl;

    // Auto with iterator
    std::cout << "Auto iterator: ";
    for (auto it = l.begin(); it != l.end(); ++it) {
        std::cout << (*it) << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.3: Using iterators with forward_list

Warning

`forward_list` does NOT support:

- `rbegin()`, `rend()` – no reverse iteration
- `--it` – no backward iteration
- `size()` – no size tracking (use `std::distance()`)
- Random access like `list[i]`

4.4 Insertion Operations

4.4.1 insert_after() Method

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l1 = {15, 20, 30};

    // insert_after(iterator, value): O(1)
    // Insert 10 after first element (15)
    auto it = l1.insert_after(l1.begin(), 10);
    // List: 15 -> 10 -> 20 -> 30
    // Returns iterator to newly inserted element (10)

    // Insert multiple copies: insert_after(iterator, count, value)
    it = l1.insert_after(it, 2, 53);
    // Inserts two 53's after the 10
    // List: 15 -> 10 -> 53 -> 53 -> 20 -> 30

    // Insert from initializer list: insert_after(iterator, {values})
    it = l1.insert_after(it, {2, 3});
    // List: 15 -> 10 -> 53 -> 2 -> 3 -> 53 -> 20 -> 30

    std::cout << "Final list: ";
    for (int val : l1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.4: Inserting elements after a position

4.4.2 emplace_after() Method

```cpp
#include <forward_list>
#include <iostream>
```

```cpp
#include <string>

class Person {
public:
    std::string name;
    int age;

    Person(const std::string& n, int a) : name(n), age(a) {
        std::cout << "Person constructed: " << name << std::endl;
    }
};

int main() {
    std::forward_list<int> l1 = {15, 10, 2, 3, 53};
    auto it = l1.begin();
    std::advance(it, 3);  // Move to 4th position

    // emplace_after: constructs in place
    it = l1.emplace_after(it, 40);
    // List: 15 -> 10 -> 2 -> 3 -> 40 -> 53

    // With custom objects
    std::forward_list<Person> people;
    people.emplace_front("Alice", 25);  // Constructs Person
   directly
    people.emplace_front("Bob", 30);

    std::cout << "\nPeople: ";
    for (const auto& p : people) {
        std::cout << p.name << "(" << p.age << ") ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.5: Efficient in-place construction

Key Point

emplace_after vs insert_after:

- `emplace_after()`: Constructs object in-place (no copy/move)
- `insert_after()`: Copies or moves existing object
- Use `emplace_after()` for efficiency with complex objects
- Can pass constructor arguments directly to `emplace_after()`

4.5 Removal Operations

4.5.1 Erasing Elements

```cpp
#include <forward_list>
```

```cpp
#include <iostream>

int main() {
    std::forward_list<int> l1 = {15, 10, 2, 3, 53, 40, 30};

    // erase_after(iterator): removes element AFTER iterator
    // Returns iterator to element after erased one
    auto it = l1.begin();
    std::advance(it, 4);  // Point to 53

    it = l1.erase_after(it);  // Removes 40
    // List: 15 -> 10 -> 2 -> 3 -> 53 -> 30
    std::cout << "After erase_after: ";
    for (int val : l1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // erase_after(iterator1, iterator2): removes range (iterator1, iterator2)
    it = l1.begin();
    std::advance(it, 1);  // Point to 10
    auto it2 = it;
    std::advance(it2, 3);  // Point to 53
    l1.erase_after(it, it2);  // Removes elements after 10 up to 53
    // List: 15 -> 10 -> 53 -> 30

    std::cout << "After range erase: ";
    for (int val : l1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.6: Removing elements

4.5.2 Removing by Value

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l = {10, 20, 10, 30, 10, 40};

    // remove(value): removes ALL instances of value, O(n)
    l.remove(10);
    // List: 20 -> 30 -> 40

    std::cout << "After remove(10): ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

```

```
    return 0;
}
```

Listing 4.7: Remove all instances of a value

4.5.3 Clearing All Elements

```
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l = {10, 20, 30};

    // Clear all elements
    l.clear();

    // Check if empty
    if (l.empty()) {
        std::cout << "List is empty" << std::endl;
    }

    return 0;
}
```

Listing 4.8: Clear the list

4.6 List Manipulation

4.6.1 Reversing

```
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l = {10, 20, 30};

    // reverse(): O(n)
    l.reverse();
    // List: 30 -> 20 -> 10

    std::cout << "Reversed: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.9: Reverse the list

4.6.2 Sorting

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l = {30, 10, 50, 20, 40};

    // sort(): O(n log n)
    l.sort();
    // List: 10 -> 20 -> 30 -> 40 -> 50

    std::cout << "Sorted: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Sort in descending order
    l.sort(std::greater<int>());
    // List: 50 -> 40 -> 30 -> 20 -> 10

    std::cout << "Sorted descending: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4.10: Sort the list

4.6.3 Merging

```cpp
#include <forward_list>
#include <iostream>

int main() {
    std::forward_list<int> l1 = {10, 30, 50};
    std::forward_list<int> l2 = {20, 40, 60};

    // merge(): transfers elements from l2 to l1
    // Both lists must be sorted
    // l2 becomes empty after merge
    l1.merge(l2);
    // l1: 10 -> 20 -> 30 -> 40 -> 50 -> 60
    // l2: (empty)

    std::cout << "Merged list: ";
    for (int val : l1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    std::cout << "l2 is empty: " << (l2.empty() ? "Yes" : "No") <<
    std::endl;

    return 0;
```

```
24 }
```

Listing 4.11: Merge two sorted lists

4.7 Time Complexity Summary

Operation	Time Complexity
`push_front()`	O(1)
`pop_front()`	O(1)
`assign(n, value)`	O(n)
`insert_after()`	O(1)
`emplace_after()`	O(1)
`erase_after()`	O(1)
`erase_after(it1, it2)`	O(distance)
`remove(value)`	O(n)
`clear()`	O(n)
`empty()`	O(1)
`reverse()`	O(n)
`sort()`	O(n log n)
`merge()`	O(n + m)

Table 4.1: forward_list time complexities

Summary

Chapter 4 Summary:
forward_list Characteristics:

- Singly-linked list (forward iteration only)
- No `size()` method (saves memory)
- Efficient front operations: O(1)
- No random access

Key Operations:

- `push_front()`, `pop_front()`: Front access
- `insert_after()`, `emplace_after()`: Insertion
- `erase_after()`: Removal at position
- `remove()`: Remove all instances of value
- `reverse()`: Reverse the list
- `sort()`: Sort in-place
- `merge()`: Merge sorted lists

When to Use forward_list:

- Need memory-efficient linked list
- Primarily insert/delete at front
- Don't need bidirectional iteration
- Don't need size tracking
- Space is more critical than features

When NOT to Use:

- Need random access (use `vector`)
- Need to know size frequently (use `list`)
- Need bidirectional iteration (use `list`)
- Need frequent insertion at back (use `deque` or `list`)

Chapter 5

STL list Container

TL;DR

`list` is a doubly-linked list providing constant time O(1) insertion and deletion at both ends. Unlike `forward_list`, it supports bidirectional iteration and has a `size()` method. Use when you need efficient insertion/removal at any position and bidirectional traversal.

5.1 Introduction to list

`list` is a container that maintains a doubly-linked sequence of elements.

5.1.1 Key Differences from forward_list

Key Point

list vs forward_list:

- **Links**: Doubly-linked (prev + next) vs singly-linked (next only)
- **Iteration**: Bidirectional vs forward-only
- **Size**: Has `size()` method vs no size tracking
- **Back operations**: `push_back()`, `pop_back()` available
- **Memory**: Slightly more per node (extra pointer)
- **Header**: `<list>`

5.2 Basic Operations

5.2.1 Front and Back Operations

```
1 #include <list>
2 #include <iostream>
3
```

```cpp
int main() {
    std::list<int> l;

    // Push at front: O(1)
    l.push_front(20);
    l.push_front(10);
    // List: 10 <-> 20

    // Push at back: O(1)
    l.push_back(30);
    l.push_back(40);
    // List: 10 <-> 20 <-> 30 <-> 40

    // Pop from front: O(1)
    l.pop_front();
    // List: 20 <-> 30 <-> 40

    // Pop from back: O(1)
    l.pop_back();
    // List: 20 <-> 30

    std::cout << "List: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 5.1: List front and back operations

5.2.2 Accessing Elements

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {10, 20, 30};

    // front(): reference to first element, O(1)
    int& first = l.front();
    std::cout << "First: " << first << std::endl;

    // back(): reference to last element, O(1)
    int& last = l.back();
    std::cout << "Last: " << last << std::endl;

    // Modify through reference
    first = 100;
    last = 300;
    // List: 100 <-> 20 <-> 300

    std::cout << "Modified list: ";
    for (int val : l) {
        std::cout << val << " ";
    }
```

```
    std::cout << std::endl;

    return 0;
}
```

Listing 5.2: Access front and back elements

5.2.3 Size Operations

```
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {10, 20, 30, 40};

    // size(): returns number of elements, O(1)
    std::cout << "Size: " << l.size() << std::endl;

    // empty(): check if list is empty, O(1)
    if (!l.empty()) {
        std::cout << "List is not empty" << std::endl;
    }

    return 0;
}
```

Listing 5.3: Size and empty

5.3 Iterators

5.3.1 Bidirectional Iteration

```
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {10, 20, 30, 40};

    // Forward iteration
    std::cout << "Forward: ";
    for (auto it = l.begin(); it != l.end(); ++it) {
        std::cout << (*it) << " ";
    }
    std::cout << std::endl;

    // Reverse iteration
    std::cout << "Reverse: ";
    for (auto it = l.rbegin(); it != l.rend(); ++it) {
        std::cout << (*it) << " ";
    }
    std::cout << std::endl;

    // Move iterator backward
    auto it = l.end();
```

```
    --it;  // Point to last element
    std::cout << "Last element: " << *it << std::endl;

    return 0;
}
```

Listing 5.4: Forward and reverse iteration

5.4 Insertion Operations

5.4.1 insert() Method

```
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {10, 20, 30};

    // insert(iterator, value): insert BEFORE iterator
    // Returns iterator to newly inserted element
    auto it = l.begin();
    ++it;  // Point to 20

    it = l.insert(it, 15);
    // List: 10 <-> 15 <-> 20 <-> 30
    std::cout << "Inserted value: " << *it << std::endl;

    // insert(iterator, count, value): insert count copies
    it = l.begin();
    ++it; ++it;  // Point to 20
    l.insert(it, 2, 7);
    // List: 10 <-> 15 <-> 7 <-> 7 <-> 20 <-> 30

    // insert(iterator, {values}): insert from initializer list
    it = l.begin();
    l.insert(it, {1, 2, 3});
    // List: 1 <-> 2 <-> 3 <-> 10 <-> 15 <-> 7 <-> 7 <-> 20 <-> 30

    std::cout << "Final list: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 5.5: Inserting at iterator position

Key Point

insert() vs insert_after():

- `list::insert(it, val)`: Inserts **before** iterator
- `forward_list::insert_after(it, val)`: Inserts **after** iterator
- Reason: Singly-linked list cannot efficiently insert before

5.5 Removal Operations

5.5.1 erase() Method

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {10, 20, 30, 40, 50};

    // erase(iterator): removes element AT iterator position
    // Returns iterator to element after erased one
    auto it = l.begin();
    ++it; ++it;  // Point to 30

    it = l.erase(it);  // Removes 30, returns iterator to 40
    // List: 10 <-> 20 <-> 40 <-> 50
    std::cout << "After erase, iterator points to: " << *it << std
    ::endl;

    // erase(iterator1, iterator2): removes range [iterator1,
    iterator2)
    it = l.begin();
    ++it;  // Point to 20
    auto it2 = it;
    ++it2; ++it2;  // Point to 50
    l.erase(it, it2);  // Removes 20 and 40
    // List: 10 <-> 50

    std::cout << "After range erase: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 5.6: Erasing elements

5.5.2 remove() Method

```cpp
#include <list>
```

```cpp
#include <iostream>

int main() {
    std::list<int> l = {10, 40, 20, 40, 30, 40};

    // remove(value): removes ALL instances, O(n)
    l.remove(40);
    // List: 10 <-> 20 <-> 30

    std::cout << "After remove(40): ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 5.7: Remove all occurrences of a value

5.6 List Manipulation

5.6.1 Merging Lists

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l1 = {10, 20, 30};
    std::list<int> l2 = {15, 25, 35};

    // merge(): transfers elements from l2 to l1
    // Both must be sorted, l2 becomes empty
    l1.merge(l2);
    // l1: 10 <-> 15 <-> 20 <-> 25 <-> 30 <-> 35

    std::cout << "Merged: ";
    for (int val : l1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    std::cout << "l2 empty: " << (l2.empty() ? "Yes" : "No") <<
   std::endl;

    return 0;
}
```

Listing 5.8: Merge sorted lists

5.6.2 Removing Duplicates

```cpp
#include <list>
#include <iostream>
```

```cpp

int main() {
    std::list<int> l = {10, 10, 20, 20, 20, 30, 30, 10};

    // unique(): removes consecutive duplicates, O(n)
    l.unique();
    // List: 10 <-> 20 <-> 30 <-> 10
    // Note: Only consecutive duplicates removed!

    std::cout << "After unique(): ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // To remove ALL duplicates, sort first
    std::list<int> l2 = {10, 30, 20, 10, 30, 20};
    l2.sort();      // 10 <-> 10 <-> 20 <-> 20 <-> 30 <-> 30
    l2.unique();    // 10 <-> 20 <-> 30

    std::cout << "After sort + unique(): ";
    for (int val : l2) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 5.9: Remove consecutive duplicates

5.6.3 Reversing and Sorting

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<int> l = {30, 10, 50, 20, 40};

    // reverse(): O(n)
    l.reverse();
    std::cout << "Reversed: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // sort(): O(n log n)
    l.sort();
    std::cout << "Sorted: ";
    for (int val : l) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Sort with custom comparator
    l.sort(std::greater<int>());
```

```
25      std::cout << "Sorted descending: ";
26      for (int val : l) {
27          std::cout << val << " ";
28      }
29      std::cout << std::endl;
30
31      return 0;
32  }
```

Listing 5.10: Reverse and sort

5.7 Time Complexity Summary

Operation	Time Complexity
`push_front()`	O(1)
`push_back()`	O(1)
`pop_front()`	O(1)
`pop_back()`	O(1)
`front()`	O(1)
`back()`	O(1)
`size()`	O(1)
`empty()`	O(1)
`insert()`	O(1)
`erase()`	O(1)
`remove(value)`	O(n)
`clear()`	O(n)
`reverse()`	O(n)
`unique()`	O(n)
`sort()`	O(n log n)
`merge()`	O(n + m)

Table 5.1: list time complexities

Summary

Chapter 5 Summary:
list Characteristics:

- Doubly-linked list (bidirectional iteration)
- Has `size()` method
- Efficient operations at both ends
- No random access

Key Operations:

- `push_front()`, `push_back()`: Add elements
- `pop_front()`, `pop_back()`: Remove elements
- `front()`, `back()`: Access ends
- `insert()`: Insert before iterator
- `erase()`: Remove at iterator
- `remove()`: Remove all instances of value
- `unique()`: Remove consecutive duplicates
- `reverse()`, `sort()`, `merge()`

When to Use list:

- Frequent insertion/deletion at arbitrary positions
- Need bidirectional iteration
- Need size tracking
- Stable iterators (insertion/deletion doesn't invalidate)

list vs vector vs forward_list:

- **list**: Best for frequent insert/delete anywhere, bidirectional
- **vector**: Best for random access, infrequent insert/delete
- **forward_list**: Most memory-efficient, forward-only, no size()

Chapter 6

STL deque Container

TL;DR

`deque` (double-ended queue) provides fast insertion and deletion at both ends and random access to elements. It combines advantages of `vector` and `list`: can grow efficiently in both directions and supports indexed access. Use when you need queue/stack operations with random access capability.

6.1 Introduction to deque

Deque (pronounced "deck") stands for double-ended queue and provides indexed sequence container.

Key Point

deque Characteristics:

- Random access (like `vector`): `operator[]`, `at()`
- Fast insert/delete at both ends (unlike `vector`)
- Can act as stack or queue
- Non-contiguous storage (unlike `vector`)
- No capacity/reserve concept
- Header: `<deque>`

6.2 Basic Operations

```
#include <deque>
#include <iostream>

int main() {
    std::deque<int> dq;

```

```
7       // Push at both ends: O(1)
8       dq.push_back(30);
9       dq.push_back(40);
10      dq.push_front(20);
11      dq.push_front(10);
12      // Deque: 10, 20, 30, 40
13
14      // Random access: O(1)
15      std::cout << "Element at index 2: " << dq[2] << std::endl;  //
        30
16      std::cout << "Element at(1): " << dq.at(1) << std::endl;    //
        20
17
18      // Access ends: O(1)
19      std::cout << "Front: " << dq.front() << std::endl;  // 10
20      std::cout << "Back: " << dq.back() << std::endl;    // 40
21
22      // Size: O(1)
23      std::cout << "Size: " << dq.size() << std::endl;    // 4
24
25      // Pop from both ends: O(1)
26      dq.pop_front();  // Remove 10
27      dq.pop_back();   // Remove 40
28      // Deque: 20, 30
29
30      std::cout << "After pops: ";
31      for (int val : dq) {
32          std::cout << val << " ";
33      }
34      std::cout << std::endl;
35
36      return 0;
37  }
```

Listing 6.1: deque basic operations

Key Point

deque vs vector:

- **Similarity**: Random access O(1), similar interface
- **Difference**: deque has O(1) `push_front()`, vector is O(n)
- **Memory**: deque uses chunks, vector uses contiguous block
- **Use deque when**: Need fast insertion at both ends
- **Use vector when**: Need guaranteed contiguous memory

Summary

Chapter 6 Summary:
Deque Features:

- Double-ended queue with random access
- O(1) insertion/deletion at both ends
- O(1) random access via index
- Efficient growth in both directions
- Can be used as stack, queue, or random-access container

Chapter 7

STL stack Adapter

TL;DR

`stack` is a container adapter providing LIFO (Last-In-First-Out) behavior. It wraps an underlying container (by default `deque`) and restricts access to only the top element. All operations are O(1). Perfect for function call stacks, expression evaluation, and backtracking algorithms.

7.1 Introduction to stack

Stack is a container adapter, not a container itself. It provides a restricted interface to an underlying container.

Key Point

Stack Properties:

- **LIFO**: Last element added is first removed
- **Container Adapter**: Wraps deque, vector, or list
- **Default**: Uses `deque` as underlying container
- **Requirements**: back(), push_back(), pop_back(), size(), empty()
- **Header**: `<stack>`

7.2 Stack Operations

```cpp
#include <stack>
#include <iostream>

int main() {
    std::stack<int> st;

    // push(): Add element to top, O(1)
    st.push(10);
```

```
    st.push(20);
    st.push(30);
    // Stack (bottom to top): 10, 20, 30

    // top(): Access top element, O(1)
    std::cout << "Top element: " << st.top() << std::endl;  // 30

    // size(): Number of elements, O(1)
    std::cout << "Size: " << st.size() << std::endl;  // 3

    // pop(): Remove top element, O(1)
    st.pop();  // Removes 30
    std::cout << "After pop, top: " << st.top() << std::endl;  //
    20

    // empty(): Check if empty, O(1)
    if (!st.empty()) {
        std::cout << "Stack is not empty" << std::endl;
    }

    // Iterate (requires popping all elements)
    std::cout << "Stack contents (top to bottom): ";
    while (!st.empty()) {
        std::cout << st.top() << " ";
        st.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 7.1: Stack operations

7.2.1 Choosing Underlying Container

```
#include <stack>
#include <vector>
#include <list>
#include <iostream>

int main() {
    // Default: uses deque
    std::stack<int> st1;

    // Using vector as underlying container
    std::stack<int, std::vector<int>> st2;

    // Using list as underlying container
    std::stack<int, std::list<int>> st3;

    // All have same interface
    st1.push(10);
    st2.push(20);
    st3.push(30);

    std::cout << "st1 top: " << st1.top() << std::endl;
    std::cout << "st2 top: " << st2.top() << std::endl;
```

```
    std::cout << "st3 top: " << st3.top() << std::endl;

    return 0;
}
```

Listing 7.2: Stack with different underlying containers

7.2.2 Practical Example: Balanced Parentheses

```
#include <stack>
#include <string>
#include <iostream>

bool isBalanced(const std::string& expr) {
    std::stack<char> st;

    for (char ch : expr) {
        if (ch == '(' || ch == '{' || ch == '[') {
            st.push(ch);
        }
        else if (ch == ')' || ch == '}' || ch == ']') {
            if (st.empty()) {
                return false;  // No matching opening bracket
            }

            char top = st.top();
            st.pop();

            if ((ch == ')' && top != '(') ||
                (ch == '}' && top != '{') ||
                (ch == ']' && top != '[')) {
                return false;  // Mismatched brackets
            }
        }
    }

    return st.empty();  // True if all brackets matched
}

int main() {
    std::string test1 = "{[()]}";
    std::string test2 = "{[(])}";
    std::string test3 = "(((()))";

    std::cout << test1 << " is "
              << (isBalanced(test1) ? "balanced" : "not balanced")
              << std::endl;
    std::cout << test2 << " is "
              << (isBalanced(test2) ? "balanced" : "not balanced")
              << std::endl;
    std::cout << test3 << " is "
              << (isBalanced(test3) ? "balanced" : "not balanced")
              << std::endl;

    return 0;
```

```
47 }
```

Listing 7.3: Check balanced parentheses using stack

Summary

Chapter 7 Summary:
Stack Characteristics:

- LIFO container adapter
- Default underlying container: `deque`
- All operations O(1)
- No iterators (restricted access)

Operations:

- `push()`: Add to top
- `pop()`: Remove from top
- `top()`: Access top element
- `size()`, `empty()`

Use Cases:

- Function call stack
- Expression evaluation (postfix, infix)
- Balanced parentheses checking
- Undo mechanisms
- Backtracking algorithms (DFS, maze solving)

Chapter 8

STL queue Adapter

TL;DR

`queue` is a container adapter providing FIFO (First-In-First-Out) behavior. Elements are added at the back and removed from the front. By default uses `deque`. All operations are O(1). Perfect for scheduling, BFS algorithms, and buffering.

8.1 Introduction to queue

Queue is a container adapter that provides FIFO access pattern.

Key Point

Queue Properties:

- **FIFO**: First element added is first removed
- **Container Adapter**: Wraps deque or list
- **Default**: Uses `deque` as underlying container
- **Requirements**: front(), back(), push_back(), pop_front(), size(), empty()
- **Cannot use vector**: No efficient `pop_front()`
- **Header**: `<queue>`

8.2 Queue Operations

```
#include <queue>
#include <iostream>

int main() {
    std::queue<int> q;

```

```
    // push(): Add element to back, O(1)
    q.push(10);
    q.push(20);
    q.push(30);
    // Queue: front[10, 20, 30]back

    // front(): Access front element, O(1)
    std::cout << "Front: " << q.front() << std::endl;  // 10

    // back(): Access back element, O(1)
    std::cout << "Back: " << q.back() << std::endl;    // 30

    // size(): Number of elements, O(1)
    std::cout << "Size: " << q.size() << std::endl;    // 3

    // pop(): Remove front element, O(1)
    q.pop();  // Removes 10
    std::cout << "After pop, front: " << q.front() << std::endl;
   // 20

    // empty(): Check if empty, O(1)
    if (!q.empty()) {
        std::cout << "Queue is not empty" << std::endl;
    }

    // Process all elements (FIFO order)
    std::cout << "Queue contents (FIFO): ";
    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 8.1: Queue operations

8.2.1 Choosing Underlying Container

```
#include <queue>
#include <list>
#include <iostream>

int main() {
    // Default: uses deque
    std::queue<int> q1;

    // Using list as underlying container
    std::queue<int, std::list<int>> q2;

    // Both have same interface
    q1.push(10);
    q1.push(20);

    q2.push(30);
    q2.push(40);
```

```
18
19      std::cout << "q1 front: " << q1.front() << std::endl;  // 10
20      std::cout << "q2 front: " << q2.front() << std::endl;  // 30
21
22      return 0;
23  }
```

Listing 8.2: Queue with different containers

8.2.2 Practical Example: Task Scheduler

```
1  #include <queue>
2  #include <string>
3  #include <iostream>
4
5  struct Task {
6      std::string name;
7      int priority;
8
9      Task(const std::string& n, int p) : name(n), priority(p) {}
10 };
11
12 int main() {
13     std::queue<Task> taskQueue;
14
15     // Add tasks to queue
16     taskQueue.push(Task("Task1", 1));
17     taskQueue.push(Task("Task2", 2));
18     taskQueue.push(Task("Task3", 1));
19     taskQueue.push(Task("Task4", 3));
20
21     std::cout << "Processing tasks in FIFO order:" << std::endl;
22
23     // Process tasks in order
24     while (!taskQueue.empty()) {
25         Task current = taskQueue.front();
26         std::cout << "Processing: " << current.name
27                   << " (Priority: " << current.priority << ")"
28                   << std::endl;
29         taskQueue.pop();
30     }
31
32     return 0;
33 }
```

Listing 8.3: Simple task scheduler using queue

Summary

Chapter 8 Summary:
Queue Characteristics:

- FIFO container adapter
- Default underlying container: `deque`
- All operations O(1)
- Access both front and back

Operations:

- `push()`: Add to back
- `pop()`: Remove from front
- `front()`: Access front element
- `back()`: Access back element
- `size()`, `empty()`

Use Cases:

- Task scheduling
- Breadth-first search (BFS)
- Print queue management
- Request buffering
- Message queues

Chapter 9

STL priority_queue Adapter

TL;DR

`priority_queue` is a heap-based container adapter where the largest (or smallest) element is always at the top. By default implements a max-heap. Push and pop operations are O(log n), top access is O(1). Perfect for scheduling with priorities, Dijkstra's algorithm, and Huffman coding.

9.1 Introduction to priority_queue

Priority queue maintains elements in a heap structure where the highest priority element is always accessible.

Key Point

Priority Queue Properties:

- **Heap**: Binary max-heap by default (largest on top)
- **Container Adapter**: Wraps vector or deque
- **Default**: Uses `vector` with `less<T>` comparator
- **Requirements**: front(), push_back(), pop_back(), random access
- **Header**: `<queue>`

9.2 Basic Operations

```cpp
#include <queue>
#include <iostream>

int main() {
    std::priority_queue<int> pq;

    // push(): Insert element, O(log n)
    pq.push(30);
```

```
    pq.push(10);
    pq.push(50);
    pq.push(20);
    // Heap maintains largest at top

    // top(): Access largest element, O(1)
    std::cout << "Top (max) element: " << pq.top() << std::endl;
   // 50

    // size(): Number of elements, O(1)
    std::cout << "Size: " << pq.size() << std::endl;  // 4

    // pop(): Remove largest element, O(log n)
    pq.pop();  // Removes 50
    std::cout << "After pop, top: " << pq.top() << std::endl;  //
   30

    // empty(): Check if empty, O(1)
    if (!pq.empty()) {
        std::cout << "Priority queue is not empty" << std::endl;
    }

    // Extract all elements in priority order
    std::cout << "Elements in priority order: ";
    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << std::endl;  // Output: 30 20 10

    return 0;
}
```

Listing 9.1: Priority queue operations - Max Heap

9.3 Min-Heap Priority Queue

9.3.1 Using greater<T> for Min-Heap

```
#include <queue>
#include <vector>
#include <iostream>

int main() {
    // Min-heap: smallest element at top
    // Syntax: priority_queue<Type, Container, Comparator>
    std::priority_queue<int, std::vector<int>, std::greater<int>>
   pq;

    pq.push(30);
    pq.push(10);
    pq.push(50);
    pq.push(20);

    std::cout << "Top (min) element: " << pq.top() << std::endl;
   // 10
```

```
16
17      std::cout << "Min-heap elements: ";
18      while (!pq.empty()) {
19          std::cout << pq.top() << " ";
20          pq.pop();
21      }
22      std::cout << std::endl;  // Output: 10 20 30 50
23
24      return 0;
25  }
```

Listing 9.2: Priority queue as Min Heap

9.3.2 Alternative: Negate Values

```
1  #include <queue>
2  #include <iostream>
3
4  int main() {
5      std::priority_queue<int> pq;  // Max-heap
6
7      int arr[] = {30, 10, 50, 20};
8      int n = 4;
9
10     // Insert negated values
11     for (int i = 0; i < n; i++) {
12         pq.push(-arr[i]);
13     }
14
15     std::cout << "Min values (by negation): ";
16     while (!pq.empty()) {
17         std::cout << -pq.top() << " ";  // Negate back
18         pq.pop();
19     }
20     std::cout << std::endl;  // Output: 10 20 30 50
21
22     return 0;
23 }
```

Listing 9.3: Min-heap by negating values

9.4 Creating from Existing Array

```
1  #include <queue>
2  #include <vector>
3  #include <iostream>
4
5  int main() {
6      int arr[] = {30, 10, 50, 20, 40};
7      int n = 5;
8
9      // Create from array (uses pointers as iterators)
10     std::priority_queue<int> pq(arr, arr + n);
11
```

```
    std::cout << "Priority queue from array: ";
    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << std::endl;  // Output: 50 40 30 20 10

    return 0;
}
```

Listing 9.4: Initialize priority queue from array

9.5 Custom Comparator

9.5.1 Priority Queue with Custom Objects

```
#include <queue>
#include <vector>
#include <string>
#include <iostream>

struct Person {
    std::string name;
    int age;

    Person(const std::string& n, int a) : name(n), age(a) {}
};

// Custom comparator: older person has higher priority
struct AgeComparator {
    bool operator()(const Person& p1, const Person& p2) {
        // Return true if p1 has lower priority than p2
        return p1.age < p2.age;  // Max-heap based on age
    }
};

int main() {
    std::priority_queue<Person, std::vector<Person>, AgeComparator
    > pq;

    pq.push(Person("Alice", 25));
    pq.push(Person("Bob", 30));
    pq.push(Person("Charlie", 20));
    pq.push(Person("David", 35));

    std::cout << "People by age (oldest first):" << std::endl;
    while (!pq.empty()) {
        Person p = pq.top();
        std::cout << p.name << " (age " << p.age << ")" << std::
    endl;
        pq.pop();
    }

    return 0;
}
\</lstlisting>
```

```
39
40 \section{Applications}
41
42 \subsection{Dijkstra's Algorithm}
43
44 \begin{example}
45 \textbf{Priority Queue in Dijkstra's Shortest Path:}
46
47 Priority queue stores vertices with their distances. Always
      processes vertex with minimum distance first.
48
49 \begin{lstlisting}
50 std::priority_queue<
51     std::pair<int, int>,  // {distance, vertex}
52     std::vector<std::pair<int, int>>,
53     std::greater<std::pair<int, int>>  // Min-heap
54 > pq;
55
56 pq.push({0, source});  // Start with source vertex
57 while (!pq.empty()) {
58     auto [dist, u] = pq.top();
59     pq.pop();
60     // Process vertex u with distance dist
61 }
```

Listing 9.5: Custom comparator for priority queue

Summary

Chapter 9 Summary:
Priority Queue Characteristics:

- Heap-based container adapter
- Max-heap by default (largest element on top)
- Can be min-heap using `greater<T>`
- Underlying container: typically `vector`
- No iteration support

Operations:

- `push()`: Insert element, O(log n)
- `pop()`: Remove top, O(log n)
- `top()`: Access top, O(1)
- `size()`, `empty()`: O(1)

Creating Min-Heap:

1. Use `std::greater<T>` comparator
2. Or negate values when inserting/extracting

Key Algorithms:

- Dijkstra's shortest path
- Prim's MST
- Huffman coding
- Heap sort
- Top-K problems

Chapter 10

STL set Container

TL;DR

`set` stores unique elements in sorted order using a Red-Black tree (balanced BST). All operations are O(log n). Elements are automatically sorted and duplicates are rejected. Perfect for maintaining sorted unique collections with fast lookup, insertion, and deletion.

10.1 Introduction to set

Set is an associative container that stores unique elements in sorted order.

Key Point

Set Characteristics:

- **Internal Structure**: Red-Black tree (self-balancing BST)
- **Height**: O(log n)
- **Ordering**: Elements sorted (ascending by default)
- **Uniqueness**: No duplicate elements
- **Immutable keys**: Cannot modify elements directly
- **Header**: `<set>`

10.2 Basic Operations

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

int main() {
    // Default: ascending order
    std::set<int> s;

    // Insert elements: O(log n)
```

```
    s.insert(30);
    s.insert(10);
    s.insert(20);
    s.insert(10);  // Duplicate - ignored

    // Set contains: 10, 20, 30 (sorted, no duplicate 10)

    std::cout << "Set elements: ";
    for (int val : s) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // Descending order set
    std::set<int, std::greater<int>> s2;
    s2.insert(30);
    s2.insert(10);
    s2.insert(20);

    std::cout << "Descending set: ";
    for (int val : s2) {
        std::cout << val << " ";  // 30, 20, 10
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 10.1: Set basic operations

10.3 Search Operations

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

int main() {
    std::set<int> s = {10, 20, 30, 40, 50};

    // find(): returns iterator to element or end(), O(log n)
    auto it = s.find(30);
    if (it != s.end()) {
        std::cout << "Found: " << *it << std::endl;
    }

    it = s.find(35);
    if (it == s.end()) {
        std::cout << "35 not found" << std::endl;
    }

    // count(): returns 0 or 1 (set has unique elements), O(log n)
    if (s.count(20)) {
        std::cout << "20 is present" << std::endl;
    }

    return 0;
```

```
}
```

Listing 10.2: Finding elements in set

10.4 Bounds Operations

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

int main() {
    std::set<int> s = {10, 20, 30, 40, 50};

    // lower_bound(val): returns iterator to first element >= val, O(log n)
    auto it = s.lower_bound(30);
    std::cout << "lower_bound(30): " << *it << std::endl;  // 30

    it = s.lower_bound(35);
    std::cout << "lower_bound(35): " << *it << std::endl;  // 40

    it = s.lower_bound(60);
    if (it == s.end()) {
        std::cout << "lower_bound(60): end()" << std::endl;
    }

    // upper_bound(val): returns iterator to first element > val, O(log n)
    it = s.upper_bound(30);
    std::cout << "upper_bound(30): " << *it << std::endl;  // 40

    it = s.upper_bound(35);
    std::cout << "upper_bound(35): " << *it << std::endl;  // 40

    return 0;
}
```

Listing 10.3: lower_bound and upper_bound

10.5 Removal Operations

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

int main() {
    std::set<int> s = {10, 20, 30, 40, 50};

    // erase(value): removes element, O(log n)
    s.erase(30);

    // erase(iterator): O(1) amortized
    auto it = s.find(40);
    if (it != s.end()) {
        s.erase(it);
```

```
    }

    // erase(it1, it2): erase range [it1, it2)
    // Set: 10, 20, 50

    // clear(): remove all elements
    s.clear();

    // empty(): check if empty, O(1)
    if (s.empty()) {
        std::cout << "Set is empty" << std::endl;
    }

    return 0;
}
```

Listing 10.4: Erasing elements from set

10.6 Iterators

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

int main() {
    std::set<int> s = {10, 20, 30, 40, 50};

    // begin(), end(): O(1)
    std::cout << "Forward: ";
    for (auto it = s.begin(); it != s.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // rbegin(), rend(): reverse iteration, O(1)
    std::cout << "Reverse: ";
    for (auto it = s.rbegin(); it != s.rend(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // cbegin(), cend(): const iterators
    // Cannot modify through const iterator

    // size(): O(1)
    std::cout << "Size: " << s.size() << std::endl;

    return 0;
}
```

Listing 10.5: Set iterators

Warning

Set elements are immutable!
You cannot modify elements directly because it would break the sorted order:

```
std::set<int> s = {10, 20, 30};
auto it = s.begin();
// *it = 25;  // ERROR! Cannot modify through iterator
```

To "modify" an element: erase it and insert new value.

10.7 Time Complexity Summary

Operation	Time Complexity
`insert()`	O(log n)
`find()`	O(log n)
`count()`	O(log n)
`lower_bound()`	O(log n)
`upper_bound()`	O(log n)
`erase(value)`	O(log n)
`erase(iterator)`	O(1) amortized
`begin()`, `end()`	O(1)
`size()`, `empty()`	O(1)
`clear()`	O(n)

Table 10.1: set time complexities

Summary

Chapter 10 Summary:
Set Features:

- Red-Black tree implementation
- Sorted unique elements
- O(log n) operations
- Bidirectional iterators
- Immutable elements

When to Use set:

- Need sorted unique collection
- Frequent search operations
- Dynamic insertion/deletion with ordering
- Range queries (lower/upper bound)

Use Cases:

- Remove duplicates while maintaining order
- Membership testing with sorting
- Finding closest elements
- Interval operations

Chapter 11

STL multiset Container

TL;DR

`multiset` is like `set` but allows duplicate elements. Still uses Red-Black tree and maintains sorted order. `count()` can return values > 1. Perfect when you need sorted collection with duplicates.

11.1 Introduction to multiset

Multiset allows multiple instances of the same value while maintaining sorted order.

Key Point

multiset vs set:

- **Duplicates**: multiset allows, set doesn't
- **count()**: Can return > 1 in multiset
- **erase(value)**: Removes all instances in multiset
- **Internal structure**: Both use Red-Black tree
- **Header**: `<set>`

11.2 Basic Operations

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

int main() {
    std::multiset<int> ms;

    // Insert duplicates: O(log n)
    ms.insert(10);
    ms.insert(20);
    ms.insert(10);
```

```
    ms.insert(30);
    ms.insert(10);

    // Multiset: 10, 10, 10, 20, 30 (sorted)

    std::cout << "Multiset elements: ";
    for (int val : ms) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // count(): returns number of occurrences, O(log n + count)
    std::cout << "Count of 10: " << ms.count(10) << std::endl;  // 3

    return 0;
}
```

Listing 11.1: Multiset with duplicates

11.3 Removal in Multiset

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

int main() {
    std::multiset<int> ms = {10, 20, 10, 30, 10};

    // erase(value): removes ALL instances, O(log n + count)
    ms.erase(10);
    // Multiset: 20, 30

    std::cout << "After erase(10): ";
    for (int val : ms) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    // To erase only ONE instance: use iterator
    ms = {10, 20, 10, 30, 10};
    auto it = ms.find(10);  // Find first occurrence
    if (it != ms.end()) {
        ms.erase(it);  // Remove only this one, O(1) amortized
    }
    // Multiset: 10, 10, 20, 30

    std::cout << "After erasing one instance: ";
    for (int val : ms) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 11.2: Erasing from multiset

11.4 Bounds with Duplicates

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

int main() {
    std::multiset<int> ms = {10, 10, 10, 20, 30};

    // lower_bound(): first occurrence, O(log n)
    auto it_lower = ms.lower_bound(10);
    std::cout << "lower_bound(10): " << *it_lower << std::endl;
   // 10

    // upper_bound(): element after last occurrence, O(log n)
    auto it_upper = ms.upper_bound(10);
    std::cout << "upper_bound(10): " << *it_upper << std::endl;
   // 20

    // equal_range(): returns pair of (lower_bound, upper_bound)
    auto range = ms.equal_range(10);
    std::cout << "equal_range(10): ";
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << *it << " ";  // 10 10 10
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 11.3: lower_bound and upper_bound with duplicates

Key Point

equal_range():

- Returns `std::pair<iterator, iterator>`
- `first`: lower_bound (first occurrence)
- `second`: upper_bound (after last occurrence)
- Range [first, second) contains all instances of value

Summary

Chapter 11 Summary:
Multiset Features:

- Sorted collection with duplicates
- Red-Black tree implementation
- `count()` returns number of occurrences
- `erase(value)` removes all instances
- `equal_range()` finds all instances

When to Use multiset:

- Need sorted collection with duplicates
- Counting occurrences
- Maintaining frequency in sorted order

Chapter 12

STL map Container

TL;DR

`map` stores key-value pairs with unique keys in sorted order (by key). Uses Red-Black tree. O(log n) operations. Provides associative array with automatic sorting. Perfect for dictionaries, lookup tables, and sorted mappings.

12.1 Introduction to map

Map is an associative container storing key-value pairs with unique keys.

Key Point

Map Characteristics:

- **Structure**: Red-Black tree (sorted by key)
- **Elements**: `std::pair<const Key, Value>`
- **Unique keys**: One value per key
- **Access**: `operator[]`, `at()`, iterators
- **Ordering**: Sorted by key (ascending by default)
- **Header**: `<map>`

12.2 Insertion and Access

```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<int, int> m;

    // insert(): O(log n)
    m.insert({10, 200});
```

```
    m.insert({20, 300});
    m.insert({10, 400});  // Ignored - key 10 already exists

    // Map: {10: 200, 20: 300}

    // operator[]: O(log n)
    // If key doesn't exist, creates with default value
    std::cout << "m[20]: " << m[20] << std::endl;  // 300
    std::cout << "m[30]: " << m[30] << std::endl;  // 0 (default int value)
    // Map now: {10: 200, 20: 300, 30: 0}

    // Modify using operator[]
    m[10] = 500;
    m[30] = 600;
    // Map: {10: 500, 20: 300, 30: 600}

    std::cout << "Map contents:" << std::endl;
    for (const auto& pair : m) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

Listing 12.1: Map insertion methods

12.3 at() Method

```
#include <map>
#include <iostream>
#include <stdexcept>

int main() {
    std::map<int, int> m = {{10, 200}, {20, 300}};

    // at(): O(log n), throws exception if key doesn't exist
    try {
        std::cout << "m.at(10): " << m.at(10) << std::endl;  // 200
        std::cout << "m.at(30): " << m.at(30) << std::endl;  // Throws!
    }
    catch (const std::out_of_range& ex) {
        std::cout << "Exception: " << ex.what() << std::endl;
    }

    // at() for modification
    m.at(10) = 250;
    std::cout << "After m.at(10) = 250: " << m.at(10) << std::endl;

    return 0;
```

```
}
```

Listing 12.2:]at() vs operator[]

Key Point

operator[] vs at():

- **operator[]**: Creates key with default value if missing
- **at()**: Throws `std::out_of_range` if key missing
- **Use []**: When default initialization is acceptable
- **Use at()**: For safety, to detect missing keys

12.4 Search Operations

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> m = {
        {"apple", 5},
        {"banana", 3},
        {"cherry", 7}
    };

    // find(): returns iterator to key-value pair, O(log n)
    auto it = m.find("banana");
    if (it != m.end()) {
        std::cout << "Found: " << it->first << " = "
                  << it->second << std::endl;
    }

    // count(): returns 0 or 1 (unique keys), O(log n)
    if (m.count("apple")) {
        std::cout << "apple is present" << std::endl;
    }

    if (!m.count("mango")) {
        std::cout << "mango is not present" << std::endl;
    }

    return 0;
}
```

Listing 12.3: Finding in map

12.5 Bounds Operations

```cpp
#include <map>
#include <iostream>

int main() {
    std::map<int, std::string> m = {
        {10, "ten"},
        {20, "twenty"},
        {30, "thirty"},
        {40, "forty"}
    };

    // lower_bound(): O(log n)
    auto it = m.lower_bound(25);
    if (it != m.end()) {
        std::cout << "lower_bound(25): " << it->first
                  << " = " << it->second << std::endl;  // 30 =
    thirty
    }

    // upper_bound(): O(log n)
    it = m.upper_bound(30);
    if (it != m.end()) {
        std::cout << "upper_bound(30): " << it->first
                  << " = " << it->second << std::endl;  // 40 =
    forty
    }

    return 0;
}
```

Listing 12.4: lower_bound and upper_bound in map

12.6 Removal Operations

```cpp
#include <map>
#include <iostream>

int main() {
    std::map<int, std::string> m = {
        {10, "ten"},
        {20, "twenty"},
        {30, "thirty"}
    };

    // erase(key): O(log n)
    m.erase(20);

    // erase(iterator): O(1) amortized
    auto it = m.find(30);
    if (it != m.end()) {
        m.erase(it);
    }

    // erase(it1, it2): erase range

    std::cout << "After erasures:" << std::endl;
```

```
    for (const auto& p : m) {
        std::cout << p.first << ": " << p.second << std::endl;
    }

    return 0;
}
```

Listing 12.5: Erasing from map

12.7 Iterating Over Map

```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> scores = {
        {"Alice", 95},
        {"Bob", 87},
        {"Charlie", 92}
    };

    // Structured bindings (C++17)
    std::cout << "Scores (C++17):" << std::endl;
    for (const auto& [name, score] : scores) {
        std::cout << name << ": " << score << std::endl;
    }

    // Traditional pair access
    std::cout << "\nScores (traditional):" << std::endl;
    for (const auto& pair : scores) {
        std::cout << pair.first << ": " << pair.second << std::
    endl;
    }

    // Iterator
    std::cout << "\nReverse iteration:" << std::endl;
    for (auto it = scores.rbegin(); it != scores.rend(); ++it) {
        std::cout << it->first << ": " << it->second << std::endl;
    }

    return 0;
}
```

Listing 12.6: Map iteration

12.8 Custom Key Comparator

```
#include <map>
#include <string>
#include <iostream>

// Map with descending order
```

```cpp
int main() {
    std::map<int, std::string, std::greater<int>> m;

    m[10] = "ten";
    m[30] = "thirty";
    m[20] = "twenty";

    std::cout << "Descending order:" << std::endl;
    for (const auto& [key, value] : m) {
        std::cout << key << ": " << value << std::endl;
    }
    // Output: 30, 20, 10

    return 0;
}
```

Listing 12.7: Map with custom comparison

12.9 Time Complexity Summary

Operation	Time Complexity
`operator[]`	O(log n)
`at()`	O(log n)
`insert()`	O(log n)
`find()`	O(log n)
`count()`	O(log n)
`lower_bound()`, `upper_bound()`	O(log n)
`erase(key)`	O(log n)
`erase(iterator)`	O(1) amortized
`size()`, `empty()`	O(1)
`begin()`, `end()`	O(1)

Table 12.1: map time complexities

Summary

Chapter 12 Summary:
Map Features:

- Sorted key-value pairs
- Unique keys, Red-Black tree
- `operator[]` creates if missing
- `at()` throws if missing
- Sorted by key

When to Use map:

- Need sorted dictionary
- Key-value associations
- Range queries on keys
- Ordered iteration

Use Cases:

- Configuration settings
- Symbol tables
- Caching with ordering
- Frequency counting (sorted)

Chapter 13

STL multimap Container

TL;DR

`multimap` allows multiple values for the same key. Like `map` but with duplicate keys. Uses Red-Black tree, sorted by key. `operator[]` is NOT available. Perfect for one-to-many relationships.

13.1 Introduction to multimap

Multimap stores key-value pairs where keys can repeat.

Key Point

multimap vs map:

- **Duplicate keys**: multimap allows, map doesn't
- **No operator[]**: Cannot use in multimap
- **count()**: Can return > 1
- **erase(key)**: Removes all pairs with that key
- **Header**: `<map>`

13.2 Basic Operations

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::multimap<std::string, int> mm;

    // Insert multiple values for same key
    mm.insert({"apple", 5});
    mm.insert({"banana", 3});
    mm.insert({"apple", 7});
```

```
    mm.insert({"apple", 2});

    // Multimap: apple->5, apple->7, apple->2, banana->3 (sorted
    by key)

    std::cout << "Multimap contents:" << std::endl;
    for (const auto& [key, value] : mm) {
        std::cout << key << ": " << value << std::endl;
    }

    // count(): number of pairs with given key
    std::cout << "\nCount of 'apple': " << mm.count("apple")
              << std::endl;  // 3

    return 0;
}
```

Listing 13.1: Multimap with duplicate keys

13.3 Finding All Values for a Key

```
#include <map>
#include <string>
#include <iostream>

int main() {
    std::multimap<std::string, int> mm = {
        {"apple", 5},
        {"banana", 3},
        {"apple", 7},
        {"apple", 2}
    };

    // equal_range(): get all pairs with key
    auto range = mm.equal_range("apple");

    std::cout << "All values for 'apple':" << std::endl;
    for (auto it = range.first; it != range.second; ++it) {
        std::cout << it->first << ": " << it->second << std::endl;
    }

    // lower_bound and upper_bound
    auto lower = mm.lower_bound("apple");
    auto upper = mm.upper_bound("apple");

    std::cout << "\nUsing lower_bound and upper_bound:" << std::
    endl;
    for (auto it = lower; it != upper; ++it) {
        std::cout << it->first << ": " << it->second << std::endl;
    }

    return 0;
}
```

Listing 13.2: equal_range in multimap

13.4 Removal in Multimap

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::multimap<std::string, int> mm = {
        {"apple", 5},
        {"apple", 7},
        {"banana", 3}
    };

    // erase(key): removes ALL pairs with this key
    mm.erase("apple");
    // Multimap: banana->3

    mm = {{"apple", 5}, {"apple", 7}, {"apple", 2}};

    // To erase only one instance: use iterator
    auto it = mm.find("apple");
    if (it != mm.end()) {
        mm.erase(it);  // Removes only first apple pair
    }
    // Multimap still has 2 apple entries

    std::cout << "After erasing one apple:" << std::endl;
    for (const auto& [key, value] : mm) {
        std::cout << key << ": " << value << std::endl;
    }

    return 0;
}
```

Listing 13.3: Erasing from multimap

Summary

Chapter 13 Summary:
Multimap Features:

- Sorted key-value pairs with duplicate keys
- No `operator[]`
- `equal_range()` for all values of key
- `count()` returns occurrence count

Use Cases:

- One-to-many relationships
- Phone book (name to multiple numbers)
- Student to courses mapping
- Word to line numbers in index

Chapter 14

STL unordered_set Container

TL;DR

`unordered_set` stores unique elements using hash table. No ordering guarantee. Average O(1) operations, worst case O(n). Faster than `set` for large datasets when ordering not needed. Perfect for fast membership testing.

14.1 Introduction to unordered_set

Unordered set uses hashing for fast access without maintaining order.

Key Point

unordered_set Characteristics:

- **Structure**: Hash table
- **Ordering**: No guaranteed order
- **Uniqueness**: No duplicates
- **Average time**: O(1) for insert, find, erase
- **Worst case**: O(n) (poor hash function)
- **Header**: `<unordered_set>`

14.2 Basic Operations

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

int main() {
    std::unordered_set<int> s;

    // insert(): average O(1), worst O(n)
    s.insert(30);
    s.insert(10);
```

```
    s.insert(20);
    s.insert(10);  // Duplicate ignored

    // No guaranteed order!
    std::cout << "Elements (unordered): ";
    for (int val : s) {
        std::cout << val << " ";  // Could be any order
    }
    std::cout << std::endl;

    // find(): average O(1)
    auto it = s.find(20);
    if (it != s.end()) {
        std::cout << "Found: " << *it << std::endl;
    }

    // count(): returns 0 or 1, average O(1)
    if (s.count(30)) {
        std::cout << "30 is present" << std::endl;
    }

    // erase(): average O(1)
    s.erase(10);

    // size() and empty(): O(1)
    std::cout << "Size: " << s.size() << std::endl;

    // clear(): O(n)
    s.clear();

    return 0;
}
```

Listing 14.1: unordered_set operations

Key Point

set vs unordered_set:

- **set**: O(log n), sorted, Red-Black tree
- **unordered_set**: O(1) average, unordered, hash table
- **Use set**: Need ordering or range queries
- **Use unordered_set**: Only need fast lookup, no ordering

Operation	Average	Worst Case
`insert()`	O(1)	O(n)
`find()`	O(1)	O(n)
`count()`	O(1)	O(n)
`erase(value)`	O(1)	O(n)
`erase(iterator)`	O(1)	O(1)
`size()`, `empty()`	O(1)	O(1)
`clear()`	O(n)	O(n)

Table 14.1: unordered_set time complexities

14.3 Time Complexity Summary

Warning

No lower_bound or upper_bound!
Hash tables don't maintain order, so range operations are not available:

- `lower_bound()`: NOT available
- `upper_bound()`: NOT available
- No `rbegin()`, `rend()`: iteration order is arbitrary

Summary

Chapter 14 Summary:
unordered_set Features:

- Hash table implementation
- O(1) average operations
- No ordering guarantee
- Unique elements only

Use Cases:

- Fast membership testing
- Removing duplicates (order not important)
- Cache implementation
- Visited nodes in graph algorithms

Chapter 15

STL unordered_map Container

TL;DR

`unordered_map` stores key-value pairs using hash table. No key ordering. Average O(1) operations. Faster than `map` for large datasets when ordering not needed. Perfect for dictionaries, caches, and frequency counting.

15.1 Introduction to unordered_map

Unordered map provides fast key-value lookup without maintaining key order.

Key Point

unordered_map Characteristics:

- **Structure**: Hash table
- **Ordering**: No guaranteed order of keys
- **Unique keys**: One value per key
- **Average time**: O(1)
- **Worst case**: O(n)
- **Header**: `<unordered_map>`

15.2 Basic Operations

```cpp
#include <unordered_map>
#include <string>
#include <iostream>

int main() {
    std::unordered_map<std::string, int> m;

    // operator[]: average O(1)
    m["gfg"] = 20;
```

```
    m["ide"] = 30;

    // If key doesn't exist, creates with default value
    std::cout << "m[\"tutorial\"]: " << m["tutorial"]
              << std::endl;  // 0 (default)

    // Update value
    m["gfg"] = 50;

    // insert(): average O(1)
    m.insert({"course", 100});

    // find(): average O(1)
    auto it = m.find("ide");
    if (it != m.end()) {
        std::cout << "Found: " << it->first << " = "
                  << it->second << std::endl;
    }

    // at(): average O(1), throws if key missing
    try {
        std::cout << "m.at(\"gfg\"): " << m.at("gfg") << std::endl
   ;
        // std::cout << m.at("missing");  // Would throw
    }
    catch (const std::out_of_range& ex) {
        std::cout << "Key not found" << std::endl;
    }

    return 0;
}
```

Listing 15.1: unordered_map operations

15.3 Iteration

```
#include <unordered_map>
#include <string>
#include <iostream>

int main() {
    std::unordered_map<std::string, int> m = {
        {"apple", 5},
        {"banana", 3},
        {"cherry", 7}
    };

    // Order is NOT guaranteed!
    std::cout << "Unordered map contents:" << std::endl;
    for (const auto& [key, value] : m) {
        std::cout << key << ": " << value << std::endl;
    }

    // begin(), end(): O(1)
    std::cout << "\nUsing iterators:" << std::endl;
    for (auto it = m.begin(); it != m.end(); ++it) {
```

```
        std::cout << it->first << ": " << it->second << std::endl;
    }

    // size(): O(1)
    std::cout << "\nSize: " << m.size() << std::endl;

    // empty(): O(1)
    if (!m.empty()) {
        std::cout << "Map is not empty" << std::endl;
    }

    return 0;
}
```

Listing 15.2: Iterating over unordered_map

15.4 Removal Operations

```
#include <unordered_map>
#include <string>
#include <iostream>

int main() {
    std::unordered_map<std::string, int> m = {
        {"a", 1},
        {"b", 2},
        {"c", 3}
    };

    // erase(key): average O(1)
    m.erase("b");

    // erase(iterator): O(1)
    auto it = m.find("a");
    if (it != m.end()) {
        m.erase(it);
    }

    // erase(it1, it2): erase range, O(distance)
    // m.erase(m.begin(), m.end());  // Same as clear()

    std::cout << "After erasures:" << std::endl;
    for (const auto& [key, value] : m) {
        std::cout << key << ": " << value << std::endl;
    }

    return 0;
}
```

Listing 15.3: Erasing from unordered_map

15.5 Practical Example: Frequency Counter

```cpp
#include <unordered_map>
#include <string>
#include <vector>
#include <iostream>

int main() {
    std::vector<std::string> words = {
        "apple", "banana", "apple", "cherry",
        "banana", "apple", "date"
    };

    std::unordered_map<std::string, int> freq;

    // Count frequencies
    for (const auto& word : words) {
        freq[word]++;  // Auto-creates if missing (default 0)
    }

    std::cout << "Word frequencies:" << std::endl;
    for (const auto& [word, count] : freq) {
        std::cout << word << ": " << count << std::endl;
    }

    return 0;
}
```

Listing 15.4: Count word frequencies

15.6 Time Complexity Summary

Operation	Average	Worst Case
`operator[]`	O(1)	O(n)
`at()`	O(1)	O(n)
`insert()`	O(1)	O(n)
`find()`	O(1)	O(n)
`erase(key)`	O(1)	O(n)
`erase(iterator)`	O(1)	O(1)
`size()`, `empty()`	O(1)	O(1)

Table 15.1: unordered_map time complexities

Key Point

map vs unordered_map:

- **map**: Sorted keys, O(log n), Red-Black tree
- **unordered_map**: Unsorted, O(1) average, hash table
- **Use map**: Need ordering, range queries, sorted iteration
- **Use unordered_map**: Only need fast lookup, no ordering

Summary

Chapter 15 Summary:
unordered_map Features:

- Hash table implementation
- O(1) average operations
- No key ordering
- Unique keys
- `operator[]` and `at()` available

Use Cases:

- Fast dictionary/lookup table
- Frequency counting
- Caching
- Memoization
- Configuration settings (unordered)

Performance Comparison:

- **Small datasets**: map and unordered_map similar
- **Large datasets**: unordered_map usually faster
- **Worst hash function**: unordered_map degrades to O(n)

Part III

Real-Time Systems and Concurrency

Chapter 16

FreeRTOS Fundamentals

TL;DR

FreeRTOS is a real-time operating system for microcontrollers. Core concepts: tasks (threads), queues (inter-task communication), semaphores (synchronization), and ISR (interrupt service routines). Naming conventions use prefixes: `x` for return values, `v` for void, `pv` for pointers.

16.1 Introduction to FreeRTOS

FreeRTOS provides multitasking capabilities on resource-constrained embedded systems.

Key Point

FreeRTOS Core Components:

- **Tasks**: Independent threads of execution
- **Queues**: FIFO message passing between tasks
- **Semaphores**: Synchronization primitives
- **Mutexes**: Mutual exclusion locks
- **Timers**: Software timers
- **Scheduler**: Preemptive task scheduling

16.2 Naming Conventions

16.2.1 Function Prefix Guide

16.2.2 Examples

```
// 'x' prefix: returns status or handle
BaseType_t result = xTaskCreate(...);
```

Prefix	Meaning / Return Type
`x`	Returns BaseType_t, TickType_t, or handle (non-void)
`v`	Returns void (performs action only)
`pv`	Returns void* pointer
`ux`	Returns unsigned value (UBaseType_t)
`pc`	Returns pointer to char string
`pd`	Macro constant (pdTRUE, pdFALSE, pdPASS)
`config`	Compile-time configuration constant
`port`	Architecture/port-specific function
`task`	Type or macro related to tasks
`queue`	Type or macro related to queues

Table 16.1: FreeRTOS naming prefixes

```
BaseType_t sent = xQueueSend(queue, &data, 0);
SemaphoreHandle_t sem = xSemaphoreCreateBinary();

// 'v' prefix: returns void
vTaskDelay(pdMS_TO_TICKS(1000));
vTaskDelete(NULL);
vQueueDelete(queueHandle);

// 'pv' prefix: returns void* pointer
void* ptr = pvPortMalloc(100);

// 'ux' prefix: returns unsigned value
UBaseType_t waiting = uxQueueMessagesWaiting(queue);
UBaseType_t highWater = uxTaskGetStackHighWaterMark(NULL);

// 'pd' constants
if (result == pdPASS) { /* success */ }
if (result == pdFAIL) { /* failure */ }
```

Listing 16.1: FreeRTOS naming examples

16.3 Complete FreeRTOS Example

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include <stdio.h>

/* Queue handle */
QueueHandle_t xQueue;

/* Task handle for counter task */
TaskHandle_t xCounterTaskHandle;

/* Global variable incremented by counter task */
```

```
volatile uint32_t ulCounter = 0;

/* Function prototypes */
void vSenderTask(void *pvParameters);
void vLedTask(void *pvParameters);
void vCounterTask(void *pvParameters);

/* Hardware-specific LED toggle function */
void ToggleLED(void) {
    printf("LED toggled\n");
}

/* Simulated hardware interrupt handler */
void EXTI0_IRQHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Notify counter task that ISR occurred */
    vTaskNotifyGiveFromISR(xCounterTaskHandle,
                           &xHigherPriorityTaskWoken);

    /* Context switch if needed */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}

int main(void) {
    /* Create a queue capable of holding 5 pointers to char
       strings */
    xQueue = xQueueCreate(5, sizeof(char *));

    if (xQueue == NULL) {
        printf("Queue creation failed!\n");
        while (1);
    }

    /* Create tasks */
    xTaskCreate(vSenderTask, "Sender", 256, NULL, 1, NULL);
    xTaskCreate(vLedTask, "LED", 256, NULL, 1, NULL);
    xTaskCreate(vCounterTask, "Counter", 256, NULL, 2,
                &xCounterTaskHandle);

    /* Start scheduler */
    vTaskStartScheduler();

    /* Should never reach here */
    for (;;);
}

/* Task 1: Send message every 1 second */
void vSenderTask(void *pvParameters) {
    const char *pcMessage = "Hello from Sender Task";
    (void) pvParameters;

    for (;;) {
        /* Send message to queue */
        xQueueSend(xQueue, &pcMessage, portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(1000)); // 1 second
    }
}
```

```

/* Task 2: Blink LED every 200 ms and read queue */
void vLedTask(void *pvParameters) {
    char *pcReceivedMessage;
    (void) pvParameters;

    for (;;) {
        ToggleLED();

        if (xQueueReceive(xQueue, &pcReceivedMessage, 0) == pdPASS
    ) {
            printf("Received: %s\n", pcReceivedMessage);
        }

        vTaskDelay(pdMS_TO_TICKS(200)); // 200 ms
    }
}

/* Task 3: Counter task, triggered by ISR */
void vCounterTask(void *pvParameters) {
    (void) pvParameters;

    for (;;) {
        /* Wait until ISR notifies us */
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        /* Increment counter */
        ulCounter++;
        printf("Counter incremented: %lu\n",
               (unsigned long)ulCounter);
    }
}
```

Listing 16.2: FreeRTOS comprehensive example

16.4 Task Creation and Management

```
// Task creation syntax:
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,      // Function pointer
    const char * const pcName,      // Task name (debugging)
    unsigned short usStackDepth,    // Stack size in words
    void *pvParameters,             // Parameters to pass
    UBaseType_t uxPriority,        // Task priority
    TaskHandle_t *pxCreatedTask    // Handle to created task
);

// Example:
TaskHandle_t myTaskHandle;
BaseType_t result = xTaskCreate(
    myTaskFunction,     // Function to run
    "MyTask",           // Name
    128,                // Stack: 128 words
    NULL,               // No parameters
    1,                  // Priority 1
    &myTaskHandle       // Store handle
```

```
);

if (result == pdPASS) {
    // Task created successfully
}
```

Listing 16.3: Task creation details

16.5 Queues for Inter-Task Communication

```
// Create queue
QueueHandle_t xQueue = xQueueCreate(
    10,               // Queue length
    sizeof(int)       // Item size
);

// Send to queue (from task)
int data = 42;
BaseType_t sent = xQueueSend(
    xQueue,           // Queue handle
    &data,            // Pointer to data
    pdMS_TO_TICKS(100)  // Timeout: 100ms
);

// Send from ISR
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
xQueueSendFromISR(xQueue, &data, &xHigherPriorityTaskWoken);
portYIELD_FROM_ISR(xHigherPriorityTaskWoken);

// Receive from queue
int received;
BaseType_t got = xQueueReceive(
    xQueue,
    &received,
    portMAX_DELAY     // Wait forever
);

if (got == pdPASS) {
    printf("Received: %d\n", received);
}
```

Listing 16.4: Queue operations

16.6 Semaphores for Synchronization

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>

SemaphoreHandle_t oddSem, evenSem;

#define N 10   // Numbers up to N
```

```
9
10 void OddTask(void *pvParameters) {
11     for (int i = 1; i <= N; i += 2) {
12         // Wait for permission to print odd
13         xSemaphoreTake(oddSem, portMAX_DELAY);
14
15         printf("%d ", i);
16
17         // Give permission to Even task
18         xSemaphoreGive(evenSem);
19     }
20     vTaskDelete(NULL);
21 }
22
23 void EvenTask(void *pvParameters) {
24     for (int i = 2; i <= N; i += 2) {
25         // Wait for permission to print even
26         xSemaphoreTake(evenSem, portMAX_DELAY);
27
28         printf("%d ", i);
29
30         // Give permission back to Odd task
31         xSemaphoreGive(oddSem);
32     }
33     vTaskDelete(NULL);
34 }
35
36 int main(void) {
37     // Create semaphores
38     oddSem = xSemaphoreCreateBinary();
39     evenSem = xSemaphoreCreateBinary();
40
41     // Start with odd task enabled
42     xSemaphoreGive(oddSem);
43
44     // Create tasks
45     xTaskCreate(OddTask, "OddTask", 128, NULL, 1, NULL);
46     xTaskCreate(EvenTask, "EvenTask", 128, NULL, 1, NULL);
47
48     // Start scheduler
49     vTaskStartScheduler();
50
51     while (1) { }
52 }
```

Listing 16.5: Binary semaphore example

Key Point

FromISR Suffix:
Functions with `FromISR` suffix are safe to call from interrupt context:

- `xQueueSendFromISR()`
- `vTaskNotifyGiveFromISR()`
- `xSemaphoreGiveFromISR()`

Always use `portYIELD_FROM_ISR()` after ISR API calls to trigger context switch if needed.

Summary

Chapter 16 Summary:
FreeRTOS Key Concepts:

- **Tasks**: Independent execution threads
- **Queues**: Inter-task message passing
- **Semaphores**: Synchronization primitives
- **Task Notifications**: Lightweight signaling
- **ISR**: Interrupt service routines

Naming Conventions:

- `x`: Returns handle/status
- `v`: Returns void
- `pv`: Returns pointer
- `FromISR`: Safe in interrupt context

Best Practices:

- Use task notifications for simple signaling
- Use queues for data passing
- Use semaphores for resource protection
- Always check return values
- Use `FromISR` functions in interrupts

Chapter 17

Process, Thread, and Task Concepts

TL;DR

Process: Independent program with own memory space. **Thread**: Execution unit within process sharing memory. **Task (RTOS)**: Lightweight thread in embedded systems. Synchronization primitives: mutex (mutual exclusion), condition_variable (signaling), semaphores (counting resources).

17.1 Process vs Thread vs Task

17.1.1 Definitions

Key Point

Process:

- Independent program in execution
- Own address space and resources
- Heavy overhead (context switch expensive)
- Inter-process communication (IPC) required
- Example: Separate instances of Chrome

Thread:

- Unit of execution within process
- Shares process memory (code, data, heap)
- Own stack
- Lighter than process (faster context switch)
- Example: UI thread, network thread in Chrome

Task (RTOS):

- Essentially a thread in embedded context
- Managed by RTOS scheduler
- Shares memory in microcontroller
- Very lightweight
- Example: FreeRTOS tasks

17.1.2 Comparison Table

17.2 C++ Multithreading

17.2.1 Basic Thread Creation

```cpp
#include <iostream>
#include <thread>
#include <chrono>

void printNumbers(int count) {
```

Feature	Process	Thread	Task (RTOS)
Address Space	Separate	Shared within process	Shared in MCU
Stack	Own	Own	Own
Heap	Own	Shared	Shared (single heap)
Overhead	High	Medium	Low
Creation Time	Slow	Faster	Very fast
Isolation	Strong (protected)	Weak (shared)	Weak (shared)
Communication	IPC (pipes, sockets)	Shared memory	Queues, semaphores
Context Switch	Expensive	Cheaper	Cheapest

Table 17.1: Process vs Thread vs Task comparison

```
    for (int i = 1; i <= count; ++i) {
        std::cout << "Thread: " << i << std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(100)
    );
    }
}

int main() {
    // Create thread
    std::thread t(printNumbers, 5);

    // Main thread continues
    std::cout << "Main thread running" << std::endl;

    // Wait for thread to finish
    t.join();

    std::cout << "Thread finished" << std::endl;

    return 0;
}
```

Listing 17.1: C++ thread basics

17.3 Synchronization with Mutex

17.3.1 Mutex for Mutual Exclusion

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;
int counter = 0;

void incrementCounter(int times) {
    for (int i = 0; i < times; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        counter++;
```

```
        // lock automatically released when lock_guard goes out of scope
    }
}

int main() {
    std::thread t1(incrementCounter, 1000);
    std::thread t2(incrementCounter, 1000);

    t1.join();
    t2.join();

    std::cout << "Final counter: " << counter << std::endl;  // 2000

    return 0;
}
```

Listing 17.2: Using std::mutex

Warning

Without mutex:
Without proper synchronization, multiple threads modifying shared data leads to race conditions and undefined behavior. Always protect shared data with mutexes or other synchronization primitives.

17.4 Condition Variables

17.4.1 Odd-Even Printing with Condition Variable

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool oddTurn = true;   // Start with odd
int n = 10;            // How many numbers to print

void printOdd() {
    for (int i = 1; i <= n; i += 2) {
        std::unique_lock<std::mutex> lock(mtx);

        // Wait until it's odd's turn
        cv.wait(lock, []{ return oddTurn; });

        std::cout << i << " ";

        // Switch turn to even
        oddTurn = false;
        cv.notify_all();
    }
```

```
}

void printEven() {
    for (int i = 2; i <= n; i += 2) {
        std::unique_lock<std::mutex> lock(mtx);

        // Wait until it's even's turn
        cv.wait(lock, []{ return !oddTurn; });

        std::cout << i << " ";

        // Switch turn to odd
        oddTurn = true;
        cv.notify_all();
    }
}

int main() {
    std::thread t1(printOdd);
    std::thread t2(printEven);

    t1.join();
    t2.join();

    std::cout << std::endl;
    return 0;
}
/* Output: 1 2 3 4 5 6 7 8 9 10 */
```

Listing 17.3: Synchronized odd-even printing

Key Point

Condition Variable Components:

- `std::condition_variable`: Signaling mechanism
- `std::unique_lock<std::mutex>`: Required for wait()
- `wait(lock, predicate)`: Blocks until condition true
- `notify_one()`: Wake one waiting thread
- `notify_all()`: Wake all waiting threads

17.5 lock_guard vs unique_lock

```
#include <mutex>

std::mutex mtx;

void useLockGuard() {
    std::lock_guard<std::mutex> lock(mtx);
    // Locks mutex, automatically unlocks when going out of scope
    // Cannot manually unlock
```

```
    // Simple and efficient
}

void useUniqueLock() {
    std::unique_lock<std::mutex> lock(mtx);
    // Can manually unlock: lock.unlock();
    // Can relock: lock.lock();
    // Required for condition_variable::wait()
    // More flexible but slightly more overhead
}
```

Listing 17.4: Comparing lock types

Key Point

When to Use:

- **lock_guard**: Simple lock/unlock, RAII-style
- **unique_lock**: Need manual unlock, use with condition_variable
- **scoped_lock (C++17)**: Lock multiple mutexes simultaneously

17.6 Producer-Consumer Pattern

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

std::queue<int> dataQueue;
std::mutex mtx;
std::condition_variable cv;
const int MAX_SIZE = 5;

void producer(int id, int count) {
    for (int i = 0; i < count; ++i) {
        std::unique_lock<std::mutex> lock(mtx);

        // Wait if queue is full
        cv.wait(lock, []{ return dataQueue.size() < MAX_SIZE; });

        int data = id * 100 + i;
        dataQueue.push(data);
        std::cout << "Producer " << id << " produced: "
                  << data << std::endl;

        cv.notify_all();  // Notify consumers
    }
}

void consumer(int id, int count) {
    for (int i = 0; i < count; ++i) {
        std::unique_lock<std::mutex> lock(mtx);
```

```
31
32          // Wait until queue has data
33          cv.wait(lock, []{ return !dataQueue.empty(); });
34
35          int data = dataQueue.front();
36          dataQueue.pop();
37          std::cout << "Consumer " << id << " consumed: "
38                    << data << std::endl;
39
40          cv.notify_all();  // Notify producers
41      }
42  }
43
44  int main() {
45      std::thread p1(producer, 1, 10);
46      std::thread p2(producer, 2, 10);
47      std::thread c1(consumer, 1, 10);
48      std::thread c2(consumer, 2, 10);
49
50      p1.join(); p2.join();
51      c1.join(); c2.join();
52
53      return 0;
54  }
```

Listing 17.5: Producer-consumer with condition variables

17.7 Embedded vs Desktop Threading

Key Point

Embedded Systems (FreeRTOS):

- Tasks share single memory space
- No process isolation
- RTOS scheduler manages tasks
- Typically preemptive scheduling
- Resource-constrained

Desktop OS (Linux, Windows):

- Processes have separate memory (with MMU)
- Threads within process share memory
- OS kernel manages scheduling
- Virtual memory, memory protection
- More resources available

Summary

Chapter 17 Summary:
Concurrency Concepts:

- **Process**: Isolated program with own memory
- **Thread**: Execution unit sharing process memory
- **Task**: Lightweight thread in RTOS

Synchronization Primitives:

- **Mutex**: Mutual exclusion (one thread at a time)
- **Condition Variable**: Thread signaling/waiting
- **Semaphore**: Counting/binary resource control

C++ Threading:

- `std::thread`: Create threads
- `std::mutex`: Protect shared data
- `std::lock_guard`: Simple RAII locking
- `std::unique_lock`: Flexible locking
- `std::condition_variable`: Thread coordination

Common Patterns:

- Producer-Consumer
- Reader-Writer
- Dining Philosophers
- Thread Pool

Appendix A

Quick Reference

A.1 STL Container Complexity Cheat Sheet

Container	Insert	Access	Search	Delete
vector	O(n)	O(1)	O(n)	O(n)
list	O(1)*	O(n)	O(n)	O(1)*
forward_list	O(1)*	O(n)	O(n)	O(1)*
deque	O(1) ends	O(1)	O(n)	O(1) ends
set	O(log n)	N/A	O(log n)	O(log n)
map	O(log n)	O(log n)	O(log n)	O(log n)
unordered_set	O(1) avg	N/A	O(1) avg	O(1) avg
unordered_map	O(1) avg	O(1) avg	O(1) avg	O(1) avg

Table A.1: STL container time complexities (* = at position)

A.2 When to Use Which Container

- **Need random access + dynamic size**: `vector`
- **Need frequent insert/delete at ends**: `deque`
- **Need frequent insert/delete anywhere**: `list`
- **Need sorted unique elements**: `set`
- **Need sorted key-value pairs**: `map`
- **Need fast lookup, no order**: `unordered_set/map`
- **Need LIFO**: `stack`
- **Need FIFO**: `queue`
- **Need priority-based access**: `priority_queue`

Appendix B

Common Pitfalls and Best Practices

B.1 Exception Handling

DO:

- Throw by value, catch by const reference
- Make destructors virtual in base classes
- Order catch blocks from specific to generic
- Use standard exceptions when appropriate

DON'T:

- Throw from destructors
- Throw pointers to local variables
- Put generic catch blocks before specific ones
- Let exceptions escape `noexcept` functions

B.2 STL Containers

DO:

- Reserve vector capacity when size is known
- Use emplace instead of insert for complex objects
- Prefer range-based for loops
- Check iterators for validity after modifications

DON'T:

- Modify container while iterating (invalidates iterators)

- Use `operator[]` on map without checking existence
- Assume ordered iteration in unordered containers
- Modify set/map elements directly (they're immutable)

B.3 Multithreading

DO:

- Always protect shared data with mutexes
- Use RAII locks (lock_guard, unique_lock)
- Join or detach threads before destruction
- Prefer atomic operations for simple counters

DON'T:

- Access shared data without synchronization
- Hold locks longer than necessary
- Forget to notify condition variables
- Create deadlocks (circular lock dependencies)

Conclusion

This comprehensive guide has covered the essential aspects of C++ programming, from fundamental string manipulation to advanced concepts in the Standard Template Library, and real-time systems programming with FreeRTOS.

Key Takeaways:

- Master exception handling for robust error management
- Understand STL containers and choose the right one for your needs
- Apply polymorphism and abstract classes for flexible designs
- Utilize FreeRTOS for embedded real-time applications
- Implement proper thread synchronization in concurrent programs

Continue practicing these concepts through projects and real-world applications. The best way to truly understand these topics is through hands-on experience and experimentation.

Happy Coding!