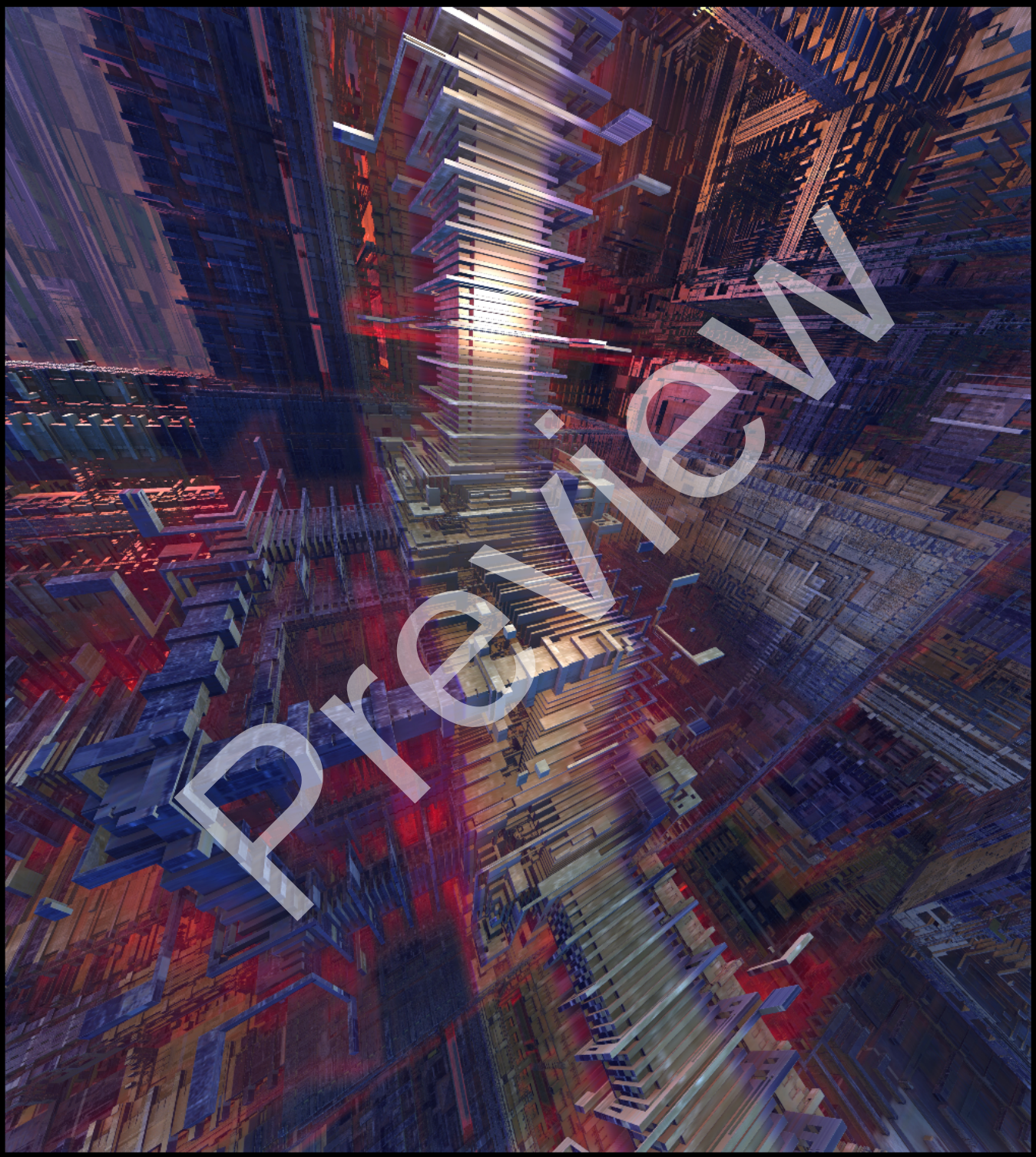
C++ Data Structures from Scratch, Vol. 3.2



Robert MacGregor

Building upon the previous book in the series, *C++ Data Structures from Scratch, Vol. 3.2* is a comprehensive guide to creating fully functional, STL-style implementations of more advanced data structures and algorithms, introducing new and powerful C++ language concepts along the way.

Key features:

- 85 complete source code files, with detailed line-by-line analysis and diagrams
- 20 sample programs directly illustrating key concepts from each chapter
- Free sample content and online support at the official website, *cppdatastructures.com*

Major topics:

- Networks
  - Maximum flow (Floyd-Fulkerson Algorithm)
  - Maximum flow of minimum cost (Dijkstra-Ford Algorithm)
- Matching algorithms
  - Bipartite
  - Stable (Gale-Shapley)
  - Perfect minimum-weight (Hungarian)
- Data compression (Huffman coding)
- Memory allocation
  - Sequential
  - Buddy
- String matching algorithms
  - Brute force
  - Knuth-Morris-Pratt
- Levenshtein distance
- ASCII and binary file I/O

About the author:

- Robert MacGregor is the developer of a C++ API for financial market trading systems. He is also a CTA (Commodity Trading Advisor) in the National Futures Association, and a Chartered Market Technician in the CMT Association.

Cover illustration: *Convergence* by Mark J. Brady (*markjaybeefractal.com*)

C++ Data Structures from Scratch, Vol. 3.2

Robert MacGregor

Published by South Coast Books

For errata, supplementary material, and contact / purchase information, visit www.cppdatastructures.com.

Cover illustration: *Convergence* by Mark J. Brady (www.markjaybeefractal.com)

ISBN-10: 0-9962115-6-X
ISBN-13: 978-0-9962115-6-7

1st Printing, September 2021

Dedicated to Liang

Table of Contents

Part 7: String Matching

Introduction and Getting Started

Chapter outline

- *A brief review of Volume 3.1*
- *Obtaining the accompanying source code*
- *Recommended study approach*
- *A brief overview of Volume 3.2*

Before we begin, let's briefly review the major topics that we covered in Volume 3.1 of *C++ Data Structures from Scratch*:

- Data structures
  - *Trie*
  - *CompressedTrie*
  - *DisjointSet*
  - *EmbeddedMap, RestrictedEmbeddedMap*
  - *Table*
  - *Graph*

- Pathfinding algorithms
  - Dijkstra
  - Bellman-Ford
  - Floyd-Warshall
  - Cycle detection

- Connectivity algorithms
  - Kruskal (minimum spanning tree)
  - Tarjan-Hopcroft (articulation points, biconnected components)
  - Kosaraju, Tarjan (strongly connected components)

- Dependency analysis (topological sort)

- Language concepts
  - ASCII file input
  - Exception handling

If you haven't yet worked through Volume 3.1, I highly recommended that you do so unless you're already familiar with the material. In addition to building upon the above concepts, we'll also reuse some of the source code from prior volumes, which won't be reexplained.

To obtain the accompanying source code for this book (which includes the pertinent source code from prior volumes), please visit the official website, *www.cppdatastructures.com*. The source code is divided into three main folders:

- *ds3*, which contains the (new) Volume 3 source code
- *ds2*, which contains (only) the reused source code from Volume 2
- *dss*, which contains (only) the reused source code from Volume 1

The *Source files and folders* section at the beginning of each chapter lists the relevant source code files and / or folders for that chapter. The root folder (*ds3*) is omitted. If a folder is listed without specific filenames, it means that we'll be using all of the files in that folder. The listing for Chapter 3.1, for example,

Source files and folders

- *CellDataPred*
- *HungarianAlgorithm/1*
- *HungarianAlgorithm/common/memberFunctions_1.h*
- *HungarianAlgorithm/operator()/operator()_1.h*

indicates that Chapter 3.1 uses:

- All of the files in the folder *ds3/CellDataPred*
- All of the files in the folder *ds3/HungarianAlgorithm/1*
- The file *ds3/HungarianAlgorithm/common/memberFunctions_1.h*, but not the other files in *ds3/HungarianAlgorithm/common*
- The file *ds3/HungarianAlgorithm/operator()/operator()_1.h*, but not the other files in *ds3/HungarianAlgorithm/operator()*

Some chapters also include a listing of relevant database files and / or diagrams. The root folder (*ds3*) is omitted. The listing for Chapter 3.1, for example,

Database files

- *DemoWeightedGraphDatabase/17.txt*

Diagrams

- *diagrams/DemoWeightedGraph/17.txt*

indicates that:

- *ds3/DemoWeightedGraphDatabase/17.txt* will be read by the accompanying program
- *ds3/diagrams/DemoWeightedGraph/17.txt* will be referenced throughout the chapter

To view the diagrams, use a plain text editor such as Microsoft Notepad or Notepad++, with the *Word Wrap* function disabled. If *Word Wrap* is enabled, the diagrams may not display correctly.

The recommended study approach for each chapter is

- Open the required source files, database files, and diagrams.

- Read the chapter, following along with the files.
- Compile the source code and run the program.
- Read the chapter again, recreating the source code from scratch.
- Compile the recreated source code and run the program, verifying the output.

Here's a brief overview of what we'll cover in Volume 3.2:

In Part 1, we'll implement a *network*, a type of graph that models the flow of resources between vertices. We'll also implement the *Ford-Fulkerson Algorithm*, which maximizes the amount of flow in a network, and the *Dijkstra-Ford Algorithm*, which finds the maximum flow of minimum cost.

In Part 2, we'll introduce the concept of matching. A *matching* is a set of edges in which no two edges share a common vertex. We'll implement two algorithms, one for finding a *maximum matching* (a matching that contains the greatest possible number of vertices), and one for finding a *stable matching* (wherein each vertex ranks possible partners in order of preference).

In Part 3, we'll implement the *Hungarian Algorithm*, which finds a *perfect matching* (a matching that contains every vertex), with the minimum total weight.

In Part 4, we'll implement *Huffman coding*, a technique for *compressing* (reducing the size of) a file.

In Part 5, we'll implement *sequential allocation*, a memory management technique that organizes a pool of memory as a linear sequence of blocks.

In Part 6, we'll implement *buddy allocation*, a memory management technique that organizes a pool of memory as a binary tree of blocks.

In Part 7, we'll implement two string matching algorithms. *String matching* is the task of searching a given string for a particular substring. We'll begin with the *brute force algorithm*, the simplest (but least efficient) approach, followed by the *Knuth-Morris-Pratt Algorithm*, a more complex (but more efficient) approach. Finally, we'll implement an algorithm to find the *Levenshtein distance*, the minimum number of *edits* (substitutions / insertions / deletions) required to transform one string into another.

C++ Data Structures from Scratch, Vol. 3.2

Part 7: String Matching

7.1: Brute Force String Matching

Source files and folders

- *bruteForceStringMatching*

Chapter outline

- *Finding a given substring, using the brute force approach*

String matching is the process of searching an existing string, called the *source string*, for a particular substring. The *brute force algorithm* begins at the desired starting position of the source string, finds the first occurrence of the substring, and returns its location. Given the source string

```
abbabbab
```

for example, suppose that we'd like to search for the substring

```
ba
```

There are 2 occurrences of *ba*, one at index 2, and the other at index 5:

```
abbabbab
01234567    // index
```

The starting position of the search is called the *start index*.

- Given a start index of 0, 1, or 2, the first occurrence of *ba* is at index 2.
- Given a start index of 3, 4, or 5, the first occurrence of *ba* is at index 5.
- Given a start index of 6 or 7, the substring *ba* isn't found.

Beginning at the start index, the brute force algorithm scans each *N*-character subsequence in the source string, where *N* is the length of the substring.

Suppose, for example, that our start index is 3. The substring *ba* is 2 characters long, so we scan each 2-character subsequence, beginning at index 3. In the following diagrams, *subsequenceBegin* is the first character (beginning) of the current subsequence:

```
Iteration 1 (subsequence ab)

  abbabbab
     |
     subsequenceBegin
```

```
  Iteration 2 (subsequence bb)

    abbabbab
        |
        subsequenceBegin
```

```
  Iteration 3 (subsequence ba)

    abbabbab
         |
         subsequenceBegin
```

```
  Iteration 4 (subsequence ab)

    abbabbab
          |
          subsequenceBegin
```

In each iteration, we compare each character in the current subsequence with the corresponding character in the substring:

- If every pair of characters matches, then we've found the substring.
- If we encounter a pair of nonmatching characters, we proceed to the next subsequence and begin the next iteration.

In the above example, we find the substring *ba* in iteration 3:

```
  Iteration 1 (subsequence ab)
    Iteration 1.1
      Character 1 of subsequence (a) != Character 1 of substring (b)
```

```
  Iteration 2 (subsequence bb)
    Iteration 2.1
      Character 1 of subsequence (b) == Character 1 of substring (b)
    Iteration 2.2
      Character 2 of subsequence (b) != Character 2 of substring (a)
```

```
  Iteration 3 (subsequence ba)
    Iteration 3.1
      Character 1 of subsequence (b) == Character 1 of substring (b)
    Iteration 3.2
      Character 2 of subsequence (a) == Character 2 of substring (a)
```

The algorithm is implemented as the standalone function (*bruteForceStringMatching.h*, lines 9-11)

```
  // Continued on next page
```

```
std::size_t bruteForceStringMatching(const std::string& source,
  const std::string& sub,
  std::size_t startIndex = 0);
```

where *source* is the source string, *sub* is the substring, and *startIndex* is the starting point. The return value (*size_t*) is the location (index) of the first occurrence of the substring. If the substring isn't found, the function returns *string::npos*, a special constant value used to indicate failure.

If the substring is blank (*bruteForceStringMatching.cpp*, line 11), we return *string::npos* (line 12) because we can't search for a blank substring.

If the substring isn't blank, we perform the search. In each iteration of the outer loop, *subsequenceBegin* (line 16) is the first character (beginning) of the current subsequence. The *sourceChar*, initialized to *subsequenceBegin* (line 20) is the current character in the subsequence. The *subChar*, initialized to the beginning of the substring (line 21) is the current character in the substring.

In each iteration of the inner loop (lines 23-29), we compare the current *sourceChar* and *subChar*. If they match, we proceed to the next pair of characters. If they don't match, or we reach the end of the *source* or *sub* string, the loop ends.

If, upon completion of the inner loop, we reach the end of the substring (line 31), then every *sourceChar* matched the corresponding *subChar*, in which case we've found the substring. To calculate the index of the beginning of the substring, we take the index of the current *sourceChar*,

```
(sourceChar - source.begin())
```

and subtract the size of the substring (*sub.size()*). We then return this value to the user (line 32).

If we complete the outer loop before finding the substring, we return *string::npos* (line 35) to indicate failure.

Given the strings

```
source = abbabbab
sub    = ba
```

and a *startIndex* of 0, for example, the function performs the following operations:

```
bruteForceStringMatching(source, sub, 0)
{
  Iteration 1
  {
    abbabbab
    |
    subsequenceBegin

    // Continued on next page
```

```
      Iteration 1.1
      {
        abbabbab      ba
        |             |
        sourceChar    subChar

        sourceChar doesn't match subChar
          break;
      }

      ++subsequenceBegin;
    }
```

```
    Iteration 2
    {
      abbabbab
       |
       subsequenceBegin

      Iteration 2.1
      {
        abbabbab       ba
         |             |
         sourceChar    subChar

        sourceChar matches subChar
          ++sourceChar;
          ++subChar;
      }

      Iteration 2.2
      {
        abbabbab       ba
          |             |
          sourceChar    subChar

        sourceChar doesn't match subChar
          break;
      }

      ++subsequenceBegin;
    }
  }
```

```
    Iteration 3
    {
      abbabbab
        |
        subsequenceBegin

      // Continued on next page
```

```
    Iteration 3.1
    {
      abbabbab       ba
        |            |
        sourceChar   subChar

      sourceChar matches subChar
        ++sourceChar;
        ++subChar;
    }

    Iteration 3.2
    {
      abbabbab       ba
         |            |
         sourceChar   subChar

      sourceChar matches subChar
        ++sourceChar;
        ++subChar;
    }

    Iteration 3.3
    {
      abbabbab       ba
          |            |
          sourceChar   subChar

      subChar is sub.end
        break;
    }

    subChar is sub.end
      return (sourceChar - source.begin) - sub.size;

    abbabbab
      |
      index 2 (return value)
  }
}
```

In addition to demonstrating the above example, our test program (*main.cpp*) finds every single occurrence of the substring.

We begin by initializing the strings (lines 11-12)

```
abbabbab    // source
ba          // sub
```

and the *startIndex* to 0 (line 13).

In each iteration of the loop, we find the first occurrence of the substring, beginning at the current

startIndex, and save its location to the variable *subIndex* (line 17).

- If the substring was found (line 19), we print its location (line 20), and advance the *startIndex* to the character immediately following the substring (line 24). If we've reached the end of the *source* string (*startIndex == source.size()*), the loop ends; otherwise, we begin the next iteration.

- If the substring wasn't found (line 21), we terminate the loop (line 22).

The loop performs the following operations:

```
a b b a b b a b
0 1 2 3 4 5 6 7
|
startIndex
```

```
Iteration 1

  subIndex   = bruteForceStringMatching(source, sub, 0)
             = 2;

  startIndex = 2 + 2
             = 4;

       a b b a b b a b
       0 1 2 3 4 5 6 7
           |   |
     subIndex   startIndex
```

```
Iteration 2

  subIndex   = bruteForceStringMatching(source, sub, 4)
             = 5;

  startIndex = 5 + 2
             = 7;

       a b b a b b a b
       0 1 2 3 4 5 6 7
                 |   |
           subIndex   startIndex
```

```
Iteration 3

  subIndex   = bruteForceStringMatching(source, sub, 7)
             = string::npos;

  subIndex is string::npos
    break;
```

The program generates the output

```
Found substring @ index 2
Found substring @ index 5
```