Examples - Using Searchers

In this lesson you will see the results of various experiments done using searchers.

WE'LL COVER THE FOLLOWING

Performance Experiments

This example builds a performance test to exercise several ways of finding a pattern in a larger text.

Here's how the test works:

- The application loads a text file (configurable via a command-line argument), for example, a book sample (like a 500KB text file)
- The whole file content is stored in one string - that will be "text" where we'll be doing the lookups.
- A pattern is selected - N letters from the input text. That way we can be sure the pattern can be found. The position of the string can be located at the front, centre or the end.
- The benchmark uses several algorithms and runs each search `ITER` times.

Example benchmarks:

- The `std::string::find` version:

```
RunAndMeasure("string::find", [&]() {
  for (size_t i = 0; i < ITERS; ++i)
  {
    std::size_t found = testString.find(needle);
    if (found == std::string::npos)
      std::cout << "The string " << needle << " not found\n";
  }
});
```

- The `default` version:

```
RunAndMeasure("default searcher", [&]() {
    for (size_t i = 0; i < ITERS; ++i)
    {
        auto it = std::search(testString.begin(), testString.end(),
            std::default_searcher(
                needle.begin(), needle.end()));
        if (it == testString.end())
            std::cout << "The string " << needle << " not found\n";
    }
});
```

- The `boyer_moore` version:

```
RunAndMeasure("boyer_moore_searcher", [&]() {
    for (size_t i = 0; i < ITERS; ++i)
    {
        auto it = std::search(testString.begin(), testString.end(),
            std::boyer_moore_searcher(
                needle.begin(), needle.end()));
        if (it == testString.end())
            std::cout << "The string " << needle << " not found\n";
    }
});
```

- The `boyer_moore_horspool` version:

```
RunAndMeasure("boyer_moore_horspool_searcher", [&]() {
for (size_t i = 0; i < ITERS; ++i)
{
  auto it = std::search(testString.begin(), testString.end(),
  std::boyer_moore_horspool_searcher(needle.begin(), needle.end()));
  if (it == testString.end())
    std::cout << "The string " << needle << " not found\n";
}
});
```

> `RunAndMeasure` is a function that takes a callable object to execute (for example a lambda). It measures the time of that execution and prints the results.

Since the input string is loaded from a file, the compiler cannot trick us and won't optimise code away.

Here is a test run for all the benchmark versions (*they are highlighted for your ease*):

main.cpp

simpleperf.h

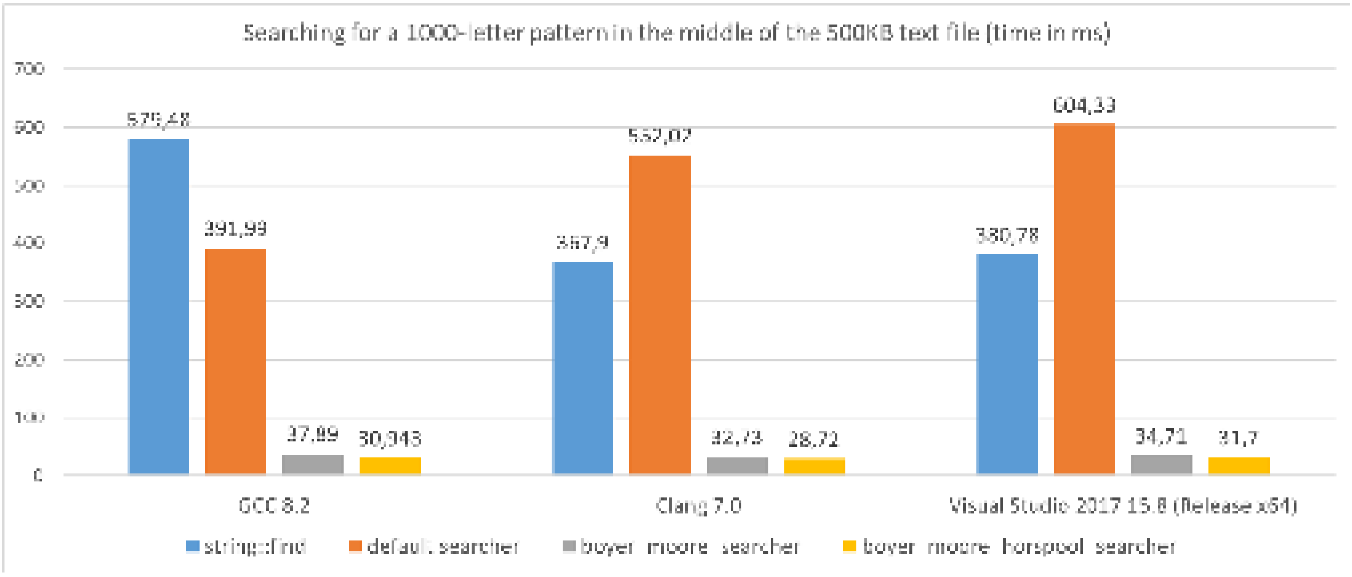
```
#include <iostream>
#include <string>
#include <algorithm>
#include <functional>
#include <chrono>
#include <fstream>
#include <string_view>
#include <numeric>
#include <sstream>
#include "simpleperf.h"

using namespace std::literals;

const std::string_view LoremIpsumStrv{ "Lorem ipsum dolor sit amet, consectetur adipiscing e
  "sed do eiusmod tempor incididuntsuperlongwordsuper ut labore et dolore magna aliqua. Ut e
  "quis nostrud exercitation ullamco laboris nisi ut aliquipsuperlongword ex ea commodo cons
  "irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariat
  "Excepteur sint occaecat cupidatatsuperlongword non proident, sunt in culpa qui officia de

std::string GetNeedleString(int argc, const char** argv, const std::string &testString)
{
  const auto testStringLen = testString.length();
  if (argc > 3)
  {
    const size_t tempLen = atoi(argv[3]);
    if (tempLen == 0) // some word?
    {
      std::cout << "needle is a string...\n";
      return argv[3];
    }
    else
    {
      const size_t PATTERN_LEN = tempLen > testStringLen ? testStringLen : tempLen;
      const int pos = argc > 4 ? atoi(argv[4]) : 0;

      if (pos == 0)
      {
        std::cout << "needle from the start...\n";
        return testString.substr(0, PATTERN_LEN);

      }
      else if (pos == 1)
      {
        std::cout << "needle from the center...\n";
        return testString.substr(testStringLen / 2 - PATTERN_LEN / 2 - 1, PATTERN_LEN);
      }
      else
      {
        std::cout << "needle from the end\n";
        return testString.substr(testStringLen - PATTERN_LEN - 1, PATTERN_LEN);

      }
    }
  }

  // just take the 1/4 of the input string from the end...
  return testString.substr(testStringLen - testStringLen/4 - 1, testStringLen/4);
}

int main(int argc, const char** argv)
{
  std::string testString{ LoremIpsumStrv };

  std::cout << "string length: " << testString.length() << '\n';

    const size_t ITERS = argc > 2 ? atoi(argv[2]) : 1000;
    std::cout << "test iterations: " << ITERS << '\n';

    const auto needle = GetNeedleString(argc, argv, testString);
    std::cout << "pattern length: " << needle.length() << '\n';

    RunAndMeasure("string::find", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            std::size_t found = testString.find(needle);
            if (found == std::string::npos)
                std::cout << "The string " << needle << " not found\n";
        }
        return 0;
    });

    RunAndMeasure("default searcher", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            auto it = std::search(testString.begin(), testString.end(),
                std::default_searcher(
                    needle.begin(), needle.end()));
            if (it == testString.end())
                std::cout << "The string " << needle << " not found\n";
        }
        return 0;
    });

    RunAndMeasure("boyer_moore_searcher init only", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            std::boyer_moore_searcher b(needle.begin(), needle.end());
            DoNotOptimizeAway(&b);
        }
        return 0;
    });

    RunAndMeasure("boyer_moore_searcher", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            auto it = std::search(testString.begin(), testString.end(),
                std::boyer_moore_searcher(
                    needle.begin(), needle.end()));
            if (it == testString.end())
                std::cout << "The string " << needle << " not found\n";
        }
        return 0;
    });

    RunAndMeasure("boyer_moore_horspool_searcher init only", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            std::boyer_moore_horspool_searcher b(needle.begin(), needle.end());
            DoNotOptimizeAway(&b);
        }
        return 0;
    });

    RunAndMeasure("boyer_moore_horspool_searcher", [&]() {
        for (size_t i = 0; i < ITERS; ++i)
        {
            auto it = std::search(testString.begin(), testString.end(),
                std::boyer_moore_horspool_searcher(
                    needle.begin(), needle.end()));
            if (it == testString.end())
                std::cout << "The string " << needle << " not found\n";
        }
        return 0;
    });
}
```

Here are some of the results running the application on Win 10 64bit, i7 8700, 3.20 GHz base frequency, 6 cores/ 12 threads (the application runs on a single thread, however). The string size is 547412 bytes (comes from a 500KB text file), and we run the benchmark 1000 times.

Algorithm	GCC 8.2	Clang 7.0	Visual Studio (Release x64)
`string::find`	579.48	367.90	380.78
`default searcher`	391.99	552.02	604.33
`boyer_moore_searcher`	37.89 (init 3.98)	32.73 (init 3.02)	34.71 (init 3.52)
`boyer_moore_horspool_searche`	30.943 (init 0)	28.72 (init 0.5)	31.70 (init 0.69)

`rspool_searcher`	30.943 (init 0)	28.72 (init 0.5)	31.70 (init 0.65)

When searching for 1000 letters from the centre of the input string, both of the new algorithms were faster than the default searcher and `string::find`. `boyer_moore` uses more time to perform the initialisation than `boyer_moore_horspool` (it creates two lookup tables, rather than one, so it will use more space and preprocessing). The results also show that `boyer_moore` usually takes a longer time to preprocess the input pattern than `boyer_moore_horspool`. And also, the second algorithm is faster in our case. But all in all, the new algorithms perform even 10…15x faster than the default versions.

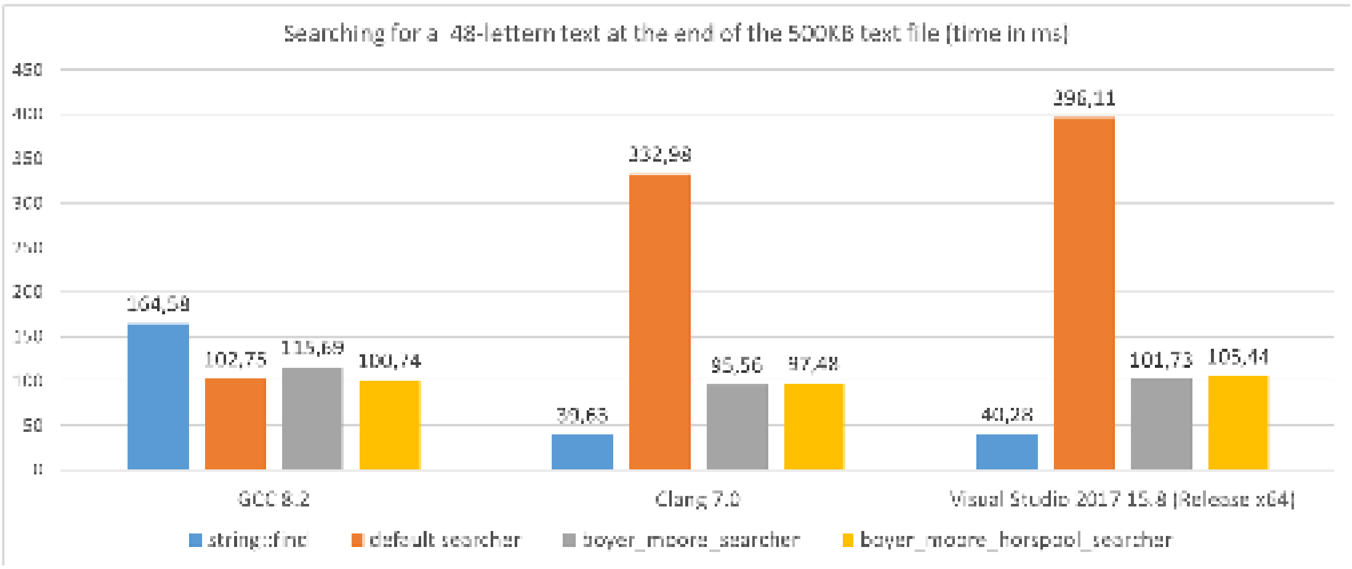


Here are the results from another run, this time we use the same input string (from a 500KB text file), we perform 1000 iterations, but the pattern is only 48 letters. It's a sentence that's located at the end of the file (a single occurrence).

Algorithm	GCC 8.2	Clang 7.0	Visual Studio (Release x64)
`string::find`	164.58	39.63	40.28
`default searcher`	102.75	332.98	396.11
`boyer_moore_searcher`	115.69 (init 0.96)	95.56 (init 0.45)	101.73 (init 0.49)

`boyer_moore_horspool_searcher`	100.74 (init 0)	97.48 (init 0.21)	105.44 (init 0.23)

In this test, Boyer-Moore algorithms in Visual Studio and Clang are 2.5x slower than `string::find`. However, on GCC `string::find` performed worse, and `boyer_moore_horspool` is the fastest.



You can run the experiments and see how your STL implementation performs. There are many ways to configure the benchmark so you can test various positions (beginning, centre, end) of the text, or check for some string pattern.

Since we are done with the example, we'll move towards DNA Matching functions in the upcoming lesson.