The Modern C++ Challenge

Become an expert programmer by solving real-world problems

By Marius Bancila

Packt
www.packt.com

The Modern C++ Challenge

Become an expert programmer by solving
real-world problems

Marius Bancila

Packt

BIRMINGHAM - MUMBAI

The Modern C++ Challenge

Commissioning Editor: Aaron Lazar
Acquisition Editors: Nitin Dasan, Chaitanya Nair
Content Development Editor: Nikhil Borkar
Technical Editor: Jijo Maliyekal
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Tania Dutta
Production Coordinator: Shantanu Zagade

First published: May 2018

Production reference: 1210518

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-386-9

www.packtpub.com

Contributors

About the author

Marius Bancila is a software engineer with 15 years of experience in developing solutions for the industrial and financial sectors. He is the author of *Modern C++ Programming Cookbook*. He focuses on Microsoft technologies and mainly develops desktop applications with C++ and C#.

He is passionate about sharing his technical expertise with others, and for that reason, he was recognized as a Microsoft MVP for more than a decade. He can be contacted on Twitter at `@mariusbancila`.

I would like to thank Nikhil Borkar, Jijo Maliyekal, Chaitanya Nair, Nitin Dasan, and all the other people at Packt who contributed to this book. I would also like to thank the reviewers who provided great feedback and steered the book in a better direction. Finally, a special thanks to my wife and family, who supported me to work on this project.

About the reviewers

Aivars Kalvāns is the lead software architect at Tieto Latvia. He has been working on a Card Suite payment card system for more than 16 years and maintains many of core C++ libraries and programs. He is also responsible for C++ programming guidelines, secure coding training, and code reviews. He organizes and speaks at internal C++ developer meetups.

> *I would like to thank my lovely wife, Anete, and sons, Kārlis, Gustavs, and Leo, for making life much more interesting.*

Arun Muralidharan is a software developer with over 8 years of experience as a systems and full-stack developer. Distributed system design, architecture, event systems, scalability, performance, and programming languages are some of the aspects of a product that interest him the most.

He is an ardent fan of C++ and its template metaprogramming; he likes how the language keeps his ego in check. So, one would find him working on C++ most of the time.

> *I would like to take this moment to thank the C++ community, from whom I have learned a lot over the years.*

Nibedit Dey is a technopreneur with a multidisciplinary technology background. He has a bachelor's in biomedical engineering and a master's in digital design and embedded systems. Before starting his entrepreneurial journey, he worked for L&T and Tektronix for several years in different R&D roles. He has been using C++ to build complex software-based systems for the last 8 years.

Table of Contents

Preface

C++ is a general-purpose programming language that combines different paradigms such as object-oriented, imperative, generic, and functional programming. C++ is designed for efficiency and is the primary choice in applications where performance is key. Over the last few decades, C++ has been one of the most widely used programming languages in industry, academia, and elsewhere. The language is standardized by the International Organization for Standardization (ISO), which is currently working on the next version of the standard, called C++20, due to be completed in 2020.

With the standard covering almost 1500 pages, C++ is not the simplest language to learn and master. Skills are not acquired only by reading about them or watching others exercising them, but by practicing them again and again. Programming is no different; we developers do not learn new languages or technologies just by reading books, articles, or watching video tutorials. Instead, we need practice to sediment and develop the new things we learn so that we can eventually master them. Many a times, however, finding good exercises to put our knowledge to test is a difficult task. Although there are many websites that feature problems for different programming languages, most of these are mathematical problems, algorithms, or problems for student competitions. These kinds of problems do not help you exercise a large variety of a programming language functionalities. That is where this book steps in.

This book is a collection of 100 real-world problems designed for you to practice a large variety of the C++ language and standard library features as well as many third-party, cross-platform libraries. Yet, a few of these problems are C++ specific and, in general, can be solved in many programming languages. Of course, the intention is to help you master C++ and therefore you are expected to solve them in C++. All the solutions provided in the book are in C++. However, you can use the book as a reference for its collection of proposed problems when you learn other programming languages, although in this case, you will not benefit from the solutions.

The problems in this book are grouped into 12 chapters. Each chapter contains problems on similar or related topics. The problems have different levels of difficulty; some of them are easy, some are moderate, and some are difficult. The book has a relatively equal number of problems for each difficulty level. Each chapter starts with the description of the proposed problems. The solutions to these problems ensue with recommendations, explanations, and source code. Although you can find the solutions in the book, it is recommended that you try to implement them by yourself first, and only afterward—or if you have difficulties completing them—look at the proposed solutions. There is only one thing that is missing in the source code presented in the book—the headers you have to include. This was left out on purpose so that you figure those out by yourself. On the other hand, the source code provided with the book is complete, and you can find all the required headers there.

At the time of writing this book, the C++20 version of the standard is in progress and will continue for the next couple of years. However, some features have already been voted in, and one of these features is the extension to the `chrono` library with calendars and time zones. There are several problems in the fifth chapter on this topic, and although no compiler supports these yet, you can solve them using the `date` library, based on which the new standard additions have been designed. Many other libraries are used for solving problems in the book. The list includes Asio, Crypto++, Curl, NLohmann/json, PDF-Writer, PNGWriter, pugixml, SQLite, and ZipLib. Also, as an alternative to the `std::optional` and the `filesystem` libraries used throughout the book, you can use Boost with compilers where these are not available. All these libraries are open source and cross-platform. They were chosen for reasons that include performance, good documentation, and wide use within the community. However, you are free to use any other libraries you would like to solve the problems.

Who this book is for

Are you trying to learn C++ and are looking for challenges to practice what you're learning? If so, this book is for you. The book is intended for people learning C++, regardless of their experience with other programming languages, as a valuable resource of practical exercises and real-world problems. This book does not teach you the features of the language or the standard library. You are expected to learn that from other resources, such as books, articles, or video tutorials. This book is a learning companion and challenges you to solve tasks of various difficulties, utilizing the skills you have previously learned from other resources. Nevertheless, many of the problems proposed in this book are language agnostic, and you can use them when learning other programming languages; however, in this case, you won't be benefiting from the solutions provided here.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Math Problems

Problems

1. Sum of naturals divisible by 3 and 5

Write a program that calculates and prints the sum of all the natural numbers divisible by either 3 or 5, up to a given limit entered by the user.

2. Greatest common divisor

Write a program that, given two positive integers, will calculate and print the greatest common divisor of the two.

3. Least common multiple

Write a program that will, given two or more positive integers, calculate and print the least common multiple of them all.

4. Largest prime smaller than given number

Write a program that computes and prints the largest prime number that is smaller than a number provided by the user, which must be a positive integer.

5. Sexy prime pairs

Write a program that prints all the sexy prime pairs up to a limit entered by the user.

6. Abundant numbers

Write a program that prints all abundant numbers and their abundance, up to a number entered by the user.

7. Amicable numbers

Write a program that prints the list of all pairs of amicable numbers smaller than 1,000,000.

8. Armstrong numbers

Write a program that prints all Armstrong numbers with three digits.

9. Prime factors of a number

Write a program that prints the prime factors of a number entered by the user.

```
      auto factors = prime_factors(number);
      std::copy(std::begin(factors), std::end(factors),
           std::ostream_iterator<unsigned long long>(std::cout, " "));
   }
```

As a further exercise, determine the largest prime factor for the number 600,851,475,143.

10. Gray code

Gray code, also known as reflected binary code or simply reflected binary, is a form of binary encoding where two consecutive numbers differ by only one bit. To perform a binary reflected Gray code encoding, we need to use the following formula:

```
if b[i-1] = 1 then g[i] = not b[i]
else g[i] = b[i]
```

This is equivalent to the following:

```
g = b xor (b logically right shifted 1 time)
```

For decoding a binary reflected Gray code, the following formula should be used:

```
b[0] = g[0]
b[i] = g[i] xor b[i-1]
```

These can be written in C++ as follows, for 32-bit unsigned integers:

```
unsigned int gray_encode(unsigned int const num)
{
   return num ^ (num >> 1);
}

unsigned int gray_decode(unsigned int gray)
{
   for (unsigned int bit = 1U << 31; bit > 1; bit >>= 1)
   {
      if (gray & bit) gray ^= bit >> 1;
   }
   return gray;
}
```

To print the all 5-bit integers, their binary representation, the encoded Gray code representation, and the decoded value, we could use the following code:

```
std::string to_binary(unsigned int value, int const digits)
{
   return std::bitset<32>(value).to_string().substr(32-digits, digits);
}

int main()
{
   std::cout << "Number\tBinary\tGray\tDecoded\n";
   std::cout << "------\t------\t----\t-------\n";

   for (unsigned int n = 0; n < 32; ++n)
   {
      auto encg = gray_encode(n);
      auto decg = gray_decode(encg);

      std::cout
         << n << "\t" << to_binary(n, 5) << "\t"
         << to_binary(encg, 5) << "\t" << decg << "\n";
   }
}
```

11. Converting numerical values to Roman

Roman numerals, as they are known today, use seven symbols: I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, and M = 1000. The system uses additions and subtractions in composing the numerical symbols. The symbols from 1 to 10 are I, II, III, IV, V, VI, VII, VIII, IX, and X. Romans did not have a symbol for zero and used to write *nulla* to represent it. In this system, the largest symbols are on the left, and the least significant are on the right. As an example, the Roman numeral for 1994 is MCMXCIV. If you are not familiar with the rules for Roman numerals, you should read more on the web.

To determine the Roman numeral of a number, use the following algorithm:

1. Check every Roman base symbol from the highest (M) to the lowest (I)
2. If the current value is greater than the value of the symbol, then concatenate the symbol to the Roman numeral and subtract its value from the current one
3. Repeat until the current value reaches zero

For example, consider 42: the first Roman base symbol smaller than 42 is XL, which is 40. We concatenate it to the numeral, resulting in XL, and subtract from the current number, resulting in 2. The first Roman base symbol smaller than 2 is I, which is 1. We add that to the numeral, resulting in XLI, and subtract 1 from the number, resulting in 1. We add one more I to the numeral, which becomes XLII, and subtract again 1 from the number, reaching 0 and therefore stopping:

```
std::string to_roman(unsigned int value)
{
   std::vector<std::pair<unsigned int, char const*>> roman {
      { 1000, "M" },{ 900, "CM" }, { 500, "D" },{ 400, "CD" },
      { 100, "C" },{ 90, "XC" }, { 50, "L" },{ 40, "XL" },
      { 10, "X" },{ 9, "IX" }, { 5, "V" },{ 4, "IV" }, { 1, "I" }};

   std::string result;
   for (auto const & kvp : roman) {
      while (value >= kvp.first) {
         result += kvp.second;
         value -= kvp.first;
      }
   }
   return result;
}
```

This function can be used as follows:

```
int main()
{
   for(int i = 1; i <= 100; ++i)
   {
      std::cout << i << "\t" << to_roman(i) << std::endl;
   }

   int number = 0;
   std::cout << "number:";
   std::cin >> number;
   std::cout << to_roman(number) << std::endl;
}
```

12. Largest Collatz sequence

The Collatz conjecture, also known as the Ulam conjecture, Kakutani's problem, the Thwaites conjecture, Hasse's algorithm, or the Syracuse problem, is an unproven conjecture that states that a sequence defined as explained in the following always reaches 1. The series is defined as follows: start with any positive integer `n` and obtain each new term from the previous one: if the previous term is even, the next term is half the previous term, or else it is 3 times the previous term plus 1.

The problem you are to solve is to generate the Collatz sequence for all positive integers up to one million, determine which of them is the longest, and print its length and the starting number that produced it. Although we could apply brute force to generate the sequence for each number and count the number of terms until reaching 1, a faster solution would be to save the length of all the sequences that have already been generated. When the current term of a sequence that started from a value `n` becomes smaller than `n`, then it is a number whose sequence has already been determined, so we could simply fetch its cached length and add it to the current length to determine the length of the sequence started from `n`. This approach, however, introduces a limit to the Collatz sequences that could be computed, because at some point the cache will exceed the amount of memory the system can allocate:

```
std::pair<unsigned long long, long> longest_collatz(
   unsigned long long const limit)
{
   long length = 0;
   unsigned long long number = 0;
   std::vector<int> cache(limit + 1, 0);
   for (unsigned long long i = 2; i <= limit; i++)
   {
      auto n = i;
      long steps = 0;
      while (n != 1 && n >= i)
      {
         if ((n % 2) == 0) n = n / 2;
         else n = n * 3 + 1;
         steps++;
      }
      cache[i] = steps + cache[n];

      if (cache[i] > length)
      {
         length = cache[i];
         number = i;
```

```
        }
    }

    return std::make_pair(number, length);
}
```

13. Computing the value of Pi

A suitable solution for approximately determining the value of Pi is using a Monte Carlo simulation. This is a method that uses random samples of inputs to explore the behavior of complex processes or systems. The method is used in a large variety of applications and domains, including physics, engineering, computing, finance, business, and others.

To do this we will rely on the following idea: the area of a circle with diameter `d` is `PI * d^2 / 4`. The area of a square that has the length of its sides equal to `d` is `d^2`. If we divide the two we get `PI/4`. If we put the circle inside the square and generate random numbers uniformly distributed within the square, then the count of numbers in the circle should be directly proportional to the circle area, and the count of numbers inside the square should be directly proportional to the square's area. That means that dividing the total number of hits in the square and circle should give `PI/4`. The more points generated, the more accurate the result shall be.

For generating pseudo-random numbers we will use a Mersenne twister and a uniform statistical distribution:

```
template <typename E = std::mt19937,
          typename D = std::uniform_real_distribution<>>
double compute_pi(E& engine, D& dist, int const samples = 1000000)
{
    auto hit = 0;
    for (auto i = 0; i < samples; i++)
    {
        auto x = dist(engine);
        auto y = dist(engine);
        if (y <= std::sqrt(1 - std::pow(x, 2))) hit += 1;
    }
    return 4.0 * hit / samples;
}

int main()
{
    std::random_device rd;
    auto seed_data = std::array<int, std::mt19937::state_size> {};
    std::generate(std::begin(seed_data), std::end(seed_data),
                  std::ref(rd));
    std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
    auto eng = std::mt19937{ seq };
    auto dist = std::uniform_real_distribution<>{ 0, 1 };

    for (auto j = 0; j < 10; j++)
        std::cout << compute_pi(eng, dist) << std::endl;
}
```

14. Validating ISBNs

The **International Standard Book Number** (**ISBN**) is a unique numeric identifier for books. Currently, a 13-digit format is used. However, for this problem, you are to validate the former format that used 10 digits. The last of the 10 digits is a checksum. This digit is chosen so that the sum of all the ten digits, each multiplied by its (integer) weight, descending from 10 to 1, is a multiple of 11.

The `validate_isbn_10` function, shown as follows, takes an ISBN as a string, and returns `true` if the length of the string is 10, all ten elements are digits, and the sum of all digits multiplied by their weight (or position) is a multiple of 11:

```
bool validate_isbn_10(std::string_view isbn)
{
   auto valid = false;
   if (isbn.size() == 10 &&
       std::count_if(std::begin(isbn), std::end(isbn), isdigit) == 10)
   {
      auto w = 10;
      auto sum = std::accumulate(
         std::begin(isbn), std::end(isbn), 0,
         [&w](int const total, char const c) {
            return total + w-- * (c - '0'); });

     valid = !(sum % 11);
   }
   return valid;
}
```

You can take it as a further exercise to improve this function to also correctly validate ISBN-10 numbers that include hyphens, such as `3-16-148410-0`. Also, you can write a function that validates ISBN-13 numbers.

2
Language Features

Problems

15. IPv4 data type

Write a class that represents an IPv4 address. Implement the functions required to be able to read and write such addresses from or to the console. The user should be able to input values in dotted form, such as `127.0.0.1` or `168.192.0.100`. This is also the form in which IPv4 addresses should be formatted to an output stream.

16. Enumerating IPv4 addresses in a range

Write a program that allows the user to input two IPv4 addresses representing a range and list all the addresses in that range. Extend the structure defined for the previous problem to implement the requested functionality.

17. Creating a 2D array with basic operations

Write a class template that represents a two-dimensional array container with methods for element access (`at()` and `data()`), capacity querying, iterators, filling, and swapping. It should be possible to move objects of this type.

18. Minimum function with any number of arguments

Write a function template that can take any number of arguments and returns the minimum value of them all, using `operator <` for comparison. Write a variant of this function template that can be parameterized with a binary comparison function to use instead of `operator <`.

19. Adding a range of values to a container

Write a general-purpose function that can add any number of elements to the end of a container that has a method `push_back(T&& value)`.

20. Container any, all, none

Write a set of general-purpose functions that enable checking whether any, all, or none of the specified arguments are present in a given container. These functions should make it possible to write code as follows:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
assert(contains_any(v, 0, 3, 30));

std::array<int, 6> a{ { 1, 2, 3, 4, 5, 6 } };
assert(contains_all(a, 1, 3, 5, 6));

std::list<int> l{ 1, 2, 3, 4, 5, 6 };
assert(!contains_none(l, 0, 6));
```

21. System handle wrapper

Consider an operating system handle, such as a file handle. Write a wrapper that handles the acquisition and release of the handle, as well as other operations such as verifying the validity of the handle and moving handle ownership from one object to another.

22. Literals of various temperature scales

Write a small library that enables expressing temperatures in the three most used scales, Celsius, Fahrenheit, and Kelvin, and converting between them. The library must enable you to write temperature literals in all these scales, such as `36.5_deg` for Celsius, `97.7_f` for Fahrenheit, and `309.65_K` for Kelvin; perform operations with these values; and convert between them.

Solutions

15. IPv4 data type

The problem requires writing a class to represent an IPv4 address. This is a 32-bit value, usually represented in decimal dotted format, such as `168.192.0.100`; each part of it is an 8-bit value, ranging from 0 to 255. For easy representation and handling, we can use four `unsigned char` to store the address value. Such a value could be constructed either from four `unsigned char` or from an `unsigned long`. In order to be able to read a value directly from the console (or any other input stream) and be able to write the value to the console (or any other output stream), we have to overload `operator>>` and `operator<<`. The following listing shows a minimal implementation that can meet the requested functionality:

```
class ipv4
{
   std::array<unsigned char, 4> data;
public:
   constexpr ipv4() : data{ {0} } {}
   constexpr ipv4(unsigned char const a, unsigned char const b,
                  unsigned char const c, unsigned char const d):
      data{{a,b,c,d}} {}
   explicit constexpr ipv4(unsigned long a) :
      data{ { static_cast<unsigned char>((a >> 24) & 0xFF),
              static_cast<unsigned char>((a >> 16) & 0xFF),
              static_cast<unsigned char>((a >> 8) & 0xFF),
              static_cast<unsigned char>(a & 0xFF) } } {}
   ipv4(ipv4 const & other) noexcept : data(other.data) {}
   ipv4& operator=(ipv4 const & other) noexcept
   {
      data = other.data;
      return *this;
   }

   std::string to_string() const
   {
      std::stringstream sstr;
      sstr << *this;
      return sstr.str();
   }

   constexpr unsigned long to_ulong() const noexcept
   {
      return (static_cast<unsigned long>(data[0]) << 24) |
              (static_cast<unsigned long>(data[1]) << 16) |
              (static_cast<unsigned long>(data[2]) << 8) |
               static_cast<unsigned long>(data[3]);
   }

   friend std::ostream& operator<<(std::ostream& os, const ipv4& a)
   {
      os << static_cast<int>(a.data[0]) << '.'
         << static_cast<int>(a.data[1]) << '.'
         << static_cast<int>(a.data[2]) << '.'
         << static_cast<int>(a.data[3]);
      return os;
   }

   friend std::istream& operator>>(std::istream& is, ipv4& a)
   {
      char d1, d2, d3;
      int b1, b2, b3, b4;
      is >> b1 >> d1 >> b2 >> d2 >> b3 >> d3 >> b4;
      if (d1 == '.' && d2 == '.' && d3 == '.')
         a = ipv4(b1, b2, b3, b4);
      else
         is.setstate(std::ios_base::failbit);
      return is;
   }
};
```

The `ipv4` class can be used as follows:

```
int main()
{
   ipv4 address(168, 192, 0, 1);
   std::cout << address << std::endl;

   ipv4 ip;
   std::cout << ip << std::endl;
   std::cin >> ip;
   if(!std::cin.fail())
      std::cout << ip << std::endl;
}
```

16. Enumerating IPv4 addresses in a range

To be able to enumerate IPv4 addresses in a given range, it should first be possible to compare IPv4 values. Therefore, we should implement at least `operator<`, but the following listing contains implementation for all comparison operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`. Also, in order to increment an IPv4 value, implementations for both the prefix and postfix `operator++` are provided. The following code is an extension of the IPv4 class from the previous problem:

```
ipv4& operator++()
{
   *this = ipv4(1 + to_ulong());
   return *this;
}

ipv4& operator++(int)
{
   ipv4 result(*this);
   ++(*this);
   return *this;
}

friend bool operator==(ipv4 const & a1, ipv4 const & a2) noexcept
{
   return a1.data == a2.data;
}

friend bool operator!=(ipv4 const & a1, ipv4 const & a2) noexcept
{
   return !(a1 == a2);
}

friend bool operator<(ipv4 const & a1, ipv4 const & a2) noexcept
{
   return a1.to_ulong() < a2.to_ulong();
}

friend bool operator>(ipv4 const & a1, ipv4 const & a2) noexcept
{
   return a2 < a1;
}

friend bool operator<=(ipv4 const & a1, ipv4 const & a2) noexcept
{
   return !(a1 > a2);
}
```

```
   return join_strings(std::begin(c), std::end(c), separator);
}

int main()
{
   using namespace std::string_literals;
   std::vector<std::string> v1{ "this","is","an","example" };
   std::vector<std::string> v2{ "example" };
   std::vector<std::string> v3{ };

   assert(join_strings(v1, " ") == "this is an example"s);
   assert(join_strings(v2, " ") == "example"s);
   assert(join_strings(v3, " ") == ""s);
}
```

As a further exercise, you should modify the overload that takes iterators as arguments so that it works with other types of iterators, such as bidirectional iterators, thereby enabling the use of this function with lists or other containers.

27. Splitting a string into tokens with a list of possible delimiters

Two different versions of a splitting function are listed as follows:

- The first one uses a single character as the delimiter. To split the input string it uses a string stream initialized with the content of the input string, using `std::getline()` to read chunks from it until the next delimiter or an end-of-line character is encountered.
- The second one uses a list of possible character delimiters, specified in an `std::string`. It uses `std:string::find_first_of()` to locate the first position of any of the delimiter characters, starting from a given position. It does so in a loop until the entire input string is being processed. The extracted substrings are added to the result vector:

```
template <class Elem>
using tstring = std::basic_string<Elem, std::char_traits<Elem>,
                                  std::allocator<Elem>>;

template <class Elem>
using tstringstream = std::basic_stringstream<
   Elem, std::char_traits<Elem>, std::allocator<Elem>>;
```

```
template<typename Elem>
inline std::vector<tstring<Elem>> split(tstring<Elem> text,
                                        Elem const delimiter)
{
   auto sstr = tstringstream<Elem>{ text };
   auto tokens = std::vector<tstring<Elem>>{};
   auto token = tstring<Elem>{};
   while (std::getline(sstr, token, delimiter))
   {
      if (!token.empty()) tokens.push_back(token);
   }
   return tokens;
}

template<typename Elem>
inline std::vector<tstring<Elem>> split(tstring<Elem> text,
                                        tstring<Elem> const & delimiters)
{
   auto tokens = std::vector<tstring<Elem>>{};
   size_t pos, prev_pos = 0;
   while ((pos = text.find_first_of(delimiters, prev_pos)) !=
   std::string::npos)
   {
      if (pos > prev_pos)
      tokens.push_back(text.substr(prev_pos, pos - prev_pos));
      prev_pos = pos + 1;
   }
   if (prev_pos < text.length())
   tokens.push_back(text.substr(prev_pos, std::string::npos));
   return tokens;
}
```

The following sample code shows two examples of how different strings can be split using either one delimiter character or multiple delimiters:

```
int main()
{
   using namespace std::string_literals;
   std::vector<std::string> expected{"this", "is", "a", "sample"};
   assert(expected == split("this is a sample"s, ' '));
   assert(expected == split("this,is a.sample!!"s, ",.! "s));
}
```

28. Longest palindromic substring

The simplest solution to this problem is to try a brute-force approach, checking if each substring is a palindrome. However, this means we need to check *C(N, 2)* substrings (where *N* is the number of characters in the string), and the time complexity would be $O(N^3)$. The complexity could be reduced to $O(N^2)$ by storing results of sub problems. To do so we need a table of Boolean values, of size $N \times N$, where the element at `[i, j]` indicates whether the substring from position `i` to `j` is a palindrome. We start by initializing all elements `[i,i]` with `true` (one-character palindromes) and all the elements `[i,i+i]` with `true` for all consecutive two identical characters (for two-character palindromes). We then go on to inspect substrings greater than two characters, setting the element at `[i,j]` to `true` if the element at `[i+i,j-1]` is `true` and the characters on the positions `i` and `j` in the string are also equal. Along the way, we retain the start position and length of the longest palindromic substring in order to extract it after finishing computing the table.

In code, this solution appears as follows:

```
std::string longest_palindrome(std::string_view str)
{
   size_t const len = str.size();
   size_t longestBegin = 0;
   size_t maxLen = 1;
   std::vector<bool> table(len * len, false);
   for (size_t i = 0; i < len; i++)
      table[i*len + i] = true;

   for (size_t i = 0; i < len - 1; i++)
   {
      if (str[i] == str[i + 1])
      {
         table[i*len + i + 1] = true;
         if (maxLen < 2)
         {
            longestBegin = i;
            maxLen = 2;
         }
      }
   }

   for (size_t k = 3; k <= len; k++)
   {
      for (size_t i = 0; i < len - k + 1; i++)
      {
         size_t j = i + k - 1;
         if (str[i] == str[j] && table[(i + 1)*len + j - 1])
```

```
            {
               table[i*len +j] = true;
               if (maxLen < k)
               {
                  longestBegin = i;
                  maxLen = k;
               }
            }
         }
      }
      return std::string(str.substr(longestBegin, maxLen));
   }
```

Here are some test cases for the `longest_palindrome()` function:

```
int main()
{
   using namespace std::string_literals;
   assert(longest_palindrome("sahararahnide") == "hararah");
   assert(longest_palindrome("level") == "level");
   assert(longest_palindrome("s") == "s");
}
```

29. License plate validation

The simplest way to solve this problem is by using regular expressions. The regular expression that meets the described format is `"[A-Z]{3}-[A-Z]{2} \d{3,4}"`.

The first function only has to validate that an input string contains only text that matches this regular expression. For that, we can use `std::regex_match()`, as follows:

```
bool validate_license_plate_format(std::string_view str)
{
   std::regex rx(R"([A-Z]{3}-[A-Z]{2} \d{3,4})");
   return std::regex_match(str.data(), rx);
}

int main()
{
   assert(validate_license_plate_format("ABC-DE 123"));
   assert(validate_license_plate_format("ABC-DE 1234"));
   assert(!validate_license_plate_format("ABC-DE 12345"));
   assert(!validate_license_plate_format("abc-de 1234"));
}
```

The second function is slightly different. Instead of matching the input string, it must identify all occurrences of the regular expression within the string. The regular expression would therefore change to `"([A-Z]{3}-[A-Z]{2} \d{3,4})*"`. To iterate through all matches we have to use `std::sregex_iterator`, which is as follows:

```
std::vector<std::string> extract_license_plate_numbers(
                            std::string const & str)
{
   std::regex rx(R"(([A-Z]{3}-[A-Z]{2} \d{3,4})*)");
   std::smatch match;
   std::vector<std::string> results;

   for(auto i = std::sregex_iterator(std::cbegin(str), std::cend(str), rx);
       i != std::sregex_iterator(); ++i)
   {
      if((*i)[1].matched)
      results.push_back(i->str());
   }
   return results;
}

int main()
{
   std::vector<std::string> expected {
      "AAA-AA 123", "ABC-DE 1234", "XYZ-WW 0001"};
   std::string text("AAA-AA 123qwe-ty 1234 ABC-DE 123456..XYZ-WW 0001");
   assert(expected == extract_license_plate_numbers(text));
}
```

30. Extracting URL parts

This problem is also suited to being solved using regular expressions. Finding a regular expression that could match any URL is, however, a difficult task. The purpose of this exercise is to help you practice your skills with the regex library, and not to find the ultimate regular expression for this particular purpose. Therefore, the regular expression used here is provided only for didactic purposes.

You can try regular expressions using online testers and debuggers, such as `https://regex101.com/`. This can be useful in order to work out your regular expressions and try them against various datasets.

For this task we will consider that a URL has the following parts: `protocol` and `domain` are mandatory, and `port`, `path`, `query`, and `fragment` are all optional. The following structure is used to return results from parsing an URL (alternatively, you could return a tuple and use structured binding to bind variables to the various sub parts of the tuple):

```
struct uri_parts
{
   std::string                protocol;
   std::string                domain;
   std::optional<int>         port;
   std::optional<std::string> path;
   std::optional<std::string> query;
   std::optional<std::string> fragment;
};
```

A function that can parse a URL and extract and return its parts could have the following implementation. Note that the return type is an `std::optional<uri_parts>` because the function might fail in matching the input string to the regular expression; in this case, the return value is `std::nullopt`:

```
std::optional<uri_parts> parse_uri(std::string uri)
{
   std::regex rx(R"(^(\w+):\/\/([\w.-
]+)(:(\d+))?([\w\/\.]+)?(\?([\w=&]*)(#?(\w+))?)?$)");
   auto matches = std::smatch{};
   if (std::regex_match(uri, matches, rx))
   {
      if (matches[1].matched && matches[2].matched)
      {
         uri_parts parts;
         parts.protocol = matches[1].str();
         parts.domain = matches[2].str();
         if (matches[4].matched)
            parts.port = std::stoi(matches[4]);
         if (matches[5].matched)
            parts.path = matches[5];
         if (matches[7].matched)
            parts.query = matches[7];
         if (matches[9].matched)
            parts.fragment = matches[9];
         return parts;
      }
   }
   return {};
}
```

42. Day and week of the year

Write a function that, given a date, returns the day of the year (from 1 to 365 or 366 for leap years) and another function that, for the same input, returns the calendar week of the year.

43. Meeting time for multiple time zones

Write a function that, given a list of meeting participants and their time zones, displays the local meeting time for each participant.

44. Monthly calendar

Write a function that, given a year and month, prints to the console the month calendar. The expected output format is as follows (the example is for December 2017):

```
Mon Tue Wed Thu Fri Sat Sun
                  1   2   3
  4   5   6   7   8   9  10
 11  12  13  14  15  16  17
 18  19  20  21  22  23  24
 25  26  27  28  29  30  31
```

Solutions

39. Measuring function execution time

To measure the execution time of a function, you should retrieve the current time before the function execution, execute the function, then retrieve the current time again and determine how much time passed between the two time points. For convenience, this can all be put in a `variadic` function template that takes as arguments the function to execute and its arguments, and:

- Uses `std::high_resolution_clock` by default to determine the current time.
- Uses `std::invoke()` to execute the function to measure, with its specified arguments.
- Returns a duration and not a number of ticks for a particular duration. This is important so that you don't lose resolution. It enables you to add execution time duration of various resolutions, such as seconds and milliseconds, which would not be possible by returning a tick count:

```
template <typename Time = std::chrono::microseconds,
          typename Clock = std::chrono::high_resolution_clock>
struct perf_timer
{
   template <typename F, typename... Args>
   static Time duration(F&& f, Args... args)
   {
      auto start = Clock::now();
      std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
      auto end = Clock::now();

      return std::chrono::duration_cast<Time>(end - start);
   }
};
```

This function template can be used as follows:

```
void f()
{
   // simulate work
   std::this_thread::sleep_for(2s);
}
```

```
void g(int const a, int const b)
{
   // simulate work
   std::this_thread::sleep_for(1s);
}

int main()
{
   auto t1 = perf_timer<std::chrono::microseconds>::duration(f);
   auto t2 = perf_timer<std::chrono::milliseconds>::duration(g, 1, 2);

   auto total = std::chrono::duration<double, std::nano>(t1 + t2).count();
}
```

40. Number of days between two dates

As of C++17, the `chrono` standard library does not have support for working with dates, weeks, calendars, time zones, and other useful related features. This will change in C++20, as time zones and calendar support have been added to the standard at the Jacksonville meeting, in March 2018. The new additions are based on an open source library called `date`, built on top of `chrono`, developed by Howard Hinnant and available on GitHub at `https://github.com/HowardHinnant/date`. We will use this library to solve several of the problems in this chapter. Although in this implementation the namespace is `date`, in C++20 it will be part of `std::chrono`. However, you should be able to simply replace the namespace without any further code changes.

To solve this task, you could use the `date::sys_days` class, available in the `date.h` header. It represents a count of days since the `std::system_clock` epoch. This is a `time_point` with a resolution of a day and is implicitly convertible to `std::system_clock::time_point`. Basically, you have to construct two objects of this type and subtract them. The result is exactly the number of days between the two dates. The following is a simple implementation of such a function:

```
inline int number_of_days(
   int const y1, unsigned int const m1, unsigned int const d1,
   int const y2, unsigned int const m2, unsigned int const d2)
{
   using namespace date;

   return (sys_days{ year{ y1 } / month{ m1 } / day{ d1 } } -
           sys_days{ year{ y2 } / month{ m2 } / day{ d2 } }).count();
}
```

```
inline int number_of_days(date::sys_days const & first,
                          date::sys_days const & last)
{
   return (last - first).count();
}
```

Here are a couple of examples of how these overloaded functions could be used:

```
int main()
{
   auto diff1 = number_of_days(2016, 9, 23, 2017, 5, 15);

   using namespace date::literals;
   auto diff2 = number_of_days(2016_y/sep/23, 15_d/may/2017);
}
```

41. Day of the week

Solving this problem is again relatively straightforward if you use the `date` library. However, this time, you have to use the following types:

- `date::year_month_day`, a structure that represents a day with fields for year, month (1 to 12), and day (1 to 31).
- `date::iso_week::year_weeknum_weekday`, from the `iso_week.h` header, is a structure that has fields for year, number of weeks in a year, and number of days in a week (1 to 7). This class is implicitly convertible to and from `date::sys_days`, which makes it explicitly convertible to any other calendar system that is implicitly convertible to and from `date::sys_days`, such as `date::year_month_day`.

With that being said, the problem resolves to creating a `year_month_day` object to represent the desired date and then a `year_weeknum_weekday` object from it, and retrieving the day of the week with `weekday()`:

```
unsigned int week_day(int const y, unsigned int const m,
                      unsigned int const d)
{
   using namespace date;

   if(m < 1 || m > 12 || d < 1 || d > 31) return 0;

   auto const dt = date::year_month_day{year{ y }, month{ m }, day{ d }};
   auto const tiso = iso_week::year_weeknum_weekday{ dt };
   return (unsigned int)tiso.weekday();
}

int main()
{
   auto wday = week_day(2018, 5, 9);
}
```

42. Day and week of the year

The solution to this two-part problem should be straightforward from the previous two:

- To compute the day of the year, you subtract two `date::sys_days` objects, one representing the given day and the other January 0 of the same year. Alternatively, you could start from January 1 and add 1 to the result.
- To determine the week number of the year, construct a `year_weeknum_weekday` object, like in the previous problem, and retrieve the `weeknum()` value:

```
int day_of_year(int const y, unsigned int const m,
                unsigned int const d)
{
   using namespace date;

   if(m < 1 || m > 12 || d < 1 || d > 31) return 0;

   return (sys_days{ year{ y } / month{ m } / day{ d } } -
           sys_days{ year{ y } / jan / 0 }).count();
}

unsigned int calendar_week(int const y, unsigned int const m,
                           unsigned int const d)
{
   using namespace date;

   if(m < 1 || m > 12 || d < 1 || d > 31) return 0;

   auto const dt = date::year_month_day{year{ y }, month{ m }, day{ d }};
   auto const tiso = iso_week::year_weeknum_weekday{ dt };

   return (unsigned int)tiso.weeknum();
}
```

These functions can be used as follows:

```
int main()
{
   int y = 0;
   unsigned int m = 0, d = 0;
   std::cout << "Year:"; std::cin >> y;
   std::cout << "Month:"; std::cin >> m;
   std::cout << "Day:"; std::cin >> d;

   std::cout << "Calendar week:" << calendar_week(y, m, d) << std::endl;
   std::cout << "Day of year:" << day_of_year(y, m, d) << std::endl;
}
```

43. Meeting time for multiple time zones

To work with time zones, you must use the `tz.h` header of the `date` library. However, this needs the *IANA Time Zone Database* to be downloaded and uncompressed on your machine.

This is how to prepare the time zone database for the date library:

- Download the latest version of the database from `https://www.iana.org/time-zones`. Currently, the latest version is called `tzdata2017c.tar.gz`.
- Uncompress this to any location on your machine, in a subdirectory called `tzdata`. Let's suppose the parent directory is `c:\work\challenges\libs\date` (on a Windows machine); this will have a sub directory called `tzdata`.
- For Windows, you need to download a file called `windowsZones.xml`, containing mappings of Windows time zones to IANA time zones. This is available at `https://unicode.org/repos/cldr/trunk/common/supplemental/windowsZones.xml`. The file must be stored in the same `tzdata` sub directory created earlier.
- In your project settings, define a preprocessor macro called `INSTALL` that indicates the parent directory for the `tzdata` sub directory. For the example given here, you should have `INSTALL=c:\\work\\challenges\\libs\\date`. (Note that the double backslash is necessary because the macro is used to create a file path using stringification and concatenation, and would otherwise result in an incorrect path.)

```
   return (unsigned int)tiso.weekday();
}

void print_month_calendar(int const y, unsigned int m)
{
   using namespace date;
   std::cout << "Mon Tue Wed Thu Fri Sat Sun" << std::endl;

   auto first_day_weekday = week_day(y, m, 1);
   auto last_day = (unsigned int)year_month_day_last(
      year{ y }, month_day_last{ month{ m } }).day();

   unsigned int index = 1;
   for (unsigned int day = 1; day < first_day_weekday; ++day, ++index)
   {
      std::cout << " ";
   }

   for (unsigned int day = 1; day <= last_day; ++day)
   {
      std::cout << std::right << std::setfill(' ') << std::setw(3)
                << day << ' ';
      if (index++ % 7 == 0) std::cout << std::endl;
   }

   std::cout << std::endl;
}

int main()
{
   print_month_calendar(2017, 12);
}
```

6
Algorithms and Data Structures

Problems

45. Priority queue

Write a data structure that represents a priority queue that provides constant time lookup for the largest element, but has logarithmic time complexity for adding and removing elements. A queue inserts new elements at the end and removes elements from the top. By default, the queue should use `operator<` to compare elements, but it should be possible for the user to provide a comparison function object that returns `true` if the first argument is less than the second. The implementation must provide at least the following operations:

- `push()` to add a new element
- `pop()` to remove the top element
- `top()` to provide access to the top element
- `size()` to indicate the number of elements in the queue
- `empty()` to indicate whether the queue is empty

46. Circular buffer

Create a data structure that represents a circular buffer of a fixed size. A circular buffer overwrites existing elements when the buffer is being filled beyond its fixed size. The class you must write should:

- Prohibit default construction
- Support the creation of objects with a specified size
- Allow checking of the buffer capacity and status (`empty()`, `full()`, `size()`, `capacity()`)
- Add a new element, an operation that could potentially overwrite the oldest element in the buffer
- Remove the oldest element from the buffer
- Support iteration through its elements

47. Double buffer

Write a class that represents a buffer that could be written and read at the same time without the two operations colliding. A read operation must provide access to the old data while a write operation is in progress. Newly written data must be available for reading upon completion of the write operation.

48. The most frequent element in a range

Write a function that, given a range, returns the most frequent element and the number of times it appears in the range. If more than one element appears the same maximum number of times then the function should return all the elements. For instance, for the range `{1,1,3,5,8,13,3,5,8,8,5}`, it should return `{5, 3}` and `{8, 3}`.

49. Text histogram

Write a program that, given a text, determines and prints a histogram with the frequency of each letter of the alphabet. The frequency is the percentage of the number of appearances of each letter from the total count of letters. The program should count only the appearances of letters and ignore digits, signs, and other possible characters. The frequency must be determined based on the count of letters and not the text size.

50. Filtering a list of phone numbers

Write a function that, given a list of phone numbers, returns only the numbers that are from a specified country. The country is indicated by its phone country code, such as 44 for Great Britain. Phone numbers may start with the country code, a + followed by the country code, or have no country code. The ones from this last category must be ignored.

51. Transforming a list of phone numbers

Write a function that, given a list of phone numbers, transforms them so they all start with a specified phone country code, preceded by the + sign. Any whitespaces from a phone number should also be removed. The following is a list of input and output examples:

```
07555 123456    => +447555123456
07555123456     => +447555123456
+44 7555 123456 => +447555123456
44 7555 123456  => +447555123456
7555 123456     => +447555123456
```

52. Generating all the permutations of a string

Write a function that, prints on the console all the possible permutations of a given string. You should provide two versions of this function: one that uses recursion, and one that does not.

53. Average rating of movies

Write a program that calculates and prints the average rating of a list of movies. Each movie has a list of ratings from 1 to 10 (where 1 is the lowest and 10 is the highest rating). In order to compute the rating, you must remove 5% of the highest and lowest ratings before computing their average. The result must be displayed with a single decimal point.

54. Pairwise algorithm

Write a general-purpose function that, given a range, returns a new range with pairs of consecutive elements from the input range. Should the input range have an odd number of elements, the last one must be ignored. For example, if the input range was `{1, 1, 3, 5, 8, 13, 21}`, the result must be `{ {1, 1}, {3, 5}, {8, 13}}`.

55. Zip algorithm

Write a function that, given two ranges, returns a new range with pairs of elements from the two ranges. Should the two ranges have different sizes, the result must contain as many elements as the smallest of the input ranges. For example, if the input ranges were `{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }` and `{ 1, 1, 3, 5, 8, 13, 21 }`, the result should be `{{1,1}, {2,1}, {3,3}, {4,5}, {5,8}, {6,13}, {7,21}}`.

56. Select algorithm

Write a function that, given a range of values and a projection function, transforms each value into a new one and returns a new range with the selected values. For instance, if you have a type book that has an `id`, `title`, and `author`, and have a range of such book values, it should be possible for the function to select only the title of the books. Here is an example of how the function should be used:

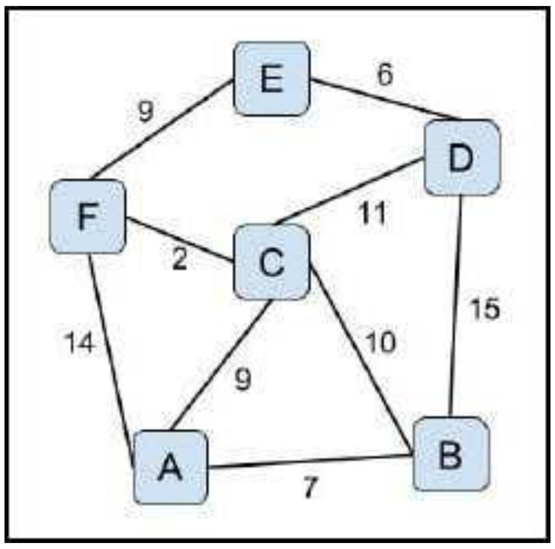
```
struct book
{
   int         id;
   std::string title;
   std::string author;
};

std::vector<book> books{
   {101, "The C++ Programming Language", "Bjarne Stroustrup"},
   {203, "Effective Modern C++", "Scott Meyers"},
   {404, "The Modern C++ Programming Cookbook", "Marius Bancila"}};

auto titles = select(books, [](book const & b) {return b.title; });
```

57. Sort algorithm

Write a function that, given a pair of random-access iterators to define its lower and upper bounds, sorts the elements of the range using the quicksort algorithm. There should be two overloads of the sort function: one that uses `operator<` to compare the elements of the range and put them in ascending order, and one that uses a user-defined binary comparison function for comparing the elements.

58. The shortest path between nodes

Write a program that, given a network of nodes and the distances between them, computes and displays the shortest distance from a specified node to all the others, as well as the path between the start and end node. As input, consider the following undirected graph:



```
      size_type size() const noexcept { return data.size(); }

      void push(value_type const & value)
      {
         data.push_back(value);
         std::push_heap(std::begin(data), std::end(data), comparer);
      }

      void pop()
      {
         std::pop_heap(std::begin(data), std::end(data), comparer);
         data.pop_back();
      }

      const_reference top() const { return data.front(); }
      void swap(priority_queue& other) noexcept
      {
         swap(data, other.data);
         swap(comparer, other.comparer);
      }
   private:
      std::vector<T> data;
      Compare comparer;
   };

   template<class T, class Compare>
   void swap(priority_queue<T, Compare>& lhs,
             priority_queue<T, Compare>& rhs)
   noexcept(noexcept(lhs.swap(rhs)))
   {
      lhs.swap(rhs);
   }
```

This class can be used as follows:

```
   int main()
   {
      priority_queue<int> q;
      for (int i : {1, 5, 3, 1, 13, 21, 8})
      {
         q.push(i);
      }

      assert(!q.empty());
      assert(q.size() == 7);

      while (!q.empty())
      {
         std::cout << q.top() << ' ';
         q.pop();
      }
   }
```

46. Circular buffer

A circular buffer is a fixed-size container that behaves as if its two ends were connected to form a virtual circular memory layout. Its main benefit is that you don't need a large amount of memory to retain data, as older entries are overwritten by newer ones. Circular buffers are used in I/O buffering, bounded logging (when you only want to retain the last messages), buffers for asynchronous processing, and others.

We can differentiate between two situations:

1. The number of elements added to the buffer has not reached its capacity (its user-defined fixed size). In this case, it behaves likes a regular container, such as a vector.
2. The number of elements added to the buffer has reached and exceeded its capacity. In this case, the buffer's memory is reused and older elements are being overwritten.

We could represent such a structure using:

- A regular container with a pre-allocated number of elements
- A head pointer to indicate the position of the last inserted element
- A size counter to indicate the number of elements in the container, which cannot exceed its capacity (since elements are being overwritten in this case)

The two main operations with a circular buffer are:

- Adding a new element to the buffer. We always insert at the next position of the head pointer (or index). This is the `push()` method shown below.
- Removing an existing element from the buffer. We always remove the oldest element. That element is at position `head - size` (this must account for the circular nature of the index). This is the `pop()` method shown below.

The implementation of such a data structure is shown here:

```
template <class T>
class circular_buffer
{
   typedef circular_buffer_iterator<T> const_iterator;

   circular_buffer() = delete;
public:
   explicit circular_buffer(size_t const size) :data_(size)
   {}

   bool clear() noexcept { head_ = -1; size_ = 0; }
   bool empty() const noexcept { return size_ == 0; }
   bool full() const noexcept { return size_ == data_.size(); }
   size_t capacity() const noexcept { return data_.size(); }
   size_t size() const noexcept { return size_; }

   void push(T const item)
   {
      head_ = next_pos();
      data_[head_] = item;
      if (size_ < data_.size()) size_++;
   }

   T pop()
   {
      if (empty()) throw std::runtime_error("empty buffer");
      auto pos = first_pos();
      size_--;
      return data_[pos];
   }

   const_iterator begin() const
   {
      return const_iterator(*this, first_pos(), empty());
   }

   const_iterator end() const
   {
      return const_iterator(*this, next_pos(), true);
   }

private:
   std::vector<T> data_;
   size_t head_ = -1;
   size_t size_ = 0;
   size_t next_pos() const noexcept
   { return size_ == 0 ? 0 : (head_ + 1) % data_.size(); }
   size_t first_pos() const noexcept
   { return size_ == 0 ? 0 : (head_ + data_.size() - size_ + 1) %
                            data_.size(); }

   friend class circular_buffer_iterator<T>;
};
```

Because of the circular nature of the indexes mapped on a contiguous memory layout, the iterator type for this class cannot be a pointer type. The iterators must be able to point elements by applying modulo operations on the index. Here is a possible implementation for such an iterator:

```
template <class T>
class circular_buffer_iterator
{
   typedef circular_buffer_iterator        self_type;
   typedef T                               value_type;
   typedef T&                              reference;
   typedef T const&                        const_reference;
   typedef T*                              pointer;
   typedef std::random_access_iterator_tag iterator_category;
   typedef ptrdiff_t                       difference_type;
public:
   circular_buffer_iterator(circular_buffer<T> const & buf,
                            size_t const pos, bool const last) :
   buffer_(buf), index_(pos), last_(last)
   {}

   self_type & operator++ ()
   {
      if (last_)
         throw std::out_of_range("Iterator cannot be incremented past the
end of range.");
      index_ = (index_ + 1) % buffer_.data_.size();
      last_ = index_ == buffer_.next_pos();
      return *this;
   }

   self_type operator++ (int)
   {
      self_type tmp = *this;
      ++*this;
      return tmp;
   }
```

```
      bool operator== (self_type const & other) const
      {
         assert(compatible(other));
         return index_ == other.index_ && last_ == other.last_;
      }

      bool operator!= (self_type const & other) const
      {
         return !(*this == other);
      }

      const_reference operator* () const
      {
         return buffer_.data_[index_];
      }

      const_reference operator-> () const
      {
         return buffer_.data_[index_];
      }
   private:
      bool compatible(self_type const & other) const
      {
         return &buffer_ == &other.buffer_;
      }

      circular_buffer<T> const & buffer_;
      size_t index_;
      bool last_;
   };
```

With all these implemented, we could write code such as the following. Notice that in the comments, the first range shows the actual content of the internal vector, and the second range shows the logical content as exposed with iterator access:

```
   int main()
   {
      circular_buffer<int> cbuf(5); // {0, 0, 0, 0, 0} -> {}

      cbuf.push(1);                 // {1, 0, 0, 0, 0} -> {1}
      cbuf.push(2);                 // {1, 2, 0, 0, 0} -> {1, 2}
      cbuf.push(3);                 // {1, 2, 3, 0, 0} -> {1, 2, 3}

      auto item = cbuf.pop();       // {1, 2, 3, 0, 0} -> {2, 3}
      cbuf.push(4);                 // {1, 2, 3, 4, 0} -> {2, 3, 4}
      cbuf.push(5);                 // {1, 2, 3, 4, 5} -> {2, 3, 4, 5}
      cbuf.push(6);                 // {6, 2, 3, 4, 5} -> {2, 3, 4, 5, 6}
      cbuf.push(7);                 // {6, 7, 3, 4, 5} -> {3, 4, 5, 6, 7}
      cbuf.push(8);                 // {6, 7, 8, 4, 5} -> {4, 5, 6, 7, 8}

      item = cbuf.pop();            // {6, 7, 8, 4, 5} -> {5, 6, 7, 8}
      item = cbuf.pop();            // {6, 7, 8, 4, 5} -> {6, 7, 8}
      item = cbuf.pop();            // {6, 7, 8, 4, 5} -> {7, 8}

      item = cbuf.pop();            // {6, 7, 8, 4, 5} -> {8}
      item = cbuf.pop();            // {6, 7, 8, 4, 5} -> {}

      cbuf.push(9);                 // {6, 7, 8, 9, 5} -> {9}
   }
```

47. Double buffer

The problem described here is a typical double buffering situation. Double buffering is the most common case of multiple buffering, which is a technique that allows a reader to see a complete version of the data and not a partially updated version produced by a writer. This is a common technique – especially in computer graphics – for avoiding flickering.

In order to implement the requested functionality, the buffer class that we should write must have two internal buffers: one that contains temporary data being written, and another one that contains completed (or committed) data. Upon the completion of a write operation, the content of the temporary buffer is written in the primary buffer. For the internal buffers, the implementation below uses `std::vector`. When the write operation completes, instead of copying data from one buffer to the other, we just swap the content of the two, which is a much faster operation. Access to the completed data is provided with either the `read()` function, which copies the content of the read buffer into a designated output, or with direct element access (overloaded `operator[]`). Access to the read buffer is synchronized with an `std::mutex` to make it safe to read from one thread while another is writing to the buffer:

```
template <typename T>
class double_buffer
{
   typedef T           value_type;
   typedef T&          reference;
   typedef T const &   const_reference;
   typedef T*          pointer;
public:
   explicit double_buffer(size_t const size) :
      rdbuf(size), wrbuf(size)
   {}
```

```
      size_t vertex_count() const { return adjacency_list.size(); }
      std::vector<Vertex> verteces() const
      {
         std::vector<Vertex> keys;
         for (auto const & kvp : adjacency_list)
            keys.push_back(kvp.first);
         return keys;
      }

      neighbor_list_type const & neighbors(Vertex const & v) const
      {
         auto pos = adjacency_list.find(v);
         if (pos == adjacency_list.end())
            throw std::runtime_error("vertex not found");
         return pos->second;
      }

      constexpr static Weight Infinity =
                 std::numeric_limits<Weight>::infinity();
   private:
      std::map<vertex_type, neighbor_list_type> adjacency_list;
   };
```

The implementation of the shortest path algorithm as described in the preceding pseudocode could look like the following. An `std::set` (that is, a self-balancing binary search tree) is used instead of the priority queue. `std::set` has the same $O(log(n))$ complexity for adding and removing the top element as a binary heap (used for a priority queue). On the other hand, `std::set` also allows finding and removing any other element in $O(log(n))$, which is helpful in order to implement the decrease-key step in logarithmic time by removing and inserting again:

```
   template <typename Vertex, typename Weight>
   void shortest_path(
      graph<Vertex, Weight> const & g,
      Vertex const source,
      std::map<Vertex, Weight>& min_distance,
      std::map<Vertex, Vertex>& previous)
   {
      auto const n = g.vertex_count();
      auto const verteces = g.verteces();

      min_distance.clear();
      for (auto const & v : verteces)
         min_distance[v] = graph<Vertex, Weight>::Infinity;
      min_distance[source] = 0;
      previous.clear();

      std::set<std::pair<Weight, Vertex> > vertex_queue;
      vertex_queue.insert(std::make_pair(min_distance[source], source));

      while (!vertex_queue.empty())
      {
         auto dist = vertex_queue.begin()->first;
         auto u = vertex_queue.begin()->second;

         vertex_queue.erase(std::begin(vertex_queue));

         auto const & neighbors = g.neighbors(u);
         for (auto const & neighbor : neighbors)
         {
            auto v = neighbor.first;
            auto w = neighbor.second;
            auto dist_via_u = dist + w;
            if (dist_via_u < min_distance[v])
            {
               vertex_queue.erase(std::make_pair(min_distance[v], v));

               min_distance[v] = dist_via_u;
               previous[v] = u;
               vertex_queue.insert(std::make_pair(min_distance[v], v));
            }
         }
      }
   }
```

The following helper functions print the results in the specified format:

```
   template <typename Vertex>
   void build_path(
      std::map<Vertex, Vertex> const & prev, Vertex const v,
      std::vector<Vertex> & result)
   {
      result.push_back(v);

      auto pos = prev.find(v);
      if (pos == std::end(prev)) return;

      build_path(prev, pos->second, result);
   }

   template <typename Vertex>
   std::vector<Vertex> build_path(std::map<Vertex, Vertex> const & prev,
                                  Vertex const v)
```

```
{
   std::vector<Vertex> result;
   build_path(prev, v, result);
   std::reverse(std::begin(result), std::end(result));
   return result;
}

template <typename Vertex>
void print_path(std::vector<Vertex> const & path)
{
   for (size_t i = 0; i < path.size(); ++i)
   {
      std::cout << path[i];
      if (i < path.size() - 1) std::cout << " -> ";
   }
}
```

The following program solves the given task:

```
int main()
{
   graph<char, double> g;
   g.add_edge('A', 'B', 7);
   g.add_edge('A', 'C', 9);
   g.add_edge('A', 'F', 14);
   g.add_edge('B', 'C', 10);
   g.add_edge('B', 'D', 15);
   g.add_edge('C', 'D', 11);
   g.add_edge('C', 'F', 2);
   g.add_edge('D', 'E', 6);
   g.add_edge('E', 'F', 9);

   char source = 'A';
   std::map<char, double> min_distance;
   std::map<char, char> previous;
   shortest_path(g, source, min_distance, previous);

   for (auto const & kvp : min_distance)
   {
      std::cout << source << " -> " << kvp.first << " : "
                << kvp.second << '\t';

      print_path(build_path(previous, kvp.first));

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

59. The Weasel program

The Weasel program is a thought experiment proposed by Richard Dawkins, intended to demonstrate how the accumulated small improvements (mutations that bring a benefit to the individual so that it is chosen by natural selection) produce fast results as opposed to the mainstream misinterpretation that evolution happens in big leaps. The algorithm for the Weasel simulation, as described on Wikipedia (see `https://en.wikipedia.org/wiki/Weasel_program`), is as follows:

1. Start with a random string of 28 characters.
2. Make 100 copies of this string, with a 5% chance per character of that character being replaced with a random character.
3. Compare each new string with the target METHINKS IT IS LIKE A WEASEL, and give each a score (the number of letters in the string that are correct and in the correct position).
4. If any of the new strings has a perfect score (28), then stop.
5. Otherwise, take the highest-scoring string and go to step 2.

A possible implementation is as follows. The `make_random()` function creates a random starting sequence of the same length as the target; the `fitness()` function computes the score of each mutated string (that is, resemblance with the target); the `mutate()` function produces a new string from a parent with a given chance for each character to mutate:

```
class weasel
{
   std::string target;
   std::uniform_int_distribution<> chardist;
   std::uniform_real_distribution<> ratedist;
   std::mt19937 mt;
   std::string const allowed_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
public:
   weasel(std::string_view t) :
      target(t), chardist(0, 26), ratedist(0, 100)
   {
      std::random_device rd;
      auto seed_data = std::array<int, std::mt19937::state_size> {};
      std::generate(std::begin(seed_data), std::end(seed_data),
      std::ref(rd));
      std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
      mt.seed(seq);
   }
```

```
      void run(int const copies)
      {
         auto parent = make_random();
         int step = 1;
         std::cout << std::left << std::setw(5) << std::setfill(' ')
                   << step << parent << std::endl;

         do
         {
            std::vector<std::string> children;
            std::generate_n(std::back_inserter(children), copies,
               [parent, this]() {return mutate(parent, 5); });

            parent = *std::max_element(
               std::begin(children), std::end(children),
               [this](std::string_view c1, std::string_view c2) {
                  return fitness(c1) < fitness(c2); });

            std::cout << std::setw(5) << std::setfill(' ') << step
                   << parent << std::endl;

            step++;
         } while (parent != target);
      }
   private:
      weasel() = delete;

      double fitness(std::string_view candidate) const
      {
         int score = 0;
         for (size_t i = 0; i < candidate.size(); ++i)
         {
            if (candidate[i] == target[i])
               score++;
         }
         return score;
      }

      std::string mutate(std::string_view parent, double const rate)
      {
         std::stringstream sstr;
         for (auto const c : parent)
         {
            auto nc = ratedist(mt) > rate ? c : allowed_chars[chardist(mt)];
            sstr << nc;
         }
         return sstr.str();
       }
      std::string make_random()
      {
         std::stringstream sstr;
         for (size_t i = 0; i < target.size(); ++i)
         {
            sstr << allowed_chars[chardist(mt)];
         }
         return sstr.str();
      }
   };
```

This is how the class can be used:

```
   int main()
   {
      weasel w("METHINKS IT IS LIKE A WEASEL");
      w.run(100);
   }
```

60. The Game of Life

The class `universe` presented below implements the game as described. There are several functions of interest:

- `initialize()` generates a starting layout; although the code accompanying the book contains more options, only two are listed here: `random`, which generates a random layout, and `ten_cell_row`, which represents a line of 10 cells in the middle of the grid.
- `reset()` sets all the cells as `dead`.
- `count_neighbors()` returns the number of alive neighbors. It uses a helper variadic function template `count_alive()`. Although this could be implemented with fold expressions, this is not yet supported in Visual C++ and therefore I have opted not to use it here.
- `next_generation()` produces a new state of the game based on the transition rules.
- `display()` shows the game status on the console; this uses a system call to erase the console, although you could use other means to do so, such as specific operating system APIs.

- `run()` initializes the starting layout and then produces a new generation at a user-specified interval, for a user-specified number of iterations, or indefinitely (if the number of iterations was set to 0).

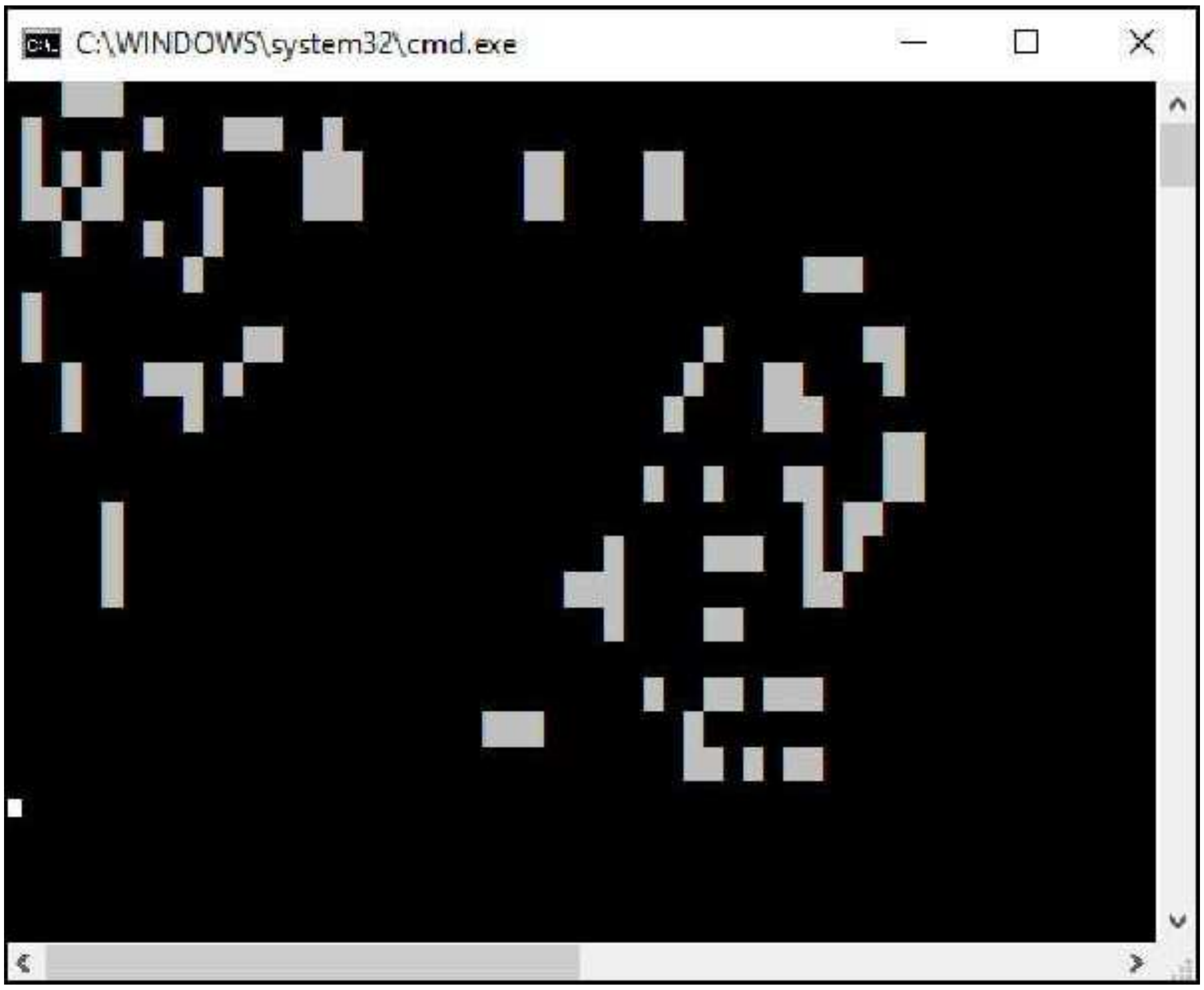
```
class universe
{
private:
   universe() = delete;
public:
   enum class seed
   {
      random, ten_cell_row
   };
public:
   universe(size_t const width, size_t const height):
      rows(height), columns(width),grid(width * height), dist(0, 4)
   {
      std::random_device rd;
      auto seed_data = std::array<int, std::mt19937::state_size> {};
      std::generate(std::begin(seed_data), std::end(seed_data),
      std::ref(rd));
      std::seed_seq seq(std::begin(seed_data), std::end(seed_data));
      mt.seed(seq);
   }

   void run(seed const s, int const generations,
            std::chrono::milliseconds const ms =
               std::chrono::milliseconds(100))
   {
      reset();
      initialize(s);
      display();

      int i = 0;
      do
      {
         next_generation();
         display();

         using namespace std::chrono_literals;
         std::this_thread::sleep_for(ms);
      } while (i++ < generations || generations == 0);
   }

private:
   void next_generation()
   {
      std::vector<unsigned char> newgrid(grid.size());

      for (size_t r = 0; r < rows; ++r)
      {
         for (size_t c = 0; c < columns; ++c)
         {
            auto count = count_neighbors(r, c);

            if (cell(c, r) == alive)
            {
               newgrid[r * columns + c] =
                  (count == 2 || count == 3) ? alive : dead;
            }
            else
            {
               newgrid[r * columns + c] = (count == 3) ? alive : dead;
            }
         }
      }

      grid.swap(newgrid);
   }

   void reset_display()
   {
#ifdef WIN32
      system("cls");
#endif
   }

   void display()
   {
      reset_display();

      for (size_t r = 0; r < rows; ++r)
      {
         for (size_t c = 0; c < columns; ++c)
         {
            std::cout << (cell(c, r) ? '*' : ' ');
         }
         std::cout << std::endl;
      }
   }

   void initialize(seed const s)
   {
      if (s == seed::ten_cell_row)
      {
```

```
         for (size_t c = columns / 2 - 5; c < columns / 2 + 5; c++)
            cell(c, rows / 2) = alive;
      }
      else
      {
         for (size_t r = 0; r < rows; ++r)
         {
            for (size_t c = 0; c < columns; ++c)
            {
               cell(c, r) = dist(mt) == 0 ? alive : dead;
            }
         }
      }
   }

   void reset()
   {
      for (size_t r = 0; r < rows; ++r)
      {
         for (size_t c = 0; c < columns; ++c)
         {
            cell(c, r) = dead;
         }
      }
   }

   int count_alive() { return 0; }

   template<typename T1, typename... T>
   auto count_alive(T1 s, T... ts) { return s + count_alive(ts...); }

   int count_neighbors(size_t const row, size_t const col)
   {
      if (row == 0 && col == 0)
         return count_alive(cell(1, 0), cell(1,1), cell(0, 1));
      if (row == 0 && col == columns - 1)
         return count_alive(cell(columns - 2, 0), cell(columns - 2, 1),
                            cell(columns - 1, 1));
      if (row == rows - 1 && col == 0)
         return count_alive(cell(0, rows - 2), cell(1, rows - 2),
                            cell(1, rows - 1));
      if (row == rows - 1 && col == columns - 1)
         return count_alive(cell(columns - 1, rows - 2),
                            cell(columns - 2, rows - 2),
                            cell(columns - 2, rows - 1));
      if (row == 0 && col > 0 && col < columns - 1)
         return count_alive(cell(col - 1, 0), cell(col - 1, 1),
                            cell(col, 1), cell(col + 1, 1),
                            cell(col + 1, 0));
      if (row == rows - 1 && col > 0 && col < columns - 1)
         return count_alive(cell(col - 1, row), cell(col - 1, row - 1),
                            cell(col, row - 1), cell(col + 1, row - 1),
                            cell(col + 1, row));
      if (col == 0 && row > 0 && row < rows - 1)
         return count_alive(cell(0, row - 1), cell(1, row - 1),
                            cell(1, row), cell(1, row + 1),
                            cell(0, row + 1));
      if (col == columns - 1 && row > 0 && row < rows - 1)
         return count_alive(cell(col, row - 1), cell(col - 1, row - 1),
                            cell(col - 1, row), cell(col - 1, row + 1),
                            cell(col, row + 1));

      return count_alive(cell(col - 1, row - 1), cell(col, row - 1),
                         cell(col + 1, row - 1), cell(col + 1, row),
                         cell(col + 1, row + 1), cell(col, row + 1),
                         cell(col - 1, row + 1), cell(col - 1, row));
   }

   unsigned char& cell(size_t const col, size_t const row)
   {
      return grid[row * columns + col];
   }

private:
   size_t rows;
   size_t columns;

   std::vector<unsigned char> grid;
   const unsigned char alive = 1;
   const unsigned char dead = 0;

   std::uniform_int_distribution<> dist;
   std::mt19937 mt;
};
```

This is how the game can be run for 100 iterations starting from a random state:

```
int main()
{
   using namespace std::chrono_literals;
   universe u(50, 20);
   u.run(universe::seed::random, 100, 100ms);
}
```

Here is an example of the program output (the screenshot represents a single iteration in the Game of Life's universe):



7
Concurrency

Problems

61. Parallel transform algorithm

Write a general-purpose algorithm that applies a given unary function to transform the elements of a range in parallel. The unary operation used to transform the range must not invalidate range iterators or modify the elements of the range. The level of parallelism, that is, the number of execution threads and the way it is achieved, is an implementation detail.

62. Parallel min and max element algorithms using threads

Implement general-purpose parallel algorithms that find the minimum value and, respectively, the maximum value in a given range. The parallelism should be implemented using threads, although the number of concurrent threads is an implementation detail.

63. Parallel min and max element algorithms using asynchronous functions

Implement general-purpose parallel algorithms that find the minimum value and, respectively, the maximum value in a given range. The parallelism should be implemented using asynchronous functions, although the number of concurrent functions is an implementation detail.

64. Parallel sort algorithm

Write a parallel version of the sort algorithm as defined for problem *53. Sort Algorithm*, in `Chapter 6`, *Algorithms and Data Structures*, which, given a pair of random access iterators to define its lower and upper bounds, sorts the elements of the range using the quicksort algorithm. The function should use the comparison operators for comparing the elements of the range. The level of parallelism and the way to achieve it is an implementation detail.

65. Thread-safe logging to the console

Write a class that enables components running in different threads to safely print log messages to the console by synchronizing access to the standard output stream to guarantee the integrity of the output. This logging component should have a method called `log()` with a string argument representing the message to be printed to the console.

66. Customer service system

Write a program that simulates the way customers are served in an office. The office has three desks where customers can be served at the same time. Customers can enter the office at any time. They take a ticket with a service number from a ticketing machine and wait until their number is next for service at one of the desks. Customers are served in the order they entered the office, or more precisely, in the order given by their ticket. Every time a service desk finishes serving a customer, the next customer in order is served. The simulation should stop after a particular number of customers have been issued tickets and served.

Solutions

61. Parallel transform algorithm

The general-purpose function `std::transform()` applies a given function to a range and stores the result in another (or the same) range. The requirement for this problem is implementing a parallel version of such a function. A general-purpose one would take iterators as arguments to define the first and one-past-last element of the range. Because the unary function is applied in the same manner to all the elements of the range, it is fairly simple to parallelize the operation. For this task, we will be using threads. Since it is not specified how many threads should be running at the same time, we could use `std::thread::hardware_concurrency()` . This function returns a hint for the number of concurrent threads supported by the implementation.

A parallel version of the algorithm performs better than a sequential implementation only if the size of the range exceeds a particular threshold, which may vary with compilation options, platform, or hardware. In the following implementation that threshold is set to 10,000 elements. As a further exercise, you could experiment with various thresholds and range sizes to see how the execution time changes.

The following function, `ptransform()`, implements the parallel transform algorithm as requested. It simply calls `std::transform()` if the range size does not exceed the defined threshold. Otherwise, it splits the range into several equal parts, one for each thread, and calls `std::transform()` on each thread for a particular subrange. In this case, the function blocks the calling thread until all the worker threads finish execution:

```
template <typename RandomAccessIterator, typename F>
void ptransform(RandomAccessIterator begin, RandomAccessIterator end,
                F&& f)
{
   auto size = std::distance(begin, end);
   if (size <= 10000)
   {
      std::transform(begin, end, begin, std::forward<F>(f));
   }
   else
   {
      std::vector<std::thread> threads;
      int thread_count = std::thread::hardware_concurrency();
      auto first = begin;
      auto last = first;
      size /= thread_count;
      for (int i = 0; i < thread_count; ++i)
      {
         first = last;
         if (i == thread_count - 1) last = end;
         else std::advance(last, size);

         threads.emplace_back([first, last, &f]() {
            std::transform(first, last, first, std::forward<F>(f));
         });
      }

      for (auto & t : threads) t.join();
   }
}
```

The function `palter()`, shown as follows, is a helper function that applies `ptransform()` to an `std::vector` and returns another `std::vector` with the result:

```
template <typename T, typename F>
std::vector<T> palter(std::vector<T> data, F&& f)
{
   ptransform(std::begin(data), std::end(data),
              std::forward<F>(f));
   return data;
}
```

The function can be used as follows (a complete example can be found in the source code accompanying this book):

```
int main()
{
   std::vector<int> data(1000000);
   // init data
   auto result = palter(data, [](int const e) {return e * e; });
}
```

TIP

In C++17, a series of standard general-purpose algorithms, including `std::transform()`, have overloads that implement a parallel version of the algorithm that can be executed according to a specified execution policy.

62. Parallel min and max element algorithms using threads

This problem, and its solution, is similar in most ways to the previous one. What is slightly different is that the function concurrently executing on each thread must return a value that represents the minimum or the maximum element in the subrange.
The `pprocess()` function template, shown as follows, is a higher-level function that implements the requested functionality generically, in the following way:

- Its arguments are the first and one-past-last iterators to the range and a function object that processes the range that we will call `f`.
- If the size of the range is smaller than a particular threshold, set to 10,000 elements here, it simply executes the function object `f` received as argument.
- Otherwise, it splits the input range into a number of subranges of equal size, one for each concurrent thread that could be executed. Each thread runs `f` for the selected subrange.
- The results of the parallel execution of `f` are collected in an `std::vector`, and after the execution of all threads is completed, `f` is used again to determine the overall result from the intermediate results:

```
template <typename Iterator, typename F>
auto pprocess(Iterator begin, Iterator end, F&& f)
{
   auto size = std::distance(begin, end);
   if (size <= 10000)
   {
      return std::forward<F>(f)(begin, end);
   }
   else
   {
      int thread_count = std::thread::hardware_concurrency();
      std::vector<std::thread> threads;
      std::vector<typename std::
         iterator_traits<Iterator>::value_type>
      mins(thread_count);

      auto first = begin;
      auto last = first;
      size /= thread_count;
      for (int i = 0; i < thread_count; ++i)
      {
         first = last;
         if (i == thread_count - 1) last = end;
```

```
            else std::advance(last, size);

            threads.emplace_back([first, last, &f, &r=mins[i]]() {
            r = std::forward<F>(f)(first, last);
            });
         }

         for (auto & t : threads) t.join();

         return std::forward<F>(f)(std::begin(mins), std::end(mins));
      }
   }
```

Two functions, called `pmin()` and `pmax()`, are provided to implement the required general-purpose min and max parallel algorithms. These two are in turn calling `pprocess()`, passing for the third argument a lambda that uses either the `std::min_element()` or the `std::max_element()` standard algorithm:

```
   template <typename Iterator>
   auto pmin(Iterator begin, Iterator end)
   {
      return pprocess(begin, end,
                      [](auto b, auto e){return *std::min_element(b, e);});
   }

   template <typename Iterator>
   auto pmax(Iterator begin, Iterator end)
   {
      return pprocess(begin, end,
                      [](auto b, auto e){return *std::max_element(b, e);});
   }
```

These functions can be used as follows:

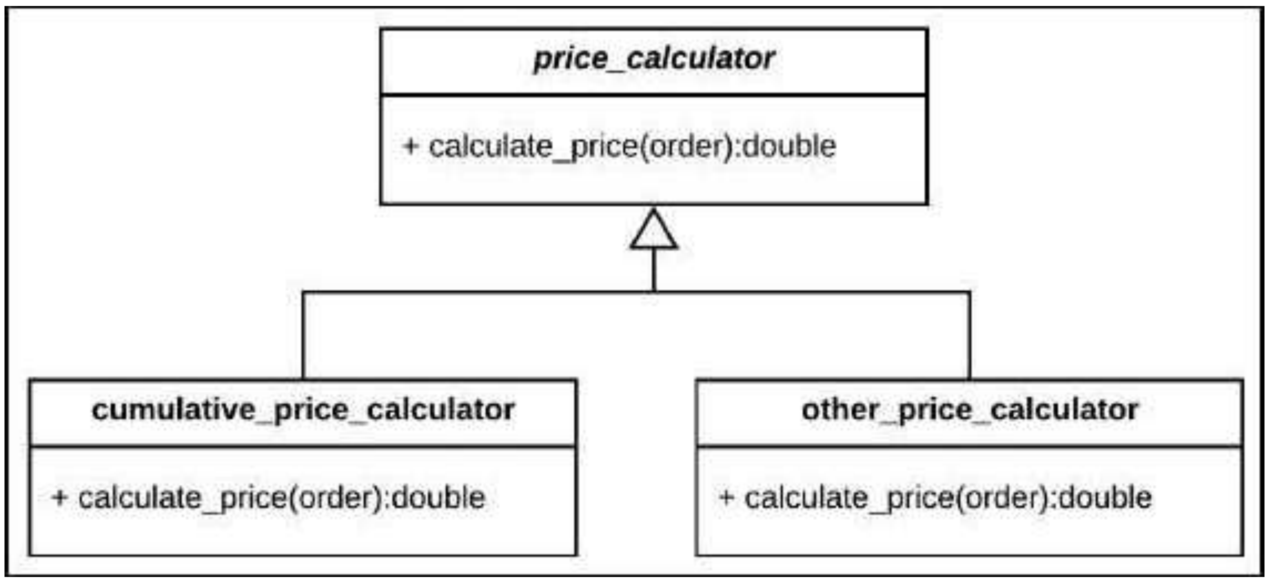
```
   int main()
   {
      std::vector<int> data(count);
      // init data
      auto rmin = pmin(std::begin(data), std::end(data));
      auto rmax = pmin(std::begin(data), std::end(data));
   }
```

```
};

struct volume_discount final : public discount_type
{
   explicit volume_discount(double const quantity,
                            double const discount) noexcept
     : discount(discount), min_quantity(quantity) {}
   virtual double discount_percent(
      double const, double const quantity) const noexcept
   {return quantity >= min_quantity ? discount : 0;}

private:
   double discount;
   double min_quantity;
};

struct price_discount : public discount_type
{
   explicit price_discount(double const price,
                           double const discount) noexcept
      : discount(discount), min_total_price(price) {}
   virtual double discount_percent(
      double const price, double const quantity) const noexcept
   {return price*quantity >= min_total_price ? discount : 0;}

private:
   double discount;
   double min_total_price;
};

struct amount_discount : public discount_type
{
   explicit amount_discount(double const price,
                            double const discount) noexcept
      : discount(discount), min_total_price(price) {}
   virtual double discount_percent(
      double const price, double const) const noexcept
   {return price >= min_total_price ? discount : 0;}

private:
   double discount;
   double min_total_price;
};
```

The classes that model customers, articles, and orders have only a minimum structure, in order to keep the solution simple. They are shown here:

```
struct customer
{
   std::string    name;
   discount_type* discount;
};

enum class article_unit
{
   piece, kg, meter, sqmeter, cmeter, liter
};

struct article
{
   int            id;
   std::string    name;
   double         price;
   article_unit   unit;
   discount_type* discount;
};

struct order_line
{
   article        product;
   int            quantity;
   discount_type* discount;
};

struct order
{
   int                     id;
   customer*               buyer;
   std::vector<order_line> lines;
   discount_type*          discount;
};
```

For computing the final price of an order, we could use various types of calculator. This is yet another instantiation of the strategy pattern:


price_calculator
+ calculate_price(order):double
cumulative_price_calculator
+ calculate_price(order):double
other_price_calculator
+ calculate_price(order):double

`price_calculator` is an abstract base class that has a pure virtual method, `calculate_price()`. The classes derived from `price_calculator`, such as `cumulative_price_calculator`, provide the actual algorithm implementation by overriding the `calculate_price()` method. For simplicity, in this implementation only one concrete strategy for price calculation is provided. As a further exercise, you can implement others:

```
struct price_calculator
{
   virtual double calculate_price(order const & o) = 0;
};

struct cumulative_price_calculator : public price_calculator
{
   virtual double calculate_price(order const & o) override
   {
      double price = 0;

      for(auto ol : o.lines)
      {
         double line_price = ol.product.price * ol.quantity;

         if(ol.product.discount != nullptr)
            line_price *= (1.0 - ol.product.discount->discount_percent(
               ol.product.price, ol.quantity));

         if(ol.discount != nullptr)
```

```
            line_price *= (1.0 - ol.discount->discount_percent(
               ol.product.price, ol.quantity));

         if(o.buyer != nullptr && o.buyer->discount != nullptr)
            line_price *= (1.0 - o.buyer->discount->discount_percent(
               ol.product.price, ol.quantity));

         price += line_price;
      }

      if(o.discount != nullptr)
         price *= (1.0 - o.discount->discount_percent(price, 0));

      return price;
   }
};
```

Here are examples of how to compute the final order price using `cumulative_price_calculator`:

```
inline bool are_equal(double const d1, double const d2,
                      double const diff = 0.001)
{
   return std::abs(d1 - d2) <= diff;
}

int()
{
   fixed_discount  d1(0.1);
   volume_discount d2(10, 0.15);
   price_discount  d3(100, 0.05);
   amount_discount d4(100, 0.05);

   customer c1 {"default", nullptr};
   customer c2 {"john", &d1};
   customer c3 {"joane", &d3};

   article a1 {1, "pen", 5, article_unit::piece, nullptr};
   article a2 {2, "expensive pen", 15, article_unit::piece, &d1};
   article a3 {3, "scissors", 10, article_unit::piece, &d2};

   cumulative_price_calculator calc;

   order o1 {101, &c1, {{a1, 1, nullptr}}, nullptr};
   assert(are_equal(calc.calculate_price(o1), 5));
```

```
    order o3 {103, &c1, {{a2, 1, nullptr}}, nullptr};
    assert(are_equal(calc.calculate_price(o3), 13.5));

    order o6 {106, &c1, {{a3, 15, nullptr}}, nullptr};
    assert(are_equal(calc.calculate_price(o6), 127.5));

    order o9 {109, &c3, {{a2, 20, &d1}}, &d4};
    assert(are_equal(calc.calculate_price(o9), 219.3075));
}
```

9
Data Serialization

Problems

73. Serializing and deserializing data to/from XML

Write a program that can serialize a list of movies to an XML file, and deserialize an XML file with a list of movies. Each movie has a numerical identifier, title, release year, length in minutes, a list of directors, a list of writers, and a list of casting roles with actor name and character name. Such an XML may look like the following:

```
<?xml version="1.0"?>
<movies>
  <movie id="9871" title="Forrest Gump" year="1994" length="202">
    <cast>
      <role star="Tom Hanks" name="Forrest Gump" />
      <role star="Sally Field" name="Mrs. Gump" />
      <role star="Robin Wright" name="Jenny Curran" />
      <role star="Mykelti Williamson" name="Bubba Blue" />
    </cast>
    <directors>
      <director name="Robert Zemeckis" />
    </directors>
    <writers>
      <writer name="Winston Groom" />
      <writer name="Eric Roth" />
    </writers>
  </movie>
  <!-- more movie elements -->
</movies>
```

74. Selecting data from XML using XPath

Consider an XML file with a list of movies as described for the previous problem. Write a program that can select and print the following:

- The title of all the movies released after a given year
- The name of the last actor in the casting list for each movie in the file

75. Serializing data to JSON

Write a program that can serialize a list of movies, as defined for the previous problems, to a JSON file. Each movie has a numerical identifier, title, release year, length in minutes, a list of directors, a list of writers, and a list of casting roles with actor name and character name. The following is an example of the expected JSON format:

```
{
  "movies": [{
    "id": 9871,
    "title": "Forrest Gump",
    "year": 1994,
    "length": 202,
    "cast": [{
        "star": "Tom Hanks",
        "name": "Forrest Gump"
      },
      {
        "star": "Sally Field",
        "name": "Mrs. Gump"
      },
      {
        "star": "Robin Wright",
        "name": "Jenny Curran"
      },
      {
        "star": "Mykelti Williamson",
        "name": "Bubba Blue"
      }
    ],
    "directors": ["Robert Zemeckis"],
    "writers": ["Winston Groom", "Eric Roth"]
  }]
}
```

76. Deserializing data from JSON

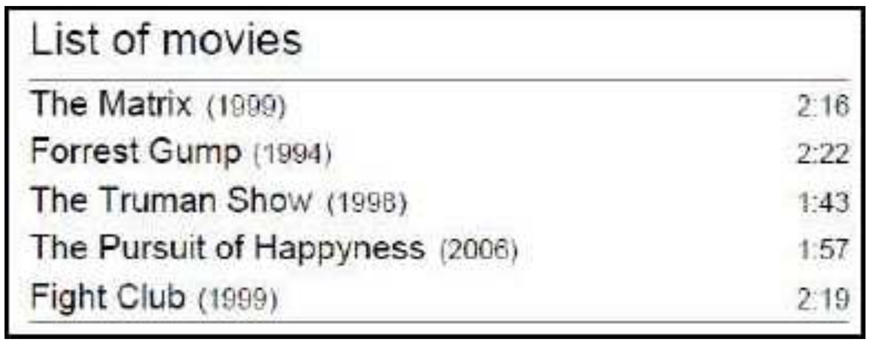
Consider a JSON file with a list of movies as shown in the previous problem. Write a program that can deserialize its content.

77. Printing a list of movies to a PDF

Write a program that can print to a PDF file a list of movies in a tabular form, with the following requirements:

- There must be a heading to the list with the content *List of movies*. This must appear only on the first page of the document.
- For each movie, it should display the title, the release year, and the length.
- The title, followed by the release year in parentheses, must be left-aligned.
- The length, in hours and minutes (for example, 2:12), must be right-aligned.
- There must be a line above and below the movie listing on each page.

Here is an example of such a PDF output:

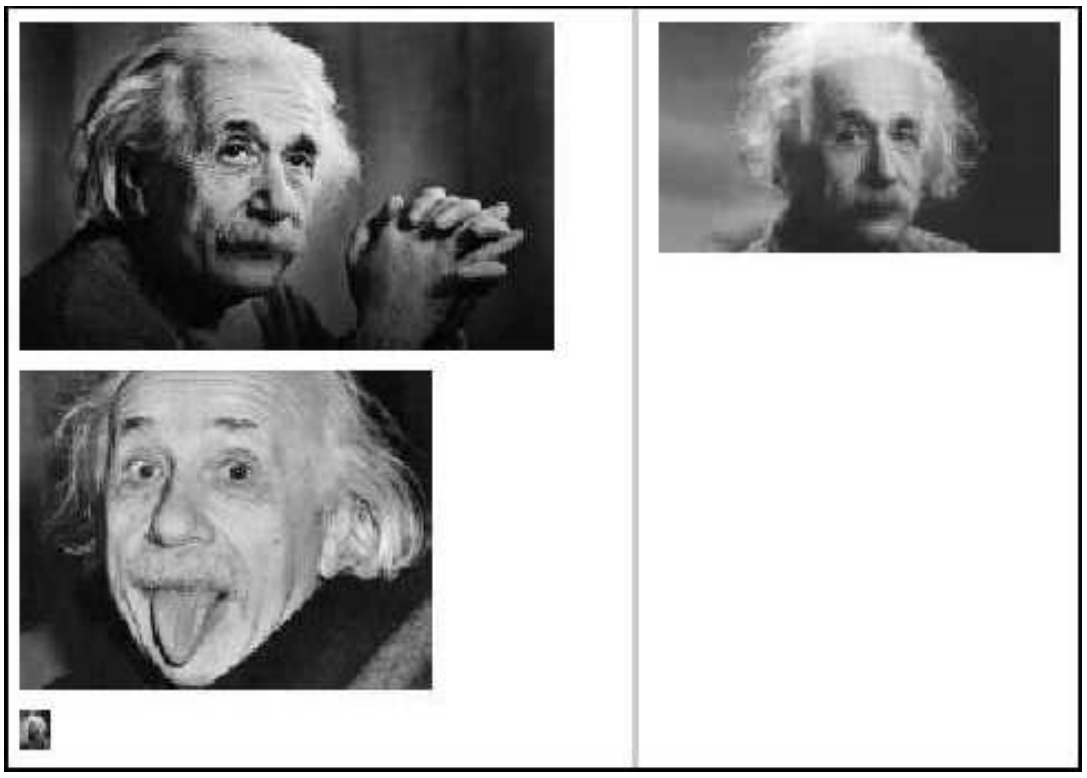


```
List of movies
The Matrix (1999)                          2:16
Forrest Gump (1994)                        2:22
The Truman Show (1998)                     1:43
The Pursuit of Happyness (2006)            1:57
Fight Club (1999)                          2:19
```

78. Creating a PDF from a collection of images

Write a program that can create a PDF document that contains images from a user-specified directory. The images must be displayed one after another. If an image does not fit on the remainder of a page, it must be placed on the next page.

The following is an example of such a PDF file, created from several images of Albert Einstein (these pictures are featured along with the source code accompanying the book):



Solutions

73. Serializing and deserializing data to/from XML

The C++ standard library does not have any support for XML, but there are multiple open source, cross-platform libraries that you can use. Some libraries are lightweight, supporting a set of basic XML features, while others are more complex and rich in functionality. It is up to you to decide which is most suitable for a particular project.

The list of libraries you may want to consider should include *Xerces-C++*, *libxml++*, *tinyxml* or *tinyxml2*, *pugixml*, *gSOAP*, and *RapidXml*. For solving this particular task I will choose *pugixml*. This is a cross-platform, lightweight library, with a fast, although non-validating, XML parser. It has a DOM-like interface with rich traversal/modification capabilities, with support for Unicode and XPath 1.0. On the limitations of the library, it should be mentioned that it lacks support for schema validation. The pugixml library is available at `https://pugixml.org/`.

To represent the movies, as described in the problem, we shall use the following structures:

```
struct casting_role
{
   std::string actor;
   std::string role;
};

struct movie
{
   unsigned int              id;
   std::string               title;
   unsigned int              year;
   unsigned int              length;
   std::vector<casting_role> cast;
   std::vector<std::string>  directors;
   std::vector<std::string>  writers;
};

using movie_list = std::vector<movie>;
```

To create an XML document you must use the `pugi::xml_document` class. After constructing the DOM tree you can save it to a file by calling `save_file()`. Nodes can be added by calling `append_child()`, and attributes with `append_attribute()`. The following method serializes a list of movies in the requested format:

```
void serialize(movie_list const & movies, std::string_view filepath)
{
   pugi::xml_document doc;
   auto root = doc.append_child("movies");

   for (auto const & m : movies)
   {
      auto movie_node = root.append_child("movie");

      movie_node.append_attribute("id").set_value(m.id);
      movie_node.append_attribute("title").set_value(m.title.c_str());
      movie_node.append_attribute("year").set_value(m.year);
      movie_node.append_attribute("length").set_value(m.length);

      auto cast_node = movie_node.append_child("cast");
      for (auto const & c : m.cast)
      {
         auto node = cast_node.append_child("role");
         node.append_attribute("star").set_value(c.actor.c_str());
         node.append_attribute("name").set_value(c.role.c_str());
      }

      auto directors_node = movie_node.append_child("directors");
      for (auto const & director : m.directors)
      {
         directors_node.append_child("director")
                       .append_attribute("name")
                       .set_value(director.c_str());
      }

      auto writers_node = movie_node.append_child("writers");
      for (auto const & writer : m.writers)
      {
         writers_node.append_child("writer")
                     .append_attribute("name")
                     .set_value(writer.c_str());
      }
   }

   doc.save_file(filepath.data());
}
```

```
            if (line == "help") print_commands();
            else if (line == "exit") break;
            else
            {
               if (starts_with(line, "find"))
                  run_find(line, db);
               else if (starts_with(line, "list"))
                  run_list(line, db);
               else if (starts_with(line, "add"))
                  run_add(line, db);
               else if (starts_with(line, "del"))
                  run_del(line, db);
               else
                  std::cout << "unknown command" << std::endl;
            }

            std::cout << std::endl;
         }
      }
      catch (sqlite::sqlite_exception const & e)
      {
         std::cerr << e.get_code() << ": " << e.what() << " during "
                   << e.get_sql() << std::endl;
      }
      catch (std::exception const & e)
      {
         std::cerr << e.what() << std::endl;
      }
   }
```

Each of the supported commands is implemented in a separate function. `run_find()`, `run_list()`, `run_add()`, and `run_del()` parse the user input, call the appropriate function for database access that we have seen earlier, and print the results to the console. These functions do not perform thorough checks on user input. The commands are case-sensitive and must be entered in lowercase.

The function `run_find()` extracts a movie title from the user input, calls `get_movie()` to retrieve the list of all the movies with that title, and prints the result to the console:

```
   void run_find(std::string_view line, sqlite::database & db)
   {
      auto title = trim(line.substr(5));

      auto movies = get_movies(title, db);
      if(movies.empty())
         std::cout << "empty" << std::endl;
      else
      {
         for (auto const m : movies)
         {
            std::cout << m.id << " | "
                      << m.title << " | "
                      << m.year << " | "
                      << m.length << "min"
                      << std::endl;
         }
      }
   }
```

The function `run_list()` extracts a movie's numerical identifier from the user input, calls `get_media()` to retrieve the list of all the media files for that movie, and prints them to the console. This function only prints the length of the blob field and not the entire object:

```
   void run_list(std::string_view line, sqlite::database & db)
   {
      auto movieid = std::stoi(trim(line.substr(5)));
      if (movieid > 0)
      {
         auto list = get_media(movieid, db);
         if (list.empty())
         {
            std::cout << "empty" << std::endl;
         }
         else
         {
            for (auto const & m : list)
            {
               std::cout
                  << m.id << " | "
                  << m.movie_id << " | "
                  << m.name << " | "
                  << m.text.value_or("(null)") << " | "
                  << m.blob.size() << " bytes"
                  << std::endl;
            }
         }
      }
      else
         std::cout << "input error" << std::endl;
   }
```

Adding a file to a movie is done with `run_add()`. This function extracts the movie identifier, the file path, and its description from the comma-separated format in the user input (as in `add <movieid>,<path>,<description>`), loads the content of the file from disk using the helper function `load_image()`, and then adds it as a new record to the `media` table. The implementation seen here does not do any checks on the file type, which makes it possible to actually add any file, not just images or videos, to a movie. You can take it as a further exercise to add additional validation to the program:

```
std::vector<char> load_image(std::string_view filepath)
{
   std::vector<char> data;

   std::ifstream ifile(filepath.data(), std::ios::binary | std::ios::ate);
   if (ifile.is_open())
   {
      auto size = ifile.tellg();
      ifile.seekg(0, std::ios::beg);

      data.resize(static_cast<size_t>(size));
      ifile.read(reinterpret_cast<char*>(data.data()), size);
   }

   return data;
}

void run_add(std::string_view line, sqlite::database & db)
{
   auto parts = split(trim(line.substr(4)), ',');
   if (parts.size() == 3)
   {
      auto movieid = std::stoi(parts[0]);
      auto path = std::experimental::filesystem::path{parts[1]};
      auto desc = parts[2];

      auto content = load_image(parts[1]);
      auto name = path.filename().string();

      auto success = add_media(movieid, name, desc, content, db);
      if (success)
         std::cout << "added" << std::endl;
      else
         std::cout << "failed" << std::endl;
   }
   else
      std::cout << "input error" << std::endl;
}
```

The last command left to implement is deleting a media file. The function `run_del()` takes the identifier of the record in the media table that is supposed to be deleted and calls `delete_media()` to remove it from the table:

```
void run_del(std::string_view line, sqlite::database & db)
{
   auto mediaid = std::stoi(trim(line.substr(4)));
   if (mediaid > 0)
   {
      auto success = delete_media(mediaid, db);
      if (success)
         std::cout << "deleted" << std::endl;
      else
         std::cout << "failed" << std::endl;
   }
   else
      std::cout << "input error" << std::endl;
}
```

In the preceding code, there are several helper functions: `split()`, which splits a text into tokens separated by a specified delimiter character; `starts_with()`, which checks whether a given string starts with a specified sub string; and `trim()`, which removes all the spaces at the beginning and the end of a string. These functions are as follows:

```
std::vector<std::string> split(std::string text, char const delimiter)
{
   auto sstr = std::stringstream{ text };
   auto tokens = std::vector<std::string>{};
   auto token = std::string{};
   while (std::getline(sstr, token, delimiter))
   {
      if (!token.empty()) tokens.push_back(token);
   }
   return tokens;
}

inline bool starts_with(std::string_view text, std::string_view part)
{
   return text.find(part) == 0;
}

inline std::string trim(std::string_view text)
{
   auto first{ text.find_first_not_of(' ') };
   auto last{ text.find_last_not_of(' ') };
   return text.substr(first, (last - first + 1)).data();
}
```

The following is a listing of running several commands, as described previously. We start by displaying all movies called *The Matrix*, although only one is found. Then we list the media files for this movie, but none exist at this point. After that, we add a file called *the_matrix.jpg* from the *res* folder and print the list of media files again. Lastly, we delete the recently added media file and display the files again to make sure the list is empty:

```
find The Matrix
1 | The Matrix | 1999 | 196min

list 1
empty

add 1,.\res\the_matrix.jpg,Main poster
added

list 1
1 | 1 | the_matrix.jpg | Main poster | 193906 bytes

del 1
deleted

list 1
empty
```

11 Cryptography

Problems

88. Caesar cipher

Write a program that can encrypt and decrypt messages using a Caesar cipher with a right rotation and any shift value. For simplicity, the program should consider only uppercase text messages and only encode letters, ignoring digits, symbols, and other types of characters.

89. Vigenère cipher

Write a program that can encrypt and decrypt messages using the Vigenère cipher. For simplicity, the input plain-text messages for encryption should consist of only uppercase letters.

90. Base64 encoding and decoding

Write a program that can encode and decode binary data using the base64 encoding scheme. You must implement the encoding and decoding functions yourself and not use a 3rd party library. The table used for encoding should be the one from the MIME specification.

91. Validating user credentials

Write a program that simulates the way users authenticate to a secured system. In order to log in, a user must be already registered with the system. The user enters a username and a password and the program checks if it matches any of its registered users; if it does, the user is granted access, otherwise, the operation fails. For security reasons, the system must not record the password but use an SHA hash instead.

92. Computing file hashes

Write a program that, given a path to a file, computes and prints to the console the SHA1, SHA256, and MD5 hash values for the content of the file.

93. Encrypting and decrypting files

Write a program that can encrypt and decrypt files using the **Advanced Encryption Standard** (**AES** or **Rijndael**). It should be possible to specify both a source file and a destination file path, as well as a password.

94. File signing

Write a program that is able to sign files and verify that a signed file has not been tampered with, using RSA cryptography. When signing a file, the signature should be written to a separate file and used later for the verification process. The program should provide at least two functions: one that signs a file (taking as arguments the path to the file, the path to the RSA private key, and the path to the file where the signature will be written) and one that verifies a file (taking as arguments the path to the file, the path to the RSA public key, and the path to the signature file).

Solutions

88. Caesar cipher

A *Caesar cipher*, also known as *Caesar's cipher*, *Caesar's code*, *Caesar shift*, or *shift cipher*, is a very old, simple, and widely known encryption technique that substitutes each letter in the plain-text with a letter some fixed number of positions down the alphabet. This method was used by Julius Caesar to protect messages of military importance. He used a shift of three letters, therefore replacing A with D, B with E, and so on. In this encoding, the text CPPCHALLENGER becomes FSSFKDOOHQJHU. The cipher is described in detail on Wikipedia at `https://en.wikipedia.org/wiki/Caesar_cipher`.

Although the Caesar cipher has no place in modern cryptography since it is trivial to break, it is still used on online forums or newsgroups as a way to scramble text to hide spoilers, offensive words, puzzle solutions, and so on. This problem is intended only as a simple exercise along these lines. You should not use such a simple substitution cipher for any cryptographic purposes.

In order to solve the proposed problem, we must implement two functions: one that performs the encryption of a plain-text and one that decrypts an encrypted text. In the code listed as follows:

- `caesar_encrypt()` is a function that takes a `string_view` representing the plain-text and a shift value that indicates how many letters down the alphabet the substitution should occur. This function accounts for and substitutes only uppercase letters and leaves the other characters from the plain-text unmodified. The alphabet is modeled in a circular sequence, so that, in the case of a right shift of 3, X becomes A, Y becomes B, and Z becomes C.
- `caesar_decrypt()` is a function that takes a `string_view` representing a Caesar-encrypted text and a shift value that indicates how many letters down the alphabet (that is, a right rotation) the substitution occurred for the encryption. Like its encryption counterpart, this function only transforms uppercase letters and leaves the others untouched.

```
std::string caesar_encrypt(std::string_view text, int const shift)
{
   std::string str;
   str.reserve(text.length());
   for (auto const c : text)
   {
      if (isalpha(c) && isupper(c))
         str += 'A' + (c - 'A' + shift) % 26;
      else
         str += c;
   }

   return str;
}

std::string caesar_decrypt(std::string_view text, int const shift)
{
   std::string str;
   str.reserve(text.length());
   for (auto const c : text)
   {
      if (isalpha(c) && isupper(c))
         str += 'A' + (26 + c - 'A' - shift) % 26;
      else
         str += c;
   }

   return str;
}
```

The following is an example of how these functions can be used. The plain-text to be encrypted is actually the entire English alphabet, and the encryption/decryption is executed for every possible shift value:

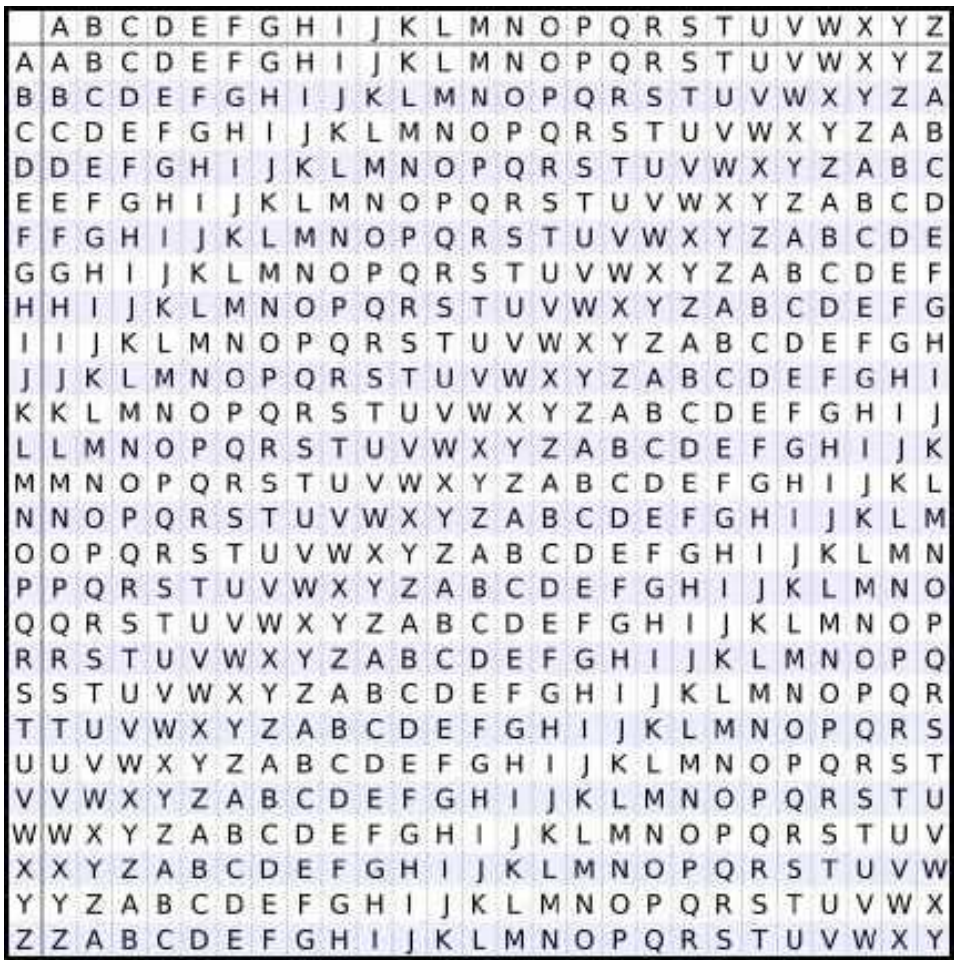
```
int main()
{
   auto text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   for (int i = 1; i <= 26; ++i)
   {
      auto enc = caesar_encrypt(text, i);
      auto dec = caesar_decrypt(enc, i);
      assert(text == dec);
   }
}
```

89. Vigenère cipher

The Vigenère cipher is an encryption technique that uses a series of interwoven Caesar ciphers. Although described in 1553 by Giovan Battista Ballaso, it was misattributed in the 19th century to Blaise de Vigenère and ended up being named after him. The cipher is described in detail on Wikipedia at `https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher`. Only a short summary is presented here.

Although the Vigenère cipher took three centuries to be broken, it is nowadays trivial to break, just as in the case of the Caesar cipher, on which it is based. Like the previous problem, this one is proposed only as a fun and simple exercise and not as an argument in favor of using this cipher for cryptographic purposes.

The technique uses a table called **tabula recta** or a **Vigenère table**. For the English alphabet, this table has 26 rows and 26 columns, where each row is the entire alphabet shifted cyclically using a Caesar cipher. The following image, from the Wikipedia article listed above, shows the content of this table:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

A key is necessary for encryption and decryption. The key is written down until it matches the length of the text to encrypt and respectively decrypt (they have the same size). Encryption is performed by looking at each letter from the plain-text, taking its corresponding letter in the key, and replacing it with the letter found at the intersection of the row corresponding to the key letter and the column corresponding to the plain-text letter. Decryption is done by going to the row that corresponds to the key letter, identifying the encrypted text letter in the row, and using the column label as the letter for the plain-text.

The function that performs the encryption is called `vigenere_encrypt()`. It takes a plain-text and a key, encrypts the plain-text according to the method described previously, and returns the encrypted text:

```
std::string vigenere_encrypt(std::string_view text, std::string_view key)
{
   std::string result;
   result.reserve(text.length());
   static auto table = build_vigenere_table();

   for (size_t i = 0; i < text.length(); ++i)
   {
      auto row = key[i%key.length()] - 'A';
      auto col = text[i] - 'A';

      result += table[row * 26 + col];
   }

   return result;
}
```

Its counterpart is called `vigenere_decrypt()`. This is a function that takes an encrypted text and the key used for encrypting it and decrypts the text using the method described previously, returning the resultant plain-text:

```
std::string vigenere_decrypt(std::string_view text, std::string_view key)
{
   std::string result;
   result.reserve(text.length());
   static auto table = build_vigenere_table();

   for (size_t i = 0; i < text.length(); ++i)
   {
      auto row = key[i%key.length()] - 'A';

      for (size_t col = 0; col < 26; col++)
      {
```

Although the implementation of base64 encoding and decoding provided here is complete, it is not the most performant one. According to my tests, it performs similarly to the implementation available in Boost.Beast. However, I do not necessarily recommend that you use it in production code. Instead, you should use a more thoroughly tested and widely used implementation, such as the ones available in Boost.Beast, Crypto++, or other libraries.

91. Validating user credentials

A good choice for a free, cross-platform C++ library for cryptographic schemes is Crypto++. This library is widely used in both non-commercial and commercial projects, as well as academia, student projects, and others, for its industry-proven implementation of cryptographic functionalities. The library provides support for AES and AES candidates, as well as other block ciphers, message authentication codes, hash functions, public key cryptography, and many other features, including non-cryptographic functionalities such as pseudo-random number generators, prime number generation and verification, DEFLATE compression/decompression, encoding schemes, checksum functions, and more. The library is available at `https://www.cryptopp.com/` and will be used to solve the cryptography problems in this chapter.

When you download the library, you will find several projects corresponding to different configurations of the library. The one you should use is `cryptolib`, which produces a static library. The dynamic library version, `cryptodll`, has been validated by NIST and CSE for FIPS 140-2 Level 1 Conformance. FIPS 140-2 is a series of US government computer security standards that specify requirements for cryptography modules. Because of this compliance, `cryptodll` does not contain anything else that does not meet the requirements, including DES and MD5.

To solve the problem, we will model a system that maintains a database of users. A user has a numerical identifier, a username, the hash value of his password, as well as optional first name and last name inputs. The following class, called `user`, is used for this purpose:

```
struct user
{
   int         id;
   std::string username;
   std::string password;
   std::string firstname;
```

```
    std::string lastname;
};
```

Computing the hash value for a password is done in the function `get_hash()`. This function takes a `string_view` that represents the password (or any text, for that matter) and returns its SHA512 hash value. Crypto++ includes a number of hash functions, including SHA-1, SHA-2 (SHA-224, SHA-256, SHA-384, and SHA-512), SHA-3, Tiger, WHIRLPOOL, RIPEMD-128, RIPEMD-256, RIPEMD-160, and RIPEMD-320, all in the `CryptoPP` namespace, as well as MD5 (in the `CryptoPP::Weak` namespace) if you use the static library version. All these hashes are derived from the class `HashTransformation` and are interchangeable. To compute the hash we must:

- Create a `HashTransformation`-derived object, such as `SHA512`.
- Define an array of bytes large enough to retrieve the hash digest.
- Call `CalculateDigest()`, passing the output buffer, the text to transform, and its length.
- The digest resulting from hashing the original text has a binary form. This can be encoded in a more human-readable form as a string containing hexadecimal digits. This can be done using the `HexEncoder` class. You can attach a sink, such as `StringSink` or `FileSink`, to accumulate the output.

The Crypto++ library uses the concept of a pipeline to flow data from a source to a sink. Within this flow, data can encounter filters that transform it before it reaches the sink. Objects within the pipeline take ownership of other objects passed to them via a pointer in the constructor and automatically destroy them when they are destroyed themselves. The following quote is taken from the library's documentation: *"If a constructor for A takes a pointer to an object B (except primitive types such as int and char), then A owns B and will delete B at A's destruction. If a constructor for A takes a reference to an object B, then the caller retains ownership of B and should not destroy it until A no longer needs it."*

Following is the implementation of the `get_hash()` function:

```
std::string get_hash(std::string_view password)
{
   CryptoPP::SHA512 sha;
   CryptoPP::byte digest[CryptoPP::SHA512::DIGESTSIZE];

   sha.CalculateDigest(
      digest,
      reinterpret_cast<CryptoPP::byte const*>(password.data()),
      password.length());

   CryptoPP::HexEncoder encoder;
   std::string result;

   encoder.Attach(new CryptoPP::StringSink(result));
   encoder.Put(digest, sizeof(digest));
   encoder.MessageEnd();

   return result;
}
```

The following program uses the class `user` and the function `get_hash()` to model the login system. `users`, as the name implies, is a list of users. Although this list is hardcoded, it could be read from a database. You can take it as an extra exercise to store the users in a SQLite database and retrieve it from there. After the user enters his username and password, the program computes the SHA512 hash of the password, checks the list of users for an exact match of the username and the password hash, and displays a message accordingly:

```
   int main()
   {
      std::vector<user> users
      {
         {
            101, "scarface",
   "07A8D53ADAB635ADDF39BAEACFB799FD7C5BFDEE365F3AA721B7E25B54A4E87D419ADDEA34
   BC3073BAC472DCF4657E50C0F6781DDD8FE883653D10F7930E78FF",
            "Tony", "Montana"
         },
         {
            202, "neo",
   "C2CC277BCC10888ECEE90F0F09EE9666199C2699922EFB41EA7E88067B2C075F3DD3FBF3CF
   E9D0EC6173668DD83C111342F91E941A2CADC46A3A814848AA9B05",
            "Thomas", "Anderson"
         },
         {
            303, "godfather",
   "0EA7A0306FE00CD22DF1B835796EC32ACC702208E0B052B15F9393BCCF5EE9ECD8BAAF2784
   0D4D3E6BCC3BB3B009259F6F73CC77480C065DDE67CD9BEA14AA4D",
            "Vito", "Corleone"
         }
      };

      std::string username, password;
      std::cout << "Username: ";
      std::cin >> username;
```

```
    std::cout << "Password: ";
    std::cin >> password;

    auto hash = get_hash(password);

    auto pos = std::find_if(
       std::begin(users), std::end(users),
       [username, hash](user const & u) {
       return u.username == username &&
          u.password == hash; });

    if (pos != std::end(users))
       std::cout << "Login successful!" << std::endl;
    else
       std::cout << "Invalid username or password" << std::endl;
}
```

92. Computing file hashes

File hashes are often used to ensure the integrity of content, such as in the case of downloading a file from the web. Although implementations of the SHA1 and MD5 hashing functions can be found in a variety of libraries, we will again use the Crypto++ library. If you did not follow the previous problem and its solution, *Validating user credentials*, you should do so before continuing with this one, because the general information about the Crypto++ library that was given there will not be repeated here.

Computing a hash for a file is relatively simple using the Crypto++ library. The following code uses several components:

- `FileSource`, which allows reading data from a file using a `BufferedTransformation`. By default, it pumps data in blocks or chunks of 4,096 bytes, although manual pumping is also possible. The constructor used here takes a path to an input file, a Boolean that indicates whether all data should be pumped or not, and a `BufferTransformation` object.
- `HashFilter`, which uses the specified hash algorithm to calculate the hash of all input data up to the first `MessageEnd` signal, at which time it outputs the resultant hash value to its attached transformation.
- `HexEncoder`, which encodes bytes in base 16 using the alphabet `0123456789ABCDEF`.

- `StringSink`, which represents a destination of string data in a pipeline. It takes a reference to a string object where data is to be stored:

`BufferedTransformation` is the basic unit of data flow in Crypto++. It represents a generalization of `BlockTransformation`, `StreamTransformation`, and `HashTransformation`. A `BufferedTransformation` is an object that takes a stream of bytes as input (this may be done in stages), does some computation on them, and then places the result into an internal buffer for later retrieval. Any partial result already in the output buffer is not modified by a further input. Objects deriving from `BufferedTransformation` can participate in pipelining, which allows data to flow from a source to a sink.

```
template <class Hash>
std::string compute_hash(fs::path const & filepath)
{
   std::string digest;
   Hash hash;

   CryptoPP::FileSource source(
      filepath.c_str(),
      true,
      new CryptoPP::HashFilter(hash,
         new CryptoPP::HexEncoder(
            new CryptoPP::StringSink(digest))));

   return digest;
}
```

The function template `compute_hash()` from the preceding code can be used as follows to determine various hashing values:

```
int main()
{
   std::string path;
   std::cout << "Path: ";
   std::cin >> path;

   try
   {
      std::cout << "SHA1: "
                << compute_hash<CryptoPP::SHA1>(path) << std::endl;
      std::cout << "SHA256: "
                << compute_hash<CryptoPP::SHA256>(path) << std::endl;
      std::cout << "MD5: "
```

```
                << compute_hash<CryptoPP::Weak::MD5>(path) << std::endl;
      }
      catch (std::exception const & ex)
      {
         std::cerr << ex.what() << std::endl;
      }
   }
```

It is important to note that the MD5 hash is obsolete and insecure and is only provided for backward compatibility. In order to use it, you must define the `CRYPTOPP_ENABLE_NAMESPACE_WEAK` macro before including the `md5.h` header, as follows:

```
#define CRYPTOPP_ENABLE_NAMESPACE_WEAK 1
#include "md5.h"
```

93. Encrypting and decrypting files

In order to solve this problem with the Crypto++ library, we need to use several components:

- `FileSource`, which allows reading data from a file using a `BufferedTransformation`. By default, it pumps data in blocks or chunks of 4,096 bytes, although manual pumping is also possible.
- `FileSink`, which allows you to write data to a file using a `BufferedTransformation`. It is the companion sink object to a `FileSource` source object.
- `DefaultEncryptorWithMAC` and `DefaultDecryptorWithMAC`, which encrypt and decrypt strings and files with an authentication tag to detect tampering. They use AES as the default block cipher and SHA256 as the default hash for the MAC. Each run through these two classes produces a different result due to the use of a time-based salt.

Two overloads are provided both for encryption and decryption:

- One overload takes a source file path, a destination file path, and a password. It encrypts or decrypts the source file, and the result is written in the destination file.
- The other overload takes a file path and a password. It encrypts or decrypts the file, writing the result in a temporary file, deletes the original file, and then moves the temporary file to the original file path. Its implementation is based on the first overload.

The functions that perform file encryption are shown here:

```
void encrypt_file(fs::path const & sourcefile,
                  fs::path const & destfile,
                  std::string_view password)
{
   CryptoPP::FileSource source(
      sourcefile.c_str(),
      true,
      new CryptoPP::DefaultEncryptorWithMAC(
      (CryptoPP::byte*)password.data(), password.size(),
         new CryptoPP::FileSink(
            destfile.c_str())
      )
   );
}

void encrypt_file(fs::path const & filepath,
                  std::string_view password)
{
   auto temppath = fs::temp_directory_path() / filepath.filename();

   encrypt_file(filepath, temppath, password);

   fs::remove(filepath);
   fs::rename(temppath, filepath);
}
```

The decrypting equivalent functions are basically identical, but instead of using `DefaultEncryptorWithMAC` for the buffered transformation they use `DefaultDecryptorWithMAC`. The two afore mentioned overloads are shown as follows:

```
void decrypt_file(fs::path const & sourcefile,
                  fs::path const & destfile,
                  std::string_view password)
{
   CryptoPP::FileSource source(
      sourcefile.c_str(),
      true,
      new CryptoPP::DefaultDecryptorWithMAC(
      (CryptoPP::byte*)password.data(), password.size(),
         new CryptoPP::FileSink(
            destfile.c_str())
      )
   );
}
```

```
void decrypt_file(fs::path const & filepath,
                  std::string_view password)
{
   auto temppath = fs::temp_directory_path() / filepath.filename();

   decrypt_file(filepath, temppath, password);

   fs::remove(filepath);
   fs::rename(temppath, filepath);
}
```

These functions can be used as follows:

```
int main()
{
   encrypt_file("sample.txt", "sample.txt.enc", "cppchallenger");
   decrypt_file("sample.txt.enc", "sample.txt.dec", "cppchallenger");

   encrypt_file("sample.txt", "cppchallenger");
   decrypt_file("sample.txt", "cppchallenger");
}
```

94. File signing

The process of signing and verifying is similar to encryption and decryption, although it differs in a fundamental way; encryption is done using the public key and decryption using the private key, while signing is done using the private key and verification is done using the public key. Signing helps a recipient that owns a public key to verify that a file is unmodified by using the signature and its public key. Having the public key, however, is not enough to change the file and sign it again. The Crypto++ library is used for solving this problem too.

Although you can use any pair of public-private RSA keys to perform the signing and verification, in the implementation provided here the keys are randomly generated when the program starts. Obviously, in practice, you would generate the keys independent of the signing and verification, and not every time you do that. The function `generate_keys()` , which is shown at the end of the following listing, creates a pair of RSA public-private 3,072-bit keys. Several helper functions, all shown here, are used for this purpose:

```
void encode(fs::path const & filepath,
            CryptoPP::BufferedTransformation const & bt)
{
   CryptoPP::FileSink file(filepath.c_str());
   bt.CopyTo(file);
```

```
      file.MessageEnd();
   }

   void encode_private_key(fs::path const & filepath,
                           CryptoPP::RSA::PrivateKey const & key)
   {
      CryptoPP::ByteQueue queue;
      key.DEREncodePrivateKey(queue);
      encode(filepath, queue);
   }

   void encode_public_key(fs::path const & filepath,
                          CryptoPP::RSA::PublicKey const & key)
   {
      CryptoPP::ByteQueue queue;
      key.DEREncodePublicKey(queue);
      encode(filepath, queue);
   }

   void decode(fs::path const & filepath,
               CryptoPP::BufferedTransformation& bt)
   {
      CryptoPP::FileSource file(filepath.c_str(), true);
      file.TransferTo(bt);
      bt.MessageEnd();
   }

   void decode_private_key(fs::path const & filepath,
                           CryptoPP::RSA::PrivateKey& key)
   {
      CryptoPP::ByteQueue queue;
      decode(filepath, queue);
      key.BERDecodePrivateKey(queue, false, queue.MaxRetrievable());
   }

   void decode_public_key(fs::path const & filepath,
                          CryptoPP::RSA::PublicKey& key)
   {
      CryptoPP::ByteQueue queue;
      decode(filepath, queue);
      key.BERDecodePublicKey(queue, false, queue.MaxRetrievable());
   }

   void generate_keys(fs::path const & privateKeyPath,
                      fs::path const & publicKeyPath,
                      CryptoPP::RandomNumberGenerator& rng)
   {
      try
      {
         CryptoPP::RSA::PrivateKey rsaPrivate;
         rsaPrivate.GenerateRandomWithKeySize(rng, 3072);

         CryptoPP::RSA::PublicKey rsaPublic(rsaPrivate);

         encode_private_key(privateKeyPath, rsaPrivate);
         encode_public_key(publicKeyPath, rsaPublic);
      }
      catch (CryptoPP::Exception const & e)
      {
         std::cerr << e.what() << std::endl;
      }
   }
```

In order to perform the signing, we use a pipeline that starts with a `FileSource`, ends with a `FileSink`, and contains a filter called `SignerFilter`, which creates a signature over a message. It uses the `RSASSA_PKCS1v15_SHA_Signer` signer to transform the source data:

```
   void rsa_sign_file(fs::path const & filepath,
                      fs::path const & privateKeyPath,
                      fs::path const & signaturePath,
                      CryptoPP::RandomNumberGenerator& rng)
   {
      CryptoPP::RSA::PrivateKey privateKey;
      decode_private_key(privateKeyPath, privateKey);

      CryptoPP::RSASSA_PKCS1v15_SHA_Signer signer(privateKey);

      CryptoPP::FileSource fileSource(
         filepath.c_str(),
         true,
         new CryptoPP::SignerFilter(
            rng,
            signer,
            new CryptoPP::FileSink(
               signaturePath.c_str())));
   }
```

The opposite process of verification is implemented in a similar way. The filter used in this case is `SignatureVerificationFilter`, which is the counterpart of `SignerFilter`, and the verifier is `RSASSA_PKCS1v15_SHA_Verifier`, which is the counterpart of `RSASSA_PKCS1v15_SHA_Signer`:

```
   bool rsa_verify_file(fs::path const & filepath,
                        fs::path const & publicKeyPath,
                        fs::path const & signaturePath)
```

```
{
   CryptoPP::RSA::PublicKey publicKey;
   decode_public_key(publicKeyPath.c_str(), publicKey);

   CryptoPP::RSASSA_PKCS1v15_SHA_Verifier verifier(publicKey);

   CryptoPP::FileSource signatureFile(signaturePath.c_str(),
                                      true);

   if (signatureFile.MaxRetrievable() != verifier.SignatureLength())
      return false;

   CryptoPP::SecByteBlock signature(verifier.SignatureLength());
   signatureFile.Get(signature, signature.size());

   auto* verifierFilter =
      new CryptoPP::SignatureVerificationFilter(verifier);
   verifierFilter->Put(signature, verifier.SignatureLength());

   CryptoPP::FileSource fileSource(
      filepath.c_str(),
      true,
      verifierFilter);

   return verifierFilter->GetLastResult();
}
```

The following program generates a pair of RSA public-private keys, then uses the private key to sign a file using the `rsa_sign_file()` function, and then uses the public key and the signature file to verify the file using the `rsa_verify_file()` counterpart function:

```
int main()
{
   CryptoPP::AutoSeededRandomPool rng;

   generate_keys("rsa-private.key", "rsa-public.key", rng);

   rsa_sign_file("sample.txt", "rsa-private.key", "sample.sign", rng);

   auto success =
      rsa_verify_file("sample.txt", "rsa-public.key", "sample.sign");

   assert(success);
}
```

12
Networking and Services

Problems

95. Finding the IP address of a host

Write a program that can retrieve and print the IPv4 address of a host. If multiple addresses are found, then all of them should be printed. The program should work on all platforms.

96. Client-server Fizz-Buzz

Write a client-server application that can be used for playing the F*izz-Buzz* game. The client sends numbers to the server that answer back with fizz, buzz, fizz-buzz, or the number itself, according to the game rules. Communication between the client and the server must be done over TCP. The server should run indefinitely. The client should run as long as the user enters numbers between 1 and 99.

Fizz-Buzz is a game for children, intended to teach them arithmetic division. A player must say a number and another player should answer with:

- Fizz, if the number is divisible by 3
- Buzz, if the number is divisible by 5
- Fizz-buzz, if the number is divisible by both 3 and 5
- The number itself in all other cases

What we get back in case of success is an XML string representing the translated text. The text is encoded with UTF-8. At this point, it is not possible to receive the result as a JSON. For the preceding example, the result from the server is:

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Salut
tout le monde !</string>
```

We can encapsulate the text translation functionality into a class that can handle application keys and endpoints in order to make the translation function simpler. The following `text_translator` class does exactly that. It is constructed from two strings, one representing the endpoint for the Text Translation API, and the other being the application key. As mentioned previously, the result from the server is returned in an XML format. The member function `deserialize_result()` extracts the actual text from its XML serialized form. However, to keep it simple, it just uses a regular expression to do so, and not an XML library, which should be enough for the purpose of this demo:

```
class text_translator
{
public:
   text_translator(std::string_view endpoint,
                   std::string_view key)
      : endpoint(endpoint), app_key(key)
   {}

   std::wstring translate_text(std::wstring_view wtext,
                               std::string_view to,
                               std::string_view from = "en");

private:
   std::string deserialize_result(std::string_view text)
   {
      std::regex rx(R"(<string.*>(.*)<\/string>)");
      std::cmatch match;
      if (std::regex_search(text.data(), match, rx))
      {
         return match[1];
      }

      return "";
   }

   std::string endpoint;
   std::string app_key;
};
```

The `translate_text()` member function performs the actual translation. Its inputs are the text to translate, the language to translate to, and the language of the text, which by default is English. The input text for this method is a UTF-16 character string, but it must be converted to UTF-8. Also, the return from the server is text encoded as UTF-8 and must be converted to UTF-16. This is done with the helper functions `utf16_to_utf8()` and `utf8_to_utf16()`:

```
std::wstring text_translator::translate_text(std::wstring_view wtext,
                                             std::string_view to,
                                             std::string_view from = "en")
{
   try
   {
      using namespace std::string_literals;

      std::stringstream str;
      std::string text = utf16_to_utf8(wtext);

      curl::curl_ios<std::stringstream> writer(str);
      curl::curl_easy easy(writer);

      curl::curl_header header;
      header.add("Ocp-Apim-Subscription-Key:" + app_key);

      easy.escape(text);
      auto url = endpoint + "/Translate";
      url += "?from="s + from.data();
      url += "&to="s + to.data();
      url += "&text="s + text;

      easy.add<CURLOPT_URL>(url.c_str());
      easy.add<CURLOPT_HTTPHEADER>(header.get());

      easy.perform();

      auto result = deserialize_result(str.str());
      return utf8_to_utf16(result);
   }
   catch (curl::curl_easy_exception const & error)
   {
      auto errors = error.get_traceback();
      error.print_traceback();
   }
   catch (std::exception const & ex)
   {
      std::err << ex.what() << std::endl;
   }

   return {};
}
```

The two helper functions for converting between UTF-8 and UTF-16 are as follows:

```
std::wstring utf8_to_utf16(std::string_view text)
{
   std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;
   std::wstring wtext = converter.from_bytes(text.data());
   return wtext;
}

std::string utf16_to_utf8(std::wstring_view wtext)
{
   std::wstring_convert<std::codecvt_utf8_utf16<wchar_t>> converter;
   std::string text = converter.to_bytes(wtext.data());
   return text;
}
```

The `text_translator` class can be used to translate texts between various languages, as shown in the following example:

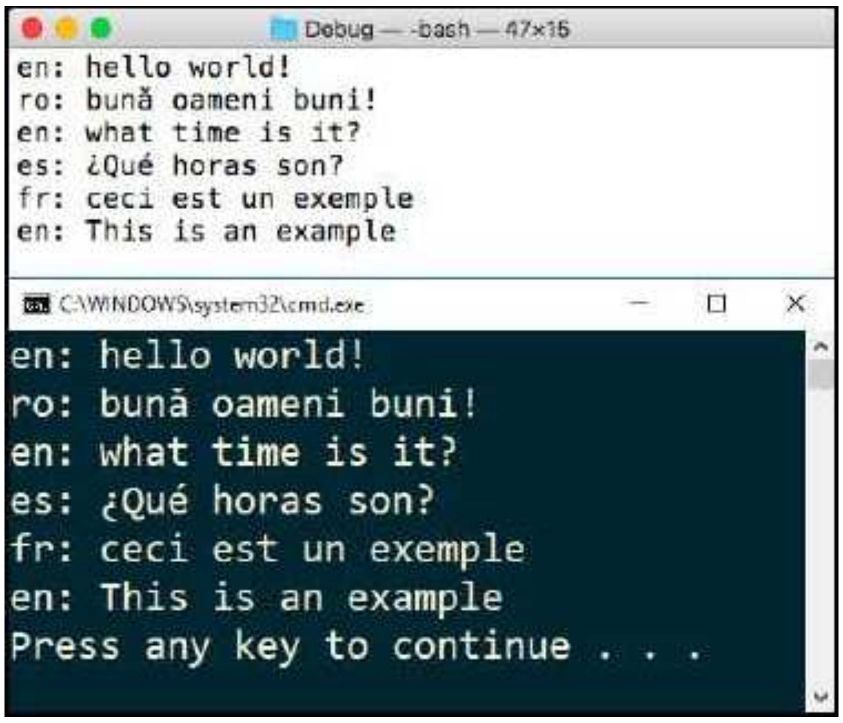
```
int main()
{
#ifdef _WIN32
   SetConsoleOutputCP(CP_UTF8);
#endif

   set_utf8_conversion(std::wcout);

   text_translator tt(
      "https://api.microsofttranslator.com/V2/Http.svc",
      "...(your app key)...");

   std::vector<std::tuple<std::wstring, std::string, std::string>> texts
   {
      { L"hello world!", "en", "ro"},
      { L"what time is it?", "en", "es" },
      { L"ceci est un exemple", "fr", "en" }
   };

   for (auto const [text, from, to] : texts)
   {
      auto result = tt.translate_text(text, to, from);

      std::cout << from << ": ";
```

```
         std::wcout << text << std::endl;
         std::cout << to << ": ";
         std::wcout << result << std::endl;
      }
   }
```

Printing UTF-8 characters to a console is, however, not straightforward. On Windows, you need to call `SetConsoleOutputCP(CP_UTF8)` to enable an appropriate code page for that. But you also need to set a proper UTF-8 locale for the output stream, which is done with the `set_utf8_conversion()` function:

```
   void set_utf8_conversion(std::wostream& stream)
   {
      auto codecvt = std::make_unique<std::codecvt_utf8<wchar_t>>();
      std::locale utf8locale(std::locale(), codecvt.get());
      codecvt.release();
      stream.imbue(utf8locale);
   }
```

The output for running the preceding example is as follows:



```
en: hello world!
ro: bună oameni buni!
en: what time is it?
es: ¿Qué horas son?
fr: ceci est un exemple
en: This is an example
```

```
en: hello world!
ro: bună oameni buni!
en: what time is it?
es: ¿Qué horas son?
fr: ceci est un exemple
en: This is an example
Press any key to continue . . .
```

100. Detecting faces in a picture

This is yet another problem that can be solved using Microsoft Cognitive Services. One of the services available in this group, called *Face API*, provides algorithms for detecting faces, gender, age, emotion, and various face landmarks and attributes, as well as the ability to find face similarities, identify people, group pictures based on visual faces similarities, and others.

Similar to the Text Translate API, there is a free plan that allows up to 30,000 transactions per month, but only 20 every minute. A transaction is basically an API call. There are several paid plans that allow for more transactions per month and per minute, but for the purpose of this problem, you can use the free tier. There is also a 30-day trial that you can use. To get started with the Face API, you have to:

1. Have an Azure account. You should create one if you don't already have one.
2. Create a new Face API resource.
3. After the resource is created, navigate to it and copy one of the two application keys generated for it and the resource endpoint. These are both necessary in order to call the service.

Documentation for the Face API is available at: `https://azure.microsoft.com/en-us/services/cognitive-services/face/`. You should read the information about the `Detect` method carefully. In short, what we have to do is the following:

- Make a `POST` request to `[endpoint]/Detect`.
- Provide optional query parameters, such as flags for returning the face ID, the face landmarks, and a string to indicate what face attributes to analyze and return.
- Provide optional and mandatory request headers. At a minimum, `Ocp-Apim-Subscription-Key` is required to pass the application key of the Azure resource.
- Provide the image to analyze. You can either pass an URL to an image in a JSON object (with the content type `application/json`), or the actual image (with the content type `application/octet-stream`). The requirement was that the picture should be loaded from a disk file, therefore we must use the latter option.

In the case of success, the response is a JSON object containing all of the requested information. In the case of failure, the response is another JSON object with information about the error.

Here is a request to analyze and return the face landmarks, age, gender, and emotion, as well as the face identifier. Information about the identified face is preserved on the server for 24 hours and can be used with other Face API algorithms:

```
POST
/face/v1.0/detect?returnFaceId=true&returnFaceLandmarks=true&returnFaceAttr
ibutes=age,gender,emotion
host: westeurope.api.cognitive.microsoft.com
ocp-apim-subscription-key: <your key here>
content-type: application/octet-stream
content-length: <length>
accept: */*
```

The JSON result returned by the server looks like the following. Notice this is only a snippet because the entire response is rather long. The actual result includes 27 different face landmarks, with only the first two being shown in the following output:

```
[{
  "faceId": "0ddb348a-6038-4cbb-b3a1-86fffe6c1f26",
  "faceRectangle": {
    "top": 86,
    "left": 165,
    "width": 72,
    "height": 72
  },
  "faceLandmarks": {
    "pupilLeft": {
      "x": 187.5,
      "y": 102.9
    },
    "pupilRight": {
      "x": 214.6,
      "y": 104.7
    }
  },
  "faceAttributes": {
    "gender": "male",
    "age": 54.9,
    "emotion": {
      "anger": 0,
      "contempt": 0,
      "disgust": 0,
      "fear": 0,
      "happiness": 1,
      "neutral": 0,
      "sadness": 0,
      "surprise": 0
      }
    }
  }]
```

We will use the `nlohmann/json` library to deserialize the JSON object and *libcurl* to make HTTP requests. The following classes model the result from the server in case of success:

```
struct face_rectangle
{
   int width = 0;
   int height = 0;
   int left = 0;
   int top = 0;
};

struct face_point
{
   double x = 0;
   double y = 0;
};

struct face_landmarks
{
   face_point pupilLeft;
   face_point pupilRight;
   face_point noseTip;
   face_point mouthLeft;
   face_point mouthRight;
   face_point eyebrowLeftOuter;
   face_point eyebrowLeftInner;
   face_point eyeLeftOuter;
   face_point eyeLeftTop;
   face_point eyeLeftBottom;
   face_point eyeLeftInner;
   face_point eyebrowRightInner;
   face_point eyebrowRightOuter;
   face_point eyeRightInner;
   face_point eyeRightTop;
   face_point eyeRightBottom;
   face_point eyeRightOuter;
   face_point noseRootLeft;
   face_point noseRootRight;
   face_point noseLeftAlarTop;
   face_point noseRightAlarTop;
   face_point noseLeftAlarOutTip;
   face_point noseRightAlarOutTip;
   face_point upperLipTop;
   face_point upperLipBottom;
   face_point underLipTop;
   face_point underLipBottom;
};

struct face_emotion
{
   double anger = 0;
   double contempt = 0;
   double disgust = 0;
   double fear = 0;
   double happiness = 0;
   double neutral = 0;
   double sadness = 0;
   double surprise = 0;
};

struct face_attributes
{
   std::string  gender;
   double       age;
   face_emotion emotion;
};

struct face_info
{
   std::string     faceId;
   face_rectangle  rectangle;
   face_landmarks  landmarks;
   face_attributes attributes;
};
```

Because a picture may contain multiple faces, the actual response from the server is an array of objects. The `face_detect_response` shown in the following code is the actual type of the response:

```
using face_detect_response = std::vector<face_info>;
```

Deserialization is done as in other cases in this book, by using `from_json()` overloaded functions. If you have already solved the other problems involving JSON deserialization, you should be very familiar with these:

```
using json = nlohmann::json;

void from_json(const json& jdata, face_rectangle& rect)
{
   rect.width = jdata.at("width").get<int>();
```

```
void from_json(const json& jdata, face_error_response& response)
{
   response.error = jdata.at("error");
}
```

Having all of these defined so far, we can write the actual calls to Face API. Like in the case of text translation, we can write a class to encapsulate this functionality (where we can add more). This can help to easily manage the application key and endpoint (instead of passing them with every single function call). The `face_manager` class is used for this purpose:

```
class face_manager
{
public:
   face_manager(std::string_view endpoint,
                std::string_view key)
      : endpoint(endpoint), app_key(key)
   {}

   face_detect_response detect_from_file(std::string_view path);

private:
   face_detect_response parse_detect_response(long const status,
                                              std::stringstream & str);

   std::string endpoint;
   std::string app_key;
};
```

The `detect_from_file()` method takes a string representing the path to an image on the disk. It loads the image, sends it to Face API, deserializes the answer, and returns a `face_detect_response` object, which is a collection of `face_info` objects. Because the actual image is passed with the call, the content type is `application/octet-stream`. We need to pass the content of the file into the `CURLOPT_POSTFIELDS` field with the `curl_easy` interface, and its length into the `CURLOPT_POSTFIELDSIZE` field:

```
face_detect_response face_manager::detect_from_file(std::string_view path)
{
   try
   {
      auto data = load_image(path);
      if (!data.empty())
      {
         std::stringstream str;
         curl::curl_ios<std::stringstream> writer(str);
         curl::curl_easy easy(writer);
```

```
            curl::curl_header header;
            header.add("Ocp-Apim-Subscription-Key:" + app_key);
            header.add("Content-Type:application/octet-stream");

            auto url = endpoint +
                       "/detect"
                       "?returnFaceId=true"
                       "&returnFaceLandmarks=true"
                       "&returnFaceAttributes=age,gender,emotion";

            easy.add<CURLOPT_URL>(url.c_str());
            easy.add<CURLOPT_HTTPHEADER>(header.get());

            easy.add<CURLOPT_POSTFIELDSIZE>(data.size());
            easy.add<CURLOPT_POSTFIELDS>(reinterpret_cast<char*>(
               data.data()));

            easy.perform();

            auto status = easy.get_info<CURLINFO_RESPONSE_CODE>();

            return parse_detect_response(status.get(), str);
         }
      }
      catch (curl::curl_easy_exception const & error)
      {
         auto errors = error.get_traceback();
         error.print_traceback();
      }
      catch (std::exception const & ex)
      {
         std::cerr << ex.what() << std::endl;
      }

      return {};
   }
```

The `parse_detect_response()` method is responsible for deserializing the JSON response from the server. It does so based on the actual HTTP response code. If the function succeeded, the status is `200`. In the case of failure, it is a `4xx` code:

```
   face_detect_response face_manager::parse_detect_response(
      long const status, std::stringstream & str)
   {
      json jdata;
      str >> jdata;
      try
      {
         if (status == 200)
         {
            face_detect_response response = jdata;

            return response;
         }
         else if (status >= 400)
         {
            face_error_response response = jdata;

            std::cout << response.error.code << std::endl
                      << response.error.message << std::endl;
         }
      }
      catch (std::exception const & ex)
      {
         std::cerr << ex.what() << std::endl;
      }

      return {};
   }
```

To read the image file from the disk, the `detect_from_file()` function uses another function called `load_image()`. This function takes a string representing the path to the file and returns the content of the file in an `std::vector<uint8_t>`. The implementation of this function is as follows:

```
   std::vector<uint8_t> load_image(std::string_view filepath)
   {
      std::vector<uint8_t> data;

      std::ifstream ifile(filepath.data(), std::ios::binary | std::ios::ate);
      if (ifile.is_open())
      {
         auto size = ifile.tellg();
         ifile.seekg(0, std::ios::beg);

         data.resize(static_cast<size_t>(size));
         ifile.read(reinterpret_cast<char*>(data.data()), size);
      }

      return data;
   }
```

At this point, we have all that is necessary to make calls to the `Detect` algorithm from Face API, deserialize the response, and print its content to the console. The following program prints information for the faces identified in a file called `albert_and_elsa.jpg` from the `res` folder of the project. Remember to use your actual endpoint and application key for your Face API resource:

```
int main()
{
   face_manager manager(
      "https://westeurope.api.cognitive.microsoft.com/face/v1.0",
      "...(your api key)...");

#ifdef _WIN32
   std::string path = R"(res\albert_and_elsa.jpg)";
#else
   std::string path = R"(./res/albert_and_elsa.jpg)";
#endif

   auto results = manager.detect_from_file(path);

   for (auto const & face : results)
   {
      std::cout << "faceId: " << face.faceId << std::endl
                << "age: " << face.attributes.age << std::endl
                << "gender: " << face.attributes.gender << std::endl
                << "rect: " << "{" << face.rectangle.left
                << "," << face.rectangle.top
                << "," << face.rectangle.width
                << "," << face.rectangle.height
                << "}" << std::endl << std::endl;
   }
}
```

The image `albert_and_elsa.jpg` is also shown here:

The following is the program's output. Keep in mind that the actual temporary face identifiers will, of course, differ with each call. As you can see in the result, two faces were identified. The first is of Albert Einstein and his detected age is 54.9 years old. This picture was taken in 1921 when he was 42. The second face is of Elsa Einstein, the wife of Albert Einstein, who was 45 at that time. In her case, the detected age is 41.6 years old. From this, you can see that the detected age is only a rough indication, and not something that is very precise:

```
faceId: 77e0536f-073d-41c5-920d-c53264d17b98
age:    54.9
gender: male
rect:   {165,86,72,72}

faceId: afb22044-14fa-46bf-9b65-16d4fe1d9817
age:    41.6
gender: female
rect:   {321,151,59,59}
```

Should the API call fail, an error message is returned instead (with the HTTP status `400`). The `parse_detect_response()` method will deserialize the error response and print a message to the console. For instance, in case a wrong API key is used, the following message is returned from the server and displayed in the console:

```
Unspecified
Access denied due to invalid subscription key. Make sure you are subscribed
to an API you are trying to call and provide the right key.
```

Bibliography

Articles

- 1337C0D3R, 2011. *Longest Palindromic Substring Part I*, https://articles.leetcode.com/longest-palindromic-substring-part-i/
- Aditya Goel, 2016. *Permutations of a given string using STL*, https://www.geeksforgeeks.org/permutations-of-a-given-string-using-stl/
- Andrei Jakab, 2010. *Using libcurl with SSH support in Visual Studio 2010*, https://curl.haxx.se/libcurl/c/Using-libcurl-with-SSH-support-in-Visual-Studio-2010.pdf
- Ashwani Gautam, 2017. *What is the analysis of quick sort?*, https://www.quora.com/What-is-the-analysis-of-quick-sort
- Ashwin Nanjappa, 2014. *How to build Boost using Visual Studio*, https://codeyarns.com/2014/06/06/how-to-build-boost-using-visual-studio/
- busycrack, 2012. *Telnet IMAP Commands Note*, https://busylog.net/telnet-imap-commands-note/
- Dan Madden, 2000. *Encrypting Log Files*, https://www.codeproject.com/Articles/644/Encrypting-Log-Files
- Georgy Gimel'farb, 2016. *Algorithm Quicksort: Analysis of Complexity*, https://www.cs.auckland.ac.nz/courses/compsci220s1c/lectures/2016S1C/CS220-Lecture10.pdf
- Jay Doshi, Chanchal Khemani, Juhi Duseja. *Dijkstra's Algorithm*, http://codersmaze.com/data-structure-explanations/graphs-data-structure/dijkstras-algorithm-for-shortest-path/
- Jeffrey Walton, 2008. *Applied Crypto++: Block Ciphers*, https://www.codeproject.com/Articles/21877/Applied-Crypto-Block-Ciphers
- Jeffrey Walton, 2007. *Product Keys Based on the Advanced Encryption Standard (AES)*, https://www.codeproject.com/Articles/16465/Product-Keys-Based-on-the-Advanced-Encryption-Stan

- Jeffrey Walton, 2006. *Compiling and Integrating Crypto++ into the Microsoft Visual C++ Environment*, `https://www.codeguru.com/cpp/v-s/devstudio_macros/openfaq/article.php/c12853/Compiling-and-Integrating-Crypto-into-the-Microsoft-Visual-C-Environment.htm`
- Jonathan Boccara, 2017. *How to split a string in C++*, `https://www.fluentcpp.com/2017/04/21/how-to-split-a-string-in-c/`
- Kenny Kerr, 2013. *Resource Management in the Windows API*, `https://visualstudiomagazine.com/articles/2013/09/01/get-a-handle-on-the-windows-api.aspx`
- Kenny Kerr, 2011. *Windows with C++ - C++ and the Windows API*, `https://msdn.microsoft.com/en-us/magazine/hh288076.aspx?f=255MSPPError=-2147217396`
- Marius Bancila, 2015. *Integrate Windows Azure Face APIs in a C++ application*, `https://www.codeproject.com/Articles/989752/Integrate-Windows-Azure-Face-APIs-in-a-Cplusplus-a`
- Marius Bancila, 2018. *Using Cognitive Services to find your Game of Thrones look-alike*, `https://www.codeproject.com/Articles/1234217/Using-Cognitive-Services-to-find-your-Game-of-Thro`
- Mary K. Vernon. *Priority Queues*, `http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html`
- Mathias Bynens. *In search of the perfect URL validation regex*, `https://mathiasbynens.be/demo/url-regex`
- O.S. Tezer, 2014. *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems*, `https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems`
- Robert Nystrom, 2014. *Game Programming patterns: Double Buffer*, `http://gameprogrammingpatterns.com/double-buffer.html`
- Robert Sedgewick, Philippe Flajolet, 2013. *Introduction to the Analysis of Algorithms*, `http://www.informit.com/articles/article.aspx?p=2017754 seqNum=5`
- Rosso Salmanzadeh, 2002. *Using libcurl in Visual Studio*, `https://curl.haxx.se/libcurl/c/visual_studio.pdf`
- Sergii Bratus, 2010. *Implementation of the Licensing System for a Software Product*, `https://www.codeproject.com/Articles/99499/Implementation-of-the-Licensing-System-for-a-Softw`
- Shubham Agrawal, 2016. *Dijkstra's Shortest Path Algorithm using priority_queue of STL*, `https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/`

- Travis Tidwell, 2013. *An Online RSA Public and Private Key Generator,* `http://travistidwell.com/blog/2013/09/06/an-online-rsa-public-and-private-key-generator/`
- Victor Volkman, 2006. *Crypto++ Holds the Key to Encrypting Your C++ Application Data,* `https://www.codeguru.com/cpp/misc/misc/cryptoapi/article.php/c11953/Cryptosupregsup-Holds-the-Key-to-Encrypting-Your-C-Application-Data.htm`
- Yang Song, 2014. *Split a string using C++,* `http://ysonggit.github.io/coding/2014/12/16/split-a-string-using-c.html`
- *Decorator Design Pattern,* `https://sourcemaking.com/design_patterns/decorator`
- *Composite Design Pattern,* `https://sourcemaking.com/design_patterns/composite`
- *Template Method Design Pattern,* `https://sourcemaking.com/design_patterns/template_method`
- *Strategy Design Pattern,* `https://sourcemaking.com/design_patterns/strategy`
- *Chain of Responsibility,* `https://sourcemaking.com/design_patterns/chain_of_responsibility`
- *Understanding the PDF File Format: Overview,* `https://blog.idrsolutions.com/2013/01/understanding-the-pdf-file-format-overview/`
- *RSA Signing is Not RSA Decryption,* `https://www.cs.cornell.edu/courses/cs5430/2015sp/notes/rsa_sign_vs_dec.php`
- *RSA Cryptography,* `https://www.cryptopp.com/wiki/RSA_Cryptography`
- *Using rand() (C/C++),* `http://eternallyconfuzzled.com/arts/jsw_art_rand.aspx`
- *Crypto++ Keys and Formats,* `https://www.cryptopp.com/wiki/Keys_and_Formats`
- *INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4 rev1,* `https://tools.ietf.org/html/rfc3501.html`
- *Internal Versus External BLOBs in SQLite,* `https://www.sqlite.org/intern-v-extern-blob.html`
- *OpenSSL Compilation and Installation,* `https://wiki.openssl.org/index.php/Compilation_and_Installation`

Library documentation

- *C/C++ JSON parser/generator benchmark,* `https://github.com/miloyip/nativejson-benchmark`
- *Crypto++,* `https://www.cryptopp.com/wiki/Main_Page`
- *Hummus PDF,* `http://pdfhummus.com/How-To`
- *JSON for Modern C++,* `https://github.com/nlohmann/json`
- *PDF-Writer,* `https://github.com/galkahana/PDF-Writer`
- *PNGWriter,* `https://github.com/pngwriter/pngwriter`
- *pugixml 1.8 quick start guide,* `https://pugixml.org/docs/quickstart.html`
- *SQLite,* `https://www.sqlite.org/docs.html`
- *sqlite_modern_cpp,* `https://github.com/SqliteModernCpp/sqlite_modern_cpp`
- *Ziplib wiki,* `https://bitbucket.org/wbenny/ziplib/wiki/Home`

Other Books You May Enjoy