4 Assignment 2 [Assignment ID: `cpp_compile_time`]

4.1 Preamble (Please Read Carefully)

Before starting work on this assignment, it is **critically important** that you **carefully** read Section 1 (titled "General Information") which starts on page 1 of this document.

4.2 Topics Covered

This assignment covers material primarily related to the following: compile-time computation, constexpr, literal types.

4.3 Problems — Part A

- 8.1 [lvalues/rvalues]
- 8.24 [moving vs. copying]
- 8.25 [temporary objects]
- 8.26 [data structures]

4.4 Problems — Part B

1. *Constexpr basic string class template (*`cexpr_basic_string`*).* In this exercise, a class template for representing a sequence of characters (i.e., a string) that can be used in constexpr contexts will be developed. This class template is called `cexpr_basic_string`. The `cexpr_basic_string` class template has two template parameters:

   (a) `T`. The type of each character in the string (e.g., **`char`**, **`unsigned char`**, **`wchar_t`**).
   (b) `M`. The maximum number of characters that can be held in the string (which does not include the dummy null character, discussed later).

   The class template uses a C-style array (embedded in the `cexpr_basic_string` object itself) for storing the character data for a string. The `cexpr_basic_string` class template has the interface given in Listing 1.

   The `cexpr_basic_string` class template always stores character string data internally in null-terminated form. That is, an additional dummy null-character (i.e., a character with the value `value_type(0)`) is always stored immediately following the last character in the string. This dummy character is always present, even in the case of an empty string. This dummy character is not considered one of the characters in the string so this dummy character is not included in the count returned by the `size` member function. It is always the case that, for any `cexpr_basic_string` object `s`, `s[s.size()]` refers to the dummy null character (which immediately follows the last character of the string). The above null-termination is employed so that the pointer returned by the `data` member function can be used in contexts where a null-terminated character array is required.

Listing 1: Interface for class template `cexpr_basic_string`

```
1  namespace ra::cexpr {
2
3      // A basic string class template for use in constexpr contexts.
4      template <class T, std::size_t M>
5      class cexpr_basic_string
```

```
6       {
7       public:
8
9           // An unsigned integral type used to represent sizes.
10          using size_type = std::size_t;
11
12          // The type of each character in the string (i.e., an alias for
13          // the template parameter T).
14          using value_type = T;
15
16          // The type of a mutating pointer to each character in the string.
17          using pointer = T*;
18
19          // The type of a non-mutating pointer to each character in the
20          // string.
21          using const_pointer = const T*;
22
23          // The type of a mutating reference to a character in the string.
24          using reference = T&;
25
26          // The type of a non-mutating reference to a character in the
27          // string.
28          using const_reference = const T&;
29
30          // A mutating iterator type for the elements in the string.
31          using iterator = pointer;
32
33          // A non-mutating iterator type for the elements in the string.
34          using const_iterator = const_pointer;
35
36          // Creates an empty string (i.e., a string containing no
37          // characters).
38          constexpr cexpr_basic_string();
39
40          // Explicitly default some special members.
41          constexpr cexpr_basic_string(const cexpr_basic_string&) =
42            default;
43          constexpr cexpr_basic_string& operator=(
44            const cexpr_basic_string&) = default;
45          ~cexpr_basic_string() = default;
46
47          // Creates a string with the contents given by the
48          // null-terminated character array pointed to by s.
49          // If the string does not have sufficient capacity to hold
50          // the character data provided, an exception of type
51          // std::runtime_error is thrown.
52          constexpr cexpr_basic_string(const value_type* s);
53
54          // Creates a string with the contents specified by the characters
55          // in the iterator range [first, last).
56          // If the string does not have sufficient capacity to hold
57          // the character data provided, an exception of type
```

```
58          // std::runtime_error is thrown.
59          constexpr cexpr_basic_string(const_iterator first,
60            const_iterator last);
61
62          // Returns the maximum number of characters that can be held by a
63          // string of this type.
64          // The value returned is the template parameter M.
65          static constexpr size_type max_size();
66
67          // Returns the maximum number of characters that the string can
68          // hold.  The value returned is always the template parameter M.
69          constexpr size_type capacity() const;
70
71          // Returns the number of characters in the string (excluding the
72          // dummy null character).
73          constexpr size_type size() const;
74
75          // Returns a pointer to the first character in the string.
76          // The pointer that is returned is guaranteed to point to a
77          // null-terminated character array.
78          // The program shall not alter the dummy null character stored
79          // at data() + size().
80          pointer data();
81          const value_type* data() const;
82
83          // Returns an iterator referring to the first character in the
84          // string.
85          constexpr iterator begin();
86          constexpr const_iterator begin() const;
87
88          // Returns an iterator referring to the fictitious
89          // one-past-the-end character in the string.
90          constexpr iterator end();
91          constexpr const_iterator end() const;
92
93          // Returns a reference to the i-th character in the string if i
94          // is less than the string size; and returns a reference to the
95          // dummy null character if i equals the string size.
96          // Precondition: The index i is such that i >= 0 and i <= size().
97          constexpr reference operator[](size_type i);
98          constexpr const_reference operator[](size_type i) const;
99
100         // Appends (i.e., adds to the end) a single character to the
101         // string.  If the size of the string is equal to the capacity,
102         // an exception of type std::runtime_error is thrown.
103         constexpr void push_back(const T& x);
104
105         // Erases the last character in the string.
106         // If the string is empty, an exception of type std::runtime_error
107         // is thrown.
108         constexpr void pop_back();
109
```

```
110         // Appends (i.e., adds to the end) to the string the
111         // null-terminated string pointed to by s.
112         // Precondition: The pointer s must be non-null.
113         // If the string has insufficient capacity to hold the new value
114         // resulting from the append operation, the string is not modified
115         // and an exception of type std::runtime_error is thrown.
116         constexpr cexpr_basic_string& append(const value_type* s);
117
118         // Appends (i.e., adds to the end) to the string another
119         // cexpr_basic_string with the same character type (but
120         // possibly a different maximum size).
121         // If the string has insufficient capacity to hold the new value
122         // resulting from the append operation, the string is not modified
123         // and an exception of type std::runtime_error is thrown.
124         template <size_type OtherM>
125         constexpr cexpr_basic_string& append(
126           const cexpr_basic_string<value_type, OtherM>& other);
127
128         // Erases all of the characters in the string, yielding an empty
129         // string.
130         constexpr void clear();
131
132     };
133
134 }
```

Since the **char** type is commonly used as the character type for strings, the following alias template is provided for convenience:

```
namespace ra::cexpr {
    template <std::size_t M>
    using cexpr_string = cexpr_basic_string<char, M>;
}
```

One additional non-member function is provided as follows:

```
namespace ra::cexpr {
    constexpr std::size_t to_string(std::size_t n, char* buffer,
      std::size_t size, char** end);
}
```

The `to_string` function converts the integer `n` to its equivalent (decimal) null-terminated string representation. The buffer to be used to store the result starts at the location pointed to by `buffer` and has a size of `size` characters. The resulting string produced by the function is null-terminated. The number of characters written to the buffer, excluding the null character, is returned. If `end` is non-null, `*end` is set to point to the null character at the end of the converted string. If the buffer provided does not have sufficient capacity to hold the string resulting from the conversion process, an exception of type `std::runtime_error` is thrown.

The code for the `cexpr_basic_string` class template and other helper code should be placed in the file include/ra/cexpr_basic_string.hpp. Note that the above identifiers (e.g., `cexpr_basic_string`, `cexpr_string`, and `to_string`) are all in the `ra::cexpr` namespace.

Write a program called `test_cexpr_basic_string` to test the code for the `cexpr_basic_string` class template and its associated helper code. The source code for the test program should be placed in the file `app/test_cexpr_basic_string.cpp`.

2. *Mandelbrot variable template (`mandelbrot`).* In this exercise, code is developed that can be used to compute an image representation of the Mandelbrot set at compile time. The computed image is made available through a variable template called `mandelbrot`, which is of type `cexpr_string`. This string variable holds the image encoded in the text-based bitmap PNM format.

   Let $\mathbb{C}$ denote that set of complex numbers. The Mandelbrot set S is the set of all $c \in \mathbb{C}$ such that the sequence $z_0, z_1, z_2, \ldots$ does not tend toward infinity, where

$$z_n = \begin{cases} z_{n-1}^2 + c & n \geq 1 \\ c & n = 0. \end{cases} \tag{2}$$

   As it turns out, the boundary of S is a fractal curve. More information about the Mandelbrot set can be found at:

   https://en.wikipedia.org/wiki/Mandelbrot_set

   The Mandelbrot set S can be represented in the form of a binary image as follows. Define the function χ_S that maps $\mathbb{C}$ to $\{0,1\}$ as

$$\chi_S(z) = \begin{cases} 1 & z \in S \\ 0 & \text{otherwise} \end{cases}$$

   (i.e., χ_S is effectively a boolean predicate that tests if a complex number is a member of S). Let F denote a binary image function defined on points in $\Lambda = \{0,1,\ldots,W-1\} \times \{0,1,\ldots,H-1\}$ (i.e., a rectangular grid of width W and height H). Define a sampling function λ that maps a point $(k,\ell) \in \Lambda$ to a point in the rectangular region $[a_0,b_0] \times [a_1,b_1]$ of $\mathbb{C}$ as given by

$$\lambda[(k,\ell)] = \left(a_0 + k\left(\tfrac{b_0-a_0}{W-1}\right), a_1 + (H-1-\ell)\left(\tfrac{b_1-a_1}{H-1}\right)\right),$$

   where $(a_0,a_1) = (-1.6,-1.1)$ and $(b_0,b_1) = (0.6,1.1)$. The function F is then given by

$$F[(k,\ell)] = \chi_S(\lambda[(k,\ell)]).$$

   The function F is effectively the Mandelbrot set represented in the form of an image.

   The function χ_S is computed from the definition of the Mandelbrot set given by (2). In order to ensure some consistency in the results obtained by different implementations, the function χ_S must be computed as follows. To determine the value of $\chi_S(c)$, the implementation must use (2) to compute z_i for successively larger values of i (starting from 0). This iteration stops when either of the following two conditions is met: 1) $|z_i| > 2$ or 2) $i = 16$. If iteration stops due to the first condition, the implementation should assume that the sequence $z_0, z_1, z_2, \ldots$ grows without bound, in which case $\chi_S(c) = 0$. If iteration stops due to the second condition, the implementation should assume that the sequence $z_0, z_1, z_2, \ldots$ remains bounded for all i, in which case $\chi_S(c) = 1$.

   The `mandelbrot` variable template has two template parameters:

   (a) `W`. The width of the image (in samples).

(b) `H`. The height of the image (in samples).

The interface for this variable template is as described in Listing 2. In order to minimize the amount of work required for this exercise, the `cexpr_string` class template and `to_string` function developed in Exercise 1 should be used. The `mandelbrot` variable template must be of type `cexpr_string<M>` for some value of `M`. The string must be encoded in the text-based PNM format for bitmap images. This particular format is described shortly. The particular value of `M` to be used is at the discretion of the implementation. Clearly, however, `M` must be chosen sufficiently large that the `cexpr_string` object can hold the complete PNM-encoded character sequence for an image of width `W` and height `H`. The source for the `mandelbrot` variable template and all of its supporting code should be placed in the file `include/ra/mandelbrot.hpp`.

Listing 2: Interface for the `mandelbrot` variable template

```
namespace ra::fractal {

    // A variable template for a string that represents an image depicting
    // the Mandelbrot set.  The image has width W and height H.
    // This object must be of type cexpr_string<M> for some appropriate M.
    // The string is a binary image encoded in the text-based bitmap PNM
    // format.
    template <std::size_t W, std::size_t H>
    constexpr auto mandelbrot = implementation-defined;

}
```

The text-based PNM format for bitmap images is very simple. This format represents a bitmap image of width W and height H using a character sequence that consists of the following (in order):

(a) A signature, which consists of the character "P" followed by the character "1".
(b) A space character.
(c) The width W of the image, which consists of a sequence of one or more decimal digits.
(d) A space character.
(e) The height H of the image, which consists of a sequence of one or more decimal digits.
(f) A newline character.
(g) The (binary) image samples. For each of the H rows in the image starting with the top row, a string of W characters that are either "0" or "1" characters, followed by a newline character.

To further illustrate this image format, we now consider a simple example. Consider an image of width 8 and height 4 whose contents resemble a crude letter "V". This image would be encoded using the following string:

```
P1 8 4
10000001
01000010
00100100
00011000
```

Since the amount of complex arithmetic in this exercise is minimal, it is probably easiest to perform the necessary computations without the formality of using a complex-number class. Unfortunately (at least, as of C++17), the class template `std::complex` does not provide constexpr versions of all of the functions that are likely to be needed for the complex arithmetic in this exercise. Consequently, the

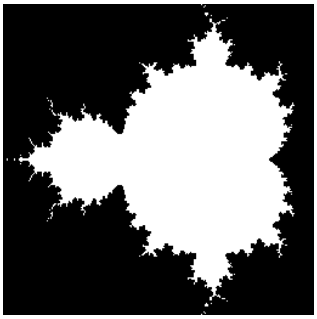

Figure 1: Mandelbrot set image.

`std::complex` class template is somewhat less helpful. In any case, regardless of whether a class such as `std::complex` is used, all real-arithmetic should be performed using double-precision (i.e., **`double`**) arithmetic.

A program called `test_mandelbrot` should be used to test the `mandelbrot` variable template. The source for this program should be placed in the file `app/test_mandelbrot.cpp`. An example of the source for a very trivial test program is shown in Listing 3. This example is intended only for illustrative purposes, however. It is probably advisable to perform more thorough testing than what is achieved by this trivial program. The image generated by this trivial test program is shown in Figure 1.

Listing 3: Source listing for a very trivial example of a `test_mandelbrot` program.

```
#include <iostream>
#include "ra/mandelbrot.hpp"

int main()
{
    // Force the image (in PNM format) to be computed at compile time.
    constexpr auto s = ra::fractal::mandelbrot<256, 256>;

    // Output the image (in PNM format).
    std::cout << s.begin() << '\n';
}
```

Admittedly, this exercise is somewhat of an abuse of compile-time computation in the sense that one would probably not normally perform this much computation at compile time. Nevertheless, this exercise serves to clearly demonstrate the power of compile-time computation. In the CMakeLists file, it may be necessary to add a compiler flag to increase the maximum allowable amount of compile-time computation for the compilation of the source file `app/test_mandelbrot.cpp`. In the case of GCC, the specification of the `-fconstexpr-loop-limit` command-line option might be necessary. This option takes a parameter that is an integer value specifying the loop limit in constexpr computation. A value of 1000000 should be sufficient to compute the value of `mandelbrot<512, 512>`. Note that this compiler option should only be used for the compilation of the file `app/test_mandelbrot.cpp` (not for any other files). This might be accomplished with a command like the following in the `CMakeLists.txt` file:

```
set_source_files_properties(app/test_mandelbrot.cpp PROPERTIES
  COMPILE_FLAGS ${CMAKE_CXX_FLAGS} -fconstexpr-loop-limit=1000000)
```

As part of the testing of the code for the `mandelbrot` variable template, confirm that the generated image does, in fact, resemble the Mandelbrot set. This can be done by printing the (string) value of the `mandelbrot` variable template to a file and then displaying this file with an image viewer that supports the PNM format. For example, the following command can be used to display a file called `mandelbrot.pnm` (in PNM format) with the ImageMagick software:

```
display mandelbrot.pnm
```

3. *Constexpr math constants and functions (i.e.,* `pi`*,* `sin`*,* `cos`*,* `tan`*,* `sqrt`*, and others).* In this exercise, numerous variable and function templates are developed that provide support for math constants (such as π) and math functions (such as sin and square root) that can be used in constexpr contexts.

   Each of these templates has a template parameter `T` that corresponds to the floating-point type to be used to represent real numbers. Only floating-point types (i.e., **`float`**, **`double`**, and **`long double`**) can be used for `T`. The interface provided by the above variable and function templates is given in Listing 4. (Note that all of these templates are in the namespace `ra::cexpr_math`.) The source for these variable and function templates is to be placed in the file `include/ra/cexpr_math.hpp`.

Listing 4: Interface for `cexpr_math` functions

```
namespace ra::cexpr_math {

    // The math constant pi.
    // The type T is a floating-point type.
    template <class T>
    constexpr T pi = boost::math::constants::pi<T>();

    // Returns the absolute value of x.
    // The type T is a floating-point type.
    template <class T>
    constexpr T abs(T x);

    // Returns the square of x.
    // The type T is a floating-point type.
    template <class T>
    constexpr T sqr(T x);

    // Returns the cube of x.
    // The type T is a floating-point type.
    template <class T>
    constexpr T cube(T x);

    // Returns the remainder after division when x is divided by y.
    // In particular, x - n y is returned where n is the result obtained by
    // dividing x by y and then rounding (to an integer value) toward zero.
    // If y is zero, an exception of type std::overflow_error is thrown.
    // The type T is a floating-point type.
    template <class T>
    constexpr T mod(T x, T y);

    // Returns the sine of x.
    // Note that a particular algorithm must be used to implement this
```

```
33      // function.
34      // The type T is a floating-point type.
35      template <class T>
36      constexpr T sin(T x);
37
38      // Returns the cosine of x.
39      // Note that a particular algorithm must be used to implement this
40      // function.
41      // The type T is a floating-point type.
42      template <class T>
43      constexpr T cos(T x);
44
45      // Returns the tangent of x.
46      // Note that a particular algorithm must be used to implement this
47      // function.
48      // If the tangent of x is infinite, an exception of type
49      // std::overflow_error is thrown.
50      // The type T is a floating-point type.
51      template <class T>
52      constexpr T tan(T x);
53
54      // Returns the square root of x.
55      // If x is negative, an exception of type std::domain_error is thrown.
56      // Note that a particular algorithm must be used to implement this
57      // function.
58      // The type T is a floating-point type.
59      template <class T>
60      constexpr T sqrt(T x);
61
62  }
```

As mentioned in the interface description, the implementation technique for some of the function templates must follow a particular approach. In what follows, these restrictions on the implementation strategies are discussed.

The `sin` function must be implemented by using the triple-angle formula and small-angle approximation for sin. The triple-angle formula and small-angle approximation are respectively given by

$$\sin 3x = 3\sin x - 4\sin^3 x \quad \text{and} \tag{3}$$

$$\sin x \approx x \ \text{ for small nonnegative } x. \tag{4}$$

By rearranging (3), we can obtain the following recursive equation for $\sin x$:

$$\sin x = 3\sin(x/3) - 4\sin^3(x/3). \tag{5}$$

To implement the `sin` function, (5) should be used recursively for the computation of $\sin x$ with the base case for the recursion corresponding to $x \leq 10^{-6}$. In this base case, $\sin x$ is simply assumed to be equal to x, with this assumption being valid for nonnegative x due to (4). The maximum recursion depth for this algorithm could potentially become large if $\sin x$ is computed for large magnitude x. To eliminate this problem, the algorithm must use the fact that sin is 2π-periodic in order to reduce the problem of computing $\sin x$ for arbitrary x to that of computing $\sin x$ for $x \in [0, 2\pi)$. (That is, do not apply (5) to the

problem of computing $\sin x$ until first ensuring that $x \in [0, 2\pi)$.) To reduce the problem in this way, the `mod` function will likely be helpful.

The `cos` function must be implemented by using the `sin` function and the fact that cos is identical to sin except for a translation (i.e., shift).

The `tan` function must be implemented by using the `sin` and `cos` functions and the fact that $\tan x = \frac{\sin x}{\cos x}$ (for $\cos x \neq 0$). The implementation must not divide by zero, under any circumstances. If a circumstance arises that would lead to a division by zero, an exception should be thrown (as explained in the interface definition).

The `sqrt` function must be implemented by using the Newton-Raphson (root-finding) method. In the Newton-Raphson method, to solve for a root of the equation $f(x) = 0$, we select an initial estimate x_0 of the root. Then, we apply the following iterative process to improve the accuracy of our initial estimate:

$$x_{n+1} = x_n - f(x_n)/f'(x_n), \tag{6}$$

where f' denotes the first derivative of f. To find the square root of c, we can simply solve for the (real nonnegative) root of the equation $f(x) = x^2 - c$. The initial estimate x_0 of the root should be chosen as 0. The iteration in (6) should continue until $|x_{n+1} - x_n| \leq \varepsilon$ (i.e., the root estimate does not change by more than ε from one iteration to the next), where the value to be used for ε is `std::numeric_limits<T>::epsilon()`. If the number whose square root is to be computed is negative, an exception should be thrown (as explained in the interface definition).

It is absolutely critical that the exact algorithms specified above be employed for `sin`, `cos`, `tan`, and `sqrt`. Failure to do so will likely result in implementations of these functions with accuracy properties that are very substantially different from what is required, which could lead to your code failing many test cases (due to results that are not sufficiently accurate).

A program called `test_cexpr_math` is to be developed to test the variable and function templates from above. The source for this test program is to be placed in the file `app/test_cexpr_math.cpp`.

4. *Biquad filter design functions.* In this exercise, code is developed that can be used to design several types of discrete-time biquad filters. This code consists of a class template `biquad_filter_coefs` that is used to represent the coefficients of a biquad filter as well as several function templates that can be used to design various types of filters. These filter-design function templates can be employed in constexpr contexts.

   In audio and music processing applications, biquad filters are often employed. A (real-coefficient) discrete-time biquad filter has a transfer function H of the form

$$H(z) = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{b_0 + b_1 z^{-1} + b_2 z^{-2}}, \tag{7}$$

   where a_0, a_1, a_2, b_0, b_1, and b_2 are real coefficients. Such a transfer function can always be normalized such that the constant term in the denominator is 1 (provided $b_0 \neq 0$, which will always be the case here). That is, such a transfer function can always be expressed in the form

$$H(z) = \frac{a'_0 + a'_1 z^{-1} + a'_2 z^{-2}}{1 + b'_1 z^{-1} + b'_2 z^{-2}}. \tag{8}$$

   This can be accomplished by simply dividing each of the numerator and denominator of $H(z)$ by b_0. For various reasons, it is often more convenient to express biquad transfer functions in this normalized form.

In discrete-time signal processing, the sampling frequency is an important quantity. Often, rather than specifying frequencies directly, it is more convenient to specify them relative to the sampling frequency, leading to the notion of normalized frequency. The normalized frequency f that corresponds to the actual frequency f' and sampling frequency f_s is given by $f = 2f'/f_s$. That is, the normalized frequency 1 corresponds to the Nyquist frequency. Note that the normalized frequency is always a value in $[0,1]$. In what follows, the term "normalized frequency" refers to this definition just introduced.

Although many types of biquad filters are possible, we only consider the design of the following basic types: 1) lowpass, 2) highpass, 3) bandpass, 4) low-frequency shelving boost filter. and 5) low-frequency shelving cut filter. A lowpass biquad filter with normalized cutoff frequency f and Q factor Q has the filter coefficients given by

$$\begin{gathered}\Omega = \tan\left(\tfrac{\pi}{2}f\right), \\ a_0 = \Omega^2, \quad a_1 = 2\Omega^2, \quad a_2 = \Omega^2, \\ b_0 = \Omega^2 + \Omega/Q + 1, \quad b_1 = 2(\Omega^2 - 1), \quad \text{and} \quad b_2 = \Omega^2 - \Omega/Q + 1.\end{gathered}$$

A highpass filter with normalized cutoff frequency f and Q factor Q has the filter coefficients given by

$$\begin{gathered}\Omega = \tan\left(\tfrac{\pi}{2}f\right), \\ a_0 = 1, \quad a_1 = -2, \quad a_2 = 1, \\ b_0 = \Omega^2 + \Omega/Q + 1, \quad b_1 = 2(\Omega^2 - 1), \quad \text{and} \quad b_2 = \Omega^2 - \Omega/Q + 1.\end{gathered}$$

A bandpass filter with normalized center frequency f and Q factor Q has the filter coefficients given by

$$\begin{gathered}\Omega = \tan\left(\tfrac{\pi}{2}f\right), \\ a_0 = \Omega/Q, \quad a_1 = 0, \quad a_2 = -\Omega/Q, \\ b_0 = \Omega^2 + \Omega/Q + 1, \quad b_1 = 2(\Omega^2 - 1), \quad \text{and} \quad b_2 = \Omega^2 - \Omega/Q + 1.\end{gathered}$$

A low-frequency shelving boost filter with normalized cutoff frequency f and gain-control parameter A has the filter coefficients given by

$$\begin{gathered}\Omega = \tan\left(\tfrac{\pi}{2}f\right), \\ a_0 = A\Omega^2 + \sqrt{2A}\Omega + 1, \quad a_1 = 2(A\Omega^2 - 1), \quad a_2 = A\Omega^2 - \sqrt{2A}\Omega + 1, \\ b_0 = \Omega^2 + \sqrt{2}\Omega + 1, \quad b_1 = 2(\Omega^2 - 1), \quad b_2 = \Omega^2 - \sqrt{2}\Omega + 1.\end{gathered}$$

A low-frequency shelving cut filter with normalized cutoff frequency f and gain-control parameter A has the filter coefficients given by

$$\begin{gathered}\Omega = \tan\left(\tfrac{\pi}{2}f\right), \\ a_0 = \Omega^2 + \sqrt{2}\Omega + 1, \quad a_1 = 2(\Omega^2 - 1), \quad a_2 = \Omega^2 - \sqrt{2}\Omega + 1, \\ b_0 = A\Omega^2 + \sqrt{2A}\Omega + 1, \quad b_1 = 2(A\Omega^2 - 1), \quad b_2 = A\Omega^2 - \sqrt{2A}\Omega + 1.\end{gathered}$$

To provide a convenient container for holding the coefficients of a biquad filter, a class template called `biquad_filter_coefs` is used. This class template has a single template parameter `Real`, which specifies the real-number type used to represent the filter coefficients. This template can be instantiated with `Real` being any floating-point type (i.e., **`float`**, **`double`**, or **`long double`**). The `biquad_filter_coefs` class template has the very simple interface provided in Listing 5. This class template only has public members and all such members are identified in the interface description.

Listing 5: Interface for `biquad_filter_coefs` class template

```
namespace ra::biquad {

    // Biquad filter coefficients class.
    template <class Real>
    struct biquad_filter_coefs
    {
        // The real number type used to represent the filter coefficients.
        using real = Real;

        // Creates a set of filter coefficients where the coefficients
        // a0, a1, a2, b0, b1, and b2 are initialized to a0_, a1_, a2_,
        // b0_, b1_, and b2_, respectively.
        constexpr biquad_filter_coefs(real a0_, real a1_, real a2_, real b0_,
          real b1_, real b2_);

        // Creates a set of filter coefficients by copying from another set.
        // Since Real and OtherReal need not be the same, this constructor
        // can be used to convert between filter coefficients of different
        // types.
        template <class OtherReal>
        constexpr biquad_filter_coefs(
          const biquad_filter_coefs<OtherReal>& coefs);

        // The filter coefficients a0, a1, a2, b0, b1, and b2.
        real a0;
        real a1;
        real a2;
        real b0;
        real b1;
        real b2;
    };

}
```

Several function templates are provided for designing various types of biquad filters. These function templates have the interface shown in Listing 6. Each of these functions has a template parameter `Real`, which controls the real-number type used for computing filter coefficients. Depending on how accurate the computed filter coefficients need to be, the template could be instantiated with `Real` being any one of the floating-point types (i.e., **`float`**, **`double`**, or **`long double`**).

Listing 6: Interface for biquad filter design functions

```
namespace ra::biquad {

    // Returns the coefficients for a biquad lowpass filter with normalized
    // cutoff frequency f and Q factor q, where f in [0,1] and q > 0.
    // The filter coefficients are always normalized such that the
    // coefficient b0 is 1.
    // The type Real is a floating-point type.
    // All real arithmetic should be performed with the Real type.
    template <class Real>
```

```
10      constexpr biquad_filter_coefs<Real> lowpass(Real f, Real q);
11
12      // Returns the coefficients for a biquad highpass filter with
13      // normalized cutoff frequency f and Q factor q, where f in [0,1]
14      // and q > 0.
15      // The filter coefficients are always normalized such that the
16      // coefficient b0 is 1.
17      // The type Real is a floating-point type.
18      // All real arithmetic should be performed with the Real type.
19      template <class Real>
20      constexpr biquad_filter_coefs<Real> highpass(Real f, Real q);
21
22      // Returns the coefficients for a biquad bandpass filter with
23      // normalized cutoff frequency f and Q factor q, where f in [0,1]
24      // and q > 0.
25      // The filter coefficients are always normalized such that the
26      // coefficient b0 is 1.
27      // The type Real is a floating-point type.
28      // All real arithmetic should be performed with the Real type.
29      template <class Real>
30      constexpr biquad_filter_coefs<Real> bandpass(Real f, Real q);
31
32      // Returns the coefficients for a biquad low-frequency shelving
33      // boost filter with normalized cutoff frequency f and gain-control
34      // parameter a, where f in [0,1] and a >= 0.
35      // For a gain in dB of G (where G > 0), choose a = 10 ^ (G / 20).
36      // The filter coefficients are always normalized such that the
37      // coefficient b0 is 1.
38      // The type Real is a floating-point type.
39      // All real arithmetic should be performed with the Real type.
40      template <class Real>
41      constexpr biquad_filter_coefs<Real> lowshelf_boost(Real f, Real a);
42
43      // Returns the coefficients for a biquad low-frequency shelving
44      // cut filter with normalized cutoff frequency f and gain-control
45      // parameter a, where f in [0,1] and a >= 0.
46      // For a gain in dB of G (where G < 0), choose a = 10 ^ (-G / 20).
47      // The filter coefficients are always normalized such that the
48      // coefficient b0 is 1.
49      // The type Real is a floating-point type.
50      // All real arithmetic should be performed with the Real type.
51      template <class Real>
52      constexpr biquad_filter_coefs<Real> lowshelf_cut(Real f, Real a);
53
54  }
```

The code for the `biquad_filter_coefs` class template and the filter-design function templates should be placed in the file `include/ra/biquad_filter.hpp`.

A program called `test_biquad_filter` should be used to test the filter-design code. The source for this program should be placed in the file `app/test_biquad_filter.cpp`. An online biquad filter coefficient calculator, which may prove useful for testing purposes, can be found at:

http://www.earlevel.com/main/2010/12/20/biquad-calculator

(In the case of low-frequency shelving filters, this online calculator selects the boost and cut cases if the gain G in dB is positive and negative, respectively.)