VE 280 Lab 5

Out: 02:00 am, June 17, 2021; **Due**: 11:59 pm, June 24, 2021.

Ex.1 *Carole & Tuesday*

Related Topics: *ADT*, *list*.

A famous band *Carole & Tuesday* now decides to release an album. They want to maintain songs as a "list". Roddy, a friend of Carole and Tuesday, provides many starter files for this "list" and lets you complete all functions in files.

A "list" is a sequence of zero or more numbers in no particular order. A list is well-formed if:

a) It is the empty list, or

b) It is an integer followed by a well-formed list.

A list is an example of a linear-recursive structure: it is "recursive" because the definition refers to itself. It is "linear" because there is only one such reference.

Here are some examples of well-formed lists:

```
( 1 2 3 4 ) // a list of four elements
( 1 2 4 ) // a list of three elements
( ) // a list of zero element--the empty list
```

The file `recursive.h` defines the type `list_t` and the following operations on lists:

```
bool list_isEmpty(list_t list);
   // EFFECTS: returns true if list is empty, false otherwise

list_t list_make();
   // EFFECTS: returns an empty list.

list_t list_make(int elt, list_t list);
   // EFFECTS: given the list make a new list consisting of
   //          the new element followed by the elements of the
   //          original list.

int list_first(list_t list);
   // REQUIRES: list is not empty
   // EFFECTS: returns the first element of list

list_t list_rest(list_t list);
   // REQUIRES: list is not empty
   // EFFECTS: returns the list containing all but the first element of list

void list_print(list_t list);
    // MODIFIES: cout
    // EFFECTS: prints list to cout.
```

They are implemented in `recursive.cpp` for you, what you need to do is to implement the functions declared in `ex1.h`.

```
int dot(list_t v1, list_t v2);
// REQUIRES: Both "v1" and "v2" are non-empty
//
// EFFECTS: Treats both lists as vectors. Returns the dot
//          product of the two vectors. If one list is longer
//          than the other, ignore the longer part of the vector.


list_t filter(list_t list, bool (*fn)(int));
// EFFECTS: Returns a list containing precisely the elements of "list"
//          for which the predicate fn() evaluates to true, in the
//          order in which they appeared in list.
//
//          For example, if predicate bool odd(int a) returns true
//          if a is odd, then the function filter(list, odd) has
//          the same behavior as the function filter_odd(list).

list_t filter_odd(list_t list);
// EFFECTS: Returns a new list containing only the elements of the
//          original "list" which are odd in value,
//          in the order in which they appeared in list.
//
//          For example, if you apply filter_odd to the list
//          ( 3 4 1 5 6 ), you would get the list ( 3 1 5 ).
```

Since `filter_odd` is a special case of `filter`, you can use `filter_odd` as a function to test `filter` if you implement it with `filter`.

Hint

You can think in the way that `recursive.h` provides an ADT for you and you need to implement the new functions declared in `ex1.h` using the methods provided.

Problem

1. Implement the functions in `ex1.cpp`.

Requirements

1. If you define **any** helper functions yourself, be sure to declare them "**static**", so that they are **not visible** outside this file.

Testing
Since you are only required to implement new methods, there is no IO requirements. However `ex1Test.cpp` is provided for you to test your correctness but you still need to design you own test cases to get full score.

Ex2. Quadratic Functions in Standard Form

Roddy is in charge of tuning for *Carole & Tuesday*. However, this reminds him of tough time dealing with **quadratic functions** in high school. As a student taking VE280, you can use your knowledge about abstract data types (ADT) to help Roddy with little knowledge in math play with quadratic functions.

Related Topics: *ADT*.

Problem: Roddy wants to represent a quadratic function in a standard form, which is $f(x) = ax^2 + bx + c$ ($a \neq 0$). He decides that the following operations should be allowed on quadratic functions:

1. Evaluate $f(x)$ at a given int x value.
2. Get the root(s) of $f(x)$, which is the value of x such that $f(x) = 0$
3. Check if two quadratic functions (f and g) intersects, which means whether there exists some real x such that $f(x) = g(x)$.

Therefore, he designed this interface to represent a quadratic function

```
class QuadraticFunction {
    // OVERVIEW: the standard form of a quadratic function f(x) = ax^2 + bx + c
    float a;
    float b;
    float c;

public:
    QuadraticFunction(float a_in, float b_in, float c_in);
    // REQUIRES: a_in is nonezero
    // EFFECTS: creates a quadratic function in standard form

    float getA() const;
    // EFFECTS: returns the value of a

    float getB() const;
    // EFFECTS: returns the value of b

    float getC() const;
    // EFFECTS: returns the value of c

    float evaluate(float x);
    // EFFECTS: returns the value of f(x)

    Root getRoot();
    // EFFECTS: returns the roots of the quadratic function

    bool intersect(QuadraticFunction q);
    // EFFECTS: returns whether g and this intersect
};
```

Here, the constructor takes 3 inputs `a_in`, `b_in` and `c_in` and uses them to represent the quadratic function $f(x) = ax^2 + bx + c$. Also, the output function for this exercise is provided.

Requirements:

1. Look through file `rootType.h`, to make the output simple, we make the following restrictions:
    - if $f(x)$ has two different real roots, then the smaller x_1 should be in `roots[0]` and the bigger x_2 should be in `roots[1]`.
    - If $f(x)$ has one real root, then $x_1 = x_2$ should be in both `roots[0]` and `roots[1]`.
    - If $f(x)$ has two complex roots, then $x_1 = m - ni$ should be in `root[0]` and $x_2 = m + ni$ should be in `roots[1]`, where $n > 0$.
2. Look through `standardForm.h` and implement the methods for QuadraticFunction class in `standardForm.cpp`.

3. `ex2.cpp` is used to test your ADT, you can just read it and run it.

Input Format: Since you only need to implement the methods of this ADT, we just provide a sample input. And there will not be cases where $a = 0$.

```
1 -3 2
1
2 -4 2
```

Output Format: Since you only need to implement the methods of this ADT, we just provide a sample output. *NOTE* that although in some cases it may be weird to have `x1 = 1.0 + -1.0i`, just ignore it.

```
f(x)=1.0x^2+-3.0x+2.0
f(1.0)=0.0
f(x) has 2 real roots.
x1 = 1.0 + 0.0i
x2 = 2.0 + 0.0i
1
```

Hint:

1. $\Delta = b^2 - 4ac$, if $\Delta \geq 0$, $x = \frac{-b \pm \sqrt{\Delta}}{2a}$. Else, $x = \frac{-b \pm i\sqrt{-\Delta}}{2a}$
2. a for $g(x)$ can be the same as a for $f(x)$.

Ex3. Quadratic Functions in Factored Form

Related Topics: *ADT*.

Problem: Roddy realizes that a quadratic function can also be represented in a factored form, which is $f(x) = a(x - r_1)(x - r_2)$ $(a \neq 0)$.

This time, the interface looks the same, but the data members are different:

```
class QuadraticFunction {
        // OVERVIEW: the factored form of a quadratic function f(x) = ax^2 + bx
+ c
    float a;
    complexNum r1;
    complexNum r2;

public:
    QuadraticFunction(float a_in, float b_in, float c_in);
    // REQUIRES: a_in is not 0
    // EFFECTS: creates a quadratic function in factored form

    float getA()const;
    // EFFECTS: returns the value of a

    float getB()const;
    // EFFECTS: returns the value of b

    float getC()const;
    // EFFECTS: returns the value of c
```

```
    float evaluate(float x);
    // EFFECTS: returns the value of f(x)

    Root getRoot();
    // EFFECTS: returns the roots of the quadratic function

    bool intersect(QuadraticFunction g);
    // EFFECTS: returns whether g and this intersect
};
```

Here, the constructor also takes 3 inputs `a_in`, `b_in` and `c_in`, but you need to do some transformation so that they can fit into the new data members. Again, the output method is provided. So you needn't implement it.

Requirements:

1. Look through `factoredForm.h` and implement the TODOs in `factoredForm.cpp`
2. Run `ex3.cpp` to test the ADT. Note that `ex3.cpp` includes `factoredForm.h`, but it uses the same code as in `ex2.cpp` to test the ADT.

Input Format: Same as ex2

Output Format: Same as ex2

Testing & Submitting

`ex1Test.cpp`, `ex2.cpp` and `ex3.cpp` are provided for your test. Please only compress `ex1.cpp`, `standardFrom.cpp` and `factoredForm.cpp` and submit each of them to the corresponding exercises on the online judge.

Created by Zhuoer Zhu
Updated by Yanjun Chen
Last update: June 13, 2021
@UM-SJTU Joint Institute