CS 311 Fall 2017
Assignment 3

Assignment 3 is due at **5 pm Tuesday, Obtober 3**. It is worth 25 points.

Procedures

This assignment is to be done individually.

Turn in answers to the exercises below on the UA Blackboard Learn site, under Assignment 3 for this class.

- Your answers should consist of two files: `da3.h` and `da3.cpp`, from Exercises A, B, C, and D. These two files should be **attached** to your submission.
- I may not look at your homework submission immediately. If you have questions, e-mail me.

Exercises (25 pts total)

General

This assignment is to be done individually.

In each of the following exercises, you are to write a function or function template. All functions & function templates are to be in the files `da3.h` and `da3.cpp`. The templates must be implemented entirely in the header file. The non-templates must be prototyped in the header and implemented in the source, as usual. Be sure to follow the coding standards. The following standards from part 3 now apply.

Standard 3A
: Requirements on template parameter types must be documented.

Standard 3B
: If a function is not a template and not a member of a class template, then the exceptions it throws must be documented.

You do not need to follow standards 3C or 3D.

In the files `da3.h` and `da3.cpp`, you may include any other functions or classes that you wish. These will not be tested; however, they must follow the coding standards. Also, use of the C++ Standard Library is legal in this assignment.

Skeleton Files

I have provided incomplete "skeleton" files `da3.h` and `da3.cpp`; these are in the Git repository. You may use these as the basis for your own work, if you wish. This is not required. However, if you do not wish to use these files, then you still need to copy the definition of `LLNode`, *exactly* as I wrote it, from the provided `da3.h`.

Test Program

A single test program for all of the exercises will be available soon: If you compile and run the program (unmodified!) with your code, then it will test whether your code works properly.

Note that your code will not compile with the test program unless all required functions exist. Therefore, you must write dummy versions of all functions—at least—or your work will not be graded. (The skeleton files make this easy; see above.)

The test program requires `catch.hpp`, the single-header version of the "Catch" unit-testing framework.

Do not turn in the test program or the Catch framework.

Exercise A — Linked List Look-Up

Purpose

In this exercise you will write code to deal with a Linked List. The code will signal an error condition—if one occurs—by throwing an exception.

Instructions

Write a function template `lookUp`, prototyped as follows.

[C++]

```
template <typename ValueType>
ValueType lookUp(const LLNode<ValueType> * head,
                 size_t index);
```

- `lookUp` is given a pointer to a null-terminated Linked List (as discussed in class, and demonstrated in the file `list_size.cpp`) and an integer index. It functions similarly to an array bracket operator, returning the item corresponding to the index, where the first item is numbered 0, the second 1, and so on, up to $size - 1$, where $size$ is the number of items in the list. The data item is to be returned by value.
- An empty (size zero) list will be given to `lookUp` by passing a null pointer as the parameter `head`.
- If `index` is out of range—negative or at least *size*—then `lookUp` should throw an exception of type `std::out_of_range`.
- `lookUp` should throw only if `index` is out of range.
- If an exception is thrown, then the exception's `what` member should return a string giving a brief description of the error. The message in the string should be a brief but informative one, aimed at a technical user with at least some knowledge of the source code.

Your header file must contain the definition of the struct template `LLNode`. This struct definition can be found in the skeleton version of `da3.h`, in the Git repository. The definition of `LLNode` in your `da3.h` must be *exactly* as I wrote it.

Exercise B — Call Between

Purpose

In this exercise you will write code to call functions that may throw an exception, catching the exception appropriately.

Background: Function Objects and `std::function`

A C++ **function object** is an object that behaves as a function. That is, it can be called, and it may possibly return a value. Consider the following code.

[C++]

```
int i = foo(7, 9);
```

The above is legal if `foo` is a function that takes two `int` parameters and returns `int`. It is also legal if `foo` is a function object with those same parameter and return types.

`std::function` is a class template prototyped in header `<functional>`. It is a wrapper for pretty much anything function-like: a function, a pointer to a function, a function object, another `std::function` object, or a pointer to a member function, as well as a couple of things we have not discussed yet: lambda expressions and bind expressions.

The type of a `std::function` is written as something like the following.

[C++]

```
std::function<return_type(param1_type, param2_type)>
```

So a `std::function` appropriate for wrapping `foo` (above), would have type `std::function<int(int,int)>`. And a `std::function<void()>` is a wrapper for a function that takes no parameters and returns nothing.

When you use a `std::function` object, you may call it exactly as it it were a function.

Instructions

Write a function (NOT a template) `callBetween`, prototyped as follows.

[C++]

```
void callBetween(const std::function<void()> & start,
                 const std::function<void()> & middle,
                 const std::function<void()> & finish);
```

Function `callBetween` is given three function objects: `start`, `middle`, and `finish`. Each of these behaves as a function that takes no parameters, and returns nothing. So, for example, you can do the following.

[C++]

```
middle();
```

The idea is to call `start`, then `middle`, then `finish`. So `middle` is *called between* the other two; thus the name `callBetween`. You might think of `start` as a function that acquires some resource, `middle` uses the resource, and `finish` releases the resource.

The tricky part of this exercise is that exceptions may be thrown by `start` and `middle`. You may assume that `finish` will not throw, as is proper for a clean-up function.

The following rules must be followed.

- `start` is called first.
- If `start` throws, then function `callBetween` terminates, with neither of the other two functions called.
- If `start` does not throw, then `middle` and `finish` should be called, in that order, no matter what.
- Any exception thrown by `start` or `middle` should be passed on to the client code.
- None of the three (`start`, `middle`, `finish`) should be called more than once by any single call to `callBetween`.

Exercise C — Count Unique Values

Purpose

In this exercise you will write a function that takes iterators as parameters and processes a range of data.

Instructions

Write a function template `uniqueCount`, prototyped as

[C++]

```
template <typename RAIter>
```

```
size_t uniqueCount(RAIter first,
                   RAIter last);
```

Parameters `first` and `last` are two random-access iterators specifying a range in the standard manner. Function `uniqueCount` should return the number of unique values in the given range.

Example usage:

[C++]

```
vector<string> v {
    "abc",
    "x",
    "x",
    "llama",
    "x",
    "abc",
    "llama"
};
cout << uniqueCount(v.begin(), v.end()) << endl;
```

The above should print

```
3
```

since there are three unique values in the given range: `"abc"`, `"x"`, and `"llama"`.

If you wish, you may require that the type of the values in the range has any of the six comparison operators (==, !=, <, <=, >, >=), along with default constructor and the Big Five.

You may assume that the range is writable, and you may modify the values in the range, if you wish.

Exercise D — Recursive GCD

Purpose

In this exercise, you will implement a simple recursive algorithm.

Background

If a and b are two nonnegative integers, not both zero, then the **greatest common divisor** (**GCD**) of a and b is the greatest integer that evenly divides both a and b.

For example, the GCD of 910 and 42 is 14, since both 910 and 42 are divisible by 14, and this is not true of any integer greater than 14. We write $\gcd(910, 42) = 14$.

The GCD can be computed quickly based on the following rules.

1. If $a = 0$, then $\gcd(a, b) = b$.
2. Otherwise, if $a > b$, then $\gcd(a, b) = \gcd(b, a)$.
3. Otherwise, $\gcd(a, b) = \gcd(b \bmod a, a)$.

Above, $b \bmod a$ is the remainder when b is divided by a.

Here is how we would apply the above rules to compute $\gcd(910, 42)$.

$$
\begin{aligned}
\gcd(910, 42) &= \gcd(42, 910) && \text{by Rule 2} \\
&= \gcd(28, 42) && \text{by Rule 3 [910 mod 42 = 28]} \\
&= \gcd(14, 28) && \text{by Rule 3 [42 mod 28 = 14]} \\
&= \gcd(0, 14) && \text{by Rule 3 [28 mod 14 = 0]} \\
&= 14 && \text{by Rule 1}
\end{aligned}
$$

The above method is a version of the **Euclidean Algorithm**, so called because it appeared in a text by the ancient Greek mathematician Euclid, written around 300 BC. It is thus among the oldest algorithms known.

Instructions

Write a function (NOT a template) `gcd`, prototyped as

```
int gcd(int a,
        int b);
```

- Function `gcd` is given two nonnegative integers, not both zero. It returns their GCD.

So, for example `gcd(910, 42)` should return `14`.

- Function `gcd` must compute the GCD using the algorithm described above.
- Function `gcd` must either be a **recursive** function, or it must do the bulk of its work by calling a recursive function.
- Neither `gcd` nor any function it calls may contain a loop.
- Whatever functions are called by `gcd` must not be available to the client code; that is, they must not be declared in the header `da3.h`.

Note that, in C++, $b \bmod a$ is computed using the `%` operator: `b % a`.