Background Theory

Definition: Two numbers are *relatively prime* if they have no common divisor aside from 1. For example, 10 and 21 are relatively prime but 12 and 9 are not.

If n and m are relatively prime, we write $m \perp n$.

The *Stern-Brocot tree* is a construction of all the non-negative ratios m/n where $m \perp n$. Start with the two fractions $\frac{0}{1}, \frac{1}{0}$ and then repeat the following operation as many times as desired:

$$\text{Insert } \frac{m+m'}{n+n'} \text{ between two adjacent fractions } \frac{m}{n} \text{ and } \frac{m'}{n'}$$

The new ratio, $\frac{m+m'}{n+n'}$ is called the *mediant* og m/n and m'/n'. The new fraction $(m+m')/(n+n')$ is called the *mediant* of m/n and $m'.n'$. For example, the first step gives us one new entry between $\frac{0}{1}$ and $\frac{1}{0}$.

$\frac{0}{1}, \frac{1}{1}, \frac{1}{0};$

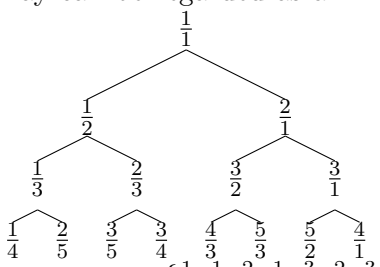
and the next gives two more:

$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0};$

The next gives four more:

$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0};$

and then we'll get 8 16, and so on. The entire array can be regarded as an infinite binary tree whose top levels look like this:



If you collapse the tree down you get an increasing sequence: $\{\frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}, \frac{4}{3}, \frac{3}{2}, \frac{5}{3}, \frac{2}{1}, \frac{5}{2}, \frac{3}{1}, \frac{4}{1}\}$

1. Overload the comparison operators, $<$, and $\leq$ for the Ratio class.
2. Make the Queue and Stack classes developed in class `template` classes so you can create a queue or a stack of `Ratio`s.
3. Write code that will accept a natural number `n` from the user and then construct a Queue of the Stern-Brocot `Ratio`s and print out the 2^n `Ratio`s of the Stern-Brocot tree in increasing order.
4. Test both the comparison operator and the Stern-Brocot tree by checking that the sequence of `Ratio`s is in increasing order.

submit your code as a zipped file containing files for the `Ratio.h, Ratio.cpp, Queue.h, Queue.cpp, Stack.h, Stack.cpp` and `main.cpp` files of your project.

A *template* in C++ is an outline, a skeleton for a family of related functions and classes. One or more types are generic, to be replaced by specific types at compilation time.

1 Function Templates

Function templates maybe best grasped through an example or two.

Consider the function `greater()` whose definition is overloaded:

```
1  //File greater.h
2
3  #ifndef GREATER_H
4  #define GREATER_H
5
6  int greater(int x, int y) {return (x>y) ? x : y; }
7  long greater(long x, long y) {return (x>y) ? x : y; }
8  double greater(double x, double y) {return (x>y) ? x : y; }
9  char greater(char x, char y) {return (x>y) ? x : y; }
10
11 #endif
12
13 //File main.cpp
14
15 #include <iostream>
16 #include "greater.h"
17 using std::cout;
18
19 int main() {
20    int a = 1, b = 2;
21    cout << greater(a,b) << endl;
22    long c = 4L, d = 3L;
23    cout << greater(c,d) << endl;
24    double x = 4.44, y = 3.33;
25    cout << greater(x,y) << endl;
26    char ch1 = 'A', ch2 = 'a';
27    cout << greater(ch1,ch2) << endl;
```

This produces the output:

```
2
4
4.44
a
```

This produces correct results, but it's too much repetitive coding. If you wanted to change the code, you'd have to change each overloaded instance. This greater() function probably won't change, but you can imagine situations where a more complicated overloaded function would make it difficult to track down all duplicates.

How to write a function template.

The function template acts as a kind of blueprint for the compiler to create functions at compile time. All templates start with the *template parameter list* which consists of

- The C++ keyword `template`
- A left brocket (`<`)

- A list of one or more *generic types* separated by commas. There are two types: *
  * The keyword `class` or `typename`
  * An identifier of your choice that will represent the generic type, typically `T`.
- A right brocket (`>`).

Here's then the revised `greater()` function as a template function:

```
//File greater.h

#ifndef GREATER_H
#define GREATER_H

template <typename T>
T greater(T x, T y) {return (x>y) ? x : y; }

#endif

//File main.cpp
#include <iostream>
#include "greater.h"
using std::cout;

int main() {
   int a = 1, b = 2;
   cout << greater(a,b) << endl;
   long c = 4L, d = 3L;
   cout << greater(c,d) << endl;
   double x = 4.44, y = 3.33;
   cout << greater(x,y) << endl;
   char ch1 = 'A', ch2 = 'a';
   cout << greater(ch1,ch2) << endl;
```

That produces the same output. When the function template is called, the compiler deduces the actual type of argument in context and substitutes it for the generic type `T`, thereby instantiating what is called a *generated function*, which it's programmed to do only once for each particular type substituted for `T`.

This template definition can be further improved by using constant reference variables, instead of passing by value. It makes little difference with primitive types (as in this example) but if you're passing a user-defined type, then you'll be unnecessarily invoking the copy constructor and finding the greater of two copies of the values, instead of the values themselves. So consider receiving and returning gheneric types by reference-to-`const` rather than by value:

```
//File greater.h

#ifndef GREATER_H
#define GREATER_H

template <typename T>
T const &greater(T const &x, T const &y) {return (x>y) ? x : y; }

#endif

//File main.cpp works the same as before
```

Of course, the design of the class `T` will beed to ensure that both `operator>()` and `operator<<()` are properly overloaded.

Also, if the user of `greater()` wants the capability of modifying whatever argument is greater, the function must return its answer by *value* instead of reference-to-`const`.

Where do function templates go?

In the *inclusion model*, all the code for the function definition is in the header file, as part of the interface. That's the simplest, safest way to avoid duplication.

Class Templates

A class template begins with a template parameter list. Use a function-style cast with no parameter to specify a default function argument.

Consider how you may want to represent a complex number which has a real and imaginary part. You may want to use different primitive types to represent these in different contexts. Here is the start of a complex number class whose abstraction consists of real and imaginary parts of some generic type. We'll throw in a static data member to boot.

```
1  //File complex.h
2
3  #ifndef COMPLEX_H
4  #define COMPLEX_H
5
6  template <typename T>
7  class Complex {
8  public:
9     Complex(T const &real = T(), T const &imga = T());
10    Complex(Complex const &); //constructors
11 private:
12    T _real; // the real part
13    T _imag; // the imaginary part
14    static int counter;
15 };
16
17 #endif
```

If `T` is a fundamental type, the expression `T()` will produce whatever value this type would take on it an instance were defined in the global space with no explicit value (i.e., zero). If, however `T()` is a user-defined type, the exptression `T()` will invoke the default constructor for type `T`, and the timporary object that is created will be used to initialize `_real` and `_imag` (unless the user provides overriding values.)

Defining members outside the class definition

When defining class members outside the class definition, you must parameterize the class name wherever it's used. This entails following the class name with brocketed parameter types like so:

```
Complex<T>
```

Here's how you'd define the constructor, for instance:

```
1  //File complex.h
2
3  #ifndef COMPLEX_H
4  #define COMPLEX_H
5
6  /* as above */
```

```
template <typename T>
Complex<T>::Complex(T const &real, T const &imag)
                : _real(arg.real), _imag(arg.imag) { }

template <typename T>
Complex<T>::Complex(Complex<T> const &arg)
                : _real(a

#endif
```

Instantiating a class template

Unlike function templates, with a *class template* you are always responsible for explicitly specifying the types to be used. This is done when the class template is instantiated (yielding a *generated class*) by following the class name with the specific type(s) written between the brockets. This can be clumsy and long, so using a `typedef` may be a good idea:

```
// Instantiate with type 'int' (without using a 'typedef')
Complex<int> x(1,2); //1+2i
Complex<int> y(3,4);

// Instantiate with type 'double' (now using a 'typedef')
typedef Complex<double> COMPLEX_DOUBLE;
COMPLEX_DOUBLE w(3.3,4.4);
COMPLEX_DOUBLE z(5.5,6.6);
```

You can instantiate a class template with another class template that has been instantiated.
Suppose we have two class templates

```
template <typename T>
class Complex {
  // class members
}

template <typename T>
class Queue {
  // class members
}
```

...then you could create a queue of complex numbers by writing the following:

```
//Instantiate a Queue with type Complex<double>
//(Note the mandatory use of a space after ``double")
Queue<Complex<double> > q;
```