Error Checking and Operator Overloading, and Templates in C++.

Hello and welcome, today we are going to make a program that has a class frac with two templated values that implements basic fraction operations via overloading the math operators. Now that is a bunch of complex programming tal, but what it means is that I will show you some basic examples of Error Checking and Operator Overloading, and Templates in C++.

First open a new project, a blank c++ project in Visual Studio, then right click the project in the solution explorer and create a class called Frac, make sure you click 'inline' here, you want that. We will start by writing a template for a new type of object. You do with the c++ keyword template, then you use the angle brackets then you say <class t>. Templating is just what it sounds like, if you have ever used a template in woodworking or some kind of design work, a template is a general outline. We are going to give the computer a general outline of what we want. What template does in c++ is say to the compiler, I want a template for data, and I want it to be literally anything. We could also say template <typename t>, using the c++ keyword typename, typename basically means 'type of thing', remembering from the last lesson that you cannot store anything in the computer that is not a number, everything in the computer must be a number, and another word for number is class. That is why we can use typename or class here , because they are synonymous in this instance, but I prefer class because it is shorter, less letters to type.

```
template <class t>
class Frac
{
Public:
```

We named our templated type t. So just like you say int a, b; to make two integers a and b. Or you say double a, b; to make doubles a and b. You say t a, b; to make two objects of the type t a and b.

The t we are using here can be any type because it is a template.

```
//data members
t a, b;
```

//Just like every other object Frac must exist. So here we are making Frac exist. With a few constructors, remembering from the last lesson that we have different constructors for different ways to construct stuff.

```
//constructors
Frac();

Frac(t, t);

Frac(t);
```

```
	//This here is operator overloading, take a note of the funny syntax, it won't get better.
```
This is why I am showing you the overloading and the templates in this way, because in the future you won't see this stuff in a simple tutorial. In the wild, templates and operator overloading will be used in conjunction with a class to do stuff, so that is what we are doing here, we are making a class that uses a templated type and an overloaded operator to do math.

```
	//functional members
	Frac<t> operator * ( Frac<t>&);
	Frac<t> operator / ( Frac<t>&);

};
```

Now that we have our declaration of the class we can write our definition. First the constructors.

One for when we just make a fraction, since you cannot divide any real number by zero, and if you could zero might go into zero one time, fractions can never have a zero on the bottom, and if the top value was zero the value would be zero and there would be no fraction. But we can have a fraction 1/1. So….

```
template<class t>
inline Frac<t>::Frac()
{
	a = 1; b = 1;
}
```

If we only have one input, one parameter for our constructor, we can just make our denominator 1 and our numerator our parameter.

```
template<class t>
inline Frac<t>::Frac(t x)
{
	a = x; b = 1;
}
```

When we have two parameters of course we just assign the first as numerator and second as denominator.

```
template<class t>
inline Frac<t>::Frac(t x, t y)
{
```

```
	a = x; b = y;
}
```

This is the first of our overloaded operators. This is the multiplication of fractions.

```
template<class t>
Frac<t> Frac<t>::operator * ( Frac<t>& otherFrac)
{
	Frac<t> tempF;                                          // make temporary Fraction
object
	tempF.a = this->a * otherFrac.a;	// a1 * a2
	tempF.b = this->b * otherFrac.b;	// b1 * b2
	return tempF;                                       // return temp
}
```

//with dividing fractions against each other we just invert one of the functions and multiply, and that is easy enough with a copy and paste and a few changes

```
template<class t>
Frac<t> Frac<t>::operator / ( Frac<t>& otherFrac)
{
	Frac<t> tempF;                                          // make temporary Fraction
object
	tempF.a = this->a * otherFrac.b;	// a1 * b2 //mult by reciprocal
	tempF.b = this->b * otherFrac.a;	// b1 * a2
	return tempF;                                       // return temp
}
```

//so that is the class now let's use it. In main. Right Click the project and make a main file.
In main we will just start with simply using the fraction and cout-ing it.
We can see here that we can put anything we want into this frac object. <float> , double, int, char, but of course If we do that we will have to actually put that stuff into the constructor call here like frac(6, 4) instead of any decimal. So if we run that we get the numerator of our one fraction.

```
int main() {
	Frac<double> frac(6.3, 6.1);

	std::cout << frac.a << std::endl;
}
```

Another thing we could do is pretty cool. We can actually take the templated object we just made, and put another one inside of it. Like this, Frac<Frac<double>> frac(4, 5); Then fix this down here, add another a frac.a.a
Because we do not have a constructor we cannot put different numbers in the denominators, Frac.a.b will be one as will frac.b.b, so this won't do much for us because we would have to write a bunch of more stuff in but that is just to show you how powerful templating and custom classes are together.

Now we are going to move on and work with our frac object to enter some numbers into the computer so I can demonstrate Error checking. When I have seen this used ‘in the wild’ it usually looks something like this.

```
int main() {
	Double x, y;
	try {

			std::cout << "Please enter two numbers for a fraction separated by a
space: ";

			std::cin >> x;

			std::cin >> y;

			if ( y == 0) {
				throw y;
			}

			Frac<double> fract(x, y);
			std::cout << "Your Fraction is " << fract.a << "/" << fract.b << std::endl;


	}
}
```

Now that we have our throw if you compile it will fail, because we need a catch. Like literally we need to use the c++ keyword catch. So that looks like this

```
	catch (int x) {
		std::cout << "Cannot divide by “<< x<< “!" << std::endl;
	}
```

Because 0 is an integer value we can catch it with an integer call that x and then cout that.

But now if we run our program we will get our fraction. That is cool, if we run our program and enter a zero, we get our error message.
That is cool, but we can do just a bit more. Let's use that overloaded operator. What I want to do is add a fraction to another fraction and check the result against a calculator.

So here is my new main using some error checking to do that, to add two fractions.

```
int main() {
	int x, y;

	try {

		std::cout << "Please enter two numbers for a fraction separated by a space: ";

		std::cin >> x;

		std::cin >> y;

		if (x == 0 || y == 0) {
			throw y;
		}

		Frac<double> fract(x, y);

		std::cout << "Please enter two numbers for a second fraction separated by a
space: ";

		std::cin >> x;

		std::cin >> y;

		if (x == 0 || y == 0) {
			throw y;
		}

		Frac<double> fract2(x, y);

		fract = fract2 + fract;

		std::cout << "Your fraction is equal to: " << fract.a / fract.b << std::endl;
	}
	catch (int) {
		std::cout << "Cannot divide by zero!"<< std::endl;
	}
```

```
	return 0;
}
```

If we run that, enter ⅝ and 6/7 we get us 1.65something and if we do that in a calculator, we get our answer. All this operator overloading, templating and error checking will allow us to run this program over and over again. And if you stick around for the micro-lecture I will show you a bit more. So about looping and the pont of all this.

1. First we never know what all types of data we will encounter, templating can help with that, Templating.
2. When we have a new type of data we may need to add, subtract or otherwise do an operation that is defined by a symbol or keyword, we can overload that operator to work with our new data. Operator Overloading
3. Always remember, whatever you write professionally as a programmer can be called trillions of times over the lifetime of software. So sometimes that code can encounter unexpected errors. So how to handle those errors is on input, and we call that error checking. Error Checking

Now the point of all of this is that we are now able to throw a while loop around our try block like this.
Notice that when you run this it will eventually somehow end up in an infinite loop.
How to fix that is to replace our ‘std::cin’ with a different function, the reason for this is kinda complex and you should read about it yourself but the function we are going to use will clear our output stream.

```
int main() {
	int x, y;
	while (true) {


		try {

			std::cout << "Please enter two numbers for a fraction separated by a
space: ";

			std::cin >> x;

			std::cin >> y;

			if (x == 0 || y == 0) {
				throw y;
			}

			Frac<double> fract(x, y);
```

```
                        std::cout << "Please enter two numbers for a second fraction separated
by a space: ";

                        std::cin >> x;

                        std::cin >> y;

                        if (x == 0 || y == 0) {
                                throw y;
                        }

                        Frac<double> fract2(x, y);

                        fract = fract2 + fract;

                        std::cout << "Your fraction is equal to: " << fract.a / fract.b << std::endl;
                }
                catch (int) {
                        std::cout << "Cannot divide by zero!" << std::endl;
                }
        }
        return 0;
}
```

So no real lecture or homework this week. If you want you can make functional members of the frac object that return the GCF or find some way to reduce the functions or something, like make the function do all the stuff that people want functions to do. That would be a hard assignment but you have all the understanding to do that.

Just get ready because the next lesson after this is the Linked list and that one will probably be significantly harder than this.