Lab 11: Overloading Operators

CPSC 1021 – Spring 2018

CLEMSON School of COMPUTING

Due Date: Friday, April 20, 2018 @ Midnight

Lab Objective

- Discuss operator overloading
- Implement a number of operators over the Pixel class

Introduction

C++ gives us programmers great flexibility in how we organize our code via classes, namespaces, etc. These language features allow us to change and mold the structure of our own code… but what if we could also change **the syntax of the language itself?** This might sound a little wild, but as a matter of fact C++ overloaded operators allow us to do just that.

We've so far seen the assignment operator and the stream insertion (print) operators, but there are over **40** different operators that you can overload for the classes you write. Today we'll talk a little more about overloaded operators and implement a few for the Pixel class.

Resources

THESE PAGES ARE AN AWESOME RESOURCE, READ THEM!

http://en.cppreference.com/w/cpp/language/operators

http://en.cppreference.com/w/cpp/language/user_literal

Assignment

Download the Pixel class from Canvas (Pixel.h/Pixel.cpp). This is the same class as in previous starter code. There is also a test driver lab11.cpp which contains tests you can run to check your solution. We will discuss operators in detail later in the write-up, but to frame the conversation here is your assignment:

Implement the following overloaded operators for the Pixel class.

Arithmetic

- Addition and subtraction of 2 Pixel objects (operator+, operator-)
- Addition and subtraction of 1 Pixel object and an int (operator+, operator-)
- Multiplication and division of 1 Pixel object and an uint (operator*, operator/)

```
Pixel p1(100, 100, 100);
Pixel p2(50, 50, 50);

// sample 2 Pixel math
p1+p2 => (150, 150, 150)
p1-p2 => (50, 50, 50)
p2-p1 => (0, 0, 0)          // min bound
```

```
p1+p1+p1 => (255,255,255)  // max bound
// sample 1 Pixel math
p1+25 => (125, 125, 125)
p1+200 => (255, 255, 255)  // max bound
p1-25 => (75, 75, 75)
p1-125 => (0, 0, 0)        // min bound
p2*2 => (100, 100, 100)
p2*10 => (255, 255, 255)   // max bound
p2/10 => (5, 5, 5)
10/p2 => ERROR: Pixel may only be numerator
```

Each operator should return a **Pixel object.** This is a new instance (don't use the new keyword, just construct and return) which has the correct values after the arithmetic operation. Note the boundary conditions, and make sure they are upheld. This means you may need to use a different type than uint8_t for the math, or some casting may be involved. For +, -, *, and / you need only handle constant value operations where the integer or uint value is on the **right side**.

```
Pixel p1(100, 100, 100);
p1 + 25 => (125, 125, 125)  => OK
25 + p1 => (125, 125, 125)  => ERROR, DON'T IMPLEMENT
```

Relational Operators

- All comparison operators: <, >, <=, >=, ==, !=
- The red value is highest priority, then green, then blue. Like when we sorted Pixels

```
Pixel p1(100, 50, 25)
Pixel p2(90, 50, 25)
Pixel p3(100, 75, 10)

p1 < p2 => false
p1 > p2 => true
p1 != p2 => true
p1 <= p3 => true
p3 >= p2 => true
```

Important to note here that you only need to write comparison code **for the < operator and the == operator**. This is important from a code reuse standpoint: How can you implement the > operator (and all others!) if we know that the < operator already works? These operators return the primitive bool type.

Stream Extraction

- The stream extraction operator <<

We've already seen how the **stream insertion** operator << can be used to print Pixels into an ostream. Review this code in the provided starter files. Now, we'd like to overload the **extraction** operator >>, which allows us to take text information from a stream (like a file ifstream) and parse the values presented.

Observe the following example:

```
stringstream s;
s << “100 100 100 200 50 100”;

Pixel p1; => (0, 0, 0)
Pixel p2; => (0, 0, 0)

s >> p1;  => (100, 100, 100)
s >> p2;  => (200, 50, 100)
```

For this lab we will only worry about streams that have literal digits in them, so P3 images. The extraction operator returns an istream&: the function will look very similar to the ostream insertion operator and the pixel reading code in the Image class. The function returns an ifsteeam&.

Discussion on Overloaded Operators

Overloading operators for a class is not a C++ specific feature; many languages allow you some form of operator control. But why do these languages provide us with such a feature when it could easily lead to confusing code? Remember that almost every new technique we have learned in C++ has aimed to make the act of **reading and writing** code easier on the programmer. We want our code to be **conversational**, that is you should be able to read your code as if it were a plain old paragraph of English.

Usually, we reserve algebraic notation like + or – for literal numbers: integers, floating points, rational numbers, etc. When we talk about or work with plain old numbers, the **meanings** of these operator symbols is clear. But what if we want to work with more complicated objects like matrices, vectors, strings, or Pixels? Is there a clear meaning behind the + or – operator when working with these entities?

Sometimes yes, and sometimes no! If we find that a **simplification of syntax** could be helpful, it might make sense to overload an operator to make our program cleaner:

```
Matrix a = {
  { 1, 0 },
  { 0, 1 }
};

// explicit scale operation
a.scale(2); => 2 0
               0 2

// implicit scale operator
a * 2; => 2 0
          0 2
```

The above code is just **syntactic sugar**, since “a * 2” will literally be compiled to:

```
a * 2 => a.operator*(2)
```

Sometimes overloading operators can make our code **harder to read** if the implied operation is **not clear from the context of the class**. For example, what would it mean to take the negative of a Pixel?

```
Pixel p(100, 100, 100);
Pixel p2 = -p;  => What should this do?
```

You might be inclined to think the above code should produce a Pixel(155, 155, 155), which is the *photographic* negative. But to someone else reading your code, that decision is completely arbitrary; they could have a completely different interpretation of what a "negative Pixel' should be. We only want to define operators that **make sense in the context of the class alone**. Make sure that if you define an operator, its meaning should be clear without any explanation.

Adding two Pixels together has a straightforward interpretation of combining the matching color values from two Pixels. **Subtracting** two Pixels has a similarly straightforward interpretation. Even adding/subtracting a constant integer to a Pixel makes some intuitive sense; simply add the constant to each color field.

However, these operators could just as easily be defined as **explicit member functions**… and they probably should be! Even though I claimed that subtraction of a constant from a Pixel makes sense, what about the other way around?

```
Pixel p(100, 100, 100);
50 – p;  => What does this mean? Makes no sense…
```

Here we see the pitfalls of overloading operators: **not all operators have clear or meaningful definitions!**

The argument could be made that defining any arithmetic operators on the Pixel class is a little foolish: I don't necessarily disagree!

Friend vs. Member Functions

If you have not, please read the first link in the resource list. It will explain operators better than I ever could. Specifically, this table is very useful:

Expression	As member function	As non-member function	Example
@a	(a).operator@ ( )	operator@ (a)	`!std::cin` calls `std::cin.operator!()`
a@b	(a).operator@ (b)	operator@ (a, b)	`std::cout << 42` calls `std::cout.operator<<(42)`
a=b	(a).operator= (b)	cannot be non-member	`std::string s; s = "abc";` calls `s.operator=("abc")`
a(b...)	(a).operator()(b...)	cannot be non-member	`std::random_device r; auto n = r();` calls `r.operator()()`
a[b]	(a).operator[](b)	cannot be non-member	`std::map<int, int> m; m[1] = 2;` calls `m.operator[](1)`
a->	(a).operator-> ( )	cannot be non-member	`auto p = std::make_unique<S>(); p->bar()` calls `p.operator->()`
a@	(a).operator@ (0)	operator@ (a, 0)	`std::vector<int>::iterator i = v.begin(); i++` calls `i.operator++(0)`
in this table, @ is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b			

Non-member functions are functions in a class that **do not have access to the "this" pointer.**

This is crucial to understand, and we've seen it before!

For example:

```
class A {
  static void do_thing(const &A);
  int do_other_thing();
};
```

For the class A, the "do_thing" function is **static** and therefore does not have access to an **instance** of the class. Trying to use the "this" pointer in the function would fail. "do_other_thing" is a member function, and therefore could use the "this" pointer to access instance data.

In general:

- Arithmetic operators are **non-member functions**
- Relational operator are **non-member functions**
- Stream insertion/extraction are **non-member functions**

In fact, most operators you'll implement are non-member. This is mostly for symmetry purposes: it's nice when all your operator functions have similar signatures because it is more **maintainable**. At the end of the day it's really up to you and your programming style. However, the insertion/extraction operators (<< and >>) should **always be non-member functions**. To see why:

```
class Pixel {
...
  friend ostream& operator << (ostream&, Pixel&);
  ostream& operator << (ostream&);
}

Pixel p;

// Using the friend non-member
cout << p;  => compiles to operator<<.(cout, p);

// Using the member function
p << cout;  => compiles to p.operator<<.(cout);
```

The second form, the member function, is clunky and looks like it is in the wrong order! Both functions will work, but the second just goes against convention and I not commonly used.

Notice also the use of the **friend** keyword! "friend" is a modifier that allows **non-member functions access to member data!** Even though a friend non-member function won't have a "this" pointer, it can still freely access the private data and functions of the class it is friends with. The starter code has an example of this for the operator<< function.

Every operator you overload in the lab should use the friend modifier!

What to Turn In

You must implement **13** overloaded operators as described at the beginning of the lab.

Please submit in an archive:

- Pixel.h and Pixel.cpp

Use the provided test program to make sure your code is working!

Tips and Tricks

I have provided a clamp() function you will probably need while doing the operator math!

Also, check out the std::tie function:

http://en.cppreference.com/w/cpp/utility/tuple/tie

Specifically, look at the example code for a quick way to compare simple classes or structs…

Compile and Execute

Use GCC to compile your code as follows:

```
g++ -Wall -g Pixel.cpp lab11.cpp -o PIXEL
```

Execute the program

```
./PIXEL
```

Formatting

Your program should be well documented!

1. Each file should include a header.
2. Your program should consist of proper and consistent indention
3. No lines of code should be more than 80 characters

5 – 10 points will be deducted for each of the above formatting infractions.

Submission Instructions

- Test your program on the School of Computing server prior to submitting.
- Use the tar utility to tar.gz all source files. **Do not tar an entire directory!**
  When I decompress your archive, I should see all of the files you included, not a top level directory!
  Failure to correctly tar may result in up to a 25 point penalty!

```
tar -czvf nglyder-lab11.tar.gz Pixel.*
```

- Name your tarred file **<username>-lab<#>.tar.gz** (ex. nglyder-lab11.tar.gz)
- Use handin (http://handin.cs.clemson.edu) to submit your archive