CPE 212 - Fundamentals of Software Engineering

Operator Overloads

Reminder:
Project 03 documentation is due this Friday by 11:59pm

Outline

- Defining Operators
- Why use them?
- Coding Examples

Operators

Operators are the fundamental tools used to do things in any programming language. You have assignment operators, arithmetic operators, compound assignment operators and the list goes on.

Operators in C++, and in most all languages, are contextual. Meaning, for different data types different operators occur.

In C++ operators are treated like functions and can be “Overloaded” much like any of our objects in our previous examples.

This allows us to create highly generic data structures and algorithms that work irrelevant of the type being used

List of Operators

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

expression	equivalent to...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units+1);`

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

&& OPERATOR (and)		
a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

|| OPERATOR (or)		
a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Goals

```cpp
#include "student.h"

template<typename Type>
void PrintEqual(const Type &one, const Type &two)
{
    std::cout << std::boolalpha << one == two << std::endl;
}

int main()
{
    PrintEqual(1, 2);
    PrintEqual(1, 1);

    Student one("Josh", "Langford", 12345678);
    Student two("Leo", "Langford", 87654321);

    PrintEqual(one, two);
}
```

How do we compare students?

Operator Overload
Declaration

```
class Student
{
private:
    std::string _firstName;
    std::string _lastName;
    unsigned int _studentID;

public:
    Student();
    Student(const std::string & first, const std::string &last,
            unsigned int id);


    unsigned int GetID() const;

    bool operator==(const Student &otherStudent);
    bool operator!=(const Student &otherStudent);
};
```

Operator Overload

Implementation

```
bool Student::operator!=(const Student &otherStudent)
{
    return otherStudent.GetID() != this->_studentID;
}

bool Student::operator==(const Student &otherStudent)
{
    return otherStudent.GetID() == this->_studentID;
}
```

Operator Overload

Format

```
Return_Type ClassName::operator op(Argument list)
{
    // Function body
}
```

2D Vector Example

```
struct vec
{
    float x;
    float y;

    // Constructors
    vec() : x(0), y(0) {}
    vec(float _x, float _y) : x(_x), y(_y) {}
};
```

More Operators

For a 2D vector let's add the following operators:

+ Add two vectors together

- Subtract two vectors

++ Increment a single vectors values

<< Print a vector

Adding Vectors

```
vec operator + (const vec& rightSide)
   {
       return vec(this->x + rightSide.x, this->y + rightSide.y);
   }
```

Subtracting Vectors

```
vec operator - (const vec& rightSide)
   {
       return vec(this->x - rightSide.x, this->y - rightSide.y);
   }
```

Incrementing a single vector

```
vec operator ++ (int)
   {
       x++;
       y++;
       return *this;
   }
```

Printing a vector

```
int main()
{
    vec a = {1.0, 2.0};
    cout << "Vector A x = " << a.x << " y = " << a.y << endl;
    return 0;
}
```

Printing a vector

```
ostream& operator << (ostream& stream, const vec<T> v)
{
    stream << "Vector x = " << v.x << " y = " << v.y << endl;
    return stream;
}
```

How do we make this 2D vector generic?

Template Declaration

```cpp
// Forward Declaration
template<typename T>
void Function(T data);

template<typename T>
void Function(T data)
{
    // do stuff with data of type T here
}
```

Adding Templates

```cpp
template <class T>
struct vec
{
    T x;
    T y;

    // Constructors
    vec() : x(0), y(0) {}
    vec(T _x, T _y) : x(_x), y(_y) {}

    // Overloads
    vec operator + (const vec& rightSide)
    {
        return vec(this->x + rightSide.x, this->y + rightSide.y);
    }

    vec operator - (const vec& rightSide)
    {
        return vec(this->x - rightSide.x, this->y - rightSide.y);
    }

    vec operator ++ (int)
    {
        x++;
        y++;
        return *this;
    }
};
```

Example

```cpp
int main()
{
    vec<float> a = {1.0, 2.5};
    vec<float> b = {4.0, 7.5};
    vec<float> c = {2.0, 3.5};

    vec<int> d = {1, 2};
    vec<int> e = {4, 5};

    a.x = b.x + c.x;

    cout << "A " << a << endl;

    b = a + c;

    cout << "B " << b << endl;

    c++;

    cout << "C " << c << endl;

    vec<int> f = d + e;
    cout << "F " << f << endl;

    return 0;
}
```

Operator Overloading Resources

Book: 6.5, page 383 -> 387.

https://www.geeksforgeeks.org/operator-overloading-c/ is also a good read.