CSE 333: Systems Programming

Section 5

Operator overloading

Operator overloading

- C++ allows for overloading of operators such as +, -, *, /, ->, [], and so forth
  - This is extremely powerful, but with great power comes great responsibility
- To overload or define an operator, declare *operator+*, *operator-*, etc. as a function inside a class (or sometimes globally)
- Let’s look at an example…

Operator overloading

```
class IntArray {
 public:
  inline IntArray(int len)
    : array_(new int[len]), len_(len) {}
  inline IntArray(const IntArray& int_array)
    : array_(new int[int_array.len_]), len_(int_array.len_) {
    memcpy(array_, int_array.array_, sizeof(int) * len_);
  }
  ~IntArray() { delete array_; }
  inline const int& operator[](int i) const {
    range_check(i);
    return array_[i];
  }
  inline int& operator[](int i) {
    range_check(i);
    return array_[i];
  }
  inline int length() const { return len_; }

 private:
  inline void range_check(int i) const {
    assert(i >= 0 && i < len_);
  }
  int* array_;
  const int len_;
};
```

Operator overloading

- We just defined a “safe” array class for storing integers. We can now do:

```
IntArray arr(10);
for (int i = 0; i < arr.length(); ++i) {
  arr[i] = i;  // okay
}
arr[15] = -1; // assertion failure!
```

- Our *range_check()* function protects against indices that are out of bounds

Operator overloading

- Let's say that we want to implement + and – operators that perform pairwise addition and subtraction

- We can write declarations for them as:

```
IntArray operator+(
    const IntArray& int_array) const { ... }
IntArray operator-(
    const IntArray& int_array) const { ... }
```

- And now if we have two *IntArrays* called *arr1* and *arr2*, we can compute *arr1 + arr2* and *arr1 - arr2*

Operators for built-in types

- In a global scope (i.e. outside of the class), we can define operators for built-in types
- To facilitate the << operator for IntArray for use with streams, we can declare the following outside of the class in the header file:

```
ostream& operator<<(
    ostream& o, IntArray int_array;
```

- The same technique can be applied to other operators as well, such as *operator+*, *operator-*, etc.

Operator misuse

- Operator overloading can easily be misused, unfortunately. For instance, I could define the following operator inside IntArray:

```
double operator+(const string& str) const;
```

- This would allow me to write:

```
IntArray arr(5);
double d = arr + "hello";
// Please, please do not do this
```

Operator design

- Now let’s imagine that we are writing a hash table in C++ that maps *uint64_ts* to *void** pointers and we want to define *operator[]* to access values
  - If *tab* is an instance of this class, I want to be able to write *tab[key] = val* to insert *val* under *key*
  - In the future, I should be able retrieve it via *tab[key]* or to overwrite it with a different value
- How should we declare *operator[]*, and how should we implement it? Keep in mind that the given key may or may not be present

Section assignment

- In section today, you will flesh out a three-dimensional vector class that stores doubles
- The provided code will not compile until you at least implement the constructors
- Uncomment the relevant test code as you implement features to see if your code works
- Submit vec3d.h to the Dropbox once you finish. Leave a comment on the Dropbox with your partner's name!