CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Operator Overloading

We talked previously about operator overloading, and we saw that it isn't always good or bad. We talked about how it can be an important tool, particularly for writability, but that we must be careful to overload in ways that mimic generally accepted standards.

For instance we probably don't want to overload a "*" operator for a car object, because that doesn't have much meaning. But overloading a matrix class to provide multiplication operator so you can multiply matrices more simply is a meaning most people would understand.

In C++ there is another reason to overload operators. C++ provides a very powerful set of libraries called the Standard Template Library (STL). In the STL we have four components:

- Algorithms
- Containers
- Functions
- Iterators

The Algorithms component provides capabilities that work on collections of elements such as:

- counting
- sorting
- moving, swapping
- searching, replacing
- reversing
- finding min/max
- etc.

Collections provided by the STL are

- arrays
- vectors
- deques
- lists
- sets
- maps
- stacks
- queues
- others

Functions provides the ability to use Functors, which are classes that can be used like functions (we won't discuss those here).

We briefly discussed iterators before -- they allow us to iterate through some group of items

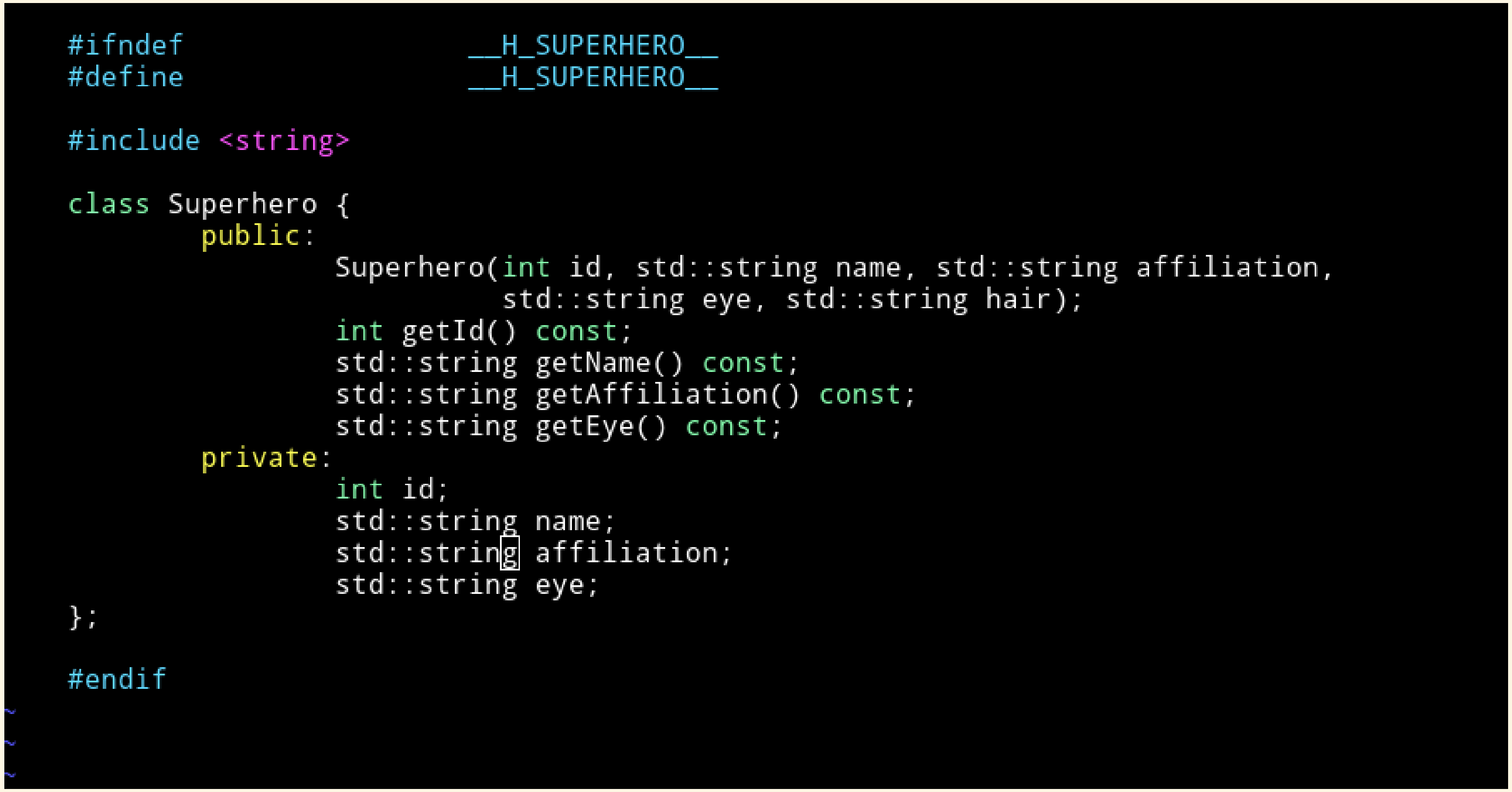
The C++ code provided to us through the STL is considerable and has many advantages. It is optimized and efficient, secure and safe, and has been (and is continuously) vetted by excellent programmers around the globe.

It is in our best interests to use it when possible.

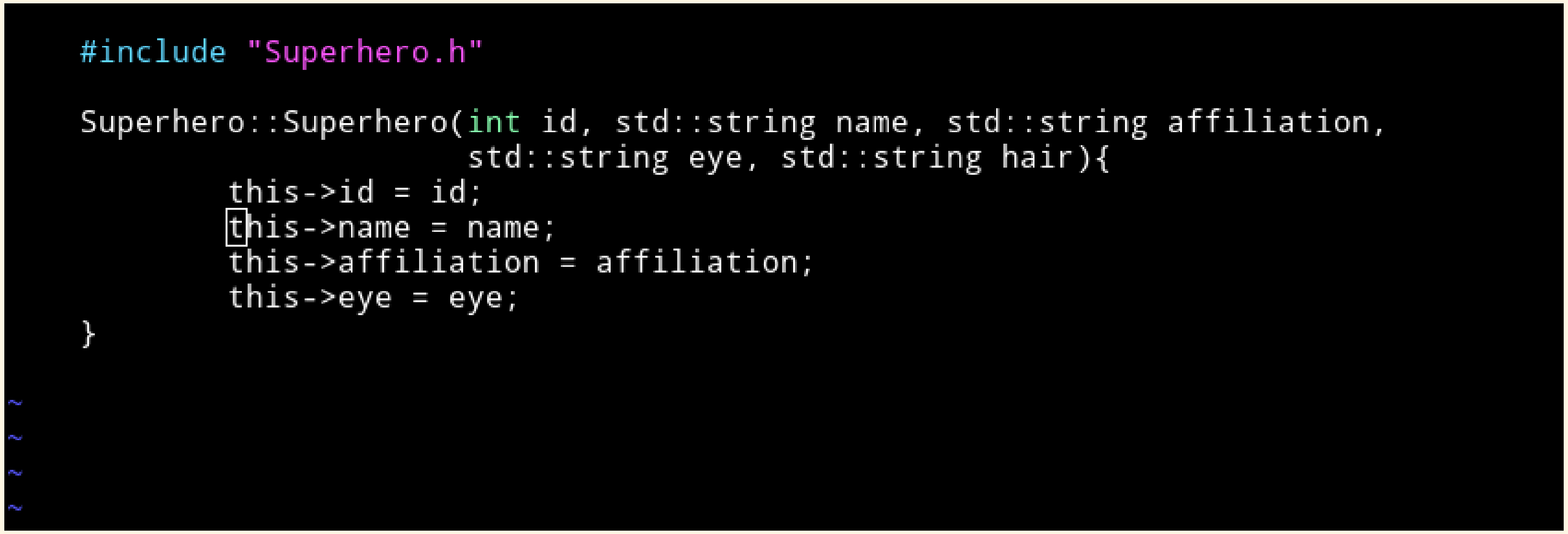
Let us consider a few examples.

With some collections we don't need to provide any sort of overloaded operators. Take std::vectors for instance; since vectors don't care about ordering of information these aren't required.

If we have the following Superhero class for instance:



```
#ifndef                 __H_SUPERHERO__
#define                 __H_SUPERHERO__

#include <string>

class Superhero {
        public:
                Superhero(int id, std::string name, std::string affiliation,
                          std::string eye, std::string hair);
                int getId() const;
                std::string getName() const;
                std::string getAffiliation() const;
                std::string getEye() const;
        private:
                int id;
                std::string name;
                std::string affiliation;
                std::string eye;
};

#endif
~
~
~
```



```
#include "Superhero.h"

Superhero::Superhero(int id, std::string name, std::string affiliation,
                     std::string eye, std::string hair){
        this->id = id;
        this->name = name;
        this->affiliation = affiliation;
        this->eye = eye;
}

~
~
~
~
```

```
#include "csv.h"
#include "Superhero.h"
#include <iostream>
#include <vector>

int main(int argc, char** argv){
        io::CSVReader<5, io::trim_chars<' '>, io::double_quote_escape<',','\"'> > in("marvel.csv");
        in.read_header(io::ignore_extra_column, "page_id", "name", "ALIGN", "EYE", "HAIR");

        std::vector<Superhero> heros;

        int id;
        std::string name;
        std::string alignment;
        std::string eye;
        std::string hair;

        while(in.read_row(id, name, alignment, eye, hair)){
                // do stuff with the data
                std::cout << name << std::endl;
                Superhero tmp(id, name, alignment, eye, hair);
                heros.push_back(tmp);
        }

        std::cout << "Entered " << heros.size() << " heros into vector." << std::endl;
}
```

We can compile and prove this example works. We can indeed create all the necessary objects and insert them into the std::vector.

But what if we wanted to insert into a priority queue?

You may not remember a priority queue, but it is a data structure that provides access to the least (or most) important element in constant time.

If we try to use a std::priority_queue instead of a std::vector to hold our Superheros, our code looks like this:

```cpp
#include "csv.h"
#include "Superhero.h"
#include <iostream>
#include <queue>

int main(int argc, char** argv){
        io::CSVReader<5, io::trim_chars<' '>, io::double_quote_escape<',','\"'> > in("marvel.csv");
        in.read_header(io::ignore_extra_column, "page_id", "name", "ALIGN", "EYE", "HAIR");

        std::priority_queue<Superhero> heros;

        int id;
        std::string name;
        std::string alignment;
        std::string eye;
        std::string hair;

        while(in.read_row(id, name, alignment, eye, hair)){
                // do stuff with the data
                std::cout << name << std::endl;
                Superhero tmp(id, name, alignment, eye, hair);
                heros.push(tmp);
        }

        std::cout << "Entered " << heros.size() << " heros into priority queue." << std::endl;
}
```

But if we attempt to compile it...

```
...
error: invalid operands to binary expression
       ('const Superhero' and 'const Superhero')
        {return __x < __y;}
                ~~~ ^ ~~~
...
```

We don't have any lines in the code we wrote to cause this...

This happens because the priority queue must be able to compare one Superhero to another in order to figure out which is the least (or most) important.

To compare one element to another the priority queue uses the < operator. Since Superhero doesn't have one, we must create it.

We are going to update our Superhero files:

```
// Add to public section of Superhero.h
friend bool operator< (const Superhero & lhs, const Superhero & rhs);

// Add to Superhero.cpp
bool operator< (const Superhero & lhs, const Superhero & rhs){
        return lhs.id < rhs.id;
}
```

The files will now compile and run as we expect.

In C++ we sometimes need to override the stream output operator. This is roughly equivalent to providing a toString() method in Java:

```cpp
// Add to Superhero.h
friend std::ostream & operator<< (std::ostream & os, const Superhero & obj);

// Add to Superhero.cpp
std::ostream & operator<< (std::ostream & os, const Superhero & obj){
        return os << obj.name << " has " << obj.eye << " eyes, " << obj.hair << " hair, and is " << obj.affiliation << ".";
```

This allows us to send the object to an output stream:

```
std::cout << myHero << std::endl;
```

For instance.

With this code, if we print out an element we get something like this:

```
Ace Maxwell (Earth-616) has  eyes, Black Hair hair, and is Bad Ch
```

Another huge bonus from having overridden the operator< is that we can now use the built in std::sort() function!

```
template <class RandomAccessIterator, class Compare>
 void sort (RandomAccessIterator frst, RandomAccessIterator lst);
```

The iterators for this function dictate where we start and stop sorting in the collection. With our vector of heroes we could then do something like this:

```
std::sort(heroes.begin(), heroes.end());
```

If we print out the first 10 heroes we now get

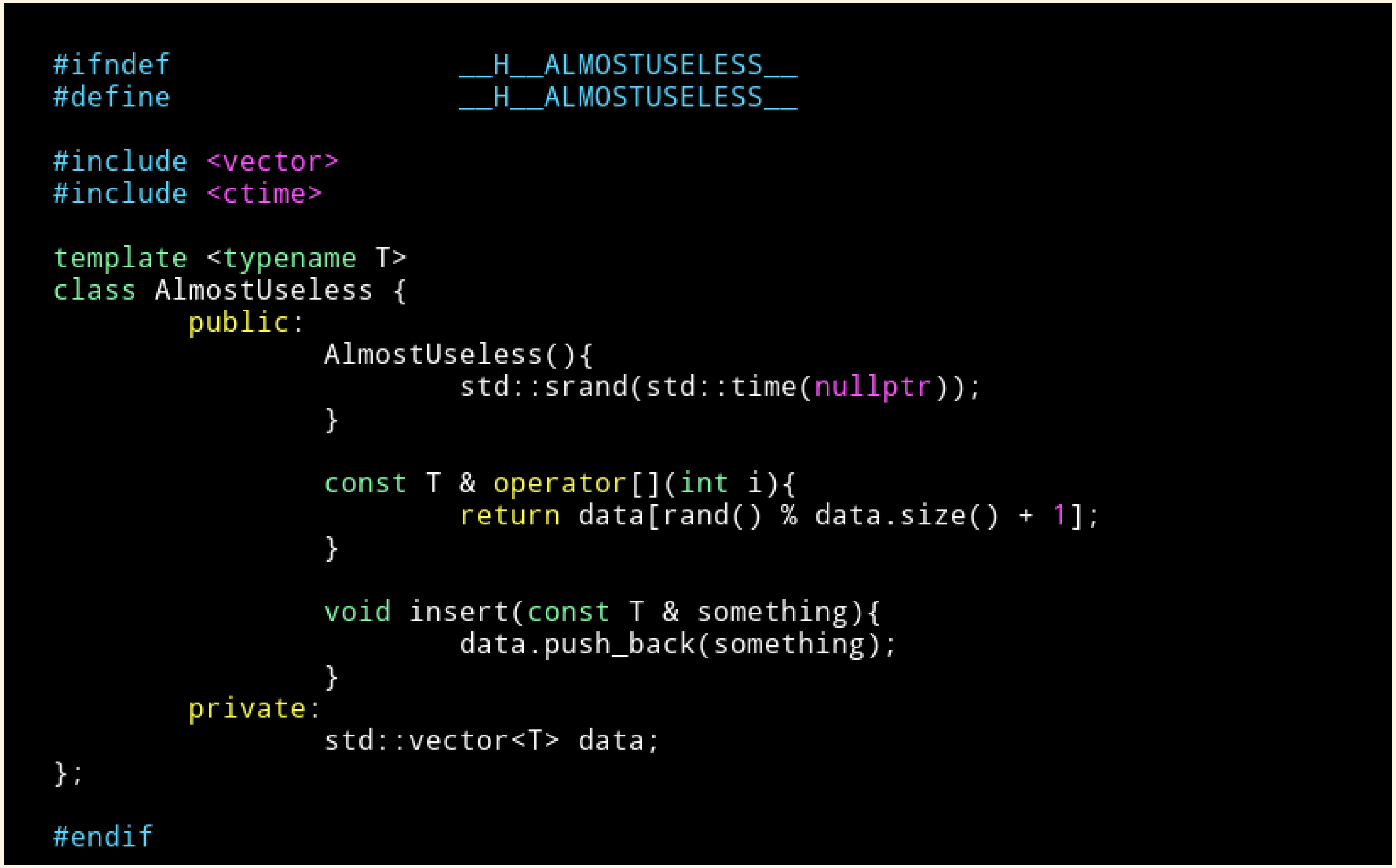
```
Emil Blonsky (Earth-616) has Blue Eyes eyes, Blond Hair hair, and
Absalom (Earth-616) has Blue Eyes eyes, Blond Hair hair, and is .
Carl Creel (Earth-616) has Blue Eyes eyes, Bald hair, and is Bad
Adam Warlock (Earth-616) has White Eyes eyes, Gold Hair hair, and
Adam Neramani (Earth-616) has Blue Eyes eyes, Blond Hair hair, an
Eric Cameron (Earth-616) has Blue Eyes eyes, Brown Hair hair, and
Adri Nital (Earth-616) has Red Eyes eyes, Black Hair hair, and is
Adversary (Earth-616) has Blue Eyes eyes,  hair, and is Bad Chara
Aeroika (Earth-616) has  eyes, Gold Hair hair, and is .
Agatha Harkness (Earth-616) has Blue Eyes eyes, White Hair hair,
```

If we look through the supplied file we see that Emil Blonsky does have the smallest id, with 1025. So this is accurate.

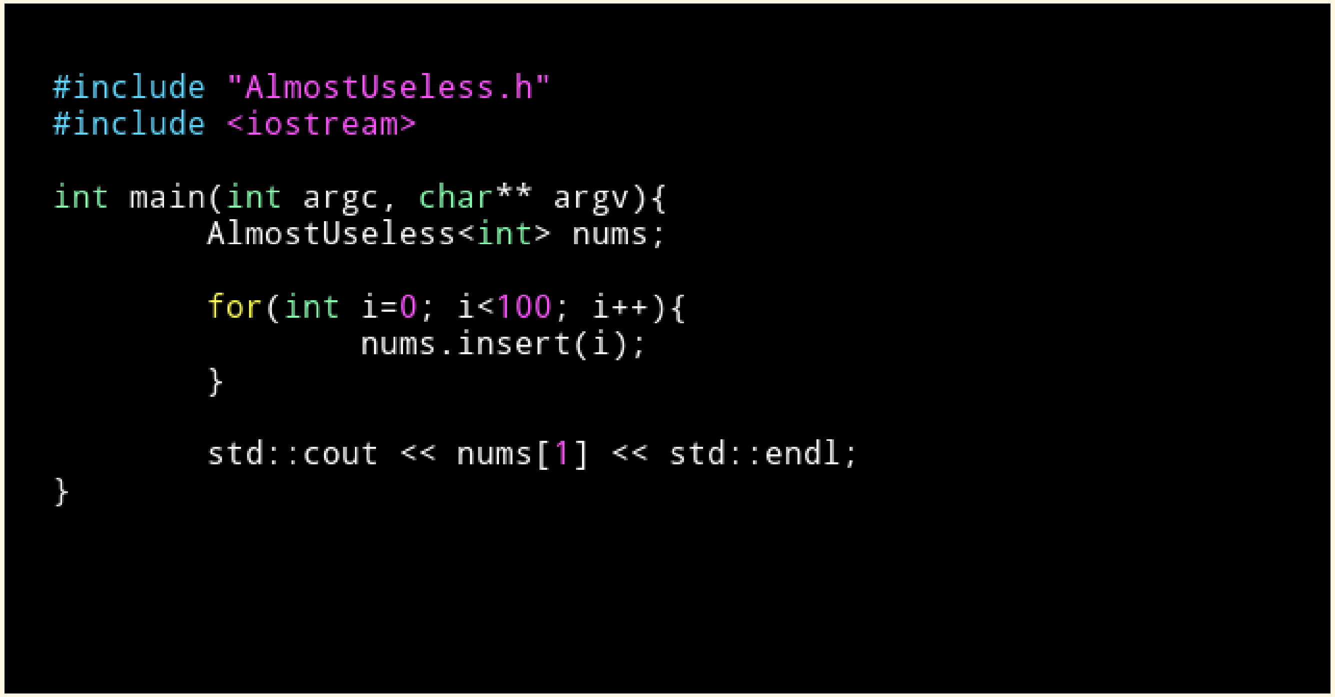
An interesting note is that if we override the operator< we can implement other relational operators in terms of it:

```
bool operator> (const X& lhs, const X& rhs){
   return rhs < lhs;
}
bool operator<=(const X& lhs, const X& rhs){
  return !(lhs > rhs);
}
bool operator>=(const X& lhs, const X& rhs){
  return !(lhs < rhs);
}
```

Another reason we may want to override an operator is if we are creating some collection and wish to allow people to access elements via an index. For instance:



```
#ifndef                 __H__ALMOSTUSELESS__
#define                 __H__ALMOSTUSELESS__

#include <vector>
#include <ctime>

template <typename T>
class AlmostUseless {
        public:
                AlmostUseless(){
                        std::srand(std::time(nullptr));
                }

                const T & operator[](int i){
                        return data[rand() % data.size() + 1];
                }

                void insert(const T & something){
                        data.push_back(something);
                }
        private:
                std::vector<T> data;
};

#endif
```



```cpp
#include "AlmostUseless.h"
#include <iostream>

int main(int argc, char** argv){
        AlmostUseless<int> nums;

        for(int i=0; i<100; i++){
                nums.insert(i);
        }

        std::cout << nums[1] << std::endl;
}
```