601.220 Intermediate Programming

Operator overloading and the friend keyword

Operator overloading

Operators such as `+` and `<<` are like functions

`a + b` is like `plus(a, b)` or `a.plus(b)`

`a + b + c` is like `plus(plus(a, b), c))`

Operator overloading

- C++ allows us to define new classes (i.e. new types), and we can define new meanings for operators so we can use them on these types
  - Overloading means piling on another definition for a name
  - Contrast overloading with overriding, where we replace a definition of a name
  - Operator syntax is familiar, and compact

- We can overload most operators (+ - * / < | & = [] == != <<, etc.)
  - Important to choose new meanings for operators that are intuitive

Operator overloading

- To specify a new definition for an operator with symbol `S`, we define a method called `operatorS`
- The compiler understands that expressions using the infix operator + applied to the types specified in the method should map to the above function.

Operator overloading

`cout <<` works with many types, but not all:

```
// insertion_eg1.cpp:
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    cout << vec << endl;
    return 0;
}
```

Operator overloading

```
$ g++ -c insertion_eg1.cpp -std=c++11 -pedantic -Wall -Wextra
insertion_eg1.cpp: In function ‘int main()’:
insertion_eg1.cpp:9:10: error: no match for ‘operator<<’ (operand types are ‘std::ostream’ {aka ‘std::basi
    9 |     cout << vec << endl;
      |     ~~~~ ^~ ~~~
      |     |       |
      |     |       std::vector<int>
      |     std::ostream {aka std::basic_ostream<char>}
In file included from /usr/include/c++/9/iostream:39,
                 from insertion_eg1.cpp:1:
/usr/include/c++/9/ostream:108:7: note: candidate: ‘std::basic_ostream<_CharT, _Traits>::__ostream_type& s
  108 |       operator<<(__ostream_type& (*__pf)(__ostream_type&))
      |       ^~~~~~~~
/usr/include/c++/9/ostream:108:36: note:   no known conversion for argument 1 from ‘std::vector<int>’ to ‘
  108 |       operator<<(__ostream_type& (*__pf)(__ostream_type&))
      |                  ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~
/usr/include/c++/9/ostream:117:7: note: candidate: ‘std::basic_ostream<_CharT, _Traits>::__ostream_type& s
  117 |       operator<<(__ios_type& (*__pf)(__ios_type&))
      |       ^~~~~~~~
/usr/include/c++/9/ostream:117:32: note:   no known conversion for argument 1 from ‘std::vector<int>’ to ‘
  117 |       operator<<(__ios_type& (*__pf)(__ios_type&))
      |                  ~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~
/usr/include/c++/9/ostream:127:7: note: candidate: ‘std::basic_ostream<_CharT, _Traits>::__ostream_type& s
  127 |       operator<<(ios_base& (*__pf) (ios_base&))
      |       ^~~~~~~~
/usr/include/c++/9/ostream:127:30: note:   no known conversion for argument 1 from ‘std::vector<int>’ to ‘
  127 |       operator<<(ios_base& (*__pf) (ios_base&))
      |                  ~~~~~~~~~~~~^~~~~~~~~~~~~~~~~
/usr/include/c++/9/ostream:166:7: note: candidate: ‘std::basic_ostream<_CharT, _Traits>::__ostream_type& s
  166 |       operator<<(long __n)
```

Operator overloading

We can *make* an operator work by defining the appropriate function:

```
// insertion_eg2.cpp:
#include <iostream>
#include <vector>

using std::cout;   using std::endl;
using std::vector; using std::ostream;

ostream& operator<<(ostream& os, const vector<int>& vec) {
    for(vector<int>::const_iterator it = vec.cbegin();
        it != vec.cend(); ++it)
    {
        os << *it << ' ';
    }
    return os;
}

int main() {
    const vector<int> vec = {1, 2, 3};
    cout << vec << endl; // now this will work!
    return 0;
}
```

Operator overloading

```
$ g++ -c insertion_eg2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o insertion_eg2 insertion_eg2.o
$ ./insertion_eg2
1 2 3
```

Operator overloading

`std::ostream` is a C++ *output stream*

Can write to it, can't read from it

It is `cout`'s type

- `cout` can be passed as parameter of type `ostream& os`
- `const ostream&` won't work, since it disallows writing

Operator overloading

What's really happening when we see this?

```
cout << "Hello " << 1 << ' ' << 2;
```

It executes the `operator<<` function in this order:

```
( ( ( ( std::cout << "Hello" ) << 1 ) << ' '' ) << 2 );
```

Operator overloading

```
ostream& operator<<(ostream& os, const vector<int>& vec) {
    for(vector<int>::const_iterator it = vec.cbegin();
        it != vec.cend(); ++it)
    {
        os << *it << ' ';
    }
    return os;
}
```

Allows `vector<int>` to appear in a typical `cout <<` chain

- Taking `ostream& os` in first parameter & returning `os` enables chaining
- Taking `const vector<int>&` as second parameter allows the `vector<int>` to appear as a right operand in a `operator<<` call

Operator overloading

- Suppose we have defined a class named `Rational` to represent rational numbers, storing an int numerator and an int denominator.
- Then, outside the class, we can declare a method named `operator+` to work on two Rational objects:

```
Rational operator+(const Rational& left,
                   const Rational& right);
```

- Note that arguments are passed in by reference, and since method shouldn't change them, they are `const` references

Operator overloading - instance methods

- This operator+ method likely needs access to the private instance variables inside the class - may make more sense as a member of the Rational class, so let's make it one (declare this inside the class itself):

```
Rational operator+(const Rational& right) const;
```

- Note that we have only one explicit argument now - member instance methods always get one implicit argument (the item pointed to by `this`)
  - the last `const` in that line promises not to modify the implicit object

Operator overloading - instance methods

```
class Rational {
public:

    //...

    Rational operator+(const Rational& right) const;

private:
    int num;  //numerator
    int den;  //denominator

};
```

Operator overloading - instance methods

```
Rational Rational::operator+(const Rational& right) const {
      int sum_num =
          this->num * right.den + right.num * this->den;
      int sum_den = this->den * right.den;
      Rational result(sum_num, sum_den);
      return result;
}
```

Returning an object by value?

Q: Notice that the return type is not a reference nor a pointer. What happens when the method on the previous slide returns its loally-declared result object?

A: The *copy constructor* of the class gets called to make a copy of result before the stack frame is popped (and the result variable is destroyed)

```
Rational(const Rational& original);
```

- If you don't define a copy constructor, a default one is created for you which performs shallow copies.

Copy constructors

- The implicit (compiler-generated) one for a class does simple field-by-field copy, but you can write a different copy constructor if you wish
  - For example, you should write one if your class manages heap memory
- A copy constructor is used in the following situations:
  - when making an explicit call to a constructor feeding it an already-created class object, e.g. Rational r2(r1);
  - when sending a class object to a function using pass-by-value
  - when a class object is returned from a function by value

Overloading the output operator

- If we have `Rational` objects r1 and r2, it's convenient to be able to write

```
cout << r1 << " " << r2 << endl;
```

- But first, how does the chaining up of << operators work?
  - `cout` is an `ostream` type object (the "hose" we put values into)
  - The << operator associates left to right, meaning we evaluate it as the parenthesized version below would suggest:

```
(((cout << r1) << " ") << r2) << endl;
```

Q: What type of value does the operation << return to make this work?

Overloading the output operator

Q: What type of value does the operation << return to make this work?

- A: The << operator returns `ostream` type (returns by reference the first argument)

So, if we want to overload the operator for the `Rational` type, we might try:

```
ostream& operator<<(ostream& os, const Rational& r);
```

Overloading the output operator

- But: to output the value, we may need access to instance variables, which are 'private'
  - So we might want to make it a member of the Rational class...
  - But we can't, since a member method would get the object of that class type as its implicit argument...
  - And the first argument for << needs to be ostream type, not `Rational` type.

Overloading the output operator - using `friend`

- Still, we can make use of the `friend` keyword to give the method "almost-member" status:

```
class Rational {
public:
    // ...
    friend ostream& operator<<(ostream& os,
                               const Rational& r);
private:
    // ...
```

- This says that the method is trusted by the class, meaning it is made allowed to access private member variables.
    - This method is not an actual member of the class.