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 operator overloading with function 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
- 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

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

```
// operator1.cpp
#include <iostream>
#include <vector>

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

```
$ g++ -o operator1 operator1.cpp -std=c++11 -pedantic -Wall -Wextra
operator1.cpp: In function int main():
operator1.cpp:6:15: error: no match for operator<< (operand types are std::ostream {aka std::basic_ostrean
     std::cout << vec << std::endl;
     ~~~~~~~~~~~~~~~~
In file included from /usr/include/c++/8/iostream:39,
                 from operator1.cpp:1:
/usr/include/c++/8/ostream:108:7: note: candidate: std::basic_ostream<_CharT, _Traits>::__ostream_type& st
       operator<<(__ostream_type& (*__pf)(__ostream_type&))
       ^~~~~~~~
/usr/include/c++/8/ostream:108:7: note:   no known conversion for argument 1 from std::vector<int> to std
/usr/include/c++/8/ostream:117:7: note: candidate: std::basic_ostream<_CharT, _Traits>::__ostream_type& st
       operator<<(__ios_type& (*__pf)(__ios_type&))
       ^~~~~~~~
/usr/include/c++/8/ostream:117:7: note:   no known conversion for argument 1 from std::vector<int> to std
/usr/include/c++/8/ostream:127:7: note: candidate: std::basic_ostream<_CharT, _Traits>::__ostream_type& st
```

Operator overloading

We can **make** it work by defining the appropriate function:

```
// operator2.cpp
#include <iostream>
#include <vector>

std::ostream& operator<<(std::ostream& os, const std::vector<int>& vec) {
    for(std::vector<int>::const_iterator it = vec.cbegin();
    it != vec.cend(); ++it) {
        os << *it << ' ';
    }
    return os;
}
int main() {
    const std::vector<int> vec = {1, 2, 3};
    std::cout << vec << std::endl; // now this will work!
    return 0;
}
```

```
$ g++ -o operator2 operator2.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./operator2
1 2 3
```

Operator overloading

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

Can write to it, cannot read from it

It is `std::cout`'s type

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

What's really happening when we see this?

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

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

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

Operator overloading

```
// operator3.cpp

1  std::ostream& operator<<(std::ostream& os, const std::vector<int>& vec) {
2      for(std::vector<int>::const_iterator it = vec.cbegin();
3      it != vec.cend(); ++it) {
4          os << *it << ' ';
5      }
6      return os;
7  }
```

It allows `std::vector<int>` to appear in a typical `std::cout <<` chain

- Taking `std::ostream& os` in the first parameter and returning `os` enables chaining
- Taking `const vector<int>&` as the second parameter allows the `std::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

```
// operator4.h
```

```
1  class Rational {
2  public:
3      //...
4      Rational operator+(const Rational& right) const;
5
6  private:
7      int num;  //numerator
8      int den;  //denominator
9  };
```

```
// operator4.cpp
```

```
1  Rational Rational::operator+(const Rational& right) const {
2      int sum_num =
3      this->num * right.den + right.num * this->den;
4      int sum_den = this->den * right.den;
5      Rational result(sum_num, sum_den);
6      return result;
7  }
```

Returning an object by value?

Q: Notice that the return type is not a reference nor a pointer. What happens when the `operator+` 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

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

- But first, how does the chaining up of << operators work?
  - `std::cout` is an `std::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:

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

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

Overloading the output operator

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

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

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

```
std::ostream& operator<<(std::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 `std::ostream` type, not `Rational` type.

Overloading the output operator

- 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.