ECE 2574: Data Structures and Algorithms - Operator Overloading

C. L. Wyatt

Today we will look at how to overload operators.

The primary use of this is to define comparisons for types in ordered containers, to define the stream extraction operator for reading/writing objects from streams, and to define *functors*.

- review the copy-assignment operator
- comparison/relational operators
- istream and ostream operators
- subscript operator
- function call operator and functors
- arithmetic operators

In C++ you can define what most operators mean for custom types (class/struct/enum).

We have already used one, copy-assignment or `operator=`.
If we write (for some type T)

```
T a,b;
b = a;
```

what really gets called is

```
T a,b;
b.operator=(a);
```

Most operators can be overloaded

- Arithmetic operators: `= + - * / ++ --` % negation
- Comparison operators/relational operators: `== != > < >= <=`
- Logical operators: `! && ||`
- Bitwise operators: `~ & | ^ << >>`
- Compound assignment operators: `+= -= *=` etc.
- Member and pointer operators: `[]`, pointer dereference, address operator, etc.
- Other: function call `()`, new, delete (and many others)

However, *most operators should not be overloaded*.

Recommendations

Only overload operators that make sense. Prefer calling free functions or method since their name helps establish the semantic meaning of code.
Example

```
Resistor r1, r2;
...
Resistor r3 = combine_series(r1, r2);
Resistor r4 = combine_parallel(r1,r2);
```

rather than

```
Resistor r1, r2;
...
Resistor r3 = r1 + r2;
Resistor r4 = r1 | r2;
```

Never redefine operators to invert thier normal meaning.

Most (but not all) operators can be defined inside (as members) or outside (as free functions).

Example: the addition operator for types `R, T, U` where `R = T + U` can be defined as

```
R T::operator+(U rhs);
```

or

```
R operator+(T lhs, U rhs);
```

Which is used when is a matter of style, but I recommend limiting member-defined operators to those that must be (e.g. `operator=`, `operator[]`) or require internal access to class members for performance reasons.

Here are what I consider the most important uses of operator overloading

- comparison/relational operators
- istream and ostream operators
- subscript operator
- function call operator and functors
- arithmetic operators (in very specific cases)

Comparison (relational) operators

`==` `!=` `>` `<` `>=` `<=` - define one, define them all.
This is trickier than it seems!
I recommend defining `<` and `==` as members and the rest as free functions.
Example: see `task_priority.cpp`

Stream (istream and ostream) operators

C++ streams reuse the bitwise shift operators for chaining writes and reads.
Example: see `task_stream.cpp` and `object_serialization.cpp`.

Subscript operator

The subscript operator is usefull for many data structures.
For example in our List interface we could define `getPosition(std::size_t position)` using `operator[]`.
See `vec_indexop.cpp`.

Function call operator

Overloading the funciton call operator allows us to define *functors*, objects that behave like functions but have state.
This is similar to closures and lambdas in other programming langauges, where functions are first-class types.
These are very usefull when combined with the standard library (`<algorithms>` in particular).
See `functor_ex.cpp`.
Note C++11 now has lambdas, but this is beyond the course scope.

Using functors to define comparisons

Consider our task sorting example above.
How could we easily switch between sorting based on task priority versus date/time?
See `task_sorting.cpp`.

Arithmetic operators

In cases where you are defining a type with a formal mathematical meaning for an operator, it is ok to overload the arithmetic operators.
Examples:

- Linear Algebra: vector-matrix and matrix-vector multiplication
- Functions (as in continuous or discrete math): f(x) + g(x), f(x) - g(x), f(g(x)), etc
- Signals: audio, images, etc

See `vec_arithop.cpp`.

Next Actions and Reminders

- Read CH pp. 425-441
- Project 4 due 11/17