Overloaded Operators, Functions, and Students

Robert Lowe

Division of Mathematics and Computer Science
Maryville College
Outline

1. Overloading Symbols
2. Overloaded Operators
3. Best Practices
A symbol is **overloaded** if it has multiple meanings. Overloading is frequently called "ad-hoc polymorphism". Also referred to as syntactic sugar. Overloading is meant to keep code close to the actual program domain. Establishes different behaviors while allowing the programmer to express intent.
Overloaded Function Example: `abs`

**In C**
```c
int abs(int j);
double fabs(double x);
```

**In C++**
```cpp
int abs(int j);
double abs(double x);
```
A Normal Function Signature Consists Of
- Name
- Number of Parameters
- Parameter Types

C++ Overloading Rules
In C++, overloaded functions share the same name. They must vary in at least one other part of the normal signature.
The Overloaded Accessor/Mutator Idiom

```cpp
std::string firstName();
void firstName(const std::string & _firstName);
```

- Both functions have the same name `firstName`
- Both functions differ by number of arguments.
- The compiler uses the function signature to determine which function to execute.
- Reusing the same name means less information for the programmer to remember!
An operator could be viewed as a function!

A binary operator takes two arguments:
\[ a + b == \text{add}(a, b) \]

A unary operator takes one argument:
\[ ++a == \text{increment}(a) \]

Like functions, operators can be overloaded.
Existing operators cannot be overridden, but you can overload types so long as they use at least one class or enumerated type.
Operator Overloading Syntax

\[ <\text{return\_type}> \operator\langle\text{symbol}\rangle(\langle\text{parameters}\rangle) \]

**Example an Overloaded \textless Operator**

```cpp
bool operator<(const PhoneRecord &lhs, const PhoneRecord &rhs)
{
    if(lhs.lastName() < rhs.lastName()) {
        return true;
    }

    return lhs.firstName() < rhs.firstName();
}
```

Robert Lowe
Overloading Arithmetic Operators

- `type operator+(const type &lhs, const type &rhs);`
- `type operator-(const type &lhs, const type &rhs);`
- `type operator*(const type &lhs, const type &rhs);`
- `type operator/(const type &lhs, const type &rhs);`
- `type operator%(const type &lhs, const type &rhs);`
Prefix Operators $++x;$
- type & operator++(type & lhs);
- type & operator--(type & lhs);

Postfix Operators $x++;$
- type operator++(type & lhs, int);
- type operator--(type & lhs, int);
Overloading Equality and Comparison Operators

- \( \text{bool } \text{operator}==\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
- \( \text{bool } \text{operator}!=\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
- \( \text{bool } \text{operator}<\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
- \( \text{bool } \text{operator}<=\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
- \( \text{bool } \text{operator}>\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
- \( \text{bool } \text{operator}>=\text{(const type } \&\text{lhs, const type } \&\text{rhs);} \)
Overloading Assignment Operators

- `type operator=(type &lhs, const type &rhs);`
- `type operator+=(type &lhs, const type &rhs);`
- `type operator-=(type &lhs, const type &rhs);`
- `type operator*=(type &lhs, const type &rhs);`
- and so on...
Often we want to create indexable containers.
Consider vector indexing: \texttt{v[0];}
This is an overloaded operator!:
\begin{verbatim}
  type & operator[](type &lhs, int index);
\end{verbatim}
Overloaded Stream Operators

Insertion Operator

\[ \text{ostream} \ & \ \text{operator}<<\ (\text{ostream} \ & \ \text{os}, \ \text{const \ type} \ & \ \text{rhs}) \];

Extraction Operator

\[ \text{istream} \ & \ \text{istream}>>\ (\text{istream} \ & \ \text{is}, \ \text{const \ type} \ & \ \text{rhs}) \];
An overloaded operator can be a member method!

When an overloaded operator is a member method, the left hand side is the current object.

Only the right hand side is specified in the parameter list.

**Example:** The PhoneRecord’s index operator.

```cpp
PhoneNumber & operator[](int index);
```
Don’t Mislead Users of Your Classes!

- Overloaded operators are meant to make code match its problem domain.
- Your overloaded operators should match the original purpose of the operator.
- Avoid side effects, unless they are expected.
- If an operator doesn’t have an obvious relationship to your class’s operations, don’t overload it!
Meet User’s Expectations!

- It is a good idea to overload at least assignment operators.
- If your class is likely to have objects in a container, provide at least `==` and `<` operators.
- Insertion and Extraction Operators are sometimes expected, but don’t use them in situations where the I/O of the object is not well defined.
- If you do provide insertion and extraction operators, the insertion operator’s output should be valid input for the extraction operator.
Keep it Inside the Class.

- Unless it's impossible, operators should be member classes.
- Operators that do not change the current object should be marked `const`.
- Avoid the use of `friend` for insertion and extraction operators.
- If you find you must make a stream operator a `friend`, chances are your class API is incomplete!