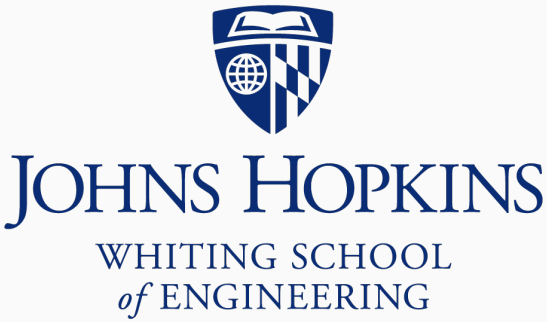
# Overloading & operator overloading

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

# Overloading & operator overloading

C++ compiler can distinguish functions with same name but different parameters

```
#include <iostream>

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

void output_type(int)   { cout << "int" << endl; }
void output_type(float) { cout << "float" << endl; }

int main() {
    output_type(1);   // int argument
    output_type(1.f); // float argument
    return 0;
}
```

# Overloading

```
$ g++ -c print_type.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o print_type print_type.o
$ ./print_type
int
float
```

But it *cannot* distinguish functions with same name & parameters but different return types

# Overloading

```
#include <iostream>

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

int   get_one() { return 1; }
float get_one() { return 1.0f; }

int main() {
    int i = get_one();
    float f = get_one();
    cout << i << ' ' << f << endl;
    return 0;
}
```

# Overloading

```
$ g++ -c print_type.cpp -std=c++11 -pedantic -Wall -Wextra
print_type.cpp: In function 'float get_one()':
print_type.cpp:6:7: error: ambiguating new declaration of 'float get_one()'
 float get_one() { return 1.0f; }
       ^~~~~~~
print_type.cpp:5:7: note: old declaration 'int get_one()'
 int   get_one() { return 1; }
       ^~~~~~~
```

# Operator overloading

Operators like `+` 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

classes override member functions to customize their behavior

*Operator overloading* is when we do something similar for *operators*

We've been using it:

```
string msg("Hello"), name;
cin >> name;                    // >> is an operator
cout << msg << ", ";            // << is an operator
cout << (name + '!') << endl; // << and + are operators
```

# Operator overloading

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

```
#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::basic_ostream<char>}' and 'std::vector<int>')
     cout << vec << endl;
     ~~~~~^~~~~~
In file included from /usr/include/c++/7/iostream:39:0,
                 from insertion_eg1.cpp:1:
/usr/include/c++/7/ostream:108:7: note: candidate: std::basic_ostream<_CharT,
_Traits>::__ostream_type& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_ostream<_CharT, _Traits>::__ostream_type&
(*)(std::basic_ostream<_CharT, _Traits>::__ostream_type&)) [with _CharT = char;
_Traits = std::char_traits<char>; std::basic_ostream<_CharT,
_Traits>::__ostream_type = std::basic_ostream<char>]
       operator<<(__ostream_type& (*__pf)(__ostream_type&))
       ^~~~~~~~
/usr/include/c++/7/ostream:108:7: note:   no known conversion for argument 1
from 'std::vector<int>' to 'std::basic_ostream<char>::__ostream_type&
(*)(std::basic_ostream<char>::__ostream_type&) {aka std::basic_ostream<char>&
(*)(std::basic_ostream<char>&)}'
/usr/include/c++/7/ostream:117:7: note: candidate: std::basic_ostream<_CharT,
_Traits>::__ostream_type& std::basic_ostream<_CharT,
_Traits>::operator<<(std::basic_ostream<_CharT, _Traits>::__ios_type&
(*)(std::basic_ostream<_CharT, _Traits>::__ios_type&)) [with _CharT = char;
_Traits = std::char_traits<char>; std::basic_ostream<_CharT,
_Traits>::__ostream_type = std::basic_ostream<char>; std::basic_ostream<_CharT,
_Traits>::__ios_type = std::basic_ios<char>]
       operator<<(__ios_type& (*__pf)(__ios_type&))
       ^~~~~~~~
```

# Operator overloading

We can make it work by defining the appropriate operator overload:

```
#include <iostream>
#include <vector>

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

std::ostream& operator<<(std::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 `std::ostream& os`
- `const std::ostream&` won't work, since it disallows writing

# Operator overloading

When we see this:

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

This is what really happens:

```
operator<<(
    operator<<(
        operator<<(
            operator<<(cout, "Hello"), 1), ' '), 2)
```

- First, we `cout << "Hello"`
- That *returns* `cout`, which becomes the new left operand
- Then we `cout << 1`, returning `cout` as new left operand
- Then we `cout << ' '`, returning `cout`
- ...

# Operator overloading

```
std::ostream& operator<<(std::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 `std::ostream& os` in 1st parameter & returning os enables chaining
- Taking `const vector<int>&` as 2nd parameter allows `vector<int>` to appear as ight operand in `operator<<` call
- Passing by const reference avoids needless copying

# Operator overloading

Can also overload operator with a member function

```
class Complex {
public:
    Complex(double r, double i) : real(r), imaginary(i) { }

    Complex operator+(const Complex& rhs) const {
        // left operand is the current object ("myself")
        // right operand is rhs
        // return value is a new Complex
        Complex result(real + rhs.real, imaginary + rhs.imaginary);
        return result; // Note: return type can't be `Complex&`
    }

private:
    double real, imaginary;
};
```

# Operator overloading

Say we want to be able to print a Complex with cout <<

```
class Complex {
public:
    ...
private:
    double real, imaginary;
};
```

Would this work?

```
std::ostream& operator<<(std::ostream& os, const Complex& d) {
    os << d.real << " + " << d.imaginary + "i";
    return os;
}
```

# Operator overloading

No, because `real` & `imaginary` are private

Can we make `operator<<` be a member function for `Complex`?

No; `operator<<` takes `std::ostream` as left operand, but a member function has to take the *object itself* as left operand

# Operator overloading

Can't be a member, but also needs access to private members??

Solution: a `friend` function:

```
class Complex {
public:
    ...

    friend std::ostream& operator<<(std::ostream& os, Complex c);

private:
    double real, imaginary;
};

std::ostream& operator<<(std::ostream& os, Complex c) {
    os << c.real << " + " << c.imaginary << 'i';
    return os;
}
```

# Operator overloading

`operator<<` is a separate, non-member function

It is also a `friend` of `Complex`, as declared here:

```
class Complex {
public:
    ...
    friend std::ostream& operator<<(std::ostream& os, Complex c);
    ...
};
```

A `friend` of a class can access its private members

- Like a “backstage pass”

# Operator overloading

| Access modifier | Any function | Derived-class members | Same-class members **& `friends`** |
|---|---|---|---|
| `public` | Yes | Yes | Yes |
| `protected` | No | Yes | Yes |
| `private` | No | No | Yes |

Often, `friend` is to be avoided, and signals bad design

A common exception is when `operator<<` must access private fields

# Operator overloading

When using a member function to overload an operator, sometimes the return value is "myself"

E.g. consider the += compound operator

```
int c = 0;
c += (c += 2);
```

Recall that the result of an assignment (including compound assignment) is the value assigned

Here, `(c += 2)` both sets c to 2 and evaluates to 2

Then outer `c += (...)` sets c to 4

# Operator overloading

Now for `Complex`:

```
Complex c(0.0, 0.0);
c += (c += Complex(2.0, 2.0));
```

# Operator overloading

Can return "myself" by returning `*this` from `operator+=` as below

Member functions have implicit pointer argument `this`: a pointer to "myself," the current object

```
class Complex {
public:
    ...
    Complex operator+=(const Complex& rhs) {
        real += rhs.real;
        imaginary += rhs.imaginary;
        return *this;
    }
    ...
};
```

# Operator overloading

Operators we can overload:

```
+   -   *   /   %   ^   &   |   ~
!   ->  =   <   >   <=  >=  ++  --
<<  >>  ==  !=  &&  ||  +=  -=  /=
%=  ^=  &=  |=  *=  <<= >>= []  ()
```

These are common:

- `+`, `-`, `+=`, `-=` (e.g. `Complex`)
- `*`, `->` (e.g. iterators)
- `[ ]` (e.g. `vector`)
- `=` (most things that can be copied)
- `==`, `<`, `>`, `<=`, `>=` (things that can be compared)

# Operator overloading

Overloading allowed only when one or both operands are a user-defined type, like a `class`

This won't work (thank goodness):

```
int operator+(int a, int b) {
    return a - b;
}
```

Can't change operator's precedence, associativity, or # operands

# Operator overloading

To allow `Date` to work with `std::sort`, we only have to add `operator<`:

```
class Date {
public:
    ...
    bool operator<(const Date& rhs) const {
        if(year < rhs.year) return true;
        if(year > rhs.year) return false;
        if(month < rhs.month) return true;
        if(month > rhs.month) return false;
        return day < rhs.day;
    }
    ...
};
```