Operator overloading (1 of 9)

Let us assume a class to handle complex numbers exist, and create a few variables (class instances, objects):

```
complex z, u, v;
```

Of course we would like to be able to write expressions using our complex numbers:

```
z = 5 * u + v * u + 4.0;
```

But what do the compiler make out of this?

```
temporary#1 = 5 * u
temporary#2 = v * u
temporary#3 = temporary#1 + temporary#2
temporary#4 = temporary#3 + 4.0
z = temporary#4
```

It will now look for a specific function matching each expression. The name must start with `operator` followed by the operator sign. For example `operator*` in the first case above. The data-type of the two values in the multiplication and the data-type of the parameters decides which functions that match.

Operator overloading (2 of 9)

This is the rules for an operator named OP:

```
            left-hand-side OP right-hand-side
member-func.   this-poiner    parameter#1
free-function  parameter#1    parameter#2

function name (member or free): operatorOP
```

Let us look at the first expression:

```
temporary#1 = 5 * u
```

The compiler search for:

```
typeA int::operator*(complex)
typeA operator*(int, complex)
```

The return value of the matching function is (only) important for further calculation.

We prefer implementing the operators as member functions in the data-type of the left-hand side, but in this case it is integer, which is built-in, and not possible to create member function for. But we can create a free function and make it a friend of out class complex.

Operator overloading (3 of 9)

Second expression:

```
temporary#2 = v * u
```

The compiler search for a matching function:

```
typeB complex::operator*(complex)
typeB operator*(complex, complex)
```

Third expression:

```
temporary#3 = temporary#1 + temporary#2
```

Possible functions to match the third expression:

```
typeC typeA::operator+(typeB)
typeC operator+(typeA, typeB)
```

If we implement a member function:

```
complex complex::operator+(complex)
```

It will match both situations with both typeA and typeB becoming the type complex.

Equivalent function in C-like syntax:

```
complex complex_add(complex*, complex)
```

Operator overloading (4 of 9)

Fourth expression:

```
temporary#4 = temporary#3 + 4.0
```

Possible functions to match the fourth expression:

```
typeD typeC::operator+(double)
typeD operator+(typeC, double)
```

In addition compiler will look for possible conversions:

```
typeX = typeX(double)
typeD typeC::operator+(typeX)
typeD operator+(typeC, typeX)
```

We can now implement a member function:

```
complex complex::operator+(double)
```

Or implement a constructor to create complex from double:

```
complex::complex(double)
```

And then reuse the previous type-complex addition:

```
complex complex::operator+(complex)
```

Operator overloading (5 of 9)

Finally it will look for fifth expression:

```
z = temporary#4
```

This requires a way to convert the type of temporart#4 (typeD) (right hand side) to complex (left hand side).

We can do it with constructor:

```
complex::complex(typeD)
```

Or with assignment operator:

```
complex& complex::operator=(typeD)
```

Or if typeD is already complex we do not have to do anything.

Operator overloading (6 of 9)

Thus we need to implement:

```
friend complex operator*(int, complex const&)

complex complex::operator*(complex const&) const

complex complex::operator+(complex const&) const

complex::complex(double)
```

Since the original expression used *, + and = this should not be a very big surprise. Although we needed slightly more that what you might have first expected.

Pay attention to the proper use of const and reference.

Also note that the free function (not class member) must be declared as friend in the class to get access to private members.

Will the above operators support this expression?

```
z = u * 5;
```

What does this do?

```
explicit complex::complex(double);
```

Operator overloading (7 of 9)

```
class array
{
public:
  array(int s); // constructor
  ~array();     // destructor

  float& index(int i); // member function

private:
  int size;
  float* data;
};

int main()
{
  array v(10); // constructor called

  v.index(0) = 0;
  v.index(1) = 1;
  for (int i = 2; i < 10; ++i)
  {
    v.index(i) = v.index(i-1) + v.index(i-2);
  }

  for (int i = 0; i < 10; ++i)
  {
    cout << "f(" << i << ") = "
         << v.index(i) << endl;
  }

  return 0; // destructor called
}
```

Operator overloading (8 of 9)

But how can we support applications like this?

```
int main()
{
  array v(10);

  v[0] = 0;
  v[1] = 1;
  for (int i = 2; i < 10; ++i)
  {
    v[i] = v[i - 1] + v[i - 2];
  }

  cout << v << endl;

  return 0;
}
```

We need operators for: [] and <<

Operator overloading (9 of 9)

```
class array
{
public:
  array(int s);
  ~array();

  double& operator[](int i);

  friend ostream& operator<<(ostream& os,
                             array const& a);
  double& index(int i);

private:
  int size;
  double* data;
};


// index operator
double& array::operator[](int i)
{
  return index(i);
}


// output operator (as free friend function)
ostream& operator<<(ostream& os, array const& a)
{
  os << setw(6) << "index"
     << setw(6) << "data" << endl;
  for (int i = 0; i < a.size; ++i)
  {
    os << setw(6) << i << setw(6) << a.data[i] << endl;
  }
  return os;
}
```

Copy Constructor / Assignment (1 of 3)

But how can we support applications like this?

```
int main()
{
  array v(10); // constructor called

  v[0] = 0;
  v[1] = 1;
  for (int i = 2; i < 10; ++i)
  {
    v[i] = v[i - 1] + v[i - 2];
  }

  array v2 = v; // copy entire array
  for (int i = 0; i < 10; ++i)
  {
    v2[i] *= v2[i]; // square
  }

  cout << "     i   fib  fib*fib" << endl;
  for (int i = 0; i < 10; ++i)
  {
    cout << setw(6) << i
         << setw(6) << v[i]
         << setw(6) << v2[i]
         << endl;
  }

  return 0; // destructor called
}
```

Copy Constructor / Assignment (2 of 3)

We can add a copy constructor and assignment operator. (If you need one you will always need the other.)

The copy constructor initiates a new instance of the class from an existing.

The assignment operator "overwrite" the content of one instance with the content of an other. (Making sure any pointers with allocated memory is handled properly.)

```
class array
{
public:
  array(int s);
  ~array();

  array(array const& a);
  array& operator=(array const& a);

  double& operator[](int i);

  friend ostream& operator<<(ostream& os,
                             array const& a);
  double& index(int i);

private:
  int size;
  double* data;
};
```

Copy Constructor / Assignment (3 of 3)

```
// copy constructor
array::array(array const& a)
{
  data = new double[a.size];
  size = a.size;

  for (int i = 0; i < size; ++i)
  {
    data[i] = a.data[i];
  }
}


// assignment operator
array& array::operator=(array const& a)
{
  if (this != &a)
  {
    array copy(a); // copy a

    // swap
    double* save = data;
    data = copy.data;
    copy.data = save;

    size = a.size;
  }
  return *this;
}
```