Operator overloading
Conversions
friend
inline

- Version 1: Dr. Ofir Pele
- Version 2: Dr. Erel Segal-Halevi

Operator Overloading

- **Operators** like +, - , * , are actually **methods**,
- and can be overloaded.

- **Syntactic sugar.**

What is it good for - 1

- Natural usage.

- compare:
  - **a.set(add(b,c))**
    - to
  - **a= b+c**

- compare:
  - **v.elementAt(i)= 3**
    - to

  v[i]= 3

What is it good for - 2

Uniformity with base types (important for templates)

```
template<typename T>
const T& min(const T& a, const T& b) {
    return a<b ? a : b;
}
```

a and b can be primitives or

user defined objects that have operator <

Complex example

Rules

1. **Don't** overload operators with **non-standard** behavior! (<< for adding,...)

2. Check how operators work on **primitives** or in the **standard library** and give the **same behavior** in your class.

Example of usage in **primitives**/**standard library**

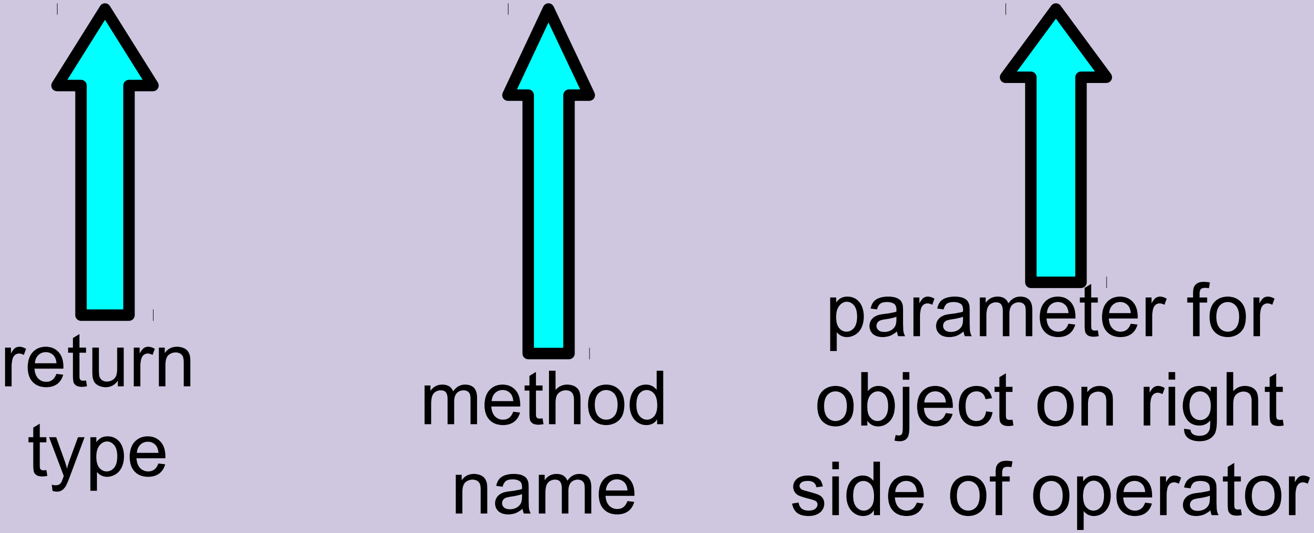
- >> << are used as bit operations for **primitives numbers** and for I/O in the **standard library** iostreams classes.

- [] is used as subscripting **primitives arrays** and vector class in the **standard library**.

- () is used for **function calls** and for functor objects in the **standard library**.

Prototype

```
X& operator=(const X& rval)
```



return type

method name

parameter for object on right side of operator

Invoking an Overloaded Operator

Operator can be invoked as a member function:

```
object1.operator=(object2);
```

It can also be used in more conventional manner:

```
object1= object2;
```

Rule of Three

- A rule of thumb:
  - When you need to make a deep copy of an object, you need to define all of these:
    1. Copy constructor
    2. Destructor
    3. Operator =
  - Or in other words:

    when you need one, you need all.

A skeleton for deep copy

```
// Copy constructor
A (const A& other) : init {
     copy_other(other);
}
```

```
// Destructor
~A() {
     clear();
}
```

```
// Operator =
A& operator=(const A& other) {
     if (this!=&other) { // preventing problems in a=a
       clear(); init // or recycle
       copy_other(other);
     } return *this; } // allows a= b= c= …
```

IntBuffer example

Operators ++ -- postfix prefix

```
// Prefix: ++n
HNum& operator++() {
 code that adds one to this HNum
 return *this; // return ref to curr
}

// Postfix : n++
const HNum operator++(int) {
 Hnum cpy(*this); // calling copy ctor
 code that adds one to this HNum
 return cpy;
}
```

A flag that makes it postfix

Operators ++ -- postfix prefix

```
// Prefix: ++n
HNum& operator++() {
 code that adds one to this HNum
 return *this; // return ref to curr
}
```

A flag that makes it postfix

```
// Postfix : n++
const HNum operator++(int) {
 Hnum cpy(*this); // calling copy ctor
 code that adds one to this HNum
 return cpy;
}
// For HNum, it might be a good idea not t
```

Conversions of types is done in two cases:

1. Explicit casting (we'll learn more about it in next lessons)

Conversions of types is done in two cases:

1. Explicit casting (we'll learn more about it in next lessons)
2. When a function gets **X** type while it was expecting to get **Y** type, and there is a casting from **X** to **Y**:

```
void foo(Y y)
...
X x;
foo(x); // a conversion from X to Y is done
```

Conversion example (conv.cpp)

Conversions danger: unexpected behavior

```
Buffer(size_t length) // ctor
…
void foo(const Buffer& v) // function
...
foo(3); // Equivalent to: foo(Buffer(3))
// Did the user really wanted this?
```

The Buffer and the size_t objects are not logically the same objects!

Conversion example (conv_explicit.cpp)

User defined conversion

```
class Fraction {
    ...
    // double --> Fraction conversion
    Fraction (const double& d) {
        ...
    }
    ...
    // Fraction --> double conversion
    operator double() const {
        ...
    }
```

friend

friend functions

Friend function in a class:

- Not a method of the class
- Have access to the class’s private and protected data members
- Defined inside the class scope

Used properly does not break encapsulation

friend functions example: Complex revisited

friend classes

- A class can allow other classes to access its private data members
- *QUESTION: Is the friendship link one-sided or two-sided? I.e:*
    - *Suppose class A is a friend of class B.*
    - *Does it mean that class B is a friend of A?*

friend classes - example

```
class IntTree {
       …
       friend class IntTreeIterator;
};


// TreeIterator can access Tree's data members
IntTreeIterator& IntTreeIterator::operator++() {
        ...
       return *this;
}
```

Inline methods

You can hint to the compiler that a method is inline **in class** declaration (inside the { }; block of a class):

```
class Tree {
    ...
    size_t size() const{ // automatically hints on inline
        return _size;
    }
};
```

Inline methods

You can hint to the compiler that a method is inline **after class** declaration:

```
class Tree {
    ...
    size_t size() const;
    ...
};
inline size_t Tree::size() const { // still in the h file
    return _size;
}
```

Inline Constructors and Destructors

Constructors and Destructors may have hidden activities inside them since the class can contain sub-objects whose constructors and destructors must be called.

You should consider its efficiency before making them inline.