C++ Operator Overloading

- Operators are language features.
- They can be given a user defined meaning to create intuitive APIs.
- Syntactic sugar, do not misuse!

Syntax for overloaded operator *op*:

returntype operator*op*(*argumentlist*)

Available Operators

Overloadable operators:

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

Member function only:

=	()	[]	->

Non-overloadable:

sizeof	typeid	.	.*	::	?:

Everyday usages

Everybody uses overloaded operators

```
int a, b = 2;
a = &b;     // int* operator&(int)
a = b & b; // int operator&(int, int)
```

```
std::string text;
text = ”hello”; // std::string& std::string::operator=(const char*)
char& c = text[1]; // char& std::string::operator[](size_t)
```

```
itk::SomeFilter::Pointer filter = itk::SomeFilter::New();
filter->Update(); // T* itk::SmartPointer::operator->() const
```

Example I

Overloading the +-operator

```
class Vector {
public:
  Vector operator+(const Vector& vec) const;
};

class Point {
public:
  Point operator+(const Vector& vec) const;
};

Vector v, w;
Point x, y;
x + y;                  // illegal: +-operator not defined
y = x + (v + w); // okay
```

Example II

How can we write ”std::cout << myClass” ?

```
class MyClass {
public:
  std::string toString() const;
};

std::ostream& operator<<(std::ostream& os, const MyClass& mc)
{
  os << mc.toString();
  return os;
}

MyClass myClass;
std::cout << myClass << 23;
```

Rules of Thumb

- Use common sense. This is the most important!
- Maintain usual arithmetic identities: x == x+y-y
- Constructive operators should return by value.
- Constructive operators should not change their operands.
- Subscript operators generally come in pairs (const and non-const)
- Avoid overloading the comma operator: x, y
- Avoid overloading short-circuiting operators: x || y or x && y. They evaluate both operands.

Copy Assignment Operator

C++ creates a default assignment operator for your class. This might not do what you want!

```
class MyClass : public SuperClass
{
  SomeClass* some; // owning semantics

public:
  MyClass() : some(new SomeClass()) { }
  ~MyClass() { delete some; }
};

MyClass* a = new MyClass();
MyClass b;
b = *a;    // b is leaking memory
delete a; // now b.some points to
          // deleted memory
```

```
MyClass&
MyClass::operator=(const MyClass& o)
{
  if (this != &o) {
    SomeClass* tmp = 0;
    try {
      tmp = new SomeClass(*o.some);
    } catch (…) {
      delete tmp; throw;
    }

    SuperClass::operator=(o);
    delete some;
    some = tmp;
  }
  return *this;
}
```