

CS202: Programming Systems

Week 3 – Operator overloading

CS202 – What will be discussed?

- ☐ What is function overloading?
- ☐ Operator overloading in C++
- ☐ Overloading >> and <<

Overloading

- ❑ There are many different “definitions” for the same name
- ❑ In C++, overloading functions are differentiated by their signatures (i.e. number/types of arguments)
- ❑ **Note:** the return type is not considered in differentiating overloading functions.

Operator Overloading

- ❑ To define operator implementations for our new user-defined types
- ❑ For example, operators such as $+$, $-$, $*$, $/$ are already defined for built-in types
- ❑ When we have a new data type, e.g. **Fraction**, we need to define new operator implementations to work with it.

Operators can be overloaded

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

- Operator `::` or `.` or `.*` cannot be defined by users.
- Operators `sizeof`, `typeid`, `?:` cannot be overloaded.
- Operators `=`, `->`, `[]`, `()` can only be overloaded by non-static functions

Overloading guidelines

- ❑ Do what users expect for that operator.
- ❑ Define them if they make logical sense.
E.g. subtraction of dates are ok but not multiplication or division
- ❑ Provide a complete set of properly related operators: $a = a + b$ and $a += b$ have the same effect

Syntax

- ❑ Declared & defined like other methods, except that the keyword ***operator*** is used.

<returned-type> **operator**<op> (arguments)

Example:

```
bool FullName::operator==(const FullName& rhs)
{
    return ((sFirstName==rhs.sFirstName) &&
            (sSurname==rhs.sSurName));
}
```

Operators in use

```
int main()
{
    FullName s1, s2;
    if (s1 == s2) //s1.operator==(s2)
    {
        ...
    }
    ...
}
```


Exercise

- ❑ Implement a **Fraction** class with basic arithmetic operators: $+$, $-$, $*$, $/$

- ❑ Remember to handle:

```
Fraction x, y;
```

```
y = x + 5;
```

```
y = 5 + x;
```

- ❑ Implement prefix and postfix increment:
 $x++$ and $++x$. Hint: using dummy `int`

The keyword: **friend**

- ❑ With the keyword **friend**, you grant access to other functions or classes
- ❑ Friend functions give a flexibility to the class. It doesn't violate the encapsulation of the class.
- ❑ Friendship is “directional”. It means if class A considers class B as its friend, it doesn't mean that class B considers A as a friend.

Example

```
class Date
{
    public:
        ...
        friend void doSomething();
    private:
        int iDay, iMonth, iYear;
}
```

- ❑ In `doSomething()`, we can have access to private data members of the class `Date`

Friend functions

- ❑ Friend functions is called like $\mathbf{f}(\mathbf{x})$ while member functions is called $\mathbf{x}.\mathbf{f}()$
- ❑ Use member functions if you can. Only choose friend functions when you have to.
- ❑ Sometimes, friend functions are good:
 - Binary infix arithmetic operators, e.g. $+$, $-$
 - Cannot modify original class, e.g. `ostream`

Member and non-member functions

```
int main()
{
    FullName s1, s2;
    if (s1 == s2)
        // member: s1.operator==(s2)
        // or non-member: operator==(s1, s2)
    {
        ...
    }
    ...
}
```

Overloading `cin` and `cout`

- ❑ We do not have access to the *istream* or *ostream* code → cannot overload `<<` or `>>` as member functions
- ❑ They cannot be members of the user-defined class because the first parameter must be an object of that type
- ❑ Operators `<<` and `>>` must be non-members, but it needs to access to private data members → make them friend functions

Typical syntax

- The general syntax for insertion and extraction operator overloadings:

```
ostream& operator<<(ostream& out, const Fraction& x)
{
    out << x.numerator << " / " << x.denominator;
    return out;
}
```

```
istream& operator>>(istream& in, Fraction& x);
```

Exercises

- Implement insertion and extraction operators for **Fraction** and **Date** class

Final notes about Op overloading

- Subscript operators often come in pair

```
const A& operator[] (int index) const;  
A&      operator[] (int index);
```

- Maintain the usual identities for $x == y$ and $x != y$
- Prefix/Postfix operators for $++$ and $--$
 - Prefix returns a reference
 - Postfix return a copy