More Advanced Class Concepts

- Operator overloading
- Inheritance
- Templates

Operator Overloading

In most programming languages some operators are overloaded. For instance, the arithmetic operators are usually overloaded for integers and reals — different addition operations are performed in the additions `1 + 2` and `1.0 + 2.0`. The compiler uses the types of the operands to determine the correct operation.

In C++, overloading can be extended to user-defined types (classes). Thus, `a + b` can mean addition of complex numbers, addition of matrices, etc.

Rules for Overloading

- All operators can be overloaded except `::` (scope resolution), `.` (member selection), `.*` (member selection through pointer to member), `? :`, `sizeof`.
- It is not possible to invent new operator symbols or to change the priority between operators.
- At least one of the operands of an overloaded operator must be an object (*not* a pointer to an object). So you cannot redefine the meaning of operators for built-in types, e.g., `int + int`.

Operator Functions

Each usage of an overloaded operator is translated into a call of an *operator function*. We have already seen one example, `operator=`. Operator functions are “normal” functions with “strange names” like `operator=`, `operator+`, . . .

Operator functions may be members or nonmembers. Rules:

- Assignment (=), subscript ([]), call (()), and member access arrow (->) *must* be members.
- Operators that change the state of their object should be members: `+=`, `++`, . . .
- Operators that don’t change the state of any object should be nonmembers: `+`, `*`, `==`, . . .

Example, Class Integer

In the following, we will use a class `Integer` to demonstrate overloading. The class is merely a wrapper for integer values, but it could easily be extended, for example to perform modulo arithmetic. Or, add an attribute to create a `Complex` class.

```
class Integer {
public:
    Integer(int v = 0) : value(v) {}
private:
    int value;
};
```

The constructor is not explicit, so you can write `Integer a = 5` as well as `Integer a(5)`.

Input and Output

You should be able to read and write `Integer` objects just as other variables, like this:

```
Integer a;
cin >> a;  // compiled into operator>>(cin, a);
cout << a; // compiled into operator<<(cout, a);
```

- The functions `operator>>` and `operator<<` must be nonmembers (otherwise they would have to be members of the `istream`/`ostream` classes, and you cannot modify the library classes).
- The input stream should be an `istream` so input can come from a file stream or a string stream, the output stream should be an `ostream`.
- The functions must be friends of the `Integer` class, so they can access the private member `value`.

Implementing operator>>

In the file `integer.h`:

```
class Integer {
    friend istream& operator>>(istream& is, Integer& i);
    ...
};

istream& operator>>(istream& is, Integer& i);
```

In the file `integer.cc`:

```
istream& operator>>(istream& is, Integer& i) {
    is >> i.value;
    return is;
}
```

Comments on next slide.

Comments, `operator>>`

- The operator function returns a reference to the stream so operations can be chained:

```
cin >> a >> b; // operator>>(operator>>(cin, a), b)
```

- The stream parameter is passed by reference (it is changed by the function).
- The `Integer` object is passed by reference (it is changed by the function).

Implementing `operator<<`

```
class Integer {
    friend ostream& operator<<(ostream& os, const Integer& i);
    ...
};

ostream& operator<<(ostream& os, const Integer& i) {
    os << i.value;
    return os;
}
```

- The function returns a reference to the stream so operations can be chained (`cout << a << b`).
- The stream is passed by reference (it is changed by the function).
- The `Integer` object is passed by constant reference (it shouldn't be copied and not changed by the function).
- The function doesn't do any extra formatting.

Compound Assignment and Arithmetic Operators

The class `Integer` should implement the arithmetic operators (+, -, ...), and to be consistent with other types also the compound assignment operators (+=, -=, ...). The arithmetic operators don't change the state of any object and should be non-members, the compound assignment operators change the state and should be members.

Always start by implementing the compound assignment operators, then use them to implement the arithmetic operators.

Example:

```
Integer a, b, sum;
a += b;       // compiled into a.operator+=(b);
sum = a + b; // sum = operator+(a, b);
```

Implementing operator+=

```
class Integer {
public:
    Integer& operator+=(const Integer& rhs) {
        value += rhs.value;
        return *this;
    }
    ...
};
```

- Like `operator=`, the function returns a reference to the current object so operations can be chained (`a += b += c`).

Implementing operator+

```
Integer operator+(const Integer& lhs, const Integer& rhs) {
    Integer sum = lhs; // local non-constant object
    sum += rhs;        // add rhs
    return sum;        // return by value
}
```

- The function returns a new `Integer` object, *not* a reference (you cannot return a reference to a local object).
- The parameters are passed by constant reference (not copied and not changed). (Pass by value would be ok here since the objects are small, but we use constant reference for all objects to be consistent.)
- The function need not be a friend of `Integer` since it calls `operator+=`, which is a member function and has access to the private member `value`.

Increment, Prefix and Postfix

`++x`	Increment `x`, return the modified value.
`x++`	Make a copy of `x`, increment `x`, return the copy. Since you have to copy an object, `x++` is less efficient than `++x`.

```
class Integer {
public:
    Integer& operator++();   // prefix, ++x
    Integer operator++(int); // postfix, x++
    ...
};
```

- Both functions have the same name; the postfix form has a dummy parameter (just an indicator, not used in the function).
- The prefix function returns a reference to the incremented value.
- The postfix function makes a copy of the object and returns the copy.

Implementing operator++

```
Integer& Integer::operator++() {
    ++value;
    return *this;
}

Integer Integer::operator++(int) {
    Integer ret = *this;
    ++*this; // use the prefix operator to do the work
    return ret;
}
```

- There is no need to give the dummy parameter to the postfix function a name.
- We could just as well have written `++value` instead of `++*this` in the second function. But `++*this` has the same effect and is better if incrementing is more complicated.

Equality Operators

```
class Integer {
    friend bool operator==(const Integer& lhs, const Integer& rhs);
    ...
};

bool operator==(const Integer& lhs, const Integer& rhs) {
    return lhs.value == rhs.value;
}

bool operator!=(const Integer& lhs, const Integer& rhs) {
    return ! (lhs == rhs);
}
```

- Global functions since they don't change any state.
- If you implement one of these you should implement both. Implement one in terms of the other.

Subscripting

We use the class String from slide 123 for the example on subscripting.

```
class String {
public:
    ...
    char& operator[](size_t index);
    const char& operator[](size_t index) const;
private:
    char* chars;
};
```

- Note two versions, for non-const and const objects. Overloading on const is possible.

Implementing operator[]

```
char& String::operator[](size_t index) {
    return chars[index];
}

const char& String::operator[](size_t index) const {
    return chars[index];
}
```

- It is essential that the non-`const` version returns a reference, so subscripting can be used on the left-hand side of an assignment.
- Here, the `const` version could just as well return a value, since the returned value is small.