Lecture 12 Operators

#CS106L

- **Operator Overloading**
- **Canonical Forms**
- **POLA**

In C++, operators can be overloaded for user-defined classes.

```
vector<string> v{"Hello", "World"};

cout << v[0];

v[1] += "!";
```

in fact,C++ tries to call these functions.

```
vector<string> v{"Hello", "World"};

cout.operator<<(v.operator[](0));

v.operator[](1).operator+=("!");
```

```
vector<string> v{"Hello", "World"};

operator<<(cout, v.operator[](0));

operator+=v.(operator[](1)), "!");
```

When overloading operators as a member function, the left hand argument is the implicit `*this`.

The subscript operator is one that must be implemented as a member.

General rule of thumb: member vs. non-member

1. Some operators must be implemented as members (eg. `[]`, `()`, `->`, `=`) due to C++ semantics.

2. Some must be implemented as non-members (eg. `<<`,if you are writing class for rhs, not lhs).

3. If unary operator (eg. `++`), implement as member.

4. If binary operator and treats both operands equally (eg.both unchanged) implement as non-member (maybe friend). Examples: `+`, `<`, `==`.

5. If binary operator and not both equally (changes lhs),implement as member (allows easy access to lhs private members). Examples: `+=`

Friend function:If you have to implement an operator as a non-member, but need access to the private members.

Principle_of_least_astonishment:If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature.

- Design operators primarily to mimic conventional usage.

- Use non-member functions for symmetric operators.

- Always provide all out of a set of related operators.

always think about:

- parameter,function,return value ->should be const/reference?

- the whole -> member vs. non-member/friend or not?

- What is the convention for overloading that operator?

Implicit conversion are dangerous,require the conversion operator to be explicit.

Q1:The `+=` operator is a binary operator that takes a left and right operand, but the parameter only has the right operand. Where did the left operand go?

Operator overloading: `vector`, `+=`

```
vector<string>& vector<string>::operator+=(const string& element) {

        push_back(element);

        return *this;

        }
```

The `+=` operator is typically a binary operator, which means it takes two operands: a left operand and a right operand. The left operand is typically the object on which the operator is being called, and the right operand is a value that is being added to the left operand.

In the code you provided, the `+=` operator is being overloaded for a vector of strings, and it only takes one parameter, which is a constant reference to a string. The left operand, which is the vector object, is not explicitly passed as a parameter. Instead, it is implicitly passed through the use of the `this` pointer.

The `this` pointer is a pointer that is implicitly passed to member functions, and it points to the object on which the member function is being called. In the case of the `+=` operator, the `this` pointer points to the vector object that is calling the operator, and it is the left operand of the operator.

Q2:member functions and non-member functions?

In C++, member functions are functions that are associated with a class or struct , while non-member functions , also known as free functions, are functions that are not associated with a class .

Here are some key differences between member functions and non-member functions:

- Access to class members: Member functions have full access to the class's private and protected members, while non-member functions do not have direct access to these members. Non-member functions can still access the class's public members, but they need to be passed an object of the class as an argument.
- Syntax: Member functions are declared within the class definition and have a special syntax. They are invoked using the dot operator (.) or the arrow operator (->) on an object of the class, while non-member functions are invoked using the function name with the appropriate number and type of arguments.
- `this` pointer: Member functions have an implicit parameter called `this` which is a pointer to the object on which the function is called. Non-member functions don't have this parameter.
- Overloading: Only member functions of a class can be overloaded, whereas non-member functions need to have unique names.
- Friend functions: Member functions can be declared as "friend" functions, which means they have access to the class's private and protected members just like member functions.
- Constness : Member functions can be overloaded with a `const` and non-const version of the function, which allows them to be called on const and non-const objects respectively.

Q3:Why did we declare these as non-member functions?

```
vector<string> operator+(const vector<string>& lhs,const vector<string>& rhs) {

        vector<string> copy = lhs;

        copy += rhs;

        return copy;

        }
```

When the operator is defined as a non-member function, it can be called on any two objects, regardless of whether they are of the same type or not, whereas if it was defined as a member function, it could only be called on objects of the same type. A non-member function is used to indicate that the operator is not part of the class and does not have access to its private data members or any other member functions of that class.

Additionally, the non-member function can also be defined as a global function, which means it can be called without having any object of the class.

In this particular example, the function `vector<string> operator+(const vector<string>& lhs, const vector<string>& rhs)` is used to concatenate two vector of strings, and it could be called on any two vector of strings, regardless of their type.

Q4:Why are we returning a reference?Why are there two versions, one that is a const member, and one that is a non-const member?

```
string& vector<string>::operator[](size_t index) {

        return _elems[index];

}

const string& vector<string>::operator[](size_t index) const {

        return _elems[index];

}
```

- Returning a reference allows the element to be accessed directly, rather than having to make a copy of it. This can be useful if the element is large and copying it would be expensive in terms of memory and performance. Additionally, a reference can be used as an lvalue (left-hand side of an assignment), which means it can be used as an input in other expressions, allowing for more efficient and expressive code.

- The client could call the subscript for both a const and non-const vector.

Q5:friend functions?

In C++, a "friend" function is a non-member function that has been granted access to the private and protected members of a class. This is done by

declaring the function as a friend of the class using the `friend` keyword, followed by the function declaration inside the class definition.

Here is an example of a friend function:

```
class MyClass {

private:

    int value;

public:

    MyClass(int v) : value(v) {}

    friend int getValue(const MyClass& obj);

};

int getValue(const MyClass& obj) {

    return obj.value;

}
```

In this example, the `getValue` function is a friend of the `MyClass` class. It can access the private member variable `value` of an object of the class, even though it is not a member function of the class.

Friend functions can be used to implement functionality that would be impractical or impossible to implement using member functions alone, such as comparing two objects of a class for equality or implementing a swap function for a class. They can also be used to provide a more flexible interface for a class.

It's important to note that, even though a friend function can access the private and protected members of a class, it is still not considered a member of the class. It does not have access to the `this` pointer and does not have the same syntax as a member function.

Also, it's important to use friend functions sparingly and only when necessary, because they break the encapsulation principle of OOP.