Lecture #14 | Polymorphism: operator overloading

SE271 Object-oriented Programming (2017)

Prof. Min-gyu Cho

Previously in Object-Oriented Programming

- In the last lecture
  - static member variable/function
  - reference
  - copy and assignment constructor
  - `std::string`
- Before that
  - Polymorphism: virtual functions
  - Polymorphism: templates

Today’s topic

- Polymorphism: operator overloading

Operator overloading

- Operators in C++ are defined as functions, and can be overloaded as other functions
- Operators can be defined for user-defined types, too
- List of operators that can be overloaded (most operators except ::, ., .*)

```
+ - * / % ^ & | ~ ! = < > += -= *= /= %=
^= &= |= << >> >>= <<= == != <= >= && ||
++ -- , ->* -> ( )
```

How to define operators

- If we want to define `operator @`
- Binary operators: `aa @ bb`
  - `aa.operator@(bb)`
  - `operator@(aa, bb)`
- Prefix unary operators: `@aa`
  - `aa.operator@()`
  - `operator@(aa)`
- Postfix unary operators: `aa@`
  - `aa.operator@(int)`
  - `operator@(aa, int)`

Example: operator+

```
class IntList {
    ...
    IntList& operator+(const IntList& v);
};
IntList& IntList::operator+(IntList& list)
{
    // FIXME: need to throw an exception
    // when the size of two lists differ
    int min_n = min(n, list.n);
    for (int i = 0; i < min_n; i++)
        elem[i] += list.elem[i];
    return *this;
}
```

```
int main()
{
    IntList list1 {2};
    list1.set(0, 42);
    list1.set(1, 23);
    IntList list2 = list1 + list1;
    list1.display();
    list2 = list1 + list2;
    list1.display();
}
```

Example: operator++

```
class IntList {
    IntList& operator++(); // prefix
    IntList operator++(int); // postfix
};

IntList& IntList::operator++()
{
    for (int i = 0; i < n; i++)
        ++elem[i];
    return *this;
}

IntList& IntList::operator++(int)
{
    IntList old {*this};
    ++(*this);
    return old;
}
```

- A dummy `int` parameter is used to distinguish prefix and postfix of incremental (and decremental) operator
- DO NOT overload operators, which is NOT intuitive!!!

Example: operator<<

```
class IntList {
    friend ostream& operator<<(ostream& output,
                               const IntList& list);
};
ostream& operator<<(ostream& output,
                    const IntList& list)
{
    output << "[";
    for (int i = 0; i < list.n; i++)
        output << list.elem[i]
               << (i != list.n - 1 ? ", " : "");
    output << "]";
    return output; // to enable chaining
}
```

- Note on friend function
  - If you put a friend specifier in front of non-member functions, they can access private or protected members
  - Do NOT abuse this!!!

Example: using operator overloading

```
int main()
{
    IntList list1 {2};
    list1.set(0, 42);
    list1.set(1, 23);
    cout << list1 << endl;

    IntList list2 = list1 + list1;
    cout << list1 << endl;;
    cout << list2 << endl;;
    cout << list1++ << endl;;
    cout << ++list1 << endl;;
}
```

```
[42, 23]
[42, 23]
[84, 46]
[42, 23]
[44, 25]
```

Notes on parameters of operator overloading

- Call-by-value v.s. call-by-reference
  - Either call-by-value or call-by-reference can be used as arguments for some operators
  - Typically, call-by-value is used for small object, and call-by-reference for large object

```
class Vector {
    Vector operator+(Vector& v); // call by reference
    Vector operator+(Vector v);  // call by value
};
```

- Specifying parameters as `const` when applicable
  - This allows functions can take non-lvalue parameter

Providing commutative operations?

```
class complex {
    double re, im;
public:
    complex(double r = 0., double i = 0.) : re(r), im(i) {}
    double real() { return re; }
    double imag() { return im; }
};
complex operator+(complex a, complex b) {
    complex tmp {a.real() + b.real(), a.imag() + b.imag() };
    return tmp;
}
complex operator+(complex a, double b) {
    complex tmp {b, 0.};
    return a + tmp;
}
complex operator+(double a, complex b) {
    complex tmp {a, 0.};
    return b + tmp;
}
```

Revisit: implicit type conversion in C/C++

- Promotion: implicit type conversion that preserve value
- `char`, `signed char`, `unsigned char`, `short int`, `unsigned short int`
  → `int` if `int` can represent all the values of the source type
  → `unsigned int` otherwise
- `char16_t`, `char32_t`, `wchar_t` or a plain enumeration type
  → `int`, `unsigned int`, `long`, `unsigned long` or `unsigned long long`
  (the first type that can represent all the avlues fo its underlying type),
- A bit-filed → `int`, `unsigned int` or no promotion
- `bool` → `int`; `true` becomes `1` and `false` becomes `0`

Exploiting implicit type conversion for less code

- What if we have only one function? `complex operator+(complex a, complex b)`

```
int main()
{
    complex x {1.2, 1.3};
    complex y = x + 2.0;
    complex z = 2.0 + x;

    std::cout << y.real() << " " << y.imag() << std::endl;
    std::cout << z.real() << " " << z.imag() << std::endl;
}
```

Other special operators

- Function call: *`expression(expression-list)`*
  - *`expression`*: left argument
  - *`expression-list`*: right argument
  - An argument list for an `operator()()` is evaluated and checked according to the usual argument-passing rules
  - Might be discussed later
- Dereferencing: `->`
- Allocation/deallocation: `new, new[], delete, delete[]`
- User-defined literal: `""`
  - You can define literals like `42.195km`, `60s`, `1.2i`

Reading list

- Learn C++
  - Operator overloading: Ch. 9.1-4,7-8

ANY QUESTIONS?