Bahasa C++:
Contoh Operator Overloading

IF2210 – Semester II 2022/2023

Tim Pengajar IF2210

Copy assignment

The assignment operator (operator=) has special properties: see copy assignment and move assignment for details.
The canonical copy-assignment operator is expected to perform no action on self-assignment, and to return the lhs by reference:

```
// assume the object holds reusable storage,
// such as a heap-allocated buffer mArray
T& operator=(const T& other) { // copy assignment

    if (this != &other) { // self-assignment check expected
        if (other.size != size) {         // storage cannot be reused
            delete[] mArray;              // destroy storage in this
            size = 0;
            mArray = nullptr; // preserve invariants in case next line throws
            mArray = new int[other.size]; // create storage in this
            size = other.size;
        }
        std::copy(other.mArray, other.mArray + other.size, mArray);
    }
    return *this;
}
```

Move assignment

The canonical move assignment is expected to leave the moved-from object in valid state (that is, a state with class invariants intact), and either do nothing or at least leave the object in a valid state on self-assignment, and return the lhs by reference to non-const, and be noexcept:

```
T& operator=(T&& other) noexcept { // move assignment

    if(this != &other) { // no-op on self-move-assignment
          //(delete[]/size=0 also ok)

        delete[] mArray; // delete this storage
        mArray = std::exchange(other.mArray, nullptr);
     // leave moved-from in valid state
        size = std::exchange(other.size, 0);
    }
    return *this;
}
```

Copy-and-swap assignment

In those situations where copy assignment cannot benefit from resource reuse (it does not manage a heap-allocated array and does not have a (possibly transitive) member that does, such as a member std::vector or std::string), there is a popular convenient shorthand: the copy-and-swap assignment operator, which takes its parameter by value (thus working as both copy- and move-assignment depending on the value category of the argument), swaps with the parameter, and lets the destructor clean it up.

```
T& T::operator=(T arg) noexcept {
// copy/move constructor is called to construct arg

    swap(arg); // resources are exchanged
               // between *this and arg
    return *this;
} // destructor of arg is called to release the resources
  // formerly held by *this
```

This form automatically provides strong exception guarantee but prohibits resource reuse.

Contoh-contoh dari TutorialsPoint

https://www.tutorialspoint.com/cplusplus/binary_operators_overloading.htm

Perkalian pecahan

```
#include <iostream>

class Fraction {
   int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
   int n, d;
  public:
   Fraction(int n, int d = 1): n(n/gcd(n, d)), d(d/gcd(n, d)) {}
   int num() const { return n; }
   int den() const { return d; }
   Fraction& operator*=(const Fraction& rhs) {
      int new_n = n * rhs.n/gcd(n * rhs.n, d * rhs.d);
      d = d * rhs.d/gcd(n * rhs.n, d * rhs.d);
      n = new_n;
      return *this;
   }
};
```

```
std::ostream& operator<<(std::ostream& out, const Fraction& f) {
  return out << f.num() << '/' << f.den() ;
}

bool operator==(const Fraction& lhs, const Fraction& rhs) {
   return lhs.num() == rhs.num() && lhs.den() == rhs.den();
}

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

Fraction operator*(Fraction lhs, const Fraction& rhs) {
   return lhs *= rhs;
}
```

```
int main() {
   Fraction f1(3, 8), f2(1, 2), f3(10, 2);
   std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n'
             << f2 << " * " << f3 << " = " << f2 * f3 << '\n'
             <<  2 << " * " << f1 << " = " <<  2 * f1 << '\n';
}
```

Contoh lain

```
#include <iostream>
using namespace std;
class A {
  public:
    A();
    A(int nn);
    A(const A& a);
    ~A();
    A& operator=(const A& a);
    A operator+(const A& a);
    friend A operator-(const A& a1, const A& a2);
    friend ostream& operator<<(ostream& os,  const A& a);
  private:
    int n;
};
```

```
A::A() { // ctor
   cout << "A::ctor 0" << endl;
   n = 0;
}

A::A(int nn) { //ctor dengan param
   cout << "A::ctor 1" << endl;
   n = nn;
}

A::A(const A& a) { //cctor
   cout << "A::cctor" << endl;
   n = a.n;
}

A::~A() { //dtor
   cout << "A::dtor" << endl;
}
```

```
A& A::operator=(const A& a) {
   cout << "A::opr =" << endl;
   n = a.n;
   return *this;
}

A A::operator+(const A& a) { //operator+ sebagai anggota kelas
   cout << "A::opr +" << endl;
   A t;
   t.n = n + a.n;
   return t;
}

A operator-(const A& a1, const A& a2) { //operator- bukan anggota kelas
   cout << "A::opr –" << endl;
   A t;
   t.n = a1.n - a2.n;
   return t;
}

ostream& operator<<(ostream& os, const A& a) {
   os << "n:" << a.n;
   return os;
}
```

Binary operator overloading example

```
#include <iostream>
using namespace std;
class Box {
  public:
    Box(double len, double bre, double hei): length(len),
                                              breadth(bre),
                                              height(hei) {}
    double volume() { return length * breadth * height; }
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
      Box box(this->length + b.length,
            this->breadth + b.breadth,
            this->height + b.height);
      return box;
    }
  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

```
// Main function for the program
int main()
{
  double volume = 0.0;    // Store the volume of a box here

  // box 1 specification
  Box Box1(6.0,7.0,5.0);

  // box 2 specification
  Box Box2(12.0,13.0,10.0);

  // volume of box 1
  volume = Box1.volume();
  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2
  volume = Box2.volume();
  cout << "Volume of Box2 : " << volume <<endl;

  // Add two object as follows:
  Box Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.volume();
  cout << "Volume of Box3 : " << volume <<endl;

  return 0;
}
```

Output :

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

```
#include <iostream>
using namespace std;
class Distance {
 private:
  int feet;    // 0 to infinite
  int inches;  // 0 to 12
 public:
  // required constructors
  Distance(int f, int i): feet(f), inches(i) {}
  Distance(): Distance(0,0) {}
  // method to display distance
  void displayDistance() {
     cout << feet << " feet " << inches << " inches" <<endl;
  }

  // overloaded minus (-) operator
  Distance operator- () {
     feet = -feet;
     inches = -inches;
     return *this;
  }
};
```

```
int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                   // apply negation
  D1.displayDistance();    // display D1

  -D2;                   // apply negation
  D2.displayDistance();    // display D2

  return 0;
}
```

Output :

```
-11 feet -10 inches
5 feet -11 inches
```