KIE1008: Programming 2

Object-Oriented Programming: Operator Overloading

Semester 1, 2025/2026

Operator Overloading

- C++ does not allow new operator to be declared, but it allows most of the existing operators to be overloaded (means perform a different task than the default one).
- Operator overloading is not automatic, programmer must write operator-overloading functions to perform the desire operations.
- Operator Overloading: C++ allows you to redefine how standard operators work when used with class objects.
  - You can change an operator's entire meaning when you overload it.
  - You cannot change the number of operands taken by an operator.

Overloaded Operators of Standard Library Class `string`

```cpp
int main() {
   string s1 = "Happy ";
   string s2 = "Birthday";
   string s3;

   cout << "Assigning s1 to s3;" << endl;
   s3 = s1; // assign s1 to s3
   cout << "s3 is \"" << s3 << "\"";

   cout << "\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   string s4(s1); // test copy constructor
   cout << "\ns4 = " << s4 << endl;
}
```

Operator Overloading

- List of operators that can be overloaded:

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

- List of operators that cannot be overloaded:

.	.*	::	?:

Overloading Arithmetic Operators

- Arithmetic operators are binary operators (i.e. take two operands).
- Syntax:

```
<return_type> operator <op_symbol>(<param_list>)
```

- An object that will store the value of the right-hand side operand of the arithmetic operator will appear in the list of arguments.

```
    Obj3 = Obj1 <op_symbol> Obj2
⇒  Obj3 = Obj1.operator <op_symbol> (Obj2)
```

<u>Example:</u>

```
   z = x + y
⇒ z = x.operator + (y)
```

Example 1: Overloading '+' Operator (1/3)

```
class Point
{
private:
   int x, y;

public:
   Point(int x = 0, int y = 0); // Constructor
   int getX() const;
   int getY() const;
   void setX(int x);
   void setY(int y);
   void print() const;
   Point operator+(const Point & rhs);
};
```

Code: W04C01

Example 1: Overloading '+' Operator (2/3)

```cpp
Point::Point(int x, int y) : x(x), y(y) { }

int Point::getX() const { return x; }
int Point::getY() const { return y; }

void Point::setX(int x) { this->x = x; }
void Point::setY(int y) { this->y = y; }

void Point::print() const {
   cout << "(" << x << "," << y << ")" << endl;
}

Point Point::operator+(const Point & rhs) {
   return Point(x + rhs.x, y + rhs.y);
}
```

Code: W04C01

Example 1: Overloading '+' Operator (3/3)

```
int main()
{
   Point p1(1, 2), p2(4, 5);
   // Use overloaded operator +
   Point p3 = p1 + p2;
   p1.print();  // (1,2)
   p2.print();  // (4,5)
   p3.print();  // (5,7)

   // Invoke via usual dot syntax, same as p1+p2
   Point p4 = p1.operator+(p2);
   p4.print();  // (5,7)

   // Chaining
   Point p5 = p1 + p2 + p3 + p4;
   p5.print();  // (15,21)
}
```

Code: W04C01

Example 2: Overloading '+' Operator (1/3)

```cpp
class Box
{
public:
   void setParam(int, int, int);
   int getVol();
   Box operator+ (const Box& b);

private:
    int length, breadth, height;
};

void Box::setParam(int x, int y, int z) {
     length = x;
     breadth = y;
     height = z;
}
```

Code: W04C02

Example 2: Overloading '+' Operator (2/3)

```
int Box::getVol()
{
    return length * breadth * height;
}

Box Box::operator+(const Box& b)
{
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
}
```

Code: W04C02

Example 2: Overloading '+' Operator (3/3)

```
int main()
{
   Box Box1, Box2, Box3;

   Box1.setParam(4, 5, 6);
   Box2.setParam(5, 6, 7);
   Box3 = Box1 + Box2;  // equivalent to Box1.operator+(Box2)

   cout << Box1.getVol() << endl;
   cout << Box2.getVol() <<endl;
   cout << Box3.getVol() <<endl;

   return 0;
}
```

Code: W04C02

Overloading << and >> Operators

```
cout << A1; ⇒ operator << (cout, A1)
```

- Since the left operand is not a class object (`cout` is an `ostream` object and `cin` is an `istream` object), we cannot use member function, but need to use non-member `friend` function for operator overloading.
- Syntax:

```
friend ostream & operator << (ostream &, const className &)
    friend istream & operator >> (istream &, className &)
```

Example 1: Overloading << and >> Operators (1/3)

```
class Point
{
private:
   int x, y;

public:
   Point(int x = 0, int y = 0); // Constructor
   void print() const;
   friend ostream &operator <<(ostream &, const Point &);
   friend istream &operator >>(istream &, Point &);
};

Point::Point(int x, int y) : x(x), y(y) { }
```

Code: W04C03

Example 1: Overloading << and >> Operators (2/3)

```
ostream &operator <<(ostream &out, const Point &point)
{
   out << "(" << point.x << "," << point.y << ")";
   return out;
}

istream & operator >>(istream &in, Point &point)
{
   cout << "Enter x and y coord: ";
   in >> point.x >> point.y;
   return in;
}
```

Code: W04C03

Example 1:
Overloading << and >> Operators (3/3)

```
int main() {
   Point p1(1, 2), p2;

   // Using overloaded operator <<
   cout << p1 << endl;     // support cascading
   operator<<(cout, p1);   // same as cout << p1
   cout << endl;

   // Using overloaded operator >>
   cin >> p1;
   cout << p1 << endl;
   operator>>(cin, p1);   // same as cin >> p1
   cout << p1 << endl;
   cin >> p1 >> p2;        // support cascading
   cout << "p1 = " << p1 <<  "; p2 = " << p2 << endl;
}
```

Code: W04C03

Example 2: Overloading << and >> Operators (1/3)

```
class Box
{
public:
  void setParam(int, int, int);
  friend ostream& operator<< (ostream& , const Box&);
  friend istream& operator>> (istream&, Box&);

private:
   int length, breadth, height;
};

void Box::setParam(int x, int y, int z)
{
    length = x;
    breadth = y;
    height = z;
}
```

Code: W04C04

Example 2:
Overloading << and >> Operators (2/3)

```
ostream& operator<< (ostream& os, const Box& b)
{
   os << "length = " << b.length << "; breath = " << b.breadth <<
   "; height = " << b.height << endl;
   return os;
}

istream& operator>> (istream& is, Box& b)
{
   cout << "Enter length, breath, height: ";
   is >> b.length;
   is >> b.breadth;
   is >> b.height;
   return is;
}
```

Code: W04C04

Example 2: Overloading << and >> Operators (3/3)

```cpp
int main()
{
   Box Box1, Box2;

   Box1.setParam(4, 5, 6);
   Box2.setParam(5, 6, 7);
   cin >> Box1;
   cout << "Box 1: " << Box1 << endl;  // output based on Cin
   cout << "Box 2: " << Box2 << endl;  // (5, 6, 7)

   return 0;
}
```

Code: W04C04

Overloading Prefix and Postfix ++

- Overloading postfix operators (such as x++, x--) ought to be differentiated from the prefix operator (++x, --x).
- Syntax to overload the pre-increment operator ++:

```
className operator ++ ()
```

- A "dummy" argument is introduced to indicate postfix operation.

```
className operator ++ (int dummy)
```

Note: postfix ++ shall save the old value, perform the increment, and then return the saved value by value.

Example: Overloading Prefix and Postfix

```
class Counter
{
private:
   int count;
public:
   Counter(int count = 0);   // Constructor
   Counter & operator++();              // ++prefix
   const Counter operator++(int dummy); // postfix++

   friend ostream &operator<<(ostream & out, const Counter &);
};

Counter::Counter(int c) : count(c) { }
```

Code: W04C05

Example: Overloading Prefix and Postfix

```
// ++prefix, return reference of this
Counter & Counter::operator++() {
   ++count;
   return *this;
}

// postfix++, return old value by value
const Counter Counter::operator++(int dummy) {
   Counter old(*this);
   ++count;
   return old;
}

// Overload stream insertion << operator
ostream & operator<<(ostream &out, const Counter &counter) {
   out << counter.count;
   return out;
}
```

operator function is declared `const` because it does not modify the original object

Code: W04C05

Example: Overloading Prefix and Postfix

```
int main()
{
   Counter c1;
   cout << c1 << endl;      // 0
   cout << ++c1 << endl;    // 1
   cout << c1 << endl;      // 1
   cout << c1++ << endl;    // 1
   cout << c1 << endl;      // 2
   cout << ++++c1 << endl;  // 4
   cout << c1++++ << endl;  // error caused by const return value
}
```

Code: W04C05

Case Study: `Complex` Class

```
class Complex {
public:
    friend ostream& operator<< (ostream&, const Complex&);
    friend istream& operator>> (istream&, Complex&);

    Complex() : real(0.0), imag(0.0) {};
    Complex(double re, double im) : real(re), imag(im) {};

    void operator=(const Complex &); // c1 = c2
    Complex operator+(const Complex &); // c1 + c2
    Complex operator+= (const Complex &); // c1 += c2
    void operator++();  // c++
    void operator+=(double); // c += double
    bool operator== (const Complex &);  // c1 == c2

private:
    double real;
    double imag;
};
```

Code: W04C06