

# Operator Overloading

Modern cpp Programming lecture 9



## Revisit previous slides

---

- OOP features in cpp
- Why OOP?
  - simulates real world
  - easy *library* | *module* development
  - easy abstraction
  - supports GUI programming
  - **enhance code reusability**
    - how can we enhance *more*???

## Revisit previous slides

---

- Advanced features for code reusability
  - Template
  - **Operator Overloading**
  - Design Pattern

# Operator

---

- To understand operator overloading...
  - you must carefully consider *operator*
  - actually, we already learned *cpp operators*...
    - Assignment
    - Arithmetic
    - Increment / Decrement
    - Relational
    - Logical
    - Bit (Optional)

# Operator

---

- Assignment

=, +=, -=, \*=, /=

- Arithmetic

+, -, \*, /, %

- Increment / Decrement

++, --

- Relational

<, >, <=, >=, !=, ==

- Logical

&&, ||, !

- And more...

# Operator

---

- Only works for primitive types
  - int, bool, long, float...
  - Even not for *string*!!
    - string + string ....OK
    - string – string ....??
    - string \* string ....??
      - actually, string is not ***a primitive type...***

# Operator

---

- How can we deal with the problem??
  - Simple solution: don't you it!!
    - string + string sounds strange...
    - You don't need to use it
  - However, if you declare a class...
    - using primitive operators helps you a lot

# Operator

---

- Suppose you want to define complex number as *Class*
  - form:  $ai + b$
  - variable: `int a, int b`
  - function: addition, subtraction, multiplication
  - Let's define a function!!



# Operator

---

- Complex class

```
class Complex {  
public:  
    int a, b;  
  
    Complex(int a, int b) {  
        this->a = a;  
        this->b = b;  
    }  
  
    Complex add(...) { ... }  
    Complex sub(...) { ... }  
    Complex mul(...) { ... }  
};
```

- Common way to define the behavior

# Operator

---

- Complex class implementation

```
Complex add(Complex other) {  
    this->a += other.a;  
    this->b += other.b;  
}
```

```
Complex sub(Complex other) {  
    this->a -= other.a;  
    this->b -= other.b;  
}
```

```
Complex mul(Complex other) {  
    this->a = (this->a * other.a - this->b * other.b);  
    this->b = (this->a * other.b + this->b * other.a);  
}
```

- Basic math...

# Operator

- Complex number class

```
class Complex {  
public:  
    int a, b;  
  
    Complex(int a, int b) {  
        this->a = a;  
        this->b = b;  
    }  
  
    Complex add(...) { ... }  
    Complex sub(...) { ... }  
    Complex mul(...) { ... }  
};
```

```
Complex X(3, 5);  
Complex Y(4, 8);
```

```
X = X.add(Y)
```

- Good to use...but not like *real* math formulas

# Operator

---

- You might want to calculate Complex numbers in...

```
Complex X(3, 5);  
Complex Y(4, 8);  
Complex Z(1, 9);
```

```
X = (X * Y) + Z
```

- easy to read
- easy to build complicated formula
- but how?
  - *operator overloading solves the problem!!*

# Operator Overloading

- Revisit overloading vs. overriding

Method Overloading	Method Overriding
Provides functionality to reuse method name for different arguments	Provides functionality to override a behavior which the class have inherited from parent class
Occurs usually within a single class (may also occur in child/parent classes)	Occurs in two classes that have child-parent or is-a relationship
Must have different argument list (signature)	Must have the same argument list
May have different return types	Must have the same or covariant return type
May have different access modifiers	Must not have a more restrictive access modifier but may have less restrictive access modifier

# Operator Overloading

- Revisit overloading vs. overriding

Method Overloading	Method Overriding
Provides functionality <b>to reuse method name</b> for different arguments	Provides functionality to override a behavior which the class have inherited from parent class
Occurs usually within a single class (may also occur in child/parent classes)	Occurs in two classes that have child-parent or is-a relationship
Must have different argument list (signature)	Must have the same argument list
May have different return types	Must have the same or covariant return type
May have different access modifiers	Must not have a more restrictive access modifier but may have less restrictive access modifier

# Operator Overloading

---

- Overloading:
  - same function name, different **signature**
    - signature:
      - function name, argument type, argument number
  - provides functionality to *reuse* the function

```
int add(int a, int b) {  
    return a + b;  
}
```

```
int add(int a, int b, int c) {  
    return a + b + c;  
}
```

```
add(1, 2)      // 3  
add(1, 2, 3)   // 6
```

# Operator Overloading

---

- Remember!! *Operator* is ***function***!!
  - arithmetic operator \*
  - binary operator which gets two number as arguments and returns their product
  - can be represented as...

```
int mul(int a, int b) {  
    int result = 0;  
    for(int i = 0; i < b; i++)  
        result += a;  
    return result;  
}
```



# Operator Overloading

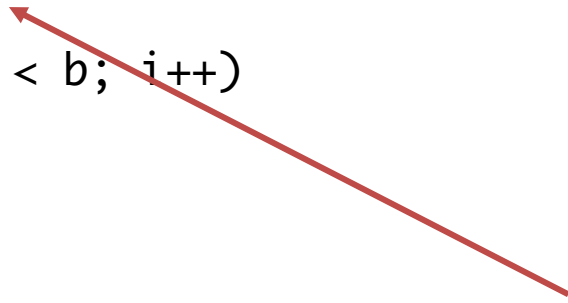
---

- Remember!! *Operator* is ***function***!!

- assignment operator `* =`

- binary operator which gets two number as arguments and assigns their product to the first element
- can be represented as...

```
int mulAssign(int& a, int b) {  
    int result = 0;  
    for(int i = 0; i < b; i++)  
        result += a;  
    a = result;  
    return;  
}
```



Why reference??  
DIY

# Operator Overloading

---

- Remember!! *Operator* is ***function***!!
  - Tricky example – Ternary operator **?:**
    - formula `formula ? a : b;`
    - returns `a` when formula returns true, else returns `b`
    - for example...

```
int b = a < 10 ? a : 10;
```

      - returns `a` when `a < 10`, else returns `b`

# Operator Overloading

---

- Remember!! *Operator* is ***function***!!
  - Tricky example – Ternary operator **?:**
    - formula `? a : b;`
    - returns `a` when formula returns true, else returns `b`
    - How can we *describe* the behavior of the operator??
      - *Argument: bool, int, int*
      - *Functionality: if bool is true, returns second element. Else returns first element*

```
int ternary(bool criteria, int first, int second) {  
    if (criteria) return first;  
    else          return second;  
}
```

# Operator Overloading

---

- Remember!! *Operator* is ***function***!!
  - If operators are functions, why can't we *overload* them??
  - ***Operator Overloading*** appears

# Operator Overloading

---

- Operator overloading rule
  - Almost every operator in cpp can be overloaded
    - Arithmetic operator `+, -, *, / , %`
    - Assignment operator `=, +=, -=, *=, /= ...`
    - Relational operator `<=, >=, <, >, ==`
    - Logical operator `||, &&, !`
    - access operator `->, .`
    - and else...
    - *Not for ternary operator!! why?*

## SubTitle

---

- Contents

## SubTitle

---

- Contents

## SubTitle

---

- Contents



## SubTitle

---

- Contents

## SubTitle

---

- Contents

## SubTitle

---

- Contents

## SubTitle

---

- Contents

## SubTitle

---

- Contents

---

Thank you!!

contact: [jeonhyun97@postech.ac.kr](mailto:jeonhyun97@postech.ac.kr)