# 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: *a*i + *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 |

reference: https://laptrinhx.com/everything-about-method-overloading-vs-method-overriding-2837559910/

# 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 |

reference: https://laptrinhx.com/everything-about-method-overloading-vs-method-overriding-2837559910/

# 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 – Tenary operator ? :
    - `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?*
      - *unnecessary…*

# Operator Overloading

- simple example

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }
};
```

- Assume that you wants to *add* or *multiply circle*

# Operator Overloading

- simple example

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }
};
```

- First we need to ***define*** *addition / multiplication* of circle

# Operator Overloading

- *Addition* of circle
  - *Circle(r1, x1, y1) + Circle(r2, x2, y2) ?*
    1. *Circle(r1 + r2, x1 + x2, y1 + y2)* *// just add elementwise*
    2. *Circle(r1 + r2, x1, y1)* *// use coordinate of the first operand*
  - *Circle(r1, x1, y1) * Circle(r2, x2, y2) ?*
    1. *Circle(r1 * r2, x1 * x2, y1 * y2)* *// just multiply elementwise*
    2. *Circle(r1 * r2, x1, y1)* *// use coordinate of the first operand*

# Operation Overloading

- Using def 1. (elementwise addition)

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }

    Circle operator+(const Circle& c) {
        Circle newCircle(this->r + c.r, this->x + c.x, this->y + c.y);
        return newCircle;
    }

    void printInfo() {
        cout << "radius: " << this->r << endl;
        cout << "x:      " << this->x << endl;
        cout << "y:      " << this->y << endl;

    }
};

Circle c1(5, 3, 4);
Circle c2(7, 3, 5);
Circle addCircle = c1 + c2;
addCircle.printInfo();
```

output:

```
radius: 12
x:      6
y:      9
```

# Operation Overloading

- Using def 1. (elementwise addition)

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }

    Circle operator*(const Circle& c) {
        Circle newCircle(this->r * c.r, this->x * c.x, this->y * c.y);
        return newCircle;
    }

    void printInfo() {
        cout << "radius: " << this->r << endl;
        cout << "x:      " << this->x << endl;
        cout << "y:      " << this->y << endl;

    }
};

Circle c1(5, 3, 4);
Circle c2(7, 3, 5);
Circle addCircle = c1 * c2;
addCircle.printInfo();
```

```
output:
radius: 35
x:      9
y:      20
```

# Operation Overloading

- Using def 2. (addition using first-element coordinate)

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }

    Circle operator+(const Circle& c) {
        Circle newCircle(this->r + c.r, this->x, this->y);
        return newCircle;
    }

    void printInfo() {
        cout << "radius: " << this->r << endl;
        cout << "x:      " << this->x << endl;
        cout << "y:      " << this->y << endl;

    }
};

Circle c1(5, 3, 4);
Circle c2(7, 3, 5);
Circle addCircle = c1 + c2;
addCircle.printInfo();
```

output:

```
radius: 12
x:      3
y:      4
```

# Operation Overloading

- Using def 2. (multiplication using first-element coordinate)

```
class Circle {
private:
    int r;      // radius
    int x, y;   // x, y coordinate

public:
    Circle(int r, int x, int y) {
        this->r = r;
        this->x = x;
        this->y = y;
    }

    Circle operator*(const Circle& c) {
        Circle newCircle(this->r * c.r, this->x * c.x, this->y * c.y);
        return newCircle;
    }

    void printInfo() {
        cout << "radius: " << this->r << endl;
        cout << "x:      " << this->x << endl;
        cout << "y:      " << this->y << endl;

    }
};

Circle c1(5, 3, 4);
Circle c2(7, 3, 5);
Circle addCircle = c1 * c2;
addCircle.printInfo();
```

```
output:
radius: 35
x:      3
y:      4
```

# Operator Overloading

- Code analysis

Q. Why constant?

A. Remember that this function is redefining an operator. It is not allowed to change the value of operand changes during operation. Therefore, we define operands (parameters) as constant to prevent such situation.

```
Circle operator+(const Circle& c) {
   Circle newCircle(this->r + c.r,
                    this->x + c.x,
                    this->y + c.y);
   return newCircle;
}
```

Q. Why reference &?

A. As the operator don’t change the value of operands, there is no need to *copy* its value. Therefore, it is efficient to use reference, as it only copies the *name, or nickname* which can access to the value.

# Operator Overloading

- Advantage
  - can handle classes easily
  - easy to perform primitive operation on classes
    - `circle1 + circle2` is better than…
    - `addCircle(circle1, circle2)` or
    - `circle1.add(circle2)`
  - can even *define* your own operator!!
    - we'll see the power…

# Operator Overloading

- Two ways
  - class method

```
Circle operator*(const Circle& c) {
        Circle newCircle(this->r * c.r,
                         this->x * c.x,
                         this->y * c.y);
        return newCircle;
}
```

  - global function

```
Circle operator*(const Circle& c1, const Circle& c2) {
        Circle newCircle(c1.r * c2.r,
                         c1.x * c2.x,
                         c1.y * c2.y);
        return newCircle;
}
```

# Operator Overloading

- Two ways
  - class method
  - global function
- highly recommend to define operator as *class method*
- why?
  - better encapsulation
  - easy to find definition
  - intuitively reasonable
    - overloaded operators *belong* to the class

# Operator Overloading

- Then why second way exists?? (global)
  - more flexibility...perhaps

```
string s1 = "Hello";
string s2 = "World";
string s3 = s1 + s2;

cout << s3 << endl;    // HelloWorld
```

  - Suppose that you needs to concatenate string a lot...
  - and wants to add blank b/w operands

# Operator Overloading

- Suppose that you needs to concatenate string a lot…
- and wants to add blank b/w operands
  - Solution 1: define function

```
string addStringWithBlank(string s1, string s2) {
   return s1 + “ “ + s2;
}
```

  - Solution 2: hardcoding

```
string s1 = "Hello";
string s2 = "World";
string s3 = s1 + “ “ + s2;

cout << s3 << endl;   // HelloWorld
```

# Operator Overloading

- Suppose that you needs to concatenate string a lot…
- and wants to add blank b/w operands
  - Problem (sol.1 & sol.2)
    - hard to read
    - harms the consistency
    - ugly code…
      - no one would prefer function names like `addStringWithBlank`
  - Redefining operator + will help the issue!!
    - by using ***global operator overloading!!***

# Operator Overloading

- Suppose that you needs to concatenate string a lot…
- and wants to add blank b/w operands
- Sol3. *Global Operator Overloading*

```
string operator+(const string& s1, const string& s2) {
    string result = "";
    result += s1;
    result += " ";
    result += s2;
    return result;
}

string s1 = "Hello";
string s2 = "World";
cout << s1 + s2 << endl;            // Hello World
```

# Operator Overloading

- Code analysis

```
string operator+(const string& s1, const string& s2) {
    string result = "";
    result += s1;
    result += " ";
    result += s2;
    return result;
}
```

Q. why not ” s1 + s2? ”

A. Remember that you *redefined* operator +. Therefore, if you use “s1 + s2”, this inner addition will recursively call your redefined operator +. This recursive function call will be executed infinitely, which will cause stack overflow.

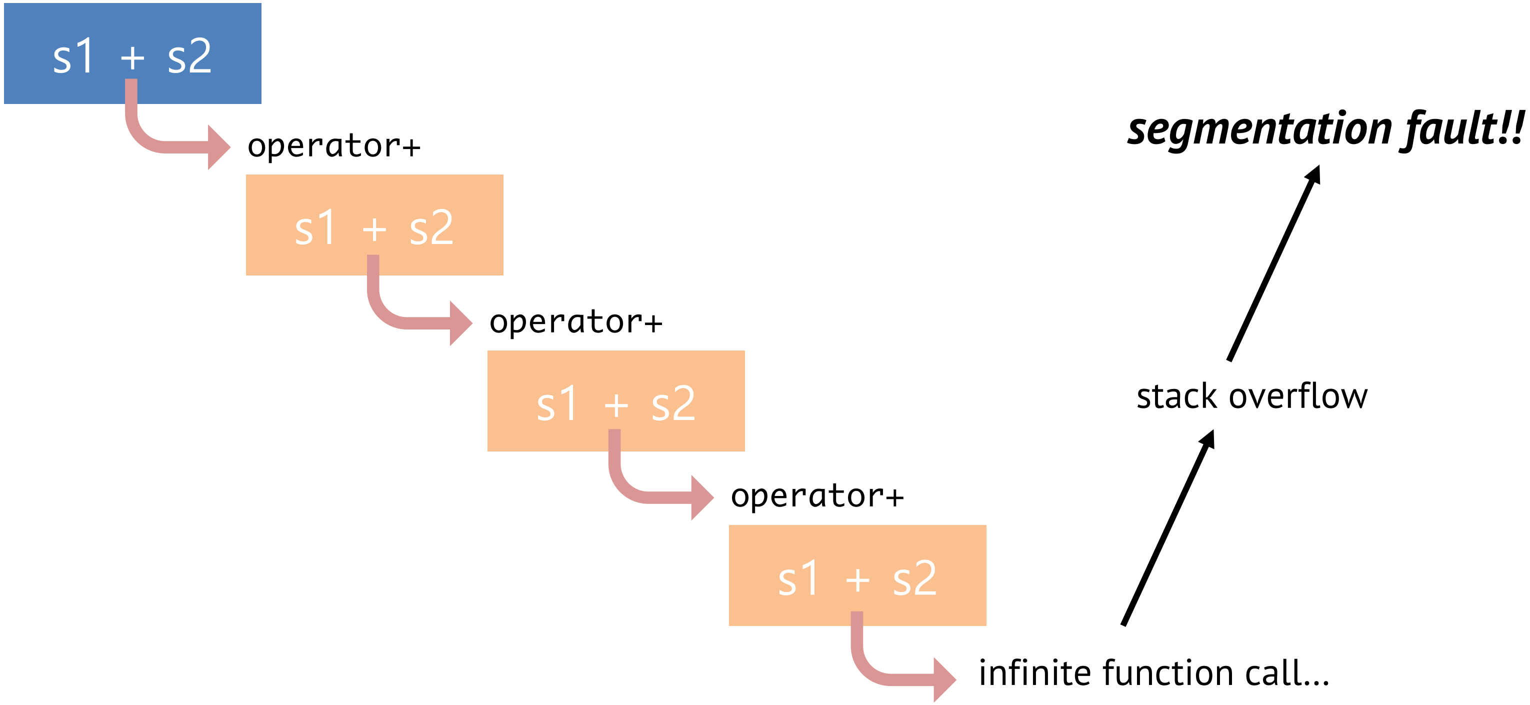
You didn’t redefine += operator, therefore free to use it

# Operator Overloading

- Code analysis

- if…

```
string operator+(const string& s1, const string& s2) {
    string result = s1 + s2;
    return result;
}
```



# Operator overloading

- Conclusion
  - great way to reuse code
  - provides the primitive operations on user-defined data types
  - supports two way
    - method
    - global function
  - also possible to redefine primitive operations

Thank you!!

contact: hj@hcil.snu.ac.kr