

Queens College, CUNY, Department of Computer Science
Object Oriented Programming in C++
CSCI 211 / 611
Summer 2018
Instructor: Dr. Sateesh Mane

© Sateesh R. Mane 2018

July 19, 2018

Operator overloading

- In this lecture we shall learn about **operator overloading**.
- We shall also learn about **operators as class methods**.

Class Point1: review for later use

- Recall the class Point1, which we have used before.

```
class Point1 {
public:
    // public methods
    void set(const double &a, const double &b)
    {
        x = a;
        y = b;
    }

    const double& getx() const { return x; }
    const double& gety() const { return y; }

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

private:
    // data
    double x, y;
};
```

1 Function overloading

- Let us write functions to add and subtract two `Point1` objects.
 1. With an obvious notation, the data values in the sum are $(x_1 + x_2, y_1 + y_2)$.
 2. Similarly, the data values in the difference are $(x_1 - x_2, y_1 - y_2)$.
 3. The C++ code for the functions `P1plus` and `P1minus` is given below.

```
Point1 P1plus(const Point1 &u, const Point1 &v) // function
{
    double xsum = u.getx() + v.getx();           // accessor methods
    double ysum = u.gety() + v.gety();
    Point1 p;
    p.set(xsum, ysum);                           // set the sum
    return p;
}

Point1 P1minus(const Point1 &u, const Point1 &v) // operator
{
    double xdiff = u.getx() - v.getx();          // accessor methods
    double ydiff = u.gety() - v.gety();
    Point1 p;
    p.set(xdiff, ydiff);                         // set the difference
    return p;
}
```

1.1 Main program

- Here is a working C++ program to instantiate `Point1` objects and call the above functions.

```
#include <iostream>
using namespace std;

class Point1 {
    // etc
};

// functions P1plus, P1minus

int main()
{
    double c1 = 2.0;
    double c2 = 3.0;
    Point1 p1, p2, p3;
    p1.set(4.4, 3.3);
    p2.set(2.2, 1.1);
    p3.set(0.4, 0.5);

    Point1 pa = P1plus(p1, p2);           // add
    Point1 pb = P1minus(p1, p2);          // subtract
    pa.print();
    pb.print();

    Point1 p123a = P1plus(P1plus(p1, p2), p3); // (p1+p2) + p3
    Point1 p123b = P1plus(p1, P1plus(p2, p3)); // p1 + (p2+p3)
    p123a.print();
    p123b.print();

    return 0;
}
```

- *The most important feature of the above program is that it is clumsy.*
- The expressions to add three objects `p1+p2+p3` are particularly ugly.

```
P1plus(P1plus(p1, p2), p3);    // (p1 + p2) + p3
P1plus(p1, P1plus(p2, p3));    // p1 + (p2 + p3)
```

- Although it works, it is a bad software design.
- *There is a better way.*

2 Operator overloading, part 1

- Note that the above functions perform the math operations $+$ and $-$.
- It would be much nicer if we could write code like this:

```
Point1 pa = p1 + p2;  
Point1 pb = p2 - p1;
```

- It would also be good to be able to multiply a `Point1` object by a `double`.

```
Point1 pc = c1 * p1;  
Point1 pd = p2 * c2;
```

- In this context, there are two different multiplication operations:
 1. `double` times `Point1`.
 2. `Point1` times `double`.
 3. To the compiler, these are distinct operations because the operands are different.
 4. The compiler does not automatically know they are equal.
- **C++ has a way to do write such operators.**
- **As well as writing functions, C++ permits us to write operators.**
- Writing code to create our own definitions of operators is called **operator overloading**.
 1. We can also overload the division operator `/`.
 2. We can overload other mathematical operators such as `++` and `--`.
 3. We can also overload Boolean operators such as `>`, `>=`, `<`, `<=`, `||` and `&&`.
 4. We can also overload the left and right shift operators `<<` and `>>`.
- **The widely used expression “`cout << (something) << endl;`” is an example of the overloading of the `<<` operator.**
- **However, we cannot overload the dot “`.`” operator.**
- There are published references online, where you can find a list of C++ operators which can and cannot be overloaded. See for example the following link.
<https://msdn.microsoft.com/en-us/library/5tk49fh2.aspx>

3 Operator overloading, part 2

- Here is the C++ code for the operators $+$, $-$ and two versions of $*$.
- **They are all examples of operator overloading.**
 1. The first two cases overload the $+$ and $-$ operators, respectively.
 2. They tell the compiler how to add and subtract two `Point1` objects.
- The third and fourth cases overload the $*$ operator in two different ways.
 1. The first overload of $*$ tells the compiler how to multiply `double` and `Point1`.
 2. The second overload of $*$ tells the compiler how to multiply `Point1` and `double`.
- ***Don't panic.***
- Before we rush too fast, in the next section(s) we shall learn step by step how to write C++ code to overload operators.

```
Point1 operator+ (const Point1 &u, const Point1 &v) // operator
{
    double xsum = u.getx() + v.getx();           // accessor methods
    double ysum = u.gety() + v.gety();
    Point1 p;
    p.set(xsum, ysum);
    return p;
}
```

```
Point1 operator- (const Point1 &u, const Point1 &v) // operator
{
    double xdiff = u.getx() - v.getx();
    double ydiff = u.gety() - v.gety();
    Point1 p;
    p.set(xdiff, ydiff);
    return p;
}
```

```
Point1 operator* (double c, const Point1 &u) // operator
{
    double x = u.getx() * c;
    double y = u.gety() * c;
    Point1 p;
    p.set(x, y);
    return p;
}
```

```
Point1 operator* (const Point1 &u, double c) // operator
{ return (c*u); }
```

4 Operator overloading, part 3

- Let us learn step by step how to write C++ code to overload operators.
- Recall the overload of the assignment operator `=`.
- Instead of a function name, we write the keyword “operator” and the symbol for the operator.
- Hence we write `operator+` for addition.
- After that we write the operator signature. This is reasonably obvious.
- The input arguments of an operator are called its **operands**.
- **However, there is an obvious difference between operands and function inputs.**
 1. Again recall the assignment operator `=`.
 2. In programming statements, the two operands are written on the left and right sides and the operator appears in the middle.
 3. Hence we write expressions of the form $p_1 + p_2$ or $p_1 - p_2$, etc.
 4. **The operator is written between the operands.**
 5. The operator `+` acts on the operands on its left and its right.
- Finally there is the return type. For the operator `+`, this is obviously a `Point1` object.
- ***Is it obvious?***
 1. The operator `+` is one of the most heavily overloaded operators.
 2. We make use of the overloading all the time without even thinking.
 3. *But what are its operands and what is its return type in each case?*

left operand	right operand	return type
int	int	int
int	double	double
double	int	double
double	double	double

4. *The C++ language must overload every other case, for `char`, `long`, etc.*
- What is the return type if we attempt to add a `double` and a `string`? This operation is not supported by the C++ language.
 - ***So ... the correct answer is that the return type is not obvious.***
 - *Can the return type be `void`?* Yes. (Try it.) But I have no idea what an operator with a `void` return type would do.

5 Operator precedence

- Consider how we would write code to evaluate the expression

$$p = p_1 + p_2 c_2 .$$

- Using the overloaded operators, the code would be written in the following way.

```
p = p1 + p2*c2;
```

- *Is this really true? Why?*
 1. Recall that `Point1` is a user-defined class.
 2. We overloaded the operators `+`, `-` and `*` to accepts operands of the class `Point1`.
 3. *But we did not say what to do if expressions involving addition and multiplication are mixed.*
 4. **The compiler implements the rules of operator precedence automatically.**
 5. *This is a truly elegant feature of operator overloading.*
- We can write clean expressions such as

```
Point1 pe = p1 + p2 + p3;  
Point1 pf = c1 * p1 * c2;  
Point1 pg = p1 + p2*c2;  
Point1 ph = c1*(p1 + p2*c2);
```

- *The compiler implements all the rules of operator precedence automatically for us.*
- **All of the above expressions will be evaluated correctly.**

6 Unary operators

- The operators $+$, $-$, $*$ and $/$ (which we did not overload) all have two operands.
- They are called **binary operators**.
- A **unary operator** has only one argument.
- Actually the operator $-$ is both a unary and a binary operator.
- We do not normally think of the expression “ $-p$ ” as an operator, but it is:

$$p_1 = -p_2 .$$

- The code to overload “ $-$ ” as a unary operator is as follows:

```
Point1 operator- (const Point1 &u)           // unary operator
{
    double xneg = -u.getx();
    double yneg = -u.gety();
    Point1 p;
    p.set(xneg, yneg);
    return p;
}
```

- There are also other unary operators.

7 Operator overloading, example program

- Here is a working C++ program employing operator overloading and `Point1` objects.
- I have skipped “pi” for obvious reasons.

```
#include <iostream>
using namespace std;

class Point1 {
    // etc
};

// code for operator overloads

int main()
{
    double c1 = 2.0;
    double c2 = 3.0;
    Point1 p1, p2, p3;
    p1.set(4.4, 3.3);
    p2.set(2.2, 1.1);
    p3.set(0.4, 0.5);

    Point1 pa = p1 + p2;           // operator +
    Point1 pb = p1 - p2;           // operator -
    Point1 pc = c1 * p1;           // operator (double * Point1)
    Point1 pd = p2 * c2;           // operator (Point1 * double)
    Point1 pe = p1 + p2 + p3;
    Point1 pf = c1 * p1 * c2;
    Point1 pg = p1 + p2*c2;
    Point1 ph = c1*(p1 + p2*c2);
    Point1 pj = -p1;

    pa.print();
    pb.print();
    pc.print();
    pd.print();
    pe.print();
    pf.print();
    pg.print();
    ph.print();
    pj.print();
    return 0;
}
```

8 Operators as class methods

- **Operators can be written as class methods.**
- Let us declare a class `Pointlop`, the same as `Point1` but with operators written as class methods.
- The binary operators `+`, `-` and one of `*` have been declared as class methods.

```
class Pointlop {                                // operators as class methods
public:
    Pointlop operator+ (const Pointlop & v)  const; // "this object" + v
    Pointlop operator- (const Pointlop & v)  const; // "this object" - v
    Pointlop operator* (double d) const;      // "this object" * d
    Pointlop operator- () const;              // unary minus no arguments

    // etc ... rest of class declaration
};
```

- The signature of the binary operators have only one argument.
- The left operand is “this object” (of the method).

```
Pointlop operator+ (const Pointlop & v)  const // "this object" + v
{
    Pointlop p;
    p.set(x + v.x, y + v.y);
    return p;
}
Pointlop operator- (const Pointlop & v)  const // "this object" - v
{
    Pointlop p;
    p.set(x - v.x, y - v.y);
    return p;
}
```

- Unlike the previous versions, the above code does not require the use of the accessor methods `getx` and `gety`.
- We are “inside the class” therefore we have access to the values of `x`, `y`, `v.x` and `v.y` directly.
- **The operators are tagged `const`, no less.**
- Unlike the assignment `operator=`, the operators `+` and `-` do not change “this” object.
- They create a new object internally and return it as the return value.
- Hence the operators can be tagged as `const`.
- [See next page\(s\).](#)

- The signature of the unary operator has no arguments (the operand is “this object”).

```
Point1op operator- () const           // unary minus, no arguments
{
    Point1op p;
    p.set(-x, -y);
    return p;
}
```

- The operator returns an object with the negated values $(-x, -y)$ of “this” object.
- **The operator is also tagged const.**
- Writing operator* as a class method presents a difficulty.
- The left operand is “this object” **hence only the operator “(object * double)” can be declared as a class method.**

```
Point1op operator* (double d) const   // "this object" * d
{
    Point1op p;
    p.set(x*d, y*d);
    return p;
}
```

- **The operator is also tagged const.**
- The operator “(double * object)” must be written outside the class, as shown previously.
- Alternatively, we can call the class method, as shown below.

```
Point1op operator* (double c, const Point1op &p)   // operator "double * object"
{
    return (p*c);                                // call the class method "object *
}
```

- The complete class declaration for Point1op is given below.
- Also the external function for operator *.
- A working main program is also displayed.

```

class Pointlop {
public:
    void set(const double &a, const double &b)
    {
        x = a;
        y = b;
    }

    const double& getx() const { return x; }
    const double& gety() const { return y; }

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

    Pointlop operator+ (const Pointlop & v) const // "this object" + v
    {
        Pointlop p;
        p.set(x + v.x, y + v.y);
        return p;
    }
    Pointlop operator- (const Pointlop & v) const // "this object" - v
    {
        Pointlop p;
        p.set(x - v.x, y - v.y);
        return p;
    }
    Pointlop operator- () const // unary minus no arguments
    {
        Pointlop p;
        p.set(-x, -y);
        return p;
    }
    Pointlop operator* (double d) const // "this object" * d
    {
        Pointlop p;
        p.set(x*d, y*d);
        return p;
    }

private:
    double x, y;
};

Pointlop operator* (double c, const Pointlop &p) // double * Pointlop
{ return (p*c); }

```

9 Operators as class methods, example program

- The program below looks basically the same as the program in Sec. 7.
- It does not really matter if the operators are written as class methods or not.
- Some operators (such as the assignment operator=) *must* be written as class methods.

```
#include <iostream>
using namespace std;

class Point1op // etc (class declaration)

// external operator*

int main()
{
    double c1 = 2.0;
    double c2 = 3.0;
    Point1op p1, p2, p3;
    p1.set(4.4, 3.3);
    p2.set(2.2, 1.1);
    p3.set(0.4, 0.5);

    Point1op pa = p1 + p2;           // operator +
    Point1op pb = p1 - p2;           // operator -
    Point1op pc = c1 * p1;           // operator (double * Point1op)
    Point1op pd = p2 * c2;           // operator (Point1op * double)
    Point1op pe = p1 + p2 + p3;
    Point1op pf = c1 * p1 * c2;
    Point1op pg = p1 + p2*c2;
    Point1op ph = c1*(p1 + p2*c2);
    Point1op pj = -p1;

    pa.print();
    pb.print();
    pc.print();
    pd.print();
    pe.print();
    pf.print();
    pg.print();
    ph.print();
    pj.print();
    return 0;
}
```

10 Summary

- Many operators can be overloaded to operate on user-defined classes.
- The list includes standard mathematical operators such as `+`, `-`, `*` and `/`.
- The list also includes comparison operators such as `==`, `<`, `≤`, `>` and `≥`, etc., which return Boolean true/false values.
- The compiler automatically implements the rules of **operator precedence** for the operators.
- Some operators such as the dot “.” operator cannot be overloaded.
- A list of C++ operators which can and cannot be overloaded can be found online.
- **Note: We cannot overload operators to act on primitive data types.**
 1. For example, we cannot overload operator `+` for (`double` and `int`).
 2. We can overload operator `+` if one of the operands is an object of a user-defined class.
 3. We can overload operator `+` for (`object` and `int`) or (`double` and `object`), etc.
- The inputs of an operator are called its **operands**.
 1. Operators with two operands are called **binary operators**.
 2. Operators with one operand are called **unary operators**, e.g. unary minus.
- Overloaded operators can be written as (i) functions (i.e. outside a class) or (i) class methods.
 1. If an operator is written as a class method, the “this object” is the left operand.
 2. Some operators, such as the assignment `operator=`, *must* be written as class methods.
 3. Some operators, such as the assignment `operator=`, modify the “this object” therefore they are not `const`.
 4. Other operators do not modify the “this object” therefore they can be tagged as `const`.
- Operators can have many return types.
 1. Comparison operators such as `==` or `!=`, etc. return a Boolean value.
 2. Operators such as `+`, `-`, `*` and `/` typically return an object.
 3. The assignment `operator=` returns a **reference to an object**.
 4. It is also possible for an operator to return a `const` object or reference.
 5. *It is also possible for an operator to return a pointer or an array.*
 6. It is also possible for the return type of an operator to be `void`.
 7. **We are free to choose the return type of an overloaded operator.**