Operator Overloading

Rupesh Nasre.

OOAIA
January 2018

Classes and Objects

- A class is a type (such as int or struct node)
- An object is its instance (such as x or n1)
- There could be multiple objects of a type.
- Each object has a single type.
- But the value stored in the object may be convertible to another type.
- A class declaration may have Methods and Variables.
- The class content may be public, private, or protected.

A Scenario

- We want to add
  - Two integers — addInt2(x, y);
  - Multiple integers — addIntMany(x, y, z, ...);
  - Multiple complex numbers — addComplex(c1, c2, c3, ...);
  - An element to a set — addElement(e);
  - A set to a set — addSet(s);
  - ... — ...

A Scenario

- We want to add
  - Two integers add(x, y);
  - Multiple integers add(x, y, z, ...);
  - Multiple complex numbers add(c1, c2, c3, ...);
  - An element to a set add(e);
  - A set to a set add(s);
  - ... ...

More Scenarios

- **+**
  - Integer addition
  - String concatenation
  - Set union
  - Appending to a queue
  - ...
- **A[i]**
  - Element of an array
  - Student in a class
  - Participant in a marathon
  - ...
- ...

Polymorphism

- The same mnemonic appears in multiple forms:
  - Poly = multiple, morph = form
- Bears potential to tremendously improve code readability.

Supported in C++, Java, ...
e.g., add(1), add(x, 4)

Supported in C++.
e.g., string * 2, obj[5]

Overloading Example

```cpp
#include <iostream>

class A {
public:
	void add(int x) { n += x; }
	void add(int x, int y) { n += x + y; }

	A():n(0) {}
	~A() { std::cout << n << std::endl; }
private:
	int n;
};

void fun(A& a) {
	a.add(1);
}
void fun(A& a, int x) {
	a.add(x, 3);
}
int main() {
	A a;
	fun(a);
	fun(a, 2);
	return 0;
}
```

Function overloading

```cpp
#include <iostream>

class A {
public:
	A& operator +(int x) { n += x; }

	A():n(0) {}
	~A() { std::cout << n << std::endl; }
private:
	int n;
};

int main() {
	A a;
	a + 2;
	return 0;
}
```

Operator overloading

Classwork

- Assuming you have a class Name, overload multiplication operator to create a concatenation of Name's value.
  - e.g., If name contains “ab”, name * 3 should return a new Name object with string “ababab”.

```
#include <iostream>

class Name {
public:
        Name operator *(int n) {
            std::string retval;
            for (int ii = 0; ii < n; ++ii) retval += str;
            return Name(retval.c_str());
        }
        void print() { std::cout << str << '\n'; }
        Name(const char *s) { str = s; }

private:
         std::string str;
};
```

```
int main() {
        Name name("abc");
        Name namemult = name * 4;
        namemult.print();
        return 0;
}
```

Rules

1. Must be overloaded for a user-defined class.
   - Cannot overload for primitive types.
2. Operator associativity remains the same.
   - Name namemult = name * 4 * 2; // left-to-right
3. Operator precedence remains the same.
   - Name namemult = name * 4 + 2; // error
4. Arity remains the same.
   - Name namemult = name ++ 4; // error
5. Cannot define a new symbol as operator.
   - Name namemult = name @ 4; // error

Non-overloadable Operators

- **.** member operator (e.g., e.g)
- **.*** pointer to member operator (e.g., x.*y)
- **?:** ternary conditional operator (e.g., a==0 ? 1 : c)
- **::** scope resolution operator (e.g., A::fun())
- **sizeof** data size operator (e.g., sizeof(int))
- **typeid** data type operator (e.g., typeid(x))

All other usual operators (+, <<, -=, ->, (), ++, [ ], new, delete, ...) can be overloaded.

Overloading Unary Operator

```
#include <iostream>

class Num {
public:
        Num operator -() {
                return Num(-n);
        }
        void print() { std::cout << n << '\n'; }
        Num(int ln) { n = ln; }

private:
         int n;
};

int main() {
        Num n1(5);
        Num n2 = -n1;
        n2.print();
        return 0;
}
```

Overloading << Operator

```
#include <iostream>

class Num {
public:
	int operator << (int by) {
		return n << by;
	}
	void print() { std::cout << n << '\n'; }
	Num(int ln) { n = ln; }

private:
	 int n;
};

int main() {
	Num n1(5);
	std::cout << n1 << 2 << std::endl;
	return 0;
}
```

What is the problem with the above code?

Overloading << Operator

```
#include <iostream>

class Num {
public:
	int operator << (int by) {
		return n << by;
	}
	void print() { std::cout << n << '\n'; }
	Num(int ln) { n = ln; }

private:
	 int n;
};

int main() {
	Num n1(5);
	std::cout << (n1 << 2) << std::endl;
	return 0;
}
```

Overloading << for cout

- We want to achieve the following:
  ```
  Num n1; std::cout << n1;
  ```
- To make << operator to work, we need to define a << operator on `cout`'s class with `Num` as a parameter.
  ```
  class ostream {
      ostream& operator << (Num &num) { ... }
      ...
  };
  ```
- We cannot do this. And definitely not for every new user-defined type!

Overloading << for cout

```
#include <iostream>

class Num {
public:
	int getNum() { return n; }
	Num(int In) { n = In; }

private:
	 int n;
};
std::ostream& operator << (
	std::ostream& stream, Num &num) {

	return (stream << num.getNum());
}

int main() {
	Num n1(5);
	std::cout << n1 << std::endl;
	return 0;
}
```

This method works if all the information to be printed is available via public methods.

Overloading << for cout

```
#include <iostream>

class Num {
public:

        Num(int In) { n = In; }

private:
         int n;
};
std::ostream& operator << (
        std::ostream& stream, Num &num) {

        return (stream << num.n);
}

int main() {
        Num n1(5);
        std::cout << n1 << std::endl;
        return 0;
}
```

Error: operator << cannot access private member n.

Overloading << for cout

```
#include <iostream>

class Num {
public:

        Num(int In) { n = In; }

private:
         int n;

friend std::ostream& operator << (
           std::ostream& stream, Num &num);
};
std::ostream& operator << (
        std::ostream& stream, Num &num) {

        return (stream << num.n);
}

int main() {
        Num n1(5);
        std::cout << n1 << std::endl;
        return 0;
}
```

- Note that we didn't have to change ostream class.
- A global operator << like this needs to take two arguments, while that inside a class needs one explicit argument.
- A similar mechanism can be used to define other operators.

Overloading + Outside Class

```
#include <iostream>

class Num {
public:
        int getNum() { return n; }
        Num(int ln) { n = ln; }

private:
         int n;
// friend int operator + (Num &num, int n2);
};
int operator + (Num &num, int n2) {
        return num.getNum() + n2;
}
int main() {
        Num n1(5);
        std::cout << (n1 + 3) << std::endl;
        return 0;
}
```

What is the issue with such a code?

Overloading + Outside Class

```
#include <iostream>

class Num {
public:
	int getNum() { return n; }
	Num(int In) { n = In; }

private:
	 int n;
// friend int operator + (Num &num, int n2);
};
int operator + (Num &num, int n2) {
	return num.getNum() + n2;
}
int main() {
	Num n1(5);
	std::cout << (n1 + 3) << std::endl;
	std::cout << (3 + n1) << std::endl;
	return 0;
}
```

Error:
+ (int, Num&) is undefined.

Overloading + Outside Class

```
#include <iostream>

class Num {
public:
	int getNum() { return n; }
	Num(int In) { n = In; }

private:
	 int n;
// friend int operator + (Num &num, int n2);
};
int operator + (Num &num, int n2) {
	return num.getNum() + n2;
}
int operator + (int n2, Num &num) {
	return num + n2;
}
int main() {
	Num n1(5);
	std::cout << (n1 + 3) << std::endl;
	std::cout << (3 + n1) << std::endl;
	return 0;
}
```

Basic integer addition.

Overloading >> for cin

```
#include <iostream>

class Num {
public:
        int getNum() { return n; }
        Num(int In) { n = In; }

private:
         int n;

friend std::istream& operator >> (
           std::istream& stream, Num &num);
};
std::istream& operator >> (
        std::istream& stream, Num &num) {

        return (stream >> num.n);
}
int main() {
        Num n1(5);
        std::cin >> n1;
        std::cout << n1.getNum() << std::endl;
        return 0;
}
```

With << and >>

```
#include <iostream>
class Num {
public:
        Num(int In) { n = In; }
private:
         int n;

friend std::istream& operator >> (
           std::istream& stream, Num &num);
friend std::ostream& operator << (
           std::ostream& stream, Num &num);
};
std::istream& operator >> (
        std::istream& stream, Num &num) {
        return (stream >> num.n);
}
std::ostream& operator << (
        std::ostream& stream, Num &num) {
        return (stream << num.n);
}
int main() {
        Num n1(5);
        std::cin >> n1;
        std::cout << n1 << std::endl;
        return 0;
}
```

Overloading new

- For custom allocation, `new` can be overloaded.
- This is useful when you want to manage memory yourself (either for efficient memory usage or for efficient execution).
- Examples:
  - reusing memory of deleted nodes
  - garbage collection
  - Improved locality
- Overloading `new` usually requires overloading `delete`.

Overloading new

```
#include <iostream>
#include <stdlib.h>

class A {
public:
	void *operator new(size_t size) {
		std::cout << "in my new.\n";
		return malloc(size);
	}
	void setData(int Id) { data = Id; }
	int getData() { return data; }

private:
	int data;
};
int main() {
	std::cout << "calling overloaded new.\n";
	A *a = new A;
	a->setData(3);
	std::cout << "in main: " << a->getData() << "\n";
	return 0;
}
```

Overloading delete

```
#include <iostream>
#include <stdlib.h>

class A {
public:
	void *operator new(size_t size) {
		std::cout << "in my new.\n";
		return malloc(size);
	}
	void operator delete(void *ptr) {
		std::cout << "in my delete.\n";
		free(ptr);
	}
	void setData(int Id) { data = Id; }
	int getData() { return data; }
private:
	int data;
};
int main() {
	A *a = new A;
	a->setData(3);
	std::cout << "in main: " << a->getData() << "\n";
	delete a;
	return 0;
}
```

Overloading [ ]

```
#include <iostream>
#include <string>
#include <vector>

class Students {
public:
	Students& operator +(std::string &onemore) {
		names.push_back(onemore);
		return *this;
	}
	Students& operator +(const char *onemore) {
		std::string onemorestr(onemore);
		return *this + onemorestr;
	}
	std::string operator [](int index) {
		return names[index];
	}
	void print() {
		for (auto it = names.begin();
				it != names.end(); ++it)
			std::cout << *it << std::endl;
	}
private:
	std::vector<std::string> names;
};
```

```
int main() {
	Students cs17;
	cs17 + "two";
	cs17 + "one";
	cs17 + "three";
	cs17 + "four";
	cs17 + "seven" + "six" +
		"nine" + "eight";
	//cs17.print();
	std::cout << cs17[6] << std::endl;
}
```

What is the issue with this code?

Needs compilation with *-std=c++11*
Or change *auto* to
std::vector<std::string>::iterator

Overloading [ ]

```
#include <iostream>
#include <string>
#include <vector>

class Students {
public:
	Students& operator +(std::string &onemore) {
		names.push_back(onemore);
		return *this;
	}
	Students& operator +(const char *onemore) {
		std::string onemorestr(onemore);
		return *this + onemorestr;
	}
	std::string operator [](int index) {
		return names[index];
	}
	void print() {
		for (auto it = names.begin();
				it != names.end(); ++it)
			std::cout << *it << std::endl;
	}
private:
	std::vector<std::string> names;
};
```

```
int main() {
	Students cs17;
	cs17 + "two";
	cs17 + "one";
	cs17 + "three";
	cs17 + "four";
	cs17 + "seven" + "six" +
		"nine" + "eight";
	//cs17.print();
	std::cout << cs17[6] << std::endl;
	cs17[6] = "NINE";
	std::cout << cs17[6] << std::endl;
}
```

We expect the output to be NINE.
But it prints nine.

Overloading [ ]

```
#include <iostream>
#include <string>
#include <vector>

class Students {
public:
	Students& operator +(std::string &onemore) {
		names.push_back(onemore);
		return *this;
	}
	Students& operator +(const char *onemore) {
		std::string onemorestr(onemore);
		return *this + onemorestr;
	}
	std::string& operator [](int index) {
		return names[index];
	}
	void print() {
		for (auto it = names.begin();
				it != names.end(); ++it)
			std::cout << *it << std::endl;
	}
private:
	std::vector<std::string> names;
};
```

```
int main() {
	Students cs17;
	cs17 + "two";
	cs17 + "one";
	cs17 + "three";
	cs17 + "four";
	cs17 + "seven" + "six" +
		"nine" + "eight";
	//cs17.print();
	std::cout << cs17[6] << std::endl;
	cs17[6] = “NINE”;
	std::cout << cs17[6] << std::endl;
}
```

Now the output is NINE.

Overloading -> (smart pointers)

- Imagine a scenario as below:
  - Roll number CS17B010 indicates a Btech student from Computer Science admitted in 2017.
  - This is done in class RollNumber. It supports a function ***getYear()***.
  - A Student has a RollNumber.
  - An application may query a student for knowing his / her enrolling year using *stud->**getYear()***.
- One way to implement is by implementing *Student::**getYear()***, which internally calls *rollno->**getYear()***.
- Another way is to make -> smart.

Overloading -> (smart pointers)

```
#include <iostream>

class RollNo {
public:
	int getYear() { return 2017; }
};

class Student {
public:
	RollNo *operator ->() {
		return &rollno;
	}
private:
	RollNo rollno;
};
int main() {
	Student stud;
	std::cout << stud->getYear() << std::endl;
	return 0;
}
```

No *getYear()* In *Student*

Summary

- Polymorphism
- Rules for operator overloading
- Overloading simple operators
- With cin and cout
- Custom memory allocation
- Array subscript operator overloading
- Smart pointers

Backup

Overloading versus Overriding

	Overloading	Overriding
Purpose	Readability	Change of functionality
Place	Within a class / globally	Derived class
Parameters	Must be different	Must be same
Polymorphism	Compile time	(in general) run time

```
#include <iostream>
class A {
public:
	void add(int x) { n += x; }
	void add(int x, int y) { n += x + y; }
	A():n(0) {}
	~A() { std::cout << n << std::endl; }
private:
	int n;
};
int main() {
	A a;
	a.add(1);
	a.add(2, 3);
	return 0;
}
```

Overloading

```
#include <iostream>
class A {
public:
	void add(int x) {
	  std::cout << "in A: " << x << std::endl;
}	};
class B: public A {
public:
	void add(int x) {
	  std::cout << "in B: " << x << std::endl;
}	};
int main() {
	A a;  B b;
	a.add(1);
	b.add(2);
	return 0;
}
```

Overriding

Pointer to Member Operator

```
#include <iostream>
class A {
public:
	A() { x = 0; }
	int x;
};
int main() {
	A a;
	int A::*p = &A::x;	// this is a type definition.
	a.*p = 10;
	std::cout << a.*p << ' ' << a.x << '\n';
	return 0;
}
```

Function Overloading

- A function in the base class can be re-implemented in the derived class.
- derived.method() calls the overloaded function.
- base.method() calls the base class method, provided base is not a derived class object.

Example

```
class A {
public:
	void fun() { cout << “in A\n”; }
};
class B: public A {
public:
	void fun() { cout << “in B\n”; }
};
int main() {
	A *a = new A;
	a->fun();
	return 0;
}
```

in A

```
class A {
public:
	void fun() { cout << “in A\n”; }
};
class B: public A {
public:
	void fun() { cout << “in B\n”; }
};
int main() {
	A *a = new B;
	a->fun();
	return 0;
}
```

in A

Example

```
class A {
public:
      void fun() { cout << “in A\n”; }
};
class B: public A {
public:
      void fun() { cout << “in B\n”; }
};
int main() {
      A *a = new A;
      a->fun();
      return 0;
}
```

in A

```
class A {
public:
      virtual void fun() { cout << “in A\n”; }
};
class B: public A {
public:
      void fun() { cout << “in B\n”; }
};
int main() {
      A *a = new B;
      a->fun();
      return 0;
}
```

in B

Why is this useful?

- Consider a hierarchy of geometric shapes
  - Shape ← Ellipse ← Circle ← CircleOrigin
  - `Shape c = new CircleOrigin; c->draw();`
  - Which `draw()` should be called?
  - Most specialized method is called.

Pure Virtual Functions and Abstract Classes

- A virtual function with *no definition* is pure.
- A class with at least one pvf is abstract.
- An abstract class cannot be instantiated.
  - But pointers and references of abstract type can be created and used.
- If a derived class does not implement all pvf, then it also becomes pure.
- In C++, a virtual function with no definition and a pvf can be different. A pvf can be specified even with a default definition.

Example

```
class A {
public:
      virtual void fun() = 0;
};
class B: public A {
public:
      void fun() { cout << “in B\n”; }
};
int main() {
      A *a = new B;
      a->fun();
      return 0;
}
```

A::fun is a pure virtual function.
A is an abstract class.

Abstract Class and Interface

- Abstract classes are useful to define an interface.
- A user method may simply use the interface to perform computation – without worrying about the derived classes.

```
class Shape {
public:
	virtual void draw() = 0;
	virtual void writeToFile(ofstream f) = 0;
protected:
	double area;
};
```

```
class Rectangle: public Shape {
public:
	virtual void draw() { ... }
	virtual void writeToFile(ofstream f) { ... }
	...
};
```

```
class Ellipse: public Shape {
public:
	virtual void draw() { ... }
	virtual void writeToFile(ofstream f) { ... }
	...
};
```

...