Operator Overloading

C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

Contents

Basic Overloading

- Operator overloading in ANSI **C**

```
int x, y, z;
double q, r, t;
z = x + y;
q = r + t;
```

The same operator can do different things.

- Overloading in **C++**

```
Array();
Array(int arraySize);
```

Overloaded constructors

```
void quit() {
   cout << "So you want to save before
    quitting?\n";
}
void quit(char *customMessage) {
   cout << customMessage << endl;
}
```

Functions with the same name can do different jobs.

Operator Overloading

- There are two possibilities for the following

  MyClass obj1, obj2;

  obj1 + obj2;

- Compiler would translate the above into one of the following function call if one of them is defined:
  - First: calling member function

    MyClass MyClass::operator+(MyClass rhs)

    i.e. obj1.operator+(obj2)
  - Second: calling global function

    MyClass operator+(MyClass lhs, MyClass rhs)

    i.e. operator+(obj1, obj2)

(If both of them are defined, the global one will be invoked.
Do not take this as a practicing rule!!)

Operator Overloading (cont'd)

- Consider the following **MenuItem** class which describes the item on a restaurant menu

```
class MenuItem {
public:
   MenuItem(int itemPrice, char *itemName);
   MenuItem(const MenuItem &src);
   ~MenuItem();
   void display() const;
private:
   int m_price;
   char *m_name;
};
```

- We would like to do the following

```
void main() {
   MenuItem item1(250, "Chicken Florentine");
   MenuItem item2(120, "Tiramisu");
   cout << "You ordered the following items:";
   item1.display(); item2.display();
   cout << "The total is $" << item1 + item2 << ".\n";
}
```


First Solution with Overloading

- Add a member function which overloads operator+()

```
class MenuItem {
public:
   MenuItem(int itemPrice, char *itemName);
   MenuItem(const MenuItem &src);
   ~MenuItem();
   void display() const;
   int operator+(const MenuItem &secondItem) const;
private:
   int m_price;
   char *m_name;
};
```

or MenuItem secondItem

- The function is defined as follows

```
int MenuItem::operator+(const MenuItem &secondItem) const {
   return m_price + secondItem.m_price;
}
```

Left operand of + Right operand of +

Behavior of Overloaded Operator

- **Add a third menu item**

```
MenuItem item1(250, "Chicken Florentine");
MenuItem item2(120, "Tiramisu");
MenuItem item3(50, "Mineral Water");
int total;
total = item1 + item2 + item3;
```

Why?

error C2677: binary '+' : no global operator defined which takes type 'class MenuItem' (or there is no acceptable conversion)

  - item1 + item2 returns an int
  - you then have int + Menuitem (item3)
    The overloaded member function can only be called by an instance of the class.

- **Solution**: make the overloaded function toplevel

```
int operator+(int currentTotal, MenuItem &secondItem) {
   return currentTotal + secondItem.m_price;
}
```

make this function a friend of MenuItem

could be reference or value

Behavior (cont'd)

- The following statement still fails

```
item1 + (item2 + item3)
```

error C2678: binary '+' : no operator defined which takes a left-hand operand of type 'class MenuItem' (or there is no acceptable conversion)

Why?

  - This is equivalent to Menuitem + int

- Solution: add another overloaded operator function

```
int MenuItem::operator+(int currentTotal) {
   return currentTotal + m_price;
}
```

Why does this function not have to be toplevel (i.e. global)?

- Conclusion

When you overload an operator, you are responsible for the correct behavior of the operator in **ALL** possible circumstances.

Alternative Solution

- Use **conversion constructor** together with global
  operator+(const MenuItem &, const MenuItem &)

```
class MenuItem {
   friend int operator+(const MenuItem &firstItem,
                        const MenuItem &secondItem);
public:
   MenuItem(int itemPrice, char *itemName);
   MenuItem(int price);
   MenuItem(const MenuItem &src);
   ~MenuItem();
   void display() const;
private:
   int m_price;
   char *m_name;
};
```

- The conversion constructor

```
MenuItem::MenuItem(int price): m_price(price), m_name(0) {
}
```

- Overload the operator at the toplevel with two MenuItem objects

```
int operator+(const MenuItem &firstItem, const MenuItem &secondItem) {
   return firstItem.m_price + secondItem.m_price;
}
```


Complex Number Example

- Complex class represents a complex number (real, imaginary), define two mathematic operations (no side effect)

```
Complex Complex::add(const Complex &secondNumber) const {
   Complex tmp(m_real+secondNumber.m_real,
               m_imaginary+secondNumber.m_imaginary);
   return tmp;
}

Complex Complex::multiply(const Complex &secondNumber) const {
   Complex tmp(m_real*secondNumber.m_real-
               m_imaginary*secondNumber.m_imaginary,
               m_real*secondNumber.m_imaginary+
               m_imaginary*secondNumber.m_real);
   return tmp;
}
```

- main()

c + z * z

```
Complex c(0.1, 0), z(0, 0);
for (int i=1; i<MaxIterations; i++) {
   z = c.add(z.multiply(z));
   if (fabs(z.getRealPart())>2.0 || fabs(z.getImaginaryPart())>2.0) break;
}
```


Complex Number (cont'd)

- Let us overload + and *

```
Complex Complex::operator+(const Complex &secondNumber) const {
   Complex tmp(m_real+secondNumber.m_real,
               m_imaginary+secondNumber.m_imaginary);
   return tmp;
}
Complex Complex::operator*(const Complex &secondNumber) const {
   Complex tmp(m_real*secondNumber.m_real-
               m_imaginary*secondNumber.m_imaginary,
               m_real*secondNumber.m_imaginary+
               m_imaginary*secondNumber.m_real);
   return tmp;
}
```

- main()

```
Complex c(0.1, 0), z(0, 0);
for (int i=1; i<MaxIterations; i++) {
   z = c + z * z;
   if (fabs(z.getRealPart())>2.0 || fabs(z.getImaginaryPart())>2.0) break;
}
```

- Related operators +=, *=

Dubious Operator Overloading

- Here are some actual examples from a textbook
  Can you guess what these operators mean?

```
Stack s;
…
s+5;
x = s--;
```

This is too far away!!

They are used to stand for the following

```
s.push(5);
x = s.pop();
```

- Overloading obscure operators can be dangerous

  Redefine ^ (bitwise XOR) to mean "power"

  It won't work as expected, ex. **Integer x;**

  x ^ 2 + 1 // if x is 5, you want to get 26, but you get 125 instead

  Reason: ^ has lower precedence than +

- Illegal overloading

```
int operator+(int number1, int number2) {
   return number1-number2;
}
```

error C2803: 'operator +' must have at least one formal parameter of class type

Operator Precedence & Association

1	::	Scope resolution	None
2	::	Global	None
3	[ ]	Array subscript	Left to right
4	( )	Function call	Left to right
5	( )	Conversion	None
6	.	Member selection	Left to right
7	–>	Member selection	Left to righ
8	++	Postfix increment	None
9	—	Postfix decrement	None
10	new	Allocate object	None
11	delete	Deallocate object	None
12	delete[]	Deallocate object	None
13	++	Prefix increment	None
14	—	Prefix decrement	None
15	*	Dereference	None
16	&	Address-of	None
17	+	Unary plus	None
18	–	Arithmetic negation (unary)	None

19	!	Logical NOT	None
20	~	Bitwise complement	None
21	sizeof	Size of object	None
22	sizeof()	Size of type	None
23	typeid()	type name	None
24	(type)	Type cast	Right to left
25	const_cast	Type cast	None
26	dynamic_cast	Type cast (conversion)	None
27	reinterpret_cast	Type cast (conversion)	None
28	static_cast	Type cast	None
29	.*	Apply pointer to class member (objects)	Left to right
30	–>*	Dereference pointer to class member	Left to right
31	*	Multiplication	Left to right
32	/	Division	Left to right

Operator Precedence & Association

33	%	Remainder (modulus)	Left to right
34	+	Addition	Left to right
35	–	Subtraction	Left to right
36	<<	Left shift	Left to right
37	>>	Right shift	Left to right
38	<	Less than	Left to right
39	>	Greater than	Left to right
40	<=	Less than or equal to	Left to right
41	>=	Greater than or equal to	Left to right
42	==	Equality	Left to right
43	!=	Inequality	Left to right
44	&	Bitwise AND	Left to right
45	^	Bitwise exclusive OR	Left to right
46	|	Bitwise OR	Left to right
47	&&	Logical AND	Left to right
48	||	Logical OR	Left to right
49	e1?e2:e3	Conditional	Right to left

50	=	Assignment	Right to left
51	*=	Multiplication assignment	Right to left
52	/=	Division assignment	Right to left
53	%=	Modulus assignment	Right to left
54	+=	Addition assignment	Right to left
55	–=	Subtraction assignment	Right to lef
56	<<=	Left-shift assignment	Right to left
57	>>=	Right-shift assignment	Right to left
58	&=	Bitwise AND assignment	Right to left
59	|=	Bitwise inclusive OR assignment	Right to left
60	^=	Bitwise exclusive OR assignment	Right to left
61	,	Comma	Left to right

Overload All Related Operators

- If you provide a + operator, you should also provide related operators such as += and ++
- Let us define a Time class that allows addition

```
class Time {
public:
   Time();
   Time(int hours, int minutes, int seconds);
   void display();
   Time operator+(Time secondTime);
private:
   int m_hours;
   int m_minutes;
   int m_seconds;
   void normalize();
};
```

```
Time::Time(): m_seconds(0), m_minutes(0), m_hours(0) {
}
```

```
Time::Time(int hours, int minutes, int seconds)
   : m_hours(hours), m_minutes(minutes), m_seconds(seconds) {
   normalize();
}
```


Overload + and *

- **operator+**

```
Time Time::operator+(Time secondTime){
   int hours, minutes, seconds;
   hours = m_hours + secondTime.m_hours;
   minutes = m_minutes + secondTime.m_minutes;
   seconds = m_seconds + secondTime.m_seconds;
   return Time(hours, minutes, seconds);
}
```

Note: we do not call normalize() in this case

- **operator*=**

```
void Time::operator*=(int num) {
   m_hours *= num;
   m_minutes *= num;
   m_seconds *= num;
   normalize();
}
```

This operator does not return anything and has side effects.

```
Time time1(20, 15, 0);
Time time2(3, 45, 10);

Time time3 = time1 + time2;
time3.display();
cout << endl;

time2 *= 3;
time2.display();
cout << endl;
```

operator++ and operator--

- ++ and -- come in **postfix** and **prefix** formats

```
int x, y;
x = 5;
y = x++;
cout << "x is " << x << " and y is " << y << "\n";
```

Output
x is 6 and y is 5

```
x = 5;
y = ++x;
cout << "x is " << x << " and y is " << y << "\n";
```

Output
x is 6 and y is 6

- How does compiler know which ++ operator you want to override?
  - **Postfix** syntax

    Time Time::operator++(int) // int argument is ignored
  - **Prefix** syntax

    Time &time::operator++() // l-value

operator++ and -- (cont'd)

- **Postfix operator**

```
Time Time::operator++(int) {
   Time tmp = *this;
   m_seconds++; normalize();
   return tmp;
}
```

  - Usage

```
Time firstTime(1, 1, 3), secondTime;
secondTime = firstTime++;
firstTime.display(); secondTime.display();
```

Output
01:01:04
01:01:03

- **Prefix operator**

```
Time &Time::operator++() {
   m_seconds++; normalize();
   return *this;
}
```

  - Usage

```
Time firstTime(1, 1, 3), secondTime;
secondTime = ++firstTime;
firstTime.display(); secondTime.display();
```

Output
01:01:04
01:01:04

ℓ value
~~++2;~~
~~2++;~~
int x;
++++x;
~~x++++;~~

operator[]

- **Example**: An array class which includes bounds checking

```
class Array {
public:
   Array();
   Array(int arraySize);
   ~Array();
   void insert(int slot, int element);
   int get(int slot) const;
private:
   int m_arraySize;
   int *m_array;
};
void Array::insert(int slot, int element) {
   if (slot<m_arraySize && slot>=0)
      m_array[slot] = element;
   else
      cout << "Subscript out of range\n";
}
int Array::get(int slot) const {
   if (slot<m_arraySize && slot>=0)
      return m_array[slot];
   cout << "Subscript out of range\n";
   return 0;
}
```

```
Array data(5);
for (int i=0; i<5; i++)
   data.insert(i, i*2);
cout << data.get(3);
```

We prefer the following: the same syntax as accessing a "normal" array.

```
Array data(5);
for (int i=0; i<5; i++)
   data[i] = i*2;
cout << data[3];
```


operator[] (cont'd)

```
class Array {
public:
   Array();
   Array(int arraySize);
   ~Array();
   int &operator[](int slot);
private:
   int m_arraySize;
   int *m_array;
};
```

ℓ-value is an object that persists beyond a simple expression
r-value is a temporary value that does not persist beyond the the expression that uses it

not address!
not address!
not address!
**int ***Array::operator[](…
Reference is an ℓ-value

```
int &Array::operator[](int slot) {
   if (slot<m_arraySize && slot>=0)
      return m_array[slot];
   cout << "Subscript out of range\n";
   return m_array[0];
}
```

The Account Example

```
class Account {
public:
   Account(const char *name, const char *phone, const char *address);
   ~Account();
   ….
private:
   char *m_name;
   char *m_phone;
   char *m_address;
};
Account::Account(const char *name,
                 const char *phone, const char *address) {
   m_name = new char[strlen(name)+1]; strcpy(m_name, name);
   m_phone = new char[strlen(phone)+1]; strcpy(m_phone, phone);
   m_address = new char[strlen(address)+1]; strcpy(m_address, address);
}
Account::~Account() {
   delete[] m_name; delete[] m_phone; delete[] m_address;
}
```


Assignment Operator

- When/where is the assignment operator invoked?
  Account customer1("abc", "1234", "ABC street");
  Account customer2, customer3; // assume default ctor defined
  customer2 = customer1;
  customer2.operator=(customer1);
  customer3 = customer2 = customer1;
- Note: Account customer2 = customer1;
  does **not** invoke the assignment operator
- What is its prototypes?
  Account **&**operator=(const Account **&**rhs);
  No extra copy ctor invoked
  Designed for continuously assignment
  customer3.operator=(customer2.operator=(customer1));
- =,(),[],-> cannot be overloaded as a non-member

Assignment Operator

- Again, if the class being designed **allocates its own resources**. It is quite often to see the **dtor**, **copy ctor**, and the **assignment operato**r occur together.
- There are **seven** important things to do in an assignment operator

```
   Account &Account::operator=(const Account &rhs)
   {
①     if (&rhs == this) return *this;        Detecting self assignments
②     delete[] m_name;  delete[] m_phone;  delete[] m_address;
③     m_name = new char[strlen(rhs.m_name)+1];
      m_phone = new char[strlen(rhs.m_phone)+1];
      m_address = new char[strlen(rhs.m_address)+1];
④     strcpy(m_name, rhs.m_name);
      strcpy(m_phone, rhs.m_phone);
      strcpy(m_address, rhs.m_address);
⑤     // invoke the base class assignment operator
⑥     // invoke the component object assignment operator
⑦     return *this;
   }
```


Related Operators of Assignment

- If you overload assignment, you might like to overload equality

```
bool Account::operator==(const Account &rhs) const {
   if ((strcmp(m_name, rhs.m_name)==0) &&
       (strcmp(m_phone, rhs.m_phone)==0) &&
       (strcmp(m_address, rhs.m_address)==0))
      return true;
   else
      return false;
}
```

- Usage

```
Account customer1("abc", "1234", "ABC street"), customer2;
customer2 = customer1;
…
if (customer2 == customer1) …
```

- Other related operators
  - **bool operator!=(const Account &rhs) const;**
  - **bool operator<(const Account &rhs) const;**
  - **bool operator<=(const Account &rhs) const;**
  - **bool operator>(const Account &rhs) const;**
  - **bool operator>=(const Account &rhs) const;**

Function Call operator()

- Overload operator() to make an object stand for a function and behave like a function

This object is called a **Functor**
Heavily used with STL and higher-order programming

```
class Polynomial {
public:
   Polynomial(double secondOrder, double firstOrder, double constant);
   double operator()(double x);
private:
   double m_coefficients[3];
};
Polynomial::Polynomial(double secondOrder, double firstOrder, double constant) {
   m_coefficients[2] = secondOrder;
   m_coefficients[1] = firstOrder;
   m_coefficients[0] = constant;
}
double Polynomial::operator()(double x) {
   return m_coefficients[2]*x*x + m_coefficients[1]*x + m_coefficients[0];
}
void main() {
   Polynomial f(2, 3, 4);
   int x = 2;
   cout << f(x);
}
```

a function with memory

Output
18

Sometimes, you might see
Polynoimial(2,3,4)(x)

Other Usage of operator()

- operator() is the only operator that can take any number of arguments
- Imagine you had a matrix class (two-dimensional array): You would like to avoid accessor and mutator functions. One idea is to overload the operator[], the subscript operator.
- How about overloading operator[][]?

```
int &operator[][](int x);
```

Illegal, no such operator

- The closest equivalent to array subscripting is to overload operator() with two arguments

```
int &Matrix::operator()(int x, int y) {
   if (x>=0 && x<m_dim1 && y>=0 && y < m_dim2)
      return m_matrix[x][y];
   cout << "out of bounds!\n";
   return m_matrix[0][0];
}
```

- Usage

```
Matrix matrix(5,10);
matrix(2,3) = 10;    cout << matrix(2,3);
```


Class Member Access Operator

- Example:

no arguments

```
class Person {
public:
   Person(char *name, int age)
   int getAge();
   Name *operator–>();
private:
   Name *m_ptrNameObject;
   int m_age;
};
```

```
class Name {
public:
   Name(char *name);
   ~Name();
   const char *getName();
private:
   char *m_name;
};
```

```
void main() {
   Person person("Harvey", 12);
   cout << person–>getName();
}
```

- The goal is to link Name::getName() to an instance of class Person

- *smart pointer* (managed pointer) are implemented with **operator–>**. The **primary purpose** to overload a *class member access* operator is ❶ to give an object **"pointer-like" behavior** and ❷ to **link a member function of a sub-object to the main object**

Class Member Access Operator (cont'd)

- The overloaded function

```
Name *Person::operator–>() {
   return m_ptrNameObject;
}
```

- Using the class member access operator

```
void main() {
   Person person("Harvey", 12);
   cout << person–>getName();
}
```

m_ptrNameObject / **person** –> getName() ?

m_ptrNameObject / **person** –> getName() ✕

Note that ***person*** is an object but behaves like a pointer.

- **Evaluating rules** of a class member access operator –> :
  1. If the target is a pointer, –> operator is evaluated as it normally is.
  2. If it is an object with an overloaded –> operator, the object is replaced by the output of the operator->() function, i.e.

     person–>getName() - - - - -> **m_ptrNameObject–>getName();**

  The process continues until evaluation occurs normally (i.e. the left hand side of –> operator is a pointer).

operator new / operator delete

- You can have your own new and delete for a particular object

```
class Random {
public:
   Random(int data);
   int getData();
   void *operator new(size_t objectSize);
   void operator delete(void *object);
private:
   int m_data;
};
```

```
void Random::operator delete(void *object) {
   cout << "delete\n";
   free(object);
}
```

new operator would determine suitable value for objectSize and invoke this function

```
void *Random::operator new(size_t objectSize) {
   cout << "new\n";
   return malloc(objectSize);
}
```

not directly

Note: mechanism is different from all other operators

```
void main() {
   Random *ptr = new Random(20);
   delete ptr;
}
```

new operator

① invokes Random::operator new(size_t)

② invokes Random::Random(int)

delete operator also does two things automatically

operator new[] / operator delete[]

```
class Random {
public:
   Random();
   int getData();
   void *operator new[](size_t objectSize);
   void operator delete[](void *object);
private:
   int m_data;
};
```

```
void *Random::operator new[](size_t objectSize) {
   cout << "new[] objectSize=" << objectSize << "\n";
   return malloc(objectSize);
}
```

```
void Random::operator delete[](void *object) {
   cout << "delete[]\n";
   free(object);
}
```

- Usage:

```
void main() {
   Random *ptr = new Random[5];
   delete[] ptr;
}
```

Note: after calling
① Random::operator new[](size_t)
new[] would invoke 5 times the default ctor
② Random::Random()

delete[] also does two things automatically

operator new / operator delete

- Why should one override **new**, **new[]**, **delete**, **delete[]**?
  - One can allocate/deallocate memory from an internal **memory pool** instead of standard malloc/free
- Can you see why **new[]/delete** or **new/delete[]** would fail?
  - For a delete[] operator, the internal mechanism should try to invoke destructors for all objects. If that block of memory was allocated with new…. Error occurs
  - For a delete operator, the internal mechanism only invoke destructor once. If that block of memory was allocated with new[] … Many objects will not be suitably destructed

Type Conversion

- Consider a simple **String** class

```
class String {
public:
   String(const char *inputData) { cout << "String(const char *)\n"; }
   String(const String &src) { cout << "String(const String &)\n"; }
private:
   char *m_string;
};
```

```
void fun(String str) {
   cout << "fun(String) \n";
}
```

- This class allows conversions from ANSI C char arrays to the object of this class through the **type conversion constructor**

```
void main() {
   String string1("str1");
   String string2 = "str2";
   String string3 = Sring("str3");
   fun(string3);
   fun("str4");
}
```

String(const char *)
String(const char *)
String(const char *)
String(const String &) fun(String)
String(const char *) fun(String)

copy ctor optimized away

- What about conversions in the other direction, from String class to ANSI C char array?

Type Conversion (cont'd)

- Type conversion operator (type coercion)

```
class String {
public:
   ....
   String(const String &src);
   operator const char *() const;
   ....
private:
   char *m_string;
};
```

Ex. ifstream ifs("data.txt");
if (!ifs) { … }

const char*() was called in either
cout << strObj; or
cout << (const char *) strObj;
But different template libraries have different behaviors.

- Implementation

```
String::operator const char *() const {
   return m_string;
}
```

  - The function has no return type, despite the fact that it does return a const char pointer!!!

- Usage:

```
void main() {
   String strObj("hello");
   cout << strlen(strObj)  << "\n";
   cout << &strObj << " " << strObj << " " << (const char *) strObj << "\n";
}
```

Output
5
00341E60 00341E60 Hello // vc98
00341E60 Hello Hello // vc 2008,10

Unary Operator -

- Binary syntax: object1 - object2

```
Complex Complex::operator-(Complex &secondNumber) const {
   Complex tmp(m_real-secondNumber.m_real,
               m_imaginary-secondNumber.m_imaginary);
   return tmp;
}
```

- Unary syntax: -object

```
Complex Complex::operator-() const {
   return Complex(-m_real, -m_imaginary);
}
```


Miscellaneous

- most operators can be overloaded as a **class member** or a **friend function** but they cannot be overloaded as a **static class member** function
- Can every operator be overloaded?
  - No.
  - some operators can only be overloaded as **class member functions**
    assignment: **operator=** class member access: **operator->**
    subscripting **operator[]** function call: **operator()**
  - some operators can be overloaded **in neither forms**
    . **.*** **::** **?:** **sizeof**
- Can you create brand new operators?
  - No. For example, you cannot do this in C++: y:=x;

.* and ->* operators

- **Pointer to member**

Compare with
int *regular_ptr = &car.fuel;

```
class Car {
public:
   int speed;
   int fuel;
};
```

```
int main() {
   int Car::*ptr = &Car::speed;
   Car car;
   car.speed = 1;      // direct access
   cout << car.speed << endl;
```

- **Dereference** of a pointer to memeber

```
   car.*ptr = 2;       // access via pointer to member
   cout << car.speed << endl;

   Car *ptrCar = &car;
   ptrCar->*ptr = 3;      // access via pointer to member
   cout << car.speed << endl;

   ptr = &Car::fuel;
   car.fuel = 4;
   cout << car.*ptr << endl;
}
```

Output is
1
2
3
4