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.

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 + 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 item1 + 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 operator 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()

```
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;
}
```

c + z * z

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.
  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

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

```
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*=**

```
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.

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 C++ know which ++ operator you want to override?
  - Postfix syntax

```
Time Time::operator++(int) // int argument is ignored
```

  - Prefix syntax

```
Time &time::operator++()
```


operator++ (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

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;
};

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];
}
```

works as an l-value

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

- 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 prototype?

```
Account &operator=(Account &rhs);
```

No extra copy ctor invoked

Designed for continuous assignment statements

```
customer3.operator=(customer2.operator=(customer1));
```

Assignment Operator (cont'd)

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

```
Account &Account operator=(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 that stands for a function behave like a function

```
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);
}
```

Output
18

Other Uses 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.
- This is illegal, no such [][] operator

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

- 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);
```

Smart Pointers

- When you overload –>, you get a *smart pointer*
  The primary purpose of a smart pointer is to link a member function of a subobject to the main object
- Example:

```
class Person {
public:
   Person(char *name, int age)
   int getAge();
   Name *operator->();
private:
   Name *m_nameObject; // must be a pointer
   int m_age;
};
class Name {
public:
   Name(char *name);
   ~Name();
   const char *getName();
private:
   char *m_name;
};
```

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

Smart Pointers (cont'd)

- The overloaded function

```
Name *Person::operator->() {
   return m_nameObject;
}
```

- Using the smart pointer

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

  Note that *person* is not a pointer.

- Evaluating rules of a smart pointer:
  If the target is a pointer, –> operator is evaluated as it normally is.
  If it is an object with an overloaded –> operator, the object is replaced by the output of the function

```
person->getName()  ——————→  m_nameObject->getName();
```

  The process continues until evaluation occurs normally (i.e. the lhs 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 new(size_t objectSize) {
   cout << "new\n";
   return malloc(objectSize);
}
void Random::operator delete(void *object) {
   cout << "delete\n";
   free(object);
}
```

  compiler would determine suitable value for objectSize and invoke this function

- Usage:

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

  Note: after calling
  ① Random::operator new() new would invoke the ctor
  ② Random::Random(int)

  delete 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[]() 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?
  - In delete[] operator, the default implementation should try to invoke destructors for all objects. If that block of memory was allocated with new…. Error occurs
  - In delete operator, the default implementation 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();
   String(char *inputData);
   String(const String &src);
   ~String();
   const char *getString() const;
private:
   char *m_string;
};
```

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

```
void main() {
   String string1("hello");
   String string2 = "bye";   // type conversion ctor then copy ctor
}
```

- 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;
};
```

- The definition

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

  - The function has no return type, despite the fact that it does return something!!!
- Usage

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

Output
5
00341E60 Hello

Overload Unary +

- 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

- Can you overload every operator?
  - No.
  - There are some operator that cannot be overloaded

    .
    .*
    ::
    ?:
    sizeof
- Can you create new operators?
  - No. For example, you cannot do this in C++: y:=x;