Operator Overloading

Sandra Batista, Mark Redekopp, and David Kempe

1.1-1.2

PARTS OF A CLASS

Parts of a C++ Class

What are the main parts of a class?

- Data members

  What data is needed to represent the object?

- Constructor(s)

  How do you build an instance?

- Member functions

  How does the user need to interact with the stored data?

- Destructor

  How do you clean up an after an instance?

```
class GroceryList {
   public:
   GroceryList();
   GroceryList(size_t n);
   ~GroceryList();
   void addItem(string item);
   void removeItem(string item);
   void merge(const GroceryList& other);
   const string& getItem(size_t position) const;
   void printList() const;
   bool empty() const;
   size_t size()const;

   private:
   size_t num_items;
   size_t list_size;
   unique_ptr<string[]> list;
   void doubleList();
};
```

Member Functions

Member functions have access to all data members of a class

Use the dot (.) operator on object or C++ reference to access data members

Use the arrow (->) operator on pointers to objects

Use the [] operator on arrays

```
// private member function
void GroceryList::doubleList(){
   unique_ptr<string[]> old_list(list.release());
   list = make_unique<string []>(list_size*2);
   for (size_t i = 0; i < num_items; i++){
        list[i] = old_list[i];
   }
    list_size *= 2;
}
// public member function
void GroceryList::addItem( string item){
   if( num_items == list_size) doubleList();
        list[num_items] = item;
        num_items++;
}
int main()
{
  GroceryList l1;
  l1.addItem("bananas");
  return 0;
}
```

'const' Keyword for member functions

const keyword can be used with member functions to protect the object

- After a member function ensures data members aren't modified by the function

- The GroceryList object on which these member functions are called is not changed.

```
/* In GroceryList class*/
const string& getItem(size_t position) const;
 void printList() const;
 bool empty() const;
size_t size() const;
/* In GroceryList class */

size_t GroceryList::size() const {
      return num_items;
}

int main()
{
  GroceryList l1;
  size_t s = l1.size();
  return 0;
}
```

'const' Keyword for return values

const keyword can be used with return value

- Before a return value indicates the value returned cannot be modified

- The GroceryList object is not changed by this const member functions.

- The string returned is not changed by this function.

```
/* In GroceryList class*/
const string& getItem(size_t position) const;
/* In GroceryList class */

const string& GroceryList::getItem(size_t position) const{
   // Exercise: Add error checking that position is in list
   return list[position];
}

int main()
{
  GroceryList l1;
  l1.addItem("bananas");
  string s = l1.getItem(0);
  const string& s = l1.getItem(0);
  //string &s = l1.getItem(0); Will not compile

  return 0;
}
```

'const' Keyword for input parameters

const keyword can be used with input parameters

Before an input argument indicates the input object cannot be modified by the function

The GroceryList object passed as an input parameter, list2, is not changed by this function.

this pointer is a pointer to the calling object. The calling object in this example is list 1.

.

```
void GroceryList::merge(const GroceryList& other){
        size_t limit = other.size();
        for (size_t i = 0; i  < limit; i++){
          this->addItem(other.getItem(i));
        }
}


int main()
{
     GroceryList list1;
      list1.addItem("apples");
      list1.addItem("bananas");
      list1.addItem("peaches");
      GroceryList list2;
      list2.addItem("onions");
      list2.addItem("peppers");
      list2.addItem("broccoli");
       list1.merge(list2);
}
```

OPERATOR OVERLOADING

List/Array Indexing

Arrays and vectors allow indexing using square brackets: [ ]

Why won’t this compile?

What should [] do for GroceryList ?

```
#include <iostream>
#include “GroceryList.h”

int main()
{
  GroceryList shopping;
  shopping.addItem(“orange juice”);
  shopping.addItem(“Oatly”);

  cout << shopping.get(0) << endl;
  cout << shopping[0] << endl;

  return 0;
}
```

What makes up a signature of a function

- name
- number and type of arguments

No two functions are allowed to have the same signature; the following 5 functions are unique:

- void f1(int); void f1(double); void f1(List<int>&);
- void f1(int, int); void f1(double, int);

f1 is overloaded

Operator Overloading

C/C++ defines operators (+,*,-,==,etc.) that work with basic data types like int, char, double, etc.

How should operators work for newly defined classes?

- For GroceryList, what should operator+ do?
- GroceryList l1,l2;

```
l1 = l1 + l2;  // should concatenate
               // l2 items to l1?
```

Goal: Write custom functions for operators

user.h

```
class User{
 public:
   User();
   User(string n);  // Constructor
   string get_name();
 private:
   int id_;
   string name_;
};
```

user.cpp

```
#include “user.h”
User::User(string n) {
  name_ = n;
}
string User::get_name(){
  return name_;
}
```

user_test.cpp

```
#include<iostream>
#include “user.h”

int main(int argc, char *argv[]) {
  User u1(“Bill”), u2(“Jane”);
  // see if same username
  // Option 1:
  //if(u1 == u2) cout << “Same”;

  // Option 2:
  if(u1.get_name() == u2.get_name())
    {   cout << “Same” << endl; }
  return 0:
  }
```

Two Approaches

There are two ways to specify an operator overload function

- Global level function
- As a member function of the class on which it will operate

Which should we choose?

- It depends on the left-hand side operand (e.g. `string + int` or `iostream + Complex`)

Method 1: Global Functions

Define global functions with name "operator{+-…}“ with two arguments:

- LHS = Left Hand side is 1st arg
- RTH = Right Hand side is 2nd arg

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

string operator+(int time, string suf)
{
  stringstream ss;
  ss << time << “ “ << suf;
  return ss.str();
}
int main()
{
  int hour = 9;
  string suffix = "p.m.";


  string time = hour + suffix;
  // WILL COMPILE TO:
  // string time = operator+(hour, suffix);
  cout << time << endl;

  return 0;
}
```

Operator Overloading w/ Global Functions

Define global functions with name "operator{+-…}" taking two arguments:

- LHS = Left Hand side is 1st arg
- RTH = Right Hand side is 2nd arg

```
class Complex
{
 public:
  Complex() { real = 0; imag = 0; };
  Complex(double r, double i) { real = r; imag= i;
}
  double getReal() const { return real; }
  double getImag() const { return imag; }
  double real;
  double imag;
};
```

```
Complex operator+(Complex c1, Complex c2){
   Complex sum;
   sum.real = c1.real + c2.real;
   sum.imag = c1.imag + c2.imag;
   return sum;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  cout << c3.getReal() << "+" << c3.getImag() << "j" << endl;
  return 0;
}
```

Operator Overloading w/ Global Functions

What happens if Complex data members are private?

```
class Complex
{
 public:
  Complex() { real = 0; imag = 0; };
  Complex(double r, double i) { real = r; imag =
i; }
  double getReal() const { return real_; }
  double getImag() const { return imag_; }
  private:
  double real;
  double imag;
};
```

```
Complex operator+(Complex c1, Complex c2){
   Complex sum;
   sum.real = c1.real + c2.real;
   sum.imag = c1.imag + c2.imag;
   return sum;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  cout << c3.getReal() << "+" << c3.getImag() << "j" << endl;
  return 0;
}
```

Method 2: Class Members

C++ permits functions that define what an operator should do for a class

- Binary operators: +, -, *, /, ++, --
- Comparison operators:
  ==, !=, <, >, <=, >=
- Assignment: =, +=, -=, *=, /=
- I/O stream operators: <<, >>

Function name is ‘operator’ followed by operator symbol

LHS is the **this** object for which function is called

RHS is the argument

```
class Complex
{
 public:

  Complex();
  Complex(double r, double i);
  ~Complex();
  Complex operator+(const Complex &rhs);

 private:
  double real;

  double imag;
};



int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  // Same as c3 = c1.operator+(c2);
  cout << c3.real << "," << c3.imag << endl;
  // can overload '<<' so we can write:
  // cout << c3 << endl;
  return 0;
}
```

Operator Overloading Member Functions

```
class Complex
{
 public:
  Complex() { real = 0; imag = 0; };
  Complex(double r, double i) { real = r; imag = i; }
  double getReal() const { return real; }
  double getImag() const { return imag; }
  Complex operator+(const Complex& rhs) const;
 private:
  double real_;
  double imag_;
};
```

```
Complex Complex::operator+(const Complex& rhs) const
{  Complex temp;
   temp.real_ = real + rhs.real;
   temp.imag_ = imag + rhs.imag;
   return temp;
}
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + c2;
  cout << c3.getReal() << "+" << c3.getImag() << "j" << endl;
  return 0;
}
```

Operator Overloading for Classes

Why won’t main compile now?

```
class Complex
{
 public:

  Complex();
  Complex(double r, double i);
  ~Complex();
  Complex operator+(const Complex &rhs) const;

 private:
  double real;

  double imag;
};

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + 2;
  return 0;
}
```

Operator Overloading for Classes

We need to add an overloaded operator that takes a Complex number as LHS and integer as the RHS

```
class Complex
{
 public:
  Complex();
  Complex(double r, double i);
  ~Complex();
  Complex operator+(const Complex &rhs) const;
  Complex operator+(int rhs) const;

 private;
  double real, imag;
};

Complex Complex::operator+(int rhs) const
{
   Complex temp;
   temp.real = real + rhs;
   temp.imag = imag;
   return temp;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = c1 + 2;
  return 0;
}
```

Binary Operator Overloading

For binary operators, perform the operation on a new object's data members and return that object

- Purpose: do not change the data members of the input operand objects

Normal order of operations and associativity apply

Each operator can be overloaded with various RHS types and that may be necessary

Binary Operator Overloading

```
class Complex
{
 public:
  Complex();
  Complex(double r, double i);
  ~Complex()
  Complex operator+(const Complex &rhs) const;
  Complex operator+(int real) const;
 private:
  double real, imag;
};
Complex Complex::operator+(const Complex &rhs) const
{
   Complex temp;
   temp.real = real + rhs.real;
   temp.imag = imag + rhs.imag;
   return temp;
}

Complex Complex::operator+( int real ) const
{
   Complex temp = *this;
   temp.real += real;
   return temp;
}
```

No special code is needed to add 3 or more operands. The compiler chains multiple calls to the binary operator in sequence.

```
int main()
{
 Complex c1(2,3), c2(4,5), c3(6,7);

  Complex c4 = c1 + c2 + c3;
  // (c1 + c2) + c3
  // c4 = c1.operator+(c2).operator+(c3)
  //    = anonymous-ret-val.operator+(c3)


  c3 = c1 + c2;
  c3 = c3 + 5;

}
```

Adding different types (`Complex + Complex` vs. `Complex + int`) requires different overloads

Relational Operator Overloading

Relational operators can also be overloaded:
==, !=, <, <=, >, >=

These operators should return bool

```
class Complex
{
 public:
  Complex();
  Complex(double r, double i);
  ~Complex();
  Complex operator+(const Complex &rhs);
  bool operator==(const Complex &rhs);
  double real, imag;
};

bool Complex::operator==(const Complex &rhs)
{
  return (real == rhs.real && imag == rhs.imag);
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  // equiv. to c1.operator==(c2);
  if(c1 == c2)
    cout << “C1 & C2 are equal!” << endl;

  return 0;
}
```

Non-Member Friend Functions

Why won’t main compile?

```
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;
  return 0;
}
```

Non-Member Friend Functions

When the LHS argument of the operator is not the class being defined:

- The operator cannot be a member function since the LHS must be an instance of class

```
int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;
  return 0;
}
```

Define a non-member that takes in two parameters, one for LHS and one for RHS

```
Complex operator+(const int& lhs, const Complex &rhs)
{
  Complex temp;
  temp.real = lhs + rhs.real;
  temp.imag = rhs.imag;
  return temp;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;    // Calls operator+(5,c1)
  return 0;
}
```

Non-Member Friend Functions

A **friend function** is a function that is

1) not a member of the class
2) has access to the private data members of instances of that class

Put keyword '**friend**' in function prototype in class definition

```
class Complex
{
 public:
   Complex();
   Complex(double r, double i);
  ~Complex();
  // this is not a member function
  friend Complex operator+(const int&, const Complex& );
 private:
  double real, imag;
};

Complex operator+(const int& lhs, const Complex &rhs)
{
  Complex temp;
  temp.real = lhs + rhs.real;
  temp.imag = rhs.imag;
  return temp;
}

int main()
{
  Complex c1(2,3);
  Complex c2(4,5);
  Complex c3 = 5 + c1;   // Calls operator+(5,c1)
  return 0;
}
```

Why Non-Member Friend Functions?

Since an ostream is LHS, consider an operator<< member function?

Ostream class cannot define such functions for every possible class (that may ever exist)

Ostream class also does not have access to private data members of class by default

```
class Complex
{
 public:
  Complex();
  Complex(double r, double i);
  ~Complex();
  Complex operator+(const Complex &rhs);
 private:
  double real, imag;
};

int main()
{
  Complex c1(2,3);
  cout << “c1 = “ << c1;
  // cout.operator<<(“c1 = “).operator<<(c1);

  // ostream::operator<<(char *str);
  // ostream::operator<<(Complex &src);

  cout << endl;
  return 0;
}
```

Ostream Overloading

Define operator functions as friend functions where LHS is first argument and RHS the second argument.

Use non-member function so LHS can be any other class

Use friend function, so function can access private data of class

Return the ostream&

1) For chaining calls to '<<'
2) os object has changed

```
class Complex
{
 public:
  Complex(int r, int i);
  ~Complex();
  Complex operator+(const Complex &rhs);
  friend std::ostream& operator<<(std::ostream& ostr, const Complex& rhs);
 private:
  int real, imag;
};

std::ostream& operator<<(std::ostream& ostr, const Complex& rhs)
{
   ostr << rhs.real_ << "+" << rhs.imag_ << "j";
   return ostr;
}

int main()
{
  Complex c1(2,3), c2(4,5);
  cout << c1 << " " << c2;
  cout << endl;
  return 0;
}
```

Template for overloading<< with ostreams and user defined type, T:
friend ostream& operator<<(ostream &os, const T &rhs);

Member Function or Friend Non-Member Function?

Is the LHS an instance of class?

YES

NO

```
C1 objA;
objA + int
```

YES the operator overload function can be a **member function**

```
C1 objA;
int + objA
cout << objA
```

NO the operator overload function should be a **global function**
If needs access to private member data: Make **friend global function**