Lec_18.md

Lecture 18 - Operator Overloading

Oct. 21/2020

Const Modifier

We want to provide protections to X and Y when calling `X+Y`

time.cpp

```
Time & Time::operator+ (const Time & rhs) const{
  int TotalSeconds;
  Time sum;

  TotalSeconds = second + 60*minute + 3600*hour;
  TotalSeconds += rhs.second +60*rhs.minute + 3600*rhs.hour;

  sum.hour = TotalSeconds/3600;
  sum.minute = (TotalSeconds - 3600*sum.hour)/60;
  sum.second = TotalSeconds%60;

  rhs.second = 0;
  second = 0;

  return (sum);
}
```

Notes:

1. `Time & Time::operator+ (const Time & rhs) const`
    - Return *by reference*
        - However, sum is a local object -> goes out of scope

So when we pass the sum value out *by value*, the value of `sum` inside `operator+` goes out of scope

- But what defines what memory gets kept in/out of scope?
    - The compiler
- In this case, the value of `sum` is copied outside the scope of `operator+`
    - A **copy constructor** is called and a **destructor** is called
    - Certain compilers will extend/prolong the life of the `sum` object to avoid repeated copy calls
        - This is called an **optimization**, don't count on it

Complex Numbers Class

We want to design and write a C++ class that allows us to create and manipulate complex numbers

- Consists of *real part* and *imaginary part*
- Example usage:

```
complex a(5.0,8.3);
complex b(a);
complex c(3.9,17.8);
```

```
complex d;

d = a + c;
c = b - a;
a = c / d;
b = c * a;
if (b == d) c.print();
```

Notes:

1. Want to be able to add, subtract, divide, multiply
   - Overload basic operators
     - `operator+` , `operator-` , `operator/` , `operator*`
2. Need two constructors
   - Default constructor
     - Want to be able to call `complex d;`
   - Constructor that takes two floats
     - Want to call `complex c(3.9,17.8);`
3. Want to overload `==` **equality comparison** operator
4. Want **accessors/mutators** to get fields of complex values
5. Want `.print()` to print value of complex class

Additionally, if we want to use `+=` or `-=` operators on the Complex Number class

- We would have to individually **overload** these operators as well!

Class Definition

Implementing the requirements:

complex.h

```
class complex{
private:
  float real;
  float imag;
public:
  complex();
  complex(float r, float i);

  float getReal() const;
  float getImag() const;
  void setReal(float r);
  void setImag(float i);

  complex operator+ (const complex & rhs) const;
  complex operator- (const complex & rhs) const;
  complex operator* (const complex & rhs) const;
  complex operator/ (const complex & rhs) const;
  bool operator== (const complex & rhs) const;

  void print() const;
};
```

Notes:

1. Do we need to write the **copy constructor**?
   - No, C++ can provide this by default
2. Do we need to write the **default destructor**?

    - No, C++ can provide this by default
3. Do we need to overload the **assignment operator**?
    - No, C++ can provide this by default

Constructors/Destructors

```
complex::complex(){
  real = 0.0;
  imag = 0.0;
}

complex::complex(float r, float i){
  real = r;
  imag = i;
}
```

Accessors/Mutators/Printing

```
float complex::getReal() const {
  return (real);
}
float complex::getImag() const {
  return (imag);
}
void complex::setReal(float r) {
  real=r;
}
void complex::setImag(float i) {
  imag=i;
}
void complex::print() const {
  cout << “(“ << real << “,” << imag << “)”;
}
```

Notes:

- The `const` term is used in certain function definitions to avoid overwriting variables

Operator+

```
  complex complex::operator+ (const complex & rhs) const {
  complex tmp;
  tmp.real = real + rhs.real;
  tmp.imag = imag + rhs.imag;
  return tmp;
}
```

Notes:

1. Argument passed *by reference*
2. Return passed *by value*
    - We want `tmp` to last beyond scope of the function
3. This does ***not*** define behaviour for `X+(-Y)`

Operator-

```
complex complex::operator- (const complex & rhs) const {
complex tmp;
tmp.real = real - rhs.real;
tmp.imag = imag - rhs.imag;
return tmp;
}
```

Operator*

```
complex complex::operator* (const complex & rhs) const {
complex tmp;
tmp.real=(real*rhs.real)-(imag*rhs.imag);
tmp.imag=(real*rhs.imag)+(imag*rhs.real);
return tmp;
}
```

Operator/

```
complex complex::operator/ (const complex & rhs) const {
  complex tmp;
  float mag;
  mag = (rhs.real*rhs.real)+(rhs.imag*rhs.imag);
  tmp.real = (real*rhs.real)+(imag*rhs.imag);
  tmp.real = tmp.real/mag;
  tmp.imag = (imag*rhs.real)-(real*rhs.imag);
  tmp.imag = tmp.imag/mag;
  return tmp;
}
```

Operator==

Another operator we want to define is the **equality** operator.

- Returns a `bool` (primitive) type

```
bool complex::operator== (const complex & rhs) const {
  if ( (real==rhs.real) && (imag==rhs.imag) ) return (true);
  else return (false);
}
```

Notes:

1. `bool complex::operator== (const complex & rhs) const{ }`
   - This **overloaded operator** returns a `bool`
     - Not every overloaded operator needs to return an object
2. Is `real==rhs.real` and `imag==rhs.imag` recursion?
   - Are we calling `operator==` again?
   - No, the types of `real`, `rhs.real`, `imag`, `rhs.imag` are floats

Exercises

Exercise 1

```
{
  complex a(5.0, 8.3);
```

```
  complex b(a);
  complex c(3.9, 17.8);
  complex d;
  complex* p = new complex();
  p = new complex(1.8,2.9);
  if (b == d) p->print();
  delete p;
}
```

Q:

1. Which constructors are called?
   - **default constructor**, constructor w/ two floats, **copy constructor**
2. How many times is the destructor called?
   - 5
     - a, b, c, d, p
3. How many times is an object copied?
   - 1
     - `p->print()`

Exercise 2

```
{
  complex a(5.0, 8.3);
  complex b(a);
  complex c(3.9, 17.8);
  complex d;
  complex* p = new complex();
  p = new complex(1.8,2.9);
  c = b - a;
  a = c + b;
  *p = c / (*p);
  if (b == d) p->print();
  delete p;
}
```

Q:

1. Which constructors are called?
   - First write operators out in their true form
     - `c = b - a; -> c.operator=( b.operator-(a) );`
     - `a = c + b; -> a.operator=( c.operator+(b) );`
   - Figure out which constructors are called by the **overloaded operators**
2. How many times is the destructor called?
   - 11
     - This might be wrong, my personal counting
     - a, b, c, d, p
     - Inside scope of:
       - 3x `operator=`
       - `operator-`
       - `operator+`
       - `operator/`
3. How many copies of object a are made?
   - 1
     - `complex b(a);`
4. How many times is an object copied?