Lec_19.md

Lecture 19 - Operator Overloading

Oct. 22/2020

Complex Numbers Class

Using the Complex Class as an example

complex.h

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

  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. `complex(const complex & src);`
    - This is the **copy constructor**
    - Want to define the implementation for the **copy constructor**
        - Instead of using the default implementation
2. `~complex();`
    - Want to define implementation for the **destructor** as well

You can use a `waitForKey()` function to temporarily block program execution

waitForKey

```
void waitForKey(){
  string anyKey;
  cout << endl << "main: press any key to continue...";
  cin >> anyKey;
  cout << endl;
}
```

Extending the Complex Class

We want to be able to do this:

```
using namespace std;
#include <iostream>
int main() {
  complex c;
  // Would like to do this
  cout << c;
  // instead of this
  c.print();
  :
}
```

Overloaded Operators and Friends

Alternative way to **overload** operators is to use non-memnber functions

- As opposed to member functions
- **Overloaded operators** are outside the class

```
class Foo {
  private:
    int x;
  public:
    Foo(int);
    int getX() const;
    void setX(int i);
};
int main () {
  Foo a(0), b(1), c(2);
  c = a + b;
  :
}
Foo operator+(const Foo& lhs, const Foo& rhs) {
  Foo t(lhs.x + rhs.x);
  return(t);
}
```

Notes:

1. `Foo operator+(const Foo& lhs, const Foo& rhs);`
   - Notice this `operator+` takes *two* parameters
     - left hand side and right hand side
2. `Foo t(lhs.x + rhs.x);`
   - Notice that `.x` are member data fields of `lhs` and `rhs`
     - The `operator+` function cannot access these fields
       - Recall **Access Control**!

One way to fix this issue is to use accessor methods:

```
class Foo {
  private:
    int x;
  public:
    Foo(int);
    int getX() const;
    void setX(int i);
};
int main () {
  Foo a(0), b(1), c(2);
  c = a + b;
  :
}
Foo operator+(const Foo& lhs, const Foo& rhs) {
  Foo t(lhs.get_x() + rhs.get_x());
  return(t);
}
```

Notes:

1. `Foo t(lhs.get_x() + rhs.get_x());`
   - Assuming `get_x()` is an **accessor** method that returns the value of `x`

Another more elegant way to approach this

- Break access control rule
- **Friends**

Friends

Friends are **non-member functions** of a class

- Able to access **private members** of the class
- Breaking a rule
  - **Encapsulation** (Access Control)
  - Private members should be private
    - However, they can be accessed by friends

```
class Foo {
  private:
    int x;
  public:
    Foo(int);
    int getX() const;
    void setX(int i);
    friend Foo operator+(const Foo& lhs, const Foo& rhs);
};
int main () {
  Foo a(0), b(1), c(2);
  c = a + b;
  :
}
Foo operator+(const Foo& lhs, const Foo& rhs) {
  Foo t(lhs.x + rhs.x);
  return(t);
}
```

Notes:

1. `friend Foo operator+(const Foo& lhs, const Foo& rhs);`
   - You can make the `operator+` function a friend of the `foo` class
     - `foo` has to declare this friendship
2. `Foo t(lhs.x + rhs.x);`
   - Now the `operator+` function can use the `lhs.x` and `rhs.x` data fields

Overloading the << Operator

We want to be able to write:

```
complex c;
cout << c;
```

First, understanding what the `<<` operator (the **insertion operator**) does:

- The function definition for the `<<` operator:
  - `ostream& operator<<(ostream& os,int i);`
  - Must pass `ostream& os` by reference
    - Too big to copy
    - C++ wants to throw a *run-time error* instead of a *compile-time error* for passing `ostream` by value
      - Make the `ostream` **copy constructor** private.

```
int main(){
  int x;
  cout << x;
}
```

Translates into:

```
int main(){
  int x;
  operator<<(cout, x);
}
```

So the function `ostream& operator<<(ostream& os,int i);` must exist

Same thing for `floats` :

```
int main(){
  float x;
  cout << x;
}
```

Translates into:

```
int main(){
  float x;
  operator<<(cout, x);
}
```

So the function `ostream& operator<<(ostream& os,float i);` must exist

Notice that the `operator<<` function is already overloaded *for primitive data types*

- `int` , `float` , `char`
  - Can print all of these to screen Extending this for the `complex` class we have written
- We need to define:
  - `ostream& operator<<(ostream& os,const complex & x);`

complex.cpp

```
ostream& operator<<(ostream& os,const complex & x){
  os << "(" << x.real << "," << x.imag << ")";
  return os;
}
```

Note:

1. `os << "(" << x.real << "," << x.imag << ")";`
    - Uses the **insertion operator** to add to the os `ostream`
        - Print to the terminal
2. Notice that `x.real` and `x.imag` are private member data fields for the complex class

Must declare the `operator<<` function a friend inside the complex class

- Why not define this function as a member function of complex?