CSC 270 – Survey of Programming Languages

C++ Lecture 4 – More on Writing Classes

MultiFile Programs

- Programs can be divided into separate files:
  - When working with classes, the basic class definition is placed in a header file (name ending with "`.h`") and the code for the methods are placed in a C++ source file (name ending with "`.cc`")

Destructors

- There are occasions when you need to do some "cleanup" after using objects. The method that is called automatically to do this is called a ***destructor***.
- In C++, destructors have the same name as the class with a tilde (~) in front of the class's name, e.g., **`~MyClass()`**

DynArray.h

```
#include    <iostream>
using namespace std;
typedef     int   *IntPtr;
class DynArray
{
public:
      DynArray(void);
      DynArray(int numrows, int numcols);
      ~DynArray(void);
      int   getArrayMember(int i, int j);
      void setArrayMember(int value, int i, int j);
private:
      IntPtr *array;
};
```

DynArray.cc

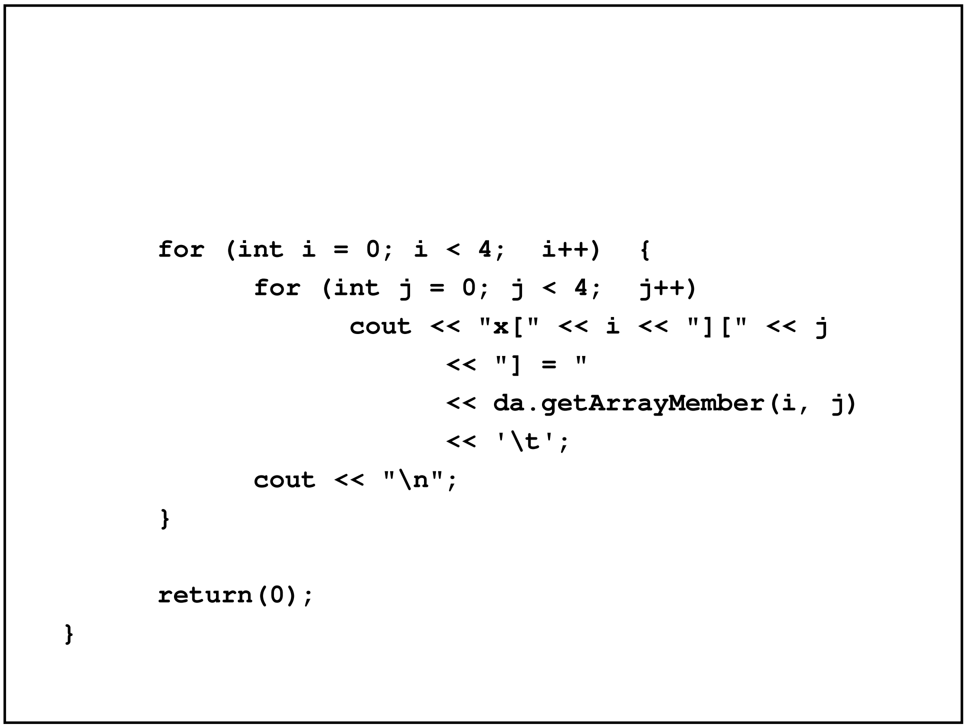
```
#include "DynArray.h"

DynArray::DynArray(void)
{
      array = new IntPtr[3];
      for (int i = 0;  i < 3; i++)
            array[i] = new int[3];
}

DynArray::DynArray(int numrows, int numcols)
{
      array = new IntPtr[numrows];
      for (int i = 0;  i < numrows; i++)
            array[i] = new int[numcols];
}
```

```
DynArray::~DynArray(void)
{
      // Free each individual row
      for (int i = 0; i < 5;  i++)
            delete [] array[i];

      // Free the array of pointers
      delete [] array;
}

int   DynArray::getArrayMember(int i, int j)
{
            return(array[i][j]);
}
```

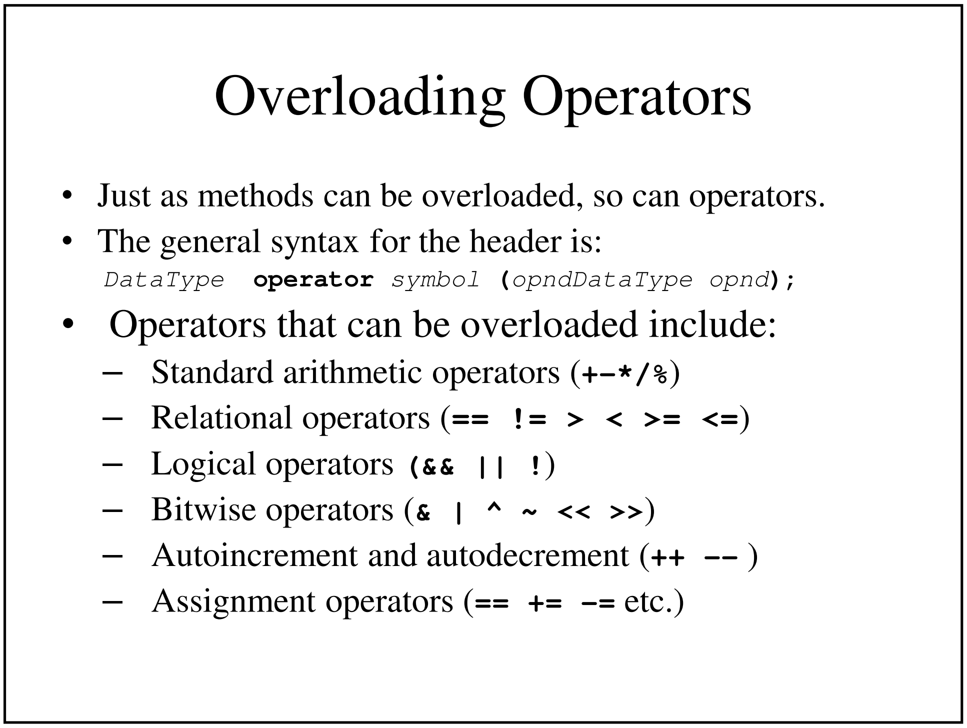
```
void DynArray::setArrayMember(int value,
                              int i, int j)
{
      array[i][j] = value;
}
```

`DynArrayDemo.cc` – the Main Program

```
#include    "DynArray.h"

int   main(void)
{
      DynArray    da(4, 4);
      int         x;
      for (int i = 0; i < 4;  i++)
            for (int j = 0; j < 4;  j++)  {
                  cin >> x;
                  da.setArrayMember(x, i, j);
            }
```

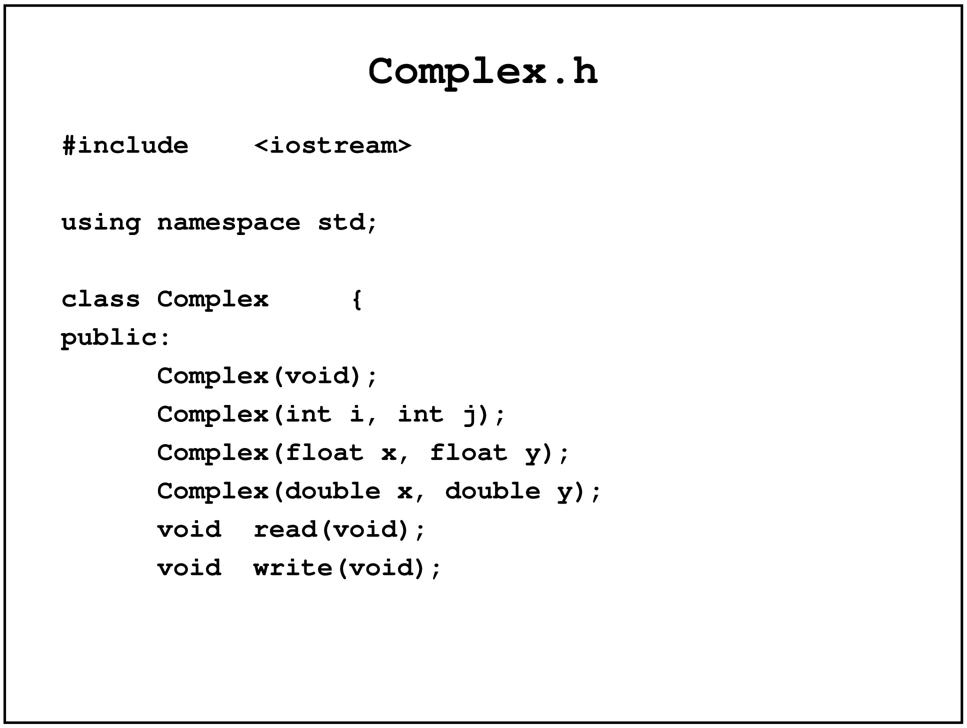


```
    for (int i = 0; i < 4;  i++)  {
          for (int j = 0; j < 4;  j++)
                cout << "x[" << i << "][" << j
                      << "] = "
                      << da.getArrayMember(i, j)
                      << '\t';
          cout << "\n";
    }

    return(0);
}
```



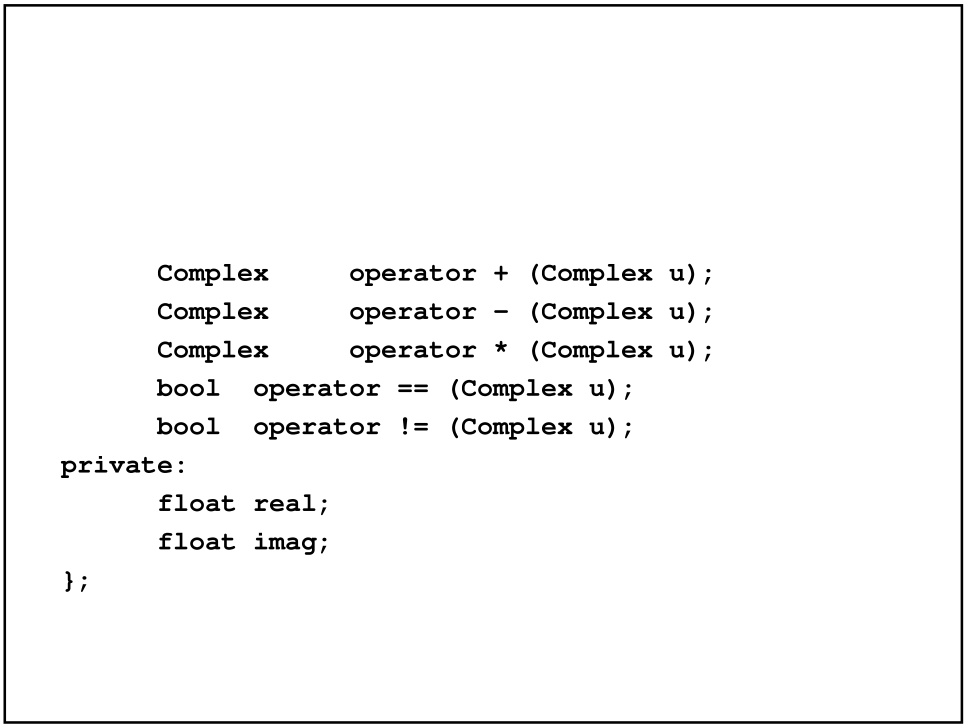
Overloading Operators

- Just as methods can be overloaded, so can operators.
- The general syntax for the header is:
  `DataType  operator symbol (opndDataType opnd);`
- Operators that can be overloaded include:
  - Standard arithmetic operators (**+-*/%**)
  - Relational operators (**== != > < >= <=**)
  - Logical operators **(&& || !**)
  - Bitwise operators (**& | ^ ~ << >>**)
  - Autoincrement and autodecrement (**++ --**)
  - Assignment operators (**== += -=** etc.)

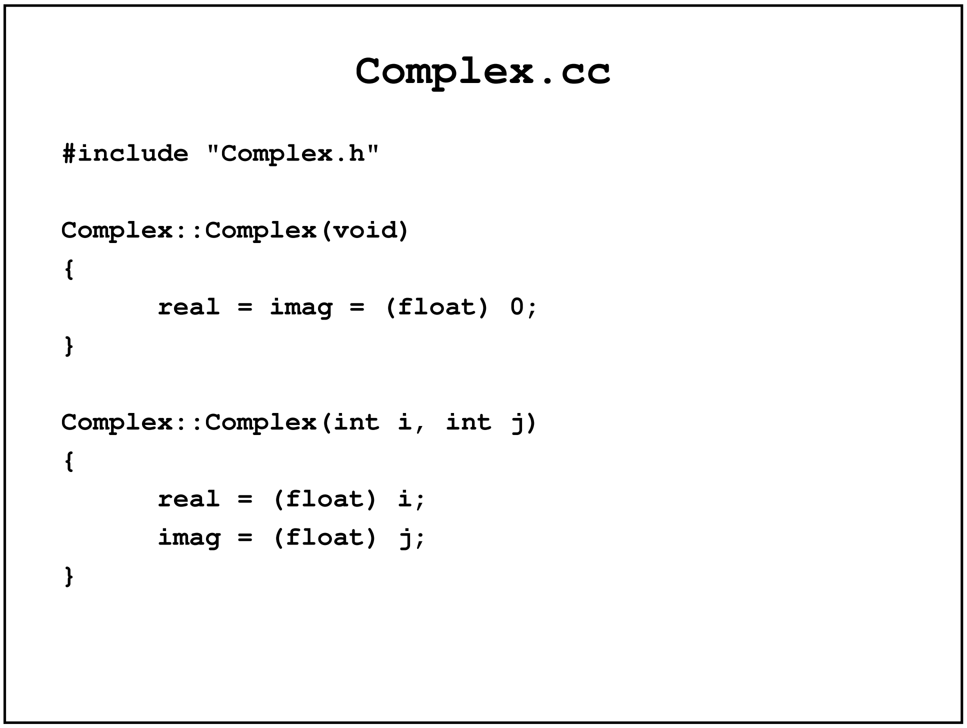


Complex.h

```
#include    <iostream>

using namespace std;

class Complex     {
public:
      Complex(void);
      Complex(int i, int j);
      Complex(float x, float y);
      Complex(double x, double y);
      void  read(void);
      void  write(void);
```

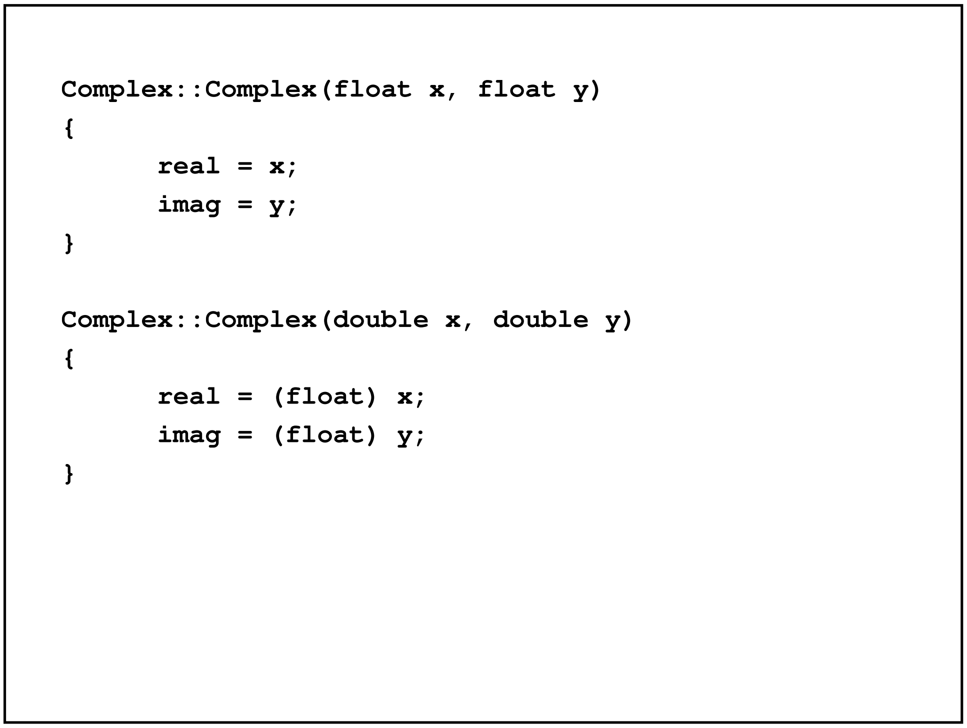


```
      Complex     operator + (Complex u);
      Complex     operator – (Complex u);
      Complex     operator * (Complex u);
      bool  operator == (Complex u);
      bool  operator != (Complex u);
private:
      float real;
      float imag;
};
```

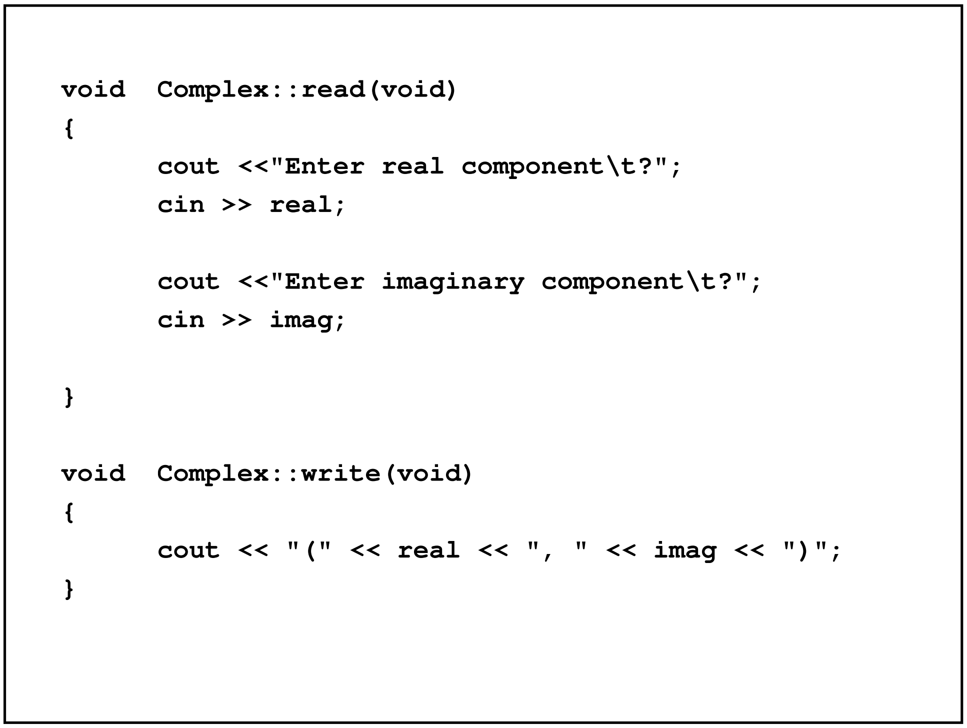


Complex.cc

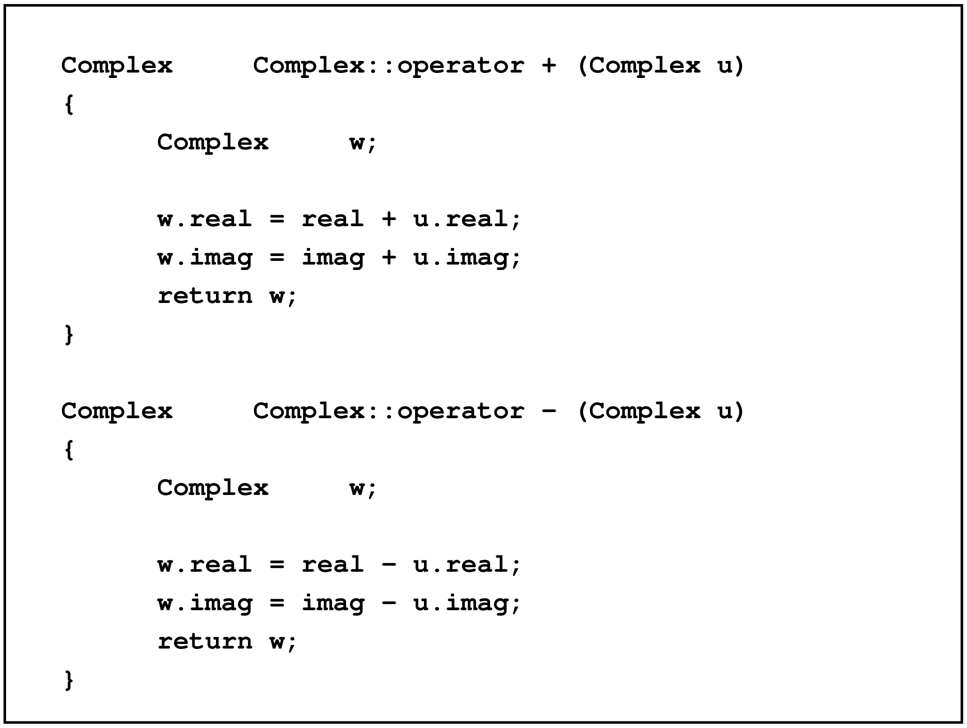
```
#include "Complex.h"

Complex::Complex(void)
{
      real = imag = (float) 0;
}

Complex::Complex(int i, int j)
{
      real = (float) i;
      imag = (float) j;
}
```



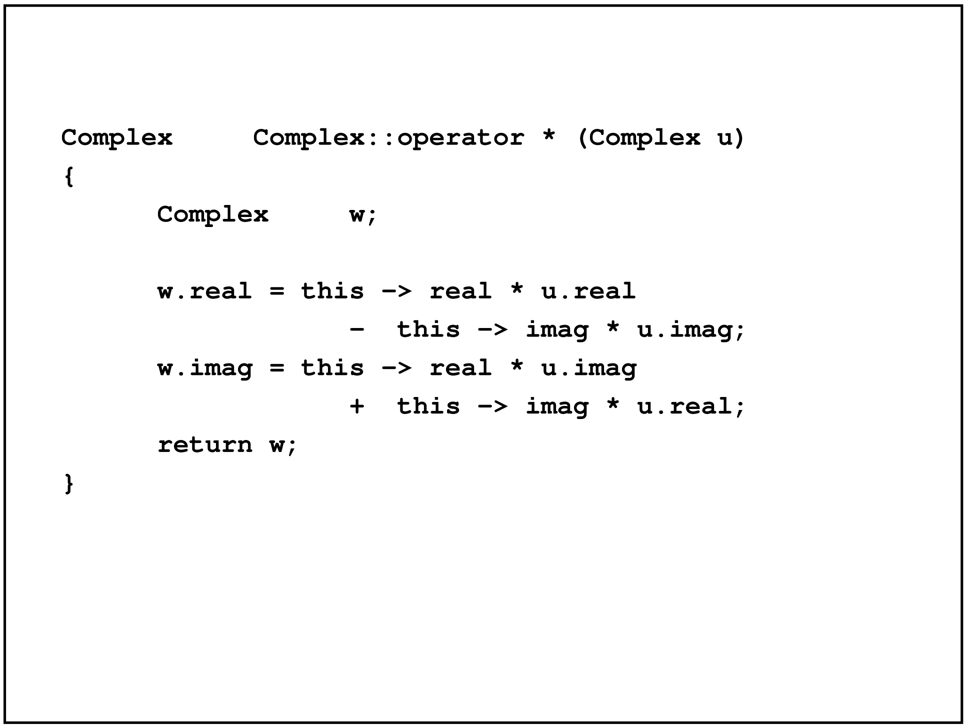
```
Complex::Complex(float x, float y)
{
      real = x;
      imag = y;
}

Complex::Complex(double x, double y)
{
      real = (float) x;
      imag = (float) y;
}
```



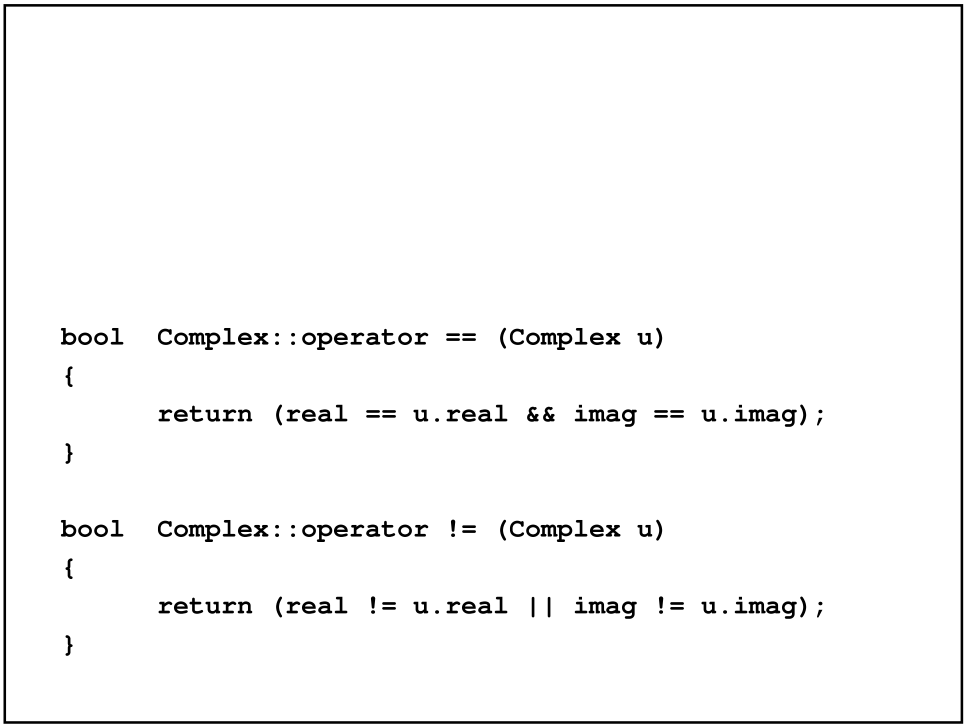
```
void  Complex::read(void)
{
      cout <<"Enter real component\t?";
      cin >> real;

      cout <<"Enter imaginary component\t?";
      cin >> imag;

}

void  Complex::write(void)
{
      cout << "(" << real << ", " << imag << ")";
}
```



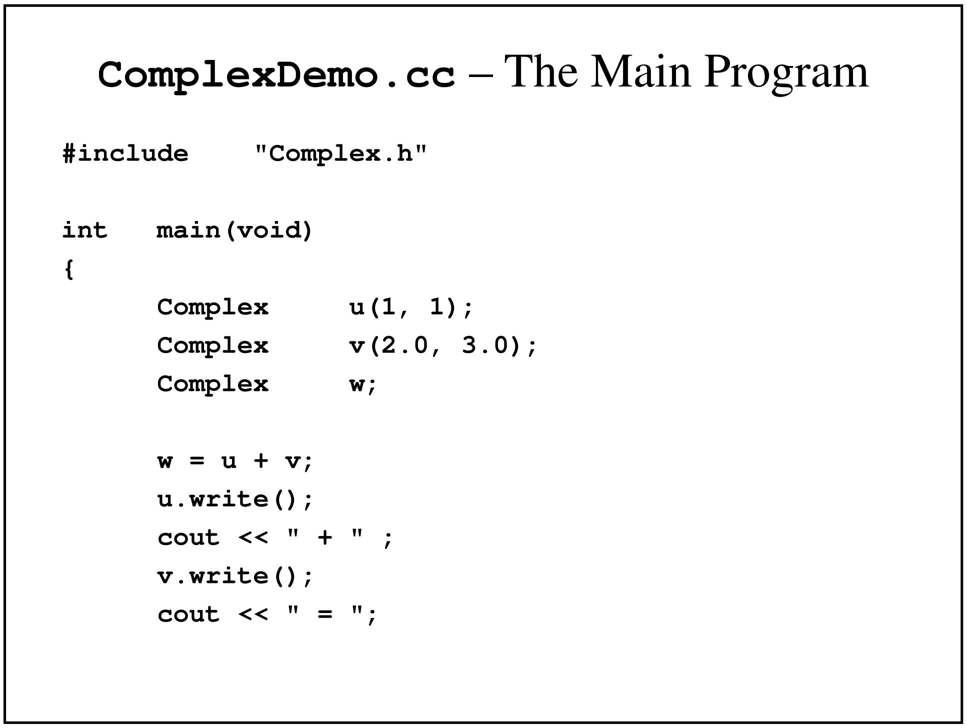
```
Complex     Complex::operator + (Complex u)
{
      Complex     w;

      w.real = real + u.real;
      w.imag = imag + u.imag;
      return w;
}

Complex     Complex::operator – (Complex u)
{
      Complex     w;

      w.real = real – u.real;
      w.imag = imag – u.imag;
      return w;
}
```



```
Complex     Complex::operator * (Complex u)
{
      Complex     w;

      w.real = this -> real * u.real
                  -  this -> imag * u.imag;
      w.imag = this -> real * u.imag
                  +  this -> imag * u.real;
      return w;
}
```

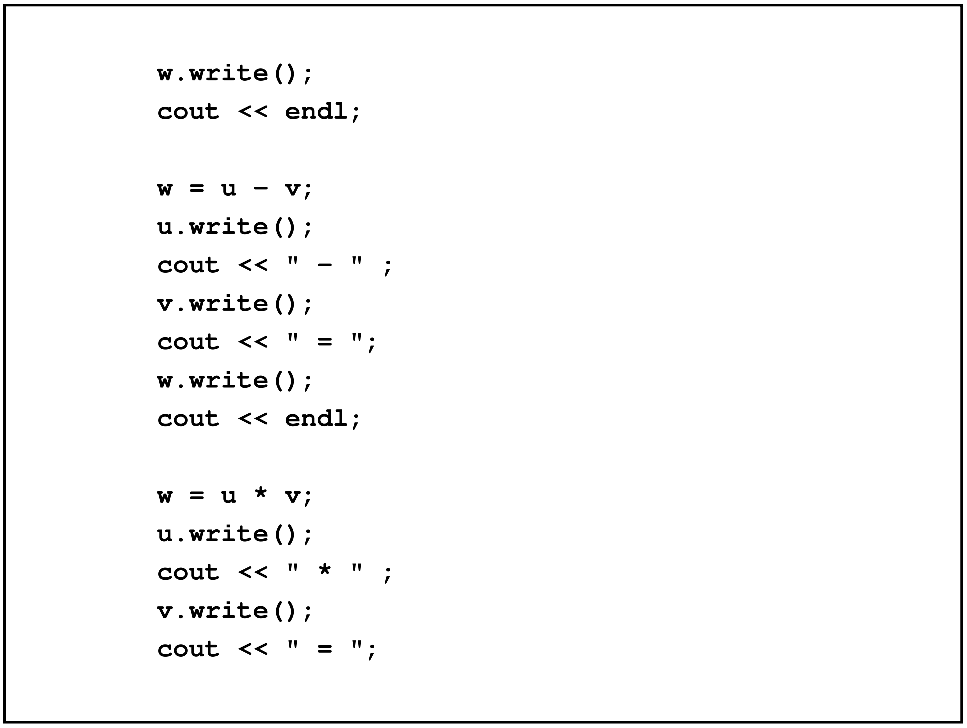


```
bool  Complex::operator == (Complex u)
{
      return (real == u.real && imag == u.imag);
}

bool  Complex::operator != (Complex u)
{
      return (real != u.real || imag != u.imag);
}
```

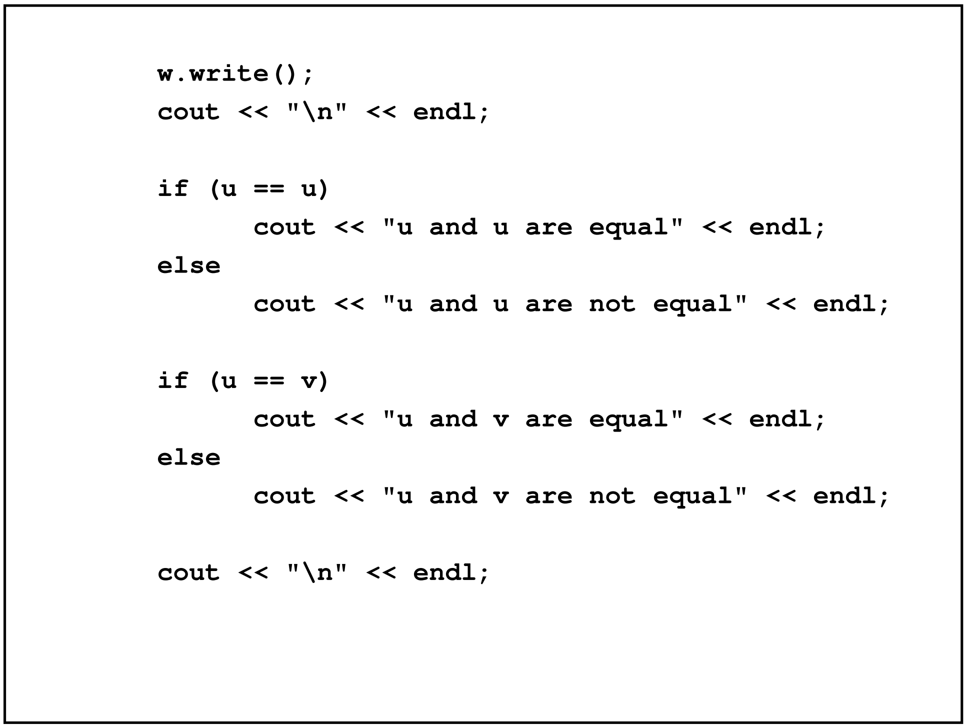


ComplexDemo.cc – The Main Program

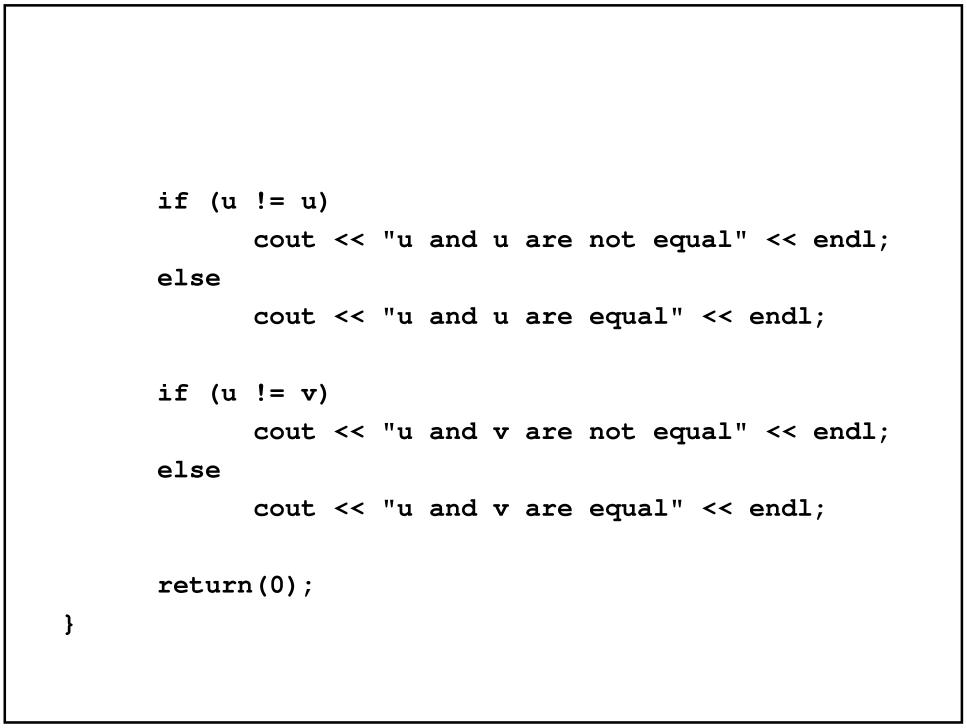
```
#include    "Complex.h"

int   main(void)
{
      Complex     u(1, 1);
      Complex     v(2.0, 3.0);
      Complex     w;

      w = u + v;
      u.write();
      cout << " + " ;
      v.write();
      cout << " = ";
```



```
      w.write();
      cout << endl;

      w = u - v;
      u.write();
      cout << " - " ;
      v.write();
      cout << " = ";
      w.write();
      cout << endl;

      w = u * v;
      u.write();
      cout << " * " ;
      v.write();
      cout << " = ";
```



```
        w.write();
        cout << "\n" << endl;

        if (u == u)
              cout << "u and u are equal" << endl;
        else
              cout << "u and u are not equal" << endl;

        if (u == v)
              cout << "u and v are equal" << endl;
        else
              cout << "u and v are not equal" << endl;

        cout << "\n" << endl;
```



```
        if (u != u)
              cout << "u and u are not equal" << endl;
        else
              cout << "u and u are equal" << endl;

        if (u != v)
              cout << "u and v are not equal" << endl;
        else
              cout << "u and v are equal" << endl;

        return(0);
  }
```