BUILDING A CLASS OF COMPLEX NUMBERS

Daniele Avitabile

University of Nottingham

In this unit we will build our first class in small incremental steps. The directory `Codes` contains material that will be used in this exercise, as well as solutions to questions. All material can be downloaded from our Moodle page.

In this unit we will

1. Recall fundamental concepts of a class
2. Familiarise with the design of a class
3. Translate mathematical operations into class members
4. Return objects as method outputs
5. Learn to override operators

QUESTION 1

Question 1

The class `ComplexNumber` written in the next slide contains the following members

1. Two double precision floating point variable `mRealPart`, `mImaginaryPart`.
2. An overridden default constructor `ComplexNumber()` which sets real and imaginary parts to 0.
3. An overridden insertion operator `<<` to print formatted output (recall Section 6.3 in Pitt-Francis&Whiteley).

Familiarise yourself with `ComplexNumber`, download the corresponding code and write a `Driver.cpp` file that produces the following output

```
Printing the complex number z1 = (0 + 0i)
```

THE INITIAL CLASS COMPLEXNUMBER

Listing 1: Codes/ComplexNumber1/ComplexNumber.hpp

```
#ifndef COMPLEXNUMBERHEADERDEF
#define COMPLEXNUMBERHEADERDEF

#include <iostream>
class ComplexNumber
{
  public:

    // Overridden default constructor
    ComplexNumber();

    // Overridden insertion operator
    friend std::ostream& operator<<(
                          std::ostream& output,
                          const ComplexNumber& z);

  private:

    // Real and imaginary parts
    double mRealPart;
    double mImaginaryPart;

};

#endif
```

Listing 2: Codes/ComplexNumber1/ComplexNumber.cpp

```
#include "ComplexNumber.hpp"

// Overridden default constructor
ComplexNumber::ComplexNumber()
{
  mRealPart = 0.0;
  mImaginaryPart = 0.0;
}

// Overridden insertion operator
std::ostream& operator<<(std::ostream& output,
                         const ComplexNumber& z)
{
  // Pretty formatting
  output << "(" << z.mRealPart << " ";
  if (z.mRealPart >= 0)
  {
    output << "+" << z.mImaginaryPart << "i";
  }
  else
  {
    output << "-" << -z.mImaginaryPart << "i";
  }
  output << ")";
}
```

QUESTION 2

Question 2 (attempt it before reading the next slide)

Let $z = re^{i\theta}$ be a complex number. Add and test the following members to `ComplexNumber` :

1. A constructor that initialises $\mathrm{Re}(z)$ and $\mathrm{Im}(x)$ (use this prototype)

```
ComplexNumber(const double x, const double y);
```

2. A method that returns the modulus r of z (use this prototype)

```
double CalculateModulus() const;
```

3. A method that returns the argument θ of z (use this prototype)

```
double CalculateArgument() const;
```

4. A method that returns the complex number z^n. You should use the de Moivre's identity $z^n = (re^{i\theta})^n = r^n[\cos(n\theta) + i\sin(n\theta)]$, the two methods `ComputeModulus`, `ComputeArgument` and the following prototype

```
ComplexNumber CalculatePower(const double n) const;
```

COMMENTS ON CALCULATEPOWER

Listing 3: From Codes/ComplexNumber2/ComplexNumber.cpp

```
1  ComplexNumber ComplexNumber::CalculatePower(const double n) const
2  {
3
4    // Retrieve modulus and arguments
5    double modulus = CalculateModulus();
6    double argument = CalculateArgument();
7
8    // Modulus and argument of z^n (using de Moivre's formula)
9    modulus = pow(modulus,n);
10   argument *= n;
11
12   // Return the new complex numbers
13   return ComplexNumber( modulus * cos(argument),
14                         modulus * sin(argument) );
15
16 }
```

Things to note

lines 5,6 `CalculateModulus` and `CalculateArgument` are accessible within `ComplexNumber`, hence we use them here.

line 13 We return an instance of `ComplexNumber` (the complex number that contains z^n)

OVERLOADING OPERATORS

Let $u, v, z \in \mathbb{C}$. The following expressions are mathematically well-defined

$$z = u, \quad z = -u, \quad z = u + w, \quad z = u - v,$$

so it would seem natural to perform similar operations in our codes

```
ComplexNumber u(1.0,2.0);
ComplexNumber w(3.0,4.0);
ComplexNumber z;
z = u;
z = -u;
z = u + w;
z = u - w;
```

Unfortunately, the operators `=`, `+`, and `-` are well defined for standard variables (for instance `double` or `int`) but they are not defined for a `ComplexNumber`, hence the code above would produce a compilation error.

However, `C++` allows to overcome this difficulty using operator overloading. For instance we can attribute a meaning to the expression `z=u` whenever `u` and `z` are `ComplexNumber`.

OVERLOADING THE ASSIGNMENT OPERATOR =

We begin by overloading the assignment operator `=`, which will be used to copy the content of `u` into `z`

```
ComplexNumber u(1.4,2.2);
ComplexNumber z;
z = u;
```

hence we add the following method to `ComplexNumber.hpp`

```
  ComplexNumber& operator=(const ComplexNumber& z);
```

and implement it in the source file `ComplexNumber.cpp`

```
ComplexNumber& ComplexNumber::
                   operator=(const ComplexNumber& z)
{
  mRealPart = z.mRealPart;
  mImaginaryPart = z.mImaginaryPart;
  return *this;
}
```

This method deserves a careful analysis, so we discuss it in depth in the next slide

OVERLOADING THE ASSIGNMENT OPERATOR =

Listing 4: From Codes/ComplexNumber3/ComplexNumber.cpp

```
1 ComplexNumber& ComplexNumber::
2                    operator=(const ComplexNumber& z)
3 {
4   mRealPart = z.mRealPart;
5   mImaginaryPart = z.mImaginaryPart;
6   return *this;
7 }
```

Things to note

Line 1 This method returns a reference to an instance of the class.

Line 2 The argument of this method is a reference to another instance of the class. This is because, by default, all method arguments are called by copy. The use of `const` guarantees that the argument won't be modified.

Lines 4-5 The real and imaginary part of the argument are “copied” to the private member of the class.

Line 6 Every `C++` object has access to its own address through an important pointer called `this`. Here we return the content of `this`, the current complex number.

Let us use the newly defined assignment operator

```
ComplexNumber u(1.4,2.2);
ComplexNumber z;
z = u;
std::cout << "u = " << u << std::endl;
std::cout << "z = " << z << std::endl;
```

When the code above is executed,

1. An object `u` is instantiated using one of the constructors. The object contains the number $1.4 + 2.2i$.
2. An object `z` is instantiated with the overloaded default constructor, so it contains the number $0 + 0i$.
3. The object `u` is passed as an argument to the method `=` of the object `z`, that is, the pointer `this` in the previous slide contains the address of `z`.
4. The content of `u` is copied into `z`.

We obtain the following output

```
u = (1.4 +2.2i)
z = (1.4 +2.2i)
```

See the full implementation in `Codes/ComplexNumber/ComplexNumber3`

QUESTION 3

Question 3

Let $u, v, z \in \mathbb{C}$. Download the code in `Codes/ComplexNumber/ComplexNumber3` and add the following members to `ComplexNumber`

1. A method that overloads the unary subtraction operator `-`, in order to perform the mathematical operation $z = -u$ (use this prototype)

```
ComplexNumber operator-() const;
```

2. A method that overloads the binary addition operator `+`, in order to perform the mathematical operation $z = u + v$ (use this prototype)

```
ComplexNumber operator+(const ComplexNumber& z) const;
```

3. A method that overloads the binary addition operator `-`, in order to perform the mathematical operation $z = u - v$ (use this prototype)

```
ComplexNumber operator-(const ComplexNumber& z) const;
```

Write a file `Driver.cpp` to test your class. You now have a fully functional `ComplexNumber` class whose modularity we will exploit in future units!