

(5 – 2) Introduction to Classes in C++

Instructor - Andrew S. O'Fallon
CptS 122 (February 19, 2021)
Washington State University

Key Concepts

- Function templates
- Defining classes with member functions
- The Rule of Three, Law of The Big Three, or The Big Three
- Constructors
 - Default and copy
- Destructors
- Setters (mutators) and getters (accessors)



Function Templates

- Overloaded functions are generally defined to perform similar operations that involve different types and/or program logic
- What happens if the program logic and operations are identical for each type?
 - Function templates may be used to more concisely overload functions



Function Template Example

```
... // template function must be placed in .h files!
template <class T>
T add (T v1, T v2)
{
    T result;
    result = v1 + v2;
    return result;
}

... // start of .cpp file!
int main (void)
{
    int n1 = 10, n2 = 20, n3 = 0;
    double d1 = 35.75, d2 = 45.5, d3 = 0.0;
    // Single function template provide capability of defining
    // a family of overloaded functions!
    n3 = add (n1, n2); // C++ generates overloaded function for integers
    d3 = add (d1, d2); // C++ generates overloaded function for doubles
}
```



Classes w/ Member Functions (I)

- Recall, in C++ we can create a user-defined type using the keyword `class`
- Also, recall, an *object* is an instantiation of a `class`
- A class consists of *data members* (attributes) and *member functions* (operations)
- A `class` controls access to its members
- Typically you cannot call a *member function* unless an object of the `class` has been instantiated
 - One exception to this rule is when you declare a member function with the keyword `static`



Classes w/ Member Functions (II)

- Classes allow the developer to separate *interfaces* from *implementation*, which is a principle of good software engineering
 - We generally place our function prototypes for member functions in the class.h file and our implementation for these in our class.cpp file
 - The function prototypes describe the classes `public` interfaces without exposing the internal implementation of the member functions



Classes w/ Member Functions (III)

- Objects can interact with each other by sending *messages*
- *Messages* are sent from one object to another by calling a method on that object
 - These methods are generally `public` member functions



Example Class ComplexNumber w/ Member Functions (I)

- Let's define the class for a complex number
- Recall, a complex number consists of a real part and imaginary part in the form:
 - $a + bi$, where a and b are real numbers, and i is the imaginary unit $i = \sqrt{-1}$
- In our design `class ComplexNumber` will consist of two data members
 - `double realPart` // we choose double because these are real numbers
 - `double imaginaryPart`



Example Class ComplexNumber w/ Member Functions (II)

- Let's add the data members to the class
 - To follow the good software engineering practice of *information hiding* we will make the data members `private`
 - `private` members may only be accessed directly by member functions of the class (or `friends`)

```
class ComplexNumber
{
    private:
        // m - represents member of a class
        double mRealPart;
        double mImaginaryPart;
}; // Don't forget the semicolon!
```



Example Class ComplexNumber w/ Member Functions (III)

- Now let's consider operations that we need to perform on the data members of the class
 - We should be able to add and subtract two complex numbers
 - We should also be able to print complex numbers in the form $a + bi$
 - We could perform each of these operations by using the C++ *operator overloading* capability, but we'll reserve that for another example



Example Class ComplexNumber w/ Member Functions (IV)

- Let's add the member functions to the class
 - The member functions will represent the well-defined interfaces to the “outside” world, thus, we'll make them `public`
 - `public` functions may be accessed by other (non-member) functions in the program as well as member functions of other classes

```
class ComplexNumber
{
    public:
        // const forces the implementation to NOT allow
        // the operand object to be modified; pass-by-ref
        // so a copy of the operand object is not made!
        ComplexNumber add (const ComplexNumber &operand);
        ComplexNumber sub (const ComplexNumber &operand);
        // Remember since print () is a member function,
        // it has access to the private data members,
        // so no parameters are required!
        void print ();

    private:
        double mRealPart; // m - represents member of a class
        double mImaginaryPart;

}; // Don't forget the semicolon!
```



Example Class ComplexNumber w/ Member Functions (V)

- Now that we've seen how to define some parts of a `class`, let's focus on how to define the definitions for the member functions
- All member functions must be associated with a `class`
 - Since we'll separate our interface (.h) from our implementation (.cpp), we'll need to use the binary scope resolution operator (`::`) to provide this association
 - Don't confuse this operator with the unary scope resolution operator!



Example Class ComplexNumber w/ Member Functions (VI)

- Let's write the definition for the `add()` member function

```
// Prototype: ComplexNumber add (const ComplexNumber &operand);

// Definition - notice the binary scope resolution operator
ComplexNumber ComplexNumber::add (const ComplexNumber &operand)
{
    // This adds the real part of the "operand" object
    // to the real part of the object that invokes the
    // call to the function; it also adds the imaginary
    // parts
    ComplexNumber result; // Declare local ComplexNumber

    // Recall we use the dot member operator (.) to access
    // members of a class; no dot (.) denotes accessing the
    // instantiated object's members; note we don't have to apply "special"
    // operators to access an object passed by reference!
    result.mRealPart = mRealPart + operand.mRealPart;
    result.mImaginaryPart = mImaginaryPart + operand.mImaginaryPart;
    // Don't want to pass back by reference; cause undefined behavior
    return result;
}
```



Example Class ComplexNumber w/ Member Functions (VII)

- Could you write the definition for the `sub()` function? Try it!



Example Class ComplexNumber w/ Member Functions (VIII)

- Let's write the definition for the `print()` member function

```
// Prototype: void print ();  
// Definition  
void ComplexNumber::print ()  
{  
    // Print in the form -> real + imaginary i  
    cout << mRealPart << " + " <<  
        mImaginaryPart << "i" << endl;  
}
```



The Rule of Three

- Also known as the Law of The Big Three or The Big Three
- The rule states that if one or more of the following are defined, then all three should be explicitly defined
 - Destructor
 - Copy constructor
 - Copy assignment operator



How to Instantiate Objects from main ()? (I)

- Continuing with our ComplexNumber example...

```
int main (void)
{
    // Instantiate three objects!
    ComplexNumber c1, c2, c3;
    // Some other code needs to be in place for this
    // to work in reality...
    c3 = c1.add (c2); // c1 invokes the add () call
    // c3 contains the result, so it invokes the
    // print () call
    c3.print ();

    return 0;
}
```



How to Instantiate Objects from `main ()`? (II)

- You should be asking yourself how do we know which values are stored in each of the `ComplexNumber` objects (`c1`, `c2`, `c3`) for each real and imaginary part
 - Right now we really don't know...most likely 0 for both data members though...
 - We need to create a means of initializing our objects
 - *Constructor* functions solve this problem for us!



Constructors for Initializing Objects! (I)

- Each `class` declared provides a *constructor* that may be used to *initialize* an object
- A *constructor* is a special member function because it **MUST** be named the same as the `class`, it cannot return a value, and it is called *implicitly* when an object is instantiated
- If a class does not *explicitly* provide a constructor, then the compiler provides a *default* constructor (a constructor with no parameters)
- Generally constructors are declared `public`
- When is an object instantiated?
 - When a variable of the type of `class` is declared
 - When the `new` operator is explicitly invoked
 - Note: `new` is used in place of `malloc ()` for C++



Constructors for Initializing Objects! (II)

- Let's add a *default* constructor to our ComplexNumber class

```
class ComplexNumber
{
    public:
        ComplexNumber (); // Default constructor
        // const forces the implementation to NOT allow
        // the operand object to be modified; pass-by-ref
        // so a copy of the operand object is not made!
        ComplexNumber add (const ComplexNumber &operand);
        ComplexNumber sub (const ComplexNumber &operand);
        // Remember since print () is a member function,
        // it has access to the private data members,
        // so no parameters are required!
        void print ();

    private:
        double mRealPart; // m - represents member of a class
        double mImaginaryPart;
}; // Don't forget the semicolon!
```



Constructors for Initializing Objects! (III)

- Let's write the definition for the *default* constructor member function

```
// Prototype: ComplexNumber ();  
// Definition  
void ComplexNumber::ComplexNumber()  
{  
    // Initialize the data members  
    mRealPart = 0.0;  
    mImaginaryPart = 0.0;  
}
```



Constructors for Initializing Objects! (IV)

- Notice the default constructor sets the real and imaginary parts to 0
- What if we want to set the parts to values other than 0?
 - We create another version of the constructor, which accepts parameters

- This implies we need to overload our constructor!

```
ComplexNumber (double real, double imaginary);
```

```
ComplexNumber::ComplexNumber (double real, double imaginary)
{
    mRealPart = real;
    mImaginaryPart = imaginary;
```



How to Initialize Objects with a Constructor?

```
int main (void)
{
    // Instantiate three objects! Use a constructor that
    // supports arguments!
    ComplexNumber c1(2.5, 3.5), c2(1.25, 5.0), c3;

    // With the addition of constructors we now know the following:
    //  $c1 = 2.5 + 3.5i$ ,  $c2 = 1.25 + 5.0i$ ,  $c3 = 0.0 + 0.0i$ 
    c3 = c1.add (c2); // c1 invokes the add () call
    // State of c3? It should be  $c3 = 3.75 + 8.5i$ 

    // c3 contains the result, so it invokes the
    // print () call
    c3.print (); // Would print  $3.75 + 8.5i$ 

    return 0;
}
```



Copy Constructor

- A *copy* constructor always accepts a parameter, which is a reference to an object of the same `class` type

```
ComplexNumber (ComplexNumber &copyObject);
```

- Copy constructors make a *copy* of an object of the same type
- A copy constructor is *implicitly* invoked when an object is *passed-by-value*!
- A *shallow* copy is made if only the data members are copied directly over to the object
- A *deep* copy is made if new memory is allocated for each of the data members
- We will explore these constructors more in the future!



Destructors

- Each `class` declared provides a *destructor*
- A *destructor* is a special member function because it **MUST** also be named the same as the `class` (with a tilde (~) in front) it cannot return a value, and it is called *implicitly* when an object is destroyed

```
~ComplexNumber ();
```

- If a class does not *explicitly* provide a destructor, then the compiler provides an “empty” destructor
- When does an object get destroyed?
 - When the object leaves scope
 - When the `delete` operator is explicitly invoked
 - Note: `delete` is used in place of `free ()` for C++



Setters and Getters

- These are `public` interfaces/functions to provide access to `private` data members
- *Setters* allow *clients* of an object to *set* or modify the data members
 - *Clients* include any statement that calls the object's member functions from *outside* the object
 - May be used to *validate* data
- *Getters* allow client to obtain/*get* a *copy* of the data members
- There generally should be 1 setter function per data member, and 1 getter function data member (of course this depends on whether or not a data member should be accessed by a client object)



References

- P.J. Deitel & H.M. Deitel, *C++: How to Program* (9th ed.), Prentice Hall, 2014
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C* (7th Ed.), Addison-Wesley, 2013



Collaborators

- Jack Hagemeister

