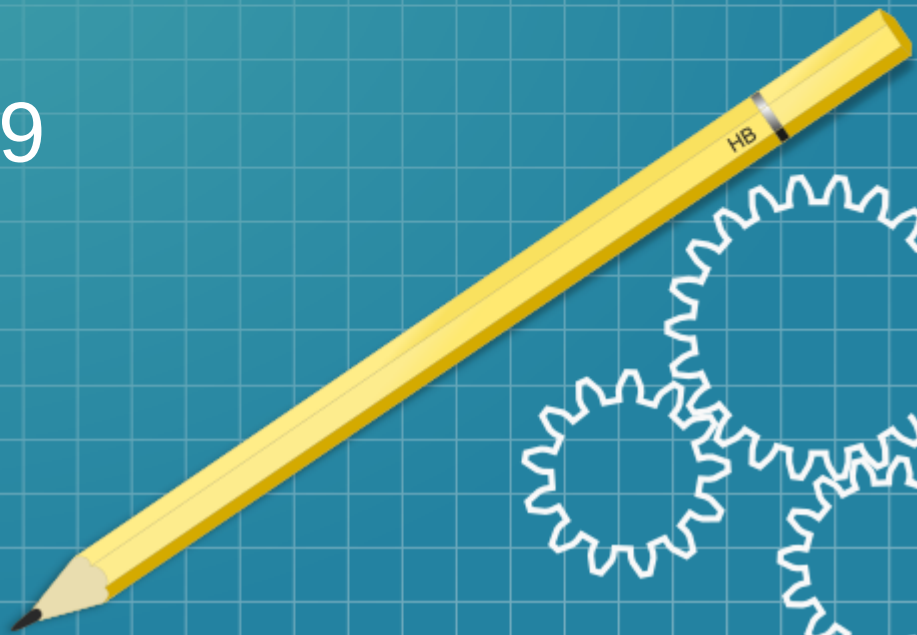
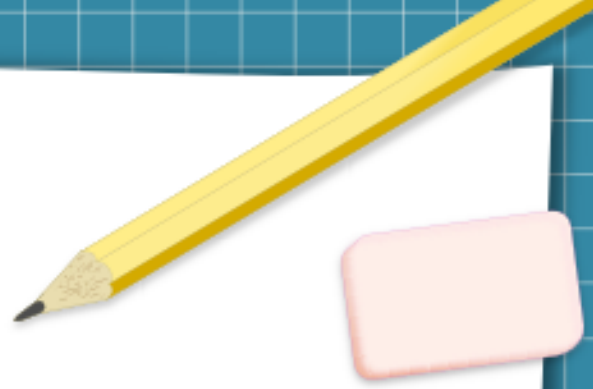
Classes: A Deeper Look (part 2)

Chapter 9



Today we will



Discuss

- return values of member functions
- copy assignment
- `const` objects and `const` Member Functions
- `friend` functions and `friend` classes
- `this` pointer

Complex numbers: returning a value

Consider the *accessor method* that returns real part:

```
double Complex::getRealPart() {
   return realPart;
}
```

Complex numbers: returning a value

Consider the *accessor method* that returns real part:

```
double Complex::getRealPart() {
   return realPart;
}
```

```
// returns reference to the real part
double& Complex::getRealPart2() {
   return realPart;
}

// returns reference to the real part
double* Complex::getRealPart3() {
   return &realPart;
}
```

See complex.h, complex.cpp, usingComplexClass.cpp

Complex numbers: returning a value

Consider the *accessor method* that returns real part:

```
double Complex::getRealPart() {
   return realPart;
}
```

The *encapsulation* of class was broken by the methods `getRealPart2()` and `getRealPart3()` : they give unrestricted and uncontrolled access to internal representation of the complex number (realPart in particular).

Complex numbers: returning a value

Consider the *accessor method* that returns real part:

```
double Complex::getRealPart() {
   return realPart;
}
```

The *encapsulation* of class was broken by the methods `getRealPart2()` and `getRealPart3()` : they give unrestricted and uncontrolled access to internal representation of the complex number (realPart in particular).

Encapsulation is when we keep the implementation details of our classes private to protect them from direct use that could complicate maintenance.
(from *Programming Principles and Practice Using C++*, by Bjarne Stroustrup) – one of many definitions

Complex numbers: returning a value

Consider the *accessor method* that returns real part:

```
double Complex::getRealPart() {
   return realPart;
}
```

The *encapsulation* of class was broken by the methods `getRealPart2()` and `getRealPart3()` : they give unrestricted and uncontrolled access to internal representation of the complex number (realPart in particular).

Encapsulation is a process of packaging some data along with the set of operations that can be performed on the data. (from *Data Structures and Algorithms Using Python and C++*, by David M. Reed and John Zelle) – one of many definitions

Default Memberwise Assignment (Copy Assignment)

Consider the following code fragment:

```
Complex d(-3, -4), a(1);

a = d;

cout << a.toString() << endl;
cout << d.toString() << endl;

cout << "modifying a ...\n";

a.setRealPart(-8);
a.setImPart(-9);

cout << a.toString() << endl;
cout << d.toString() << endl;
```

Default Memberwise Assignment (Copy Assignment)

Consider the following code fragment:

```
Complex d(-3, -4), a(1);

a = d;

cout << a.toString() << endl;
cout << d.toString() << endl;

cout << "modifying a ...\n";

a.setRealPart(-8);
a.setImPart(-9);

cout << a.toString() << endl;
cout << d.toString() << endl;
```

each data member of d is assigned individually to the same data member in the object a

See complex.h, complex.cpp, dma.cpp

Default Memberwise Assignment (Copy Assignment)

We can pass an object to a function as an argument or return an object from a function.

In such case, by default, *pass-by-value* is used, i.e. a copy of the object is passed/returned.

See complex.h, complex.cpp, dma.cpp

`const` Objects and `const` Member Functions

The statement

```
const Complex a(2,5);
```

declares a *constant object* a of type `Complex`, and initializes it to 2+5i.

`const` Objects and `const` Member Functions

The statement

```
const Complex a(2,5);
```

declares a *constant object* a of type `Complex`, and initializes it to 2+5i.

Function `double getRealPart()` can be converted to constant method: `double getRealPart() const;` which will prohibit/prevent it from modifying the data attributes of class `Complex`.

`const` Objects and `const` Member Functions

The statement

```
const Complex a(2,5);
```

declares a *constant object* a of type `Complex`, and initializes it to 2+5i.

Function `double getRealPart()` can be converted to constant method: `double getRealPart() const;` which will prohibit/prevent it from modifying the data attributes of class `Complex`.

A constructor must be a non-constant member function.

See complex2.h, complex2.cpp, constOandMF.cpp

Objects as Members of Classes

Consider a set of complex numbers …
We can find intersection, union, difference … of sets.

If we decide to define a class Set, we might consider objects of type Complex to be its members.

In this case we observe *has-a relationship*:
a class can have objects of other class as members

`friend` Functions and `friend` Classes

A `friend` function of a class is a non-member function that has the right to access the public and non-public class members.

Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

`friend` Functions and `friend` Classes

A `friend` function of a class is a non-member function that has the right to access the public and non-public class members.

Standalone functions, entire classes or member functions of other classes may be declared to be *friends* of another class.

In the next chapters we will be using it to overload certain operators, like displaying on the screen.

Recall our complex number class: wouldn't it be easier to just cout an object? `cout << a;`

See a silly example of `friend` function declaration in friendFunction.cpp

Using `this` Pointer

As we know there may be many objects of the same class/type.
How do member functions know which object to manipulate?

Using `this` Pointer

As we know there may be many objects of the same class/type.
How do member functions know which object to manipulate?
- every object has access to its own address through the pointer called `this`.

Using `this` Pointer

As we know there may be many objects of the same class/type.
How do member functions know which object to manipulate?
- every object has access to its own address through the pointer called `this`.

`this` is not a part of the object itself, so the `sizeof` operation will not reflect it in the result.

`this` pointer is passed by the compiler as an *implicit* parameter/argument to each of the object's *non-static member functions*.

Using `this` Pointer

We can use `this` pointer to avoid naming conflicts, for example:

```
void Complex::setImPart(double imaginaryPart) {

   this->imaginaryPart = imaginaryPart;
}
```

Using `this` Pointer

We can use `this` pointer to avoid naming conflicts, for example:

```
void Complex::setImPart(double imaginaryPart) {

   this->imaginaryPart = imaginaryPart;
}
```

The type of the `this` pointer depends on the type of the object and whether the member function in which this is used is declared `const`:

- in a `non-const` member function, the this pointer has the type `Complex* const` - a constant pointer to a non-constant object
- in a `const` member function, this has the type `const Complex* const` – a constant pointer to a constant object

HW assignment

1) Exercise 9.23 – from previous class meeting

Suggested exercises
(*not for grade, but the questions related to these will appear on a quiz or a test*):
1) Chapter 9, Summary and all Self-Review Exercises
2) Chapter 9, Exercise: 9.16