OBJECT-ORIENTATION IN C++

LECTURE 11-1

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN: A RETURN TO C++

We've, so far, used only the C-like features of C++. Today we look at OO.

OUTLINE:

- COMPLEX NUMBERS
- OUR HOPE FOR A COMPLEX NUMBER PACKAGE "CMPX.HH"
- CLASS DEFINITIONS IN C++
- CONSTRUCTOR AND METHOD DEFINITIONS; IMPLICIT/EXPLICIT THIS
- CLIENT CODE; LIMITING CLIENT ACCESS (PUBLIC VS. PRIVATE)
- NEXT: DESTRUCTORS, OVERLOADING, INHERITANCE, TEMPLATES

A QUICK OVERVIEW OF COMPLEX NUMBERS

Complex Numbers (Review?)

A complex number is written

$$z = a + bi$$

where

a, b are real numbers

i is the imaginary number $\sqrt{-1}$

Examples: 3 $\quad$ $3 - i$ $\quad$ $1 + 2i$ $\quad$ $4i$

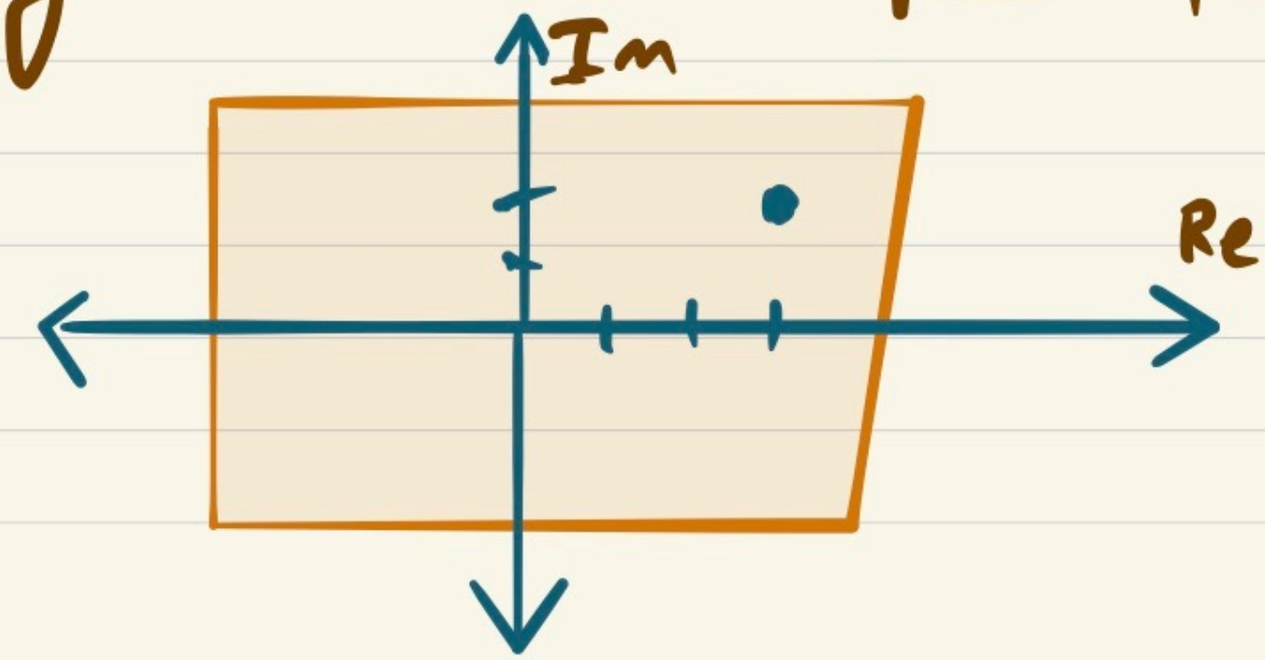
→ lots of uses in math & science
- especially physics & engrg

A QUICK OVERVIEW OF COMPLEX NUMBERS

Can be thought of as a pair of numbers

$3+2i$ is just $(3,2)$

living in the complex plane:

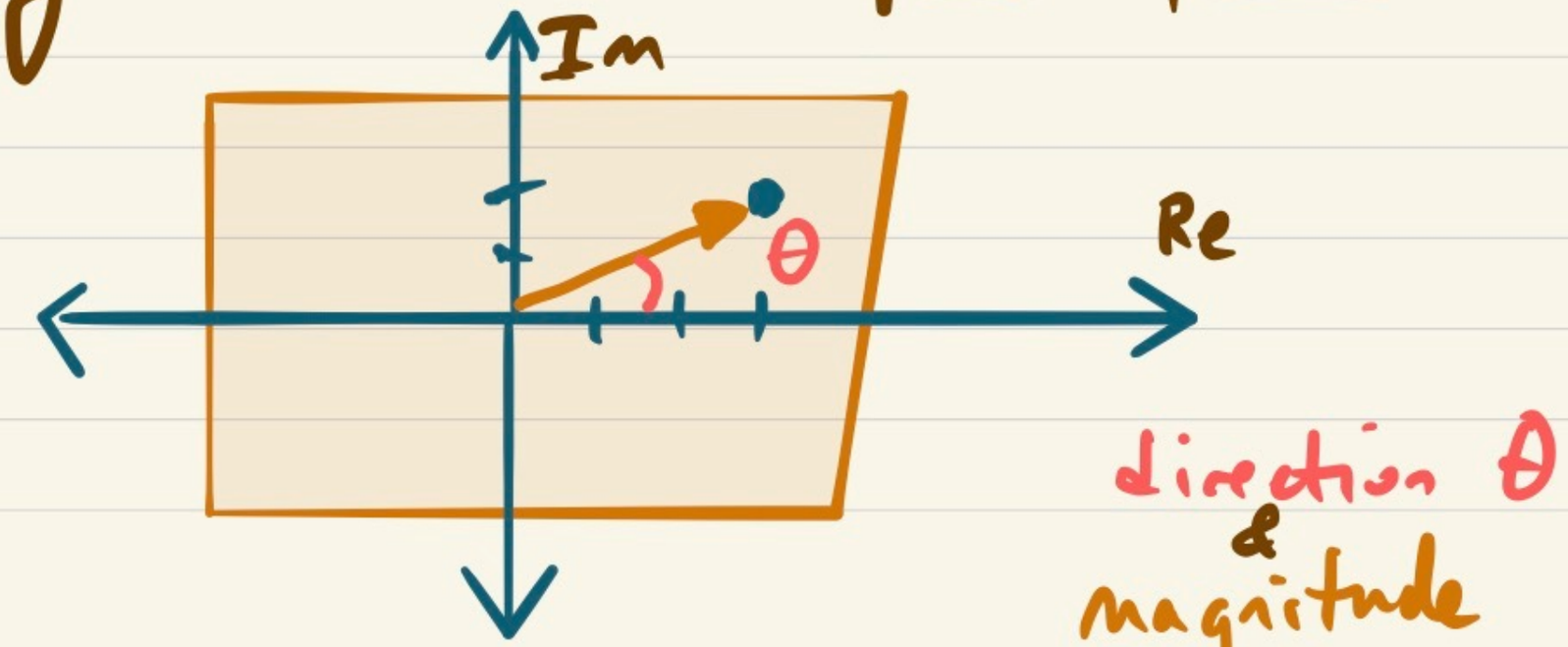


A QUICK OVERVIEW OF COMPLEX NUMBERS

Can be thought of as a pair of numbers

$3+2i$ is just $(3,2)$

living in the <u>complex plane</u>:



direction θ & magnitude

Can treat algebraically...

- have an addition operation +
- and also a multiplication operation ·

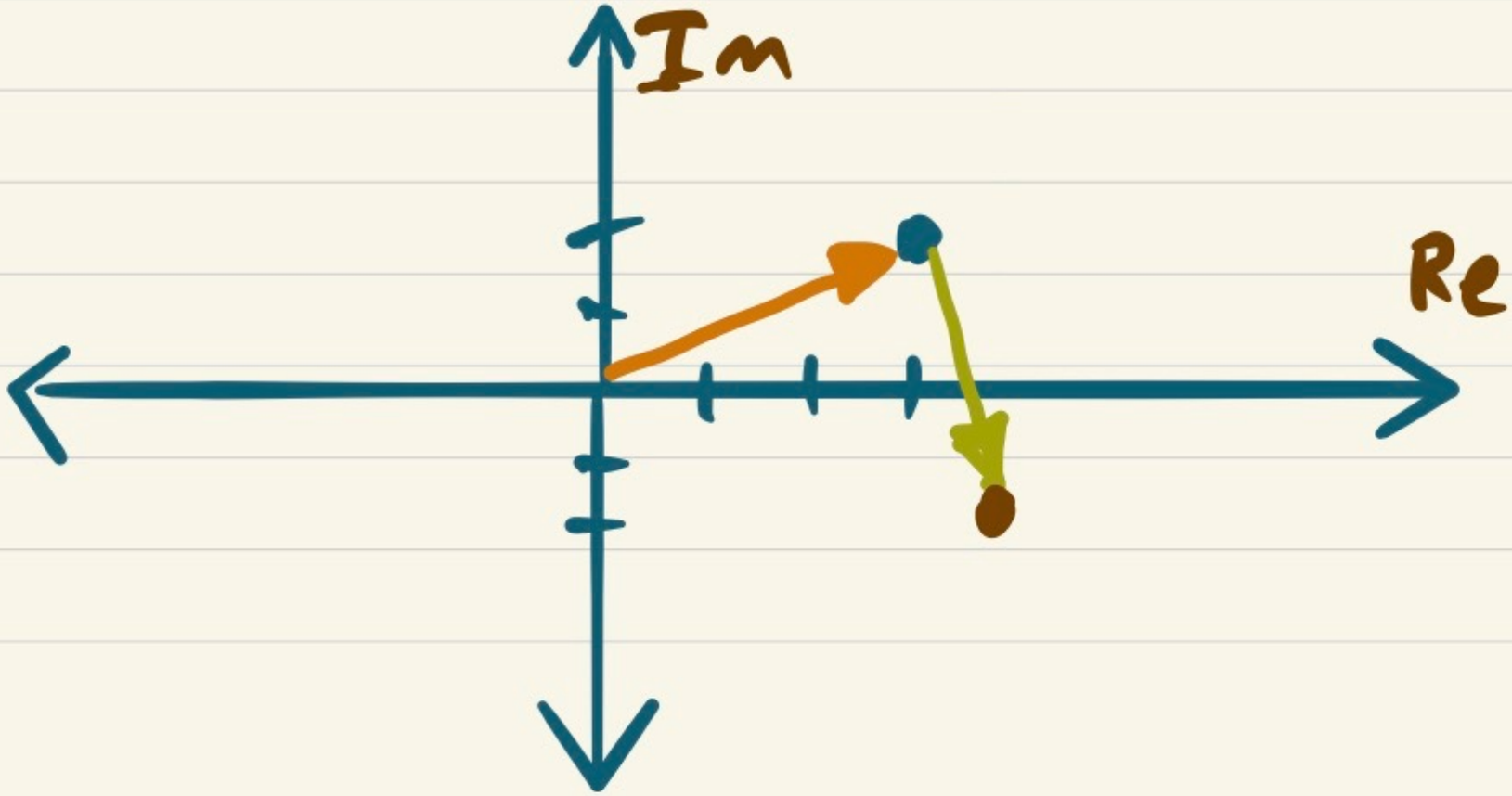
let $z_1 = a + bi$

$z_2 = c + di$

Then $z_1 + z_2$ is just $(a+c) + (b+d)i$

Addition is just vector offset.

Example: $3+2i + 1-4i = 4-2i$



And $z_1 \cdot z_2$ is given by

$(a+bi) \cdot (c+di)$ ↘ F.O.I.L.

$= ac + a \cdot di + b \cdot ci + b \cdot d \cdot i^2$

And $z_1 \cdot z_2$ is given by

$$(a+bi)\cdot(c+di) \quad \text{F.O.I.L.}$$

$$= ac + a\cdot di + b\cdot ci + b\cdot d\cdot i^2 \qquad -1$$

$$= (ac-bd) + (ad+bc)i$$

And $z_1 \cdot z_2$ is given by

$$(a+bi)\cdot(c+di) \quad \text{F.O.I.L.}$$

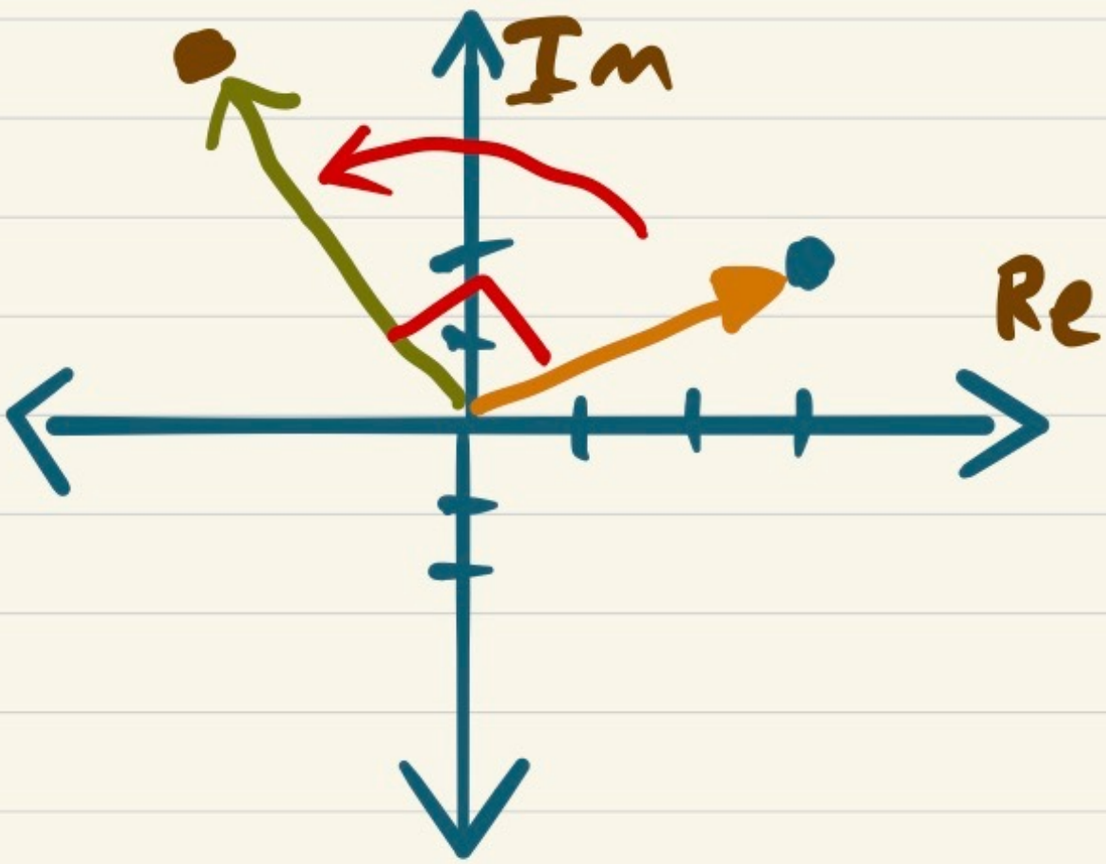
$$= ac + a\cdot di + b\cdot ci + b\cdot d\cdot i^2 \qquad -1$$

$$= (ac-bd) + (ad+bc)\,i$$

real part

imaginary part

If one number has magnitude 1, then multiplication by it rotates the other



$$(4+2i)\cdot(0+1i)$$

$$= -2+4i$$

Can also define ...

$z^* = a - bi$ conjugate

$|z|^2 = z \cdot z^*$ modulus (squared)

$\frac{1}{z} = \frac{z^*}{|z|^2}$ reciprocal

A QUICK OVERVIEW OF COMPLEX NUMBERS

Let's develop this as a new type of data in C++...

A new type cmpx.

HEADER FILE: CMPX.HH

- Here is the C++ code for implementing a new complex number type...

```
namespace cmpx {

  struct cmpx {
    double re;
    double im;
  };

  cmpx build(void);
  cmpx build(double r, double i);
  cmpx build(std::string s);

  cmpx sum(cmpx this, cmpx that);
  cmpx product(cmpx z1, cmpx z2);
  std::string to_string(cmpx z);

}
```

IMPLEMENTATION FILE: CMPX.CC

- Here are functions that operate on complex number "objects" as structs:

```
namespace cmpx {

cmpx sum(cmpx this, cmpx that) {
  cmpx z;
  z.re = this.re + that.re;
  z.im = this.im + that.im;
  return z;
}
cmpx product(cmpx z1, cmpx z2) {
  cmpx z;
  z.re = z1.re*z2.re - z1.im*z2.im;
  z.im = z1.im*z2.re + z1.re*z2.im;
  return z;
}
std::string to_string(cmpx z) {
  return std::to_string(z.re) + "+" + std::to_string(z.im) + "i";
}
```

IMPLEMENTATION FILE: CMPX.CC

- Maybe we'd write several functions that "construct" a complex number.

```
namespace cmpx {

  void parse(std::string, double& rp, double& ip) { ... }

  cmpx build(void) {
    cmpx z {0.0,0.0};
    return z;
  }

  cmpx build(double rp, double ip) {
    cmpx z {rp,ip};
    return z;
  }

  cmpx build(std::string s) {
    cmpx z;
    parse(s,z.re,z.im);
    return z;
  }
}
```

CLIENT CODE

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx::cmpx z1 = cmpx::build(6.7,2.0);
6.    cmpx::cmpx z2 = cmpx::build(6.7,-2.0);
7.
8.    cmpx::cmpx sum = cmpx::sum(z1,z2);
9.    cout << "The sum of " << cmpx::to_string(z1);
10.   cout << " and " << cmpx::to_string(z2);
11.   cout << " is " << cmpx::to_string(sum);
12.   cout << "." << endl;
13.
14.   cmpx::cmpx product = cmpx::product(z1,z2);
15.   cout << "Their product is " << cmpx::to_string(product) << endl;
16.
17.   return 0;
18. }
```

CLIENT CODE USING INITIALIZER LISTS

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx::cmpx z1 {6.7,  2.0};
6.    cmpx::cmpx z2 {6.7, -2.0};
7.
8.    cmpx::cmpx sum = cmpx::sum(z1,z2);
9.    cout << "The sum of " << cmpx::to_string(z1);
10.   cout << " and " << cmpx::to_string(z2);
11.   cout << " is " << cmpx::to_string(sum);
12.   cout << "." << endl;
13.
14.   cmpx::cmpx product = cmpx::product(z1,z2);
15.   cout << "Their product is " << cmpx::to_string(product) << endl;
16.
17.   return 0;
18. }
```

C-LIKE OBJECT CODING

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx::cmpx z1 = cmpx::build("6.7+2.0i");
6.    cmpx::cmpx z2 = cmpx::build("6.7-2.0i");
7.
8.    cmpx::cmpx sum = cmpx::sum(z1,z2);
9.    cout << "The sum of " << cmpx::to_string(z1);
10.   cout << " and " << cmpx::to_string(z2);
11.   cout << " is " << cmpx::to_string(sum);
12.   cout << "." << endl;
13.
14.   cmpx::cmpx product = cmpx::product(z1,z2);
15.   cout << "Their product is " << cmpx::to_string(product) << endl;
16.
17.   return 0;
18. }
```

SAME CODE BUT OO C++ USING CONSTRUCTORS AND METHODS

- Consider this C++ program instead:

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {"6.7 + 2.0i"};
6.    Cmpx z2 {"6.7 - 2.0i"};
7.
8.    Cmpx sum = z1.plus(z2);
9.    cout << "The sum of " << z1.to_string();
10.   cout << " and " << z2.to_string();
11.   cout << " is " << sum.to_string();
12.   cout << "." << endl;
13.
14.   Cmpx product = z1.times(z2);
15.   cout << "Their product is " << product.to_string() << endl;
16.
17.   return 0;
18. }
```

OO C++ USING OPERATORS AND HOOKS

- Or consider this C++ program:

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {"6.7 + 2.0i"};
6.    Cmpx z2 {"6.7 - 2.0i"};
7.
8.    cout << "The sum of " << z1;
9.    cout << " and " << z2;
10.   cout << " is " << z1+z2;
11.   cout << "." << endl;
12.
13.   Cmpx product = z1.times(z2);
14.   cout << "Their product is " << z1*z2 << endl;
15.
16.   return 0;
17. }
```

LONGER VERSION OF THE SAME USING OO C++

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {"6.7 + 2.0i"};
6.    Cmpx z2 {"6.7 - 2.0i"};
7.    Cmpx sum = z1.plus(z2);
8.    cout << "The sum of " << z1.to_string();
9.    cout << " and " << z2.to_string();
10.   cout << " is " << sum.to_string();
11.   cout << "." << endl;
12.   Cmpx product = z1.times(z2);
13.   cout << "Their product is " << product.to_string() << endl;
14.   Cmpz z1p = product.over(z2);
15.   cout << "Dividing out the 2nd to obtain the 1st: " << z1p.to_string() << endl;
16.   Cmpz z2p = product.over(z1);
17.   cout << "Dividing out the 1st to obtain the 2nd: " << z2p.to_string() << endl;
18.   Cmpx i {0.0,1.0};
19.   cout << "Rotating 1st by 90 degrees: " << z1.times(i).to_string() << endl;
20.   cout << "Rotating 2nd by 90 degrees: " << z2.times(i).to_string() << endl;
21.   return 0;
22. }
```

LONGER VERSION OF OO C++ USING OPERATORS AND HOOKS

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {"6.7 + 2.0i"};
6.    Cmpx z2 {"6.7 - 2.0i"};
7.
8.    cout << "The sum of " << z1;
9.    cout << " and " << z2;
10.   cout << " is " << z1+z2;
11.   cout << "." << endl;
12.
13.   Cmpx product = z1.times(z2);
14.   cout << "Their product is " << z1*z2 << endl;
15.
16.   cout << "Dividing out the 2nd to obtain the 1st: " << product/z2 << endl;
17.   cout << "Dividing out the 1st to obtain the 2nd: " << product/z1 << endl;
18.
19.   Cmpx i {0.0,1.0};
20.   cout << "Rotating 1st by 90 degrees: " << z1*i << endl;
21.   cout << "Rotating 2nd by 90 degrees: " << z2*i << endl;
22.
23.   return 0;
24. }
```

OBJECT-ORIENTATION IN C++

- GOAL: cover the key OO syntax of C++ to reach these code examples.
- Outline. explain the code in the **lb11** sample code:
  - **Cmpx.hh**: the specification "header" file for the `class Cmpx`
  - **Cmpx.cc:** the implementation of (methods) of `class Cmpx`
  - **test_cmpx.cc**: a sample client of `class Cmpx`

TL;DR OF OO C++: VERSUS PYTHON

- We declare instance variables "*statically*". (Can't be added "*dynamically*.")
- **`this`** is used instead of **`self`**. It's is not an explicit method parameter.
- Object instances don't have to be heap-allocated as pointers, but can be.
- Object instances are passed *by value*. Can also pass pointers or *by reference*.
- **`__init__`** replaced by *constructors*, one or more, including a *default* one.
- Need *destructors* because there is no garbage collector.
- Can limit access with **`public`** versus **`private`** and can have **`friend`**s.
- Can *overload* methods. Can define **`operator`** like **`+`**, and others.

SPECIFICATION (I.E. HEADER) FILE: CMPX.HH

- Here is (the start of) **Cmpx.hh**:

```
1.  class Cmpx {
2.    // instance variables
3.    double re;
4.    double im;
5.    // constructors
6.    Cmpx(void);                    // "default" constructor
7.    Cmpx(std::string);
8.    Cmpx(double rp, double ip);
9.    Cmpx(const Cmpx& that);        // "copy" constructor (later)
10.   // methods
11.   Cmpx plus(Cmpx that);
12.   Cmpx times(Cmpx that);
13.   std::string to_string();
14. };
```

SPECIFICATION (I.E. HEADER) FILE: CMPX.HH

- Here is (the start of) **Cmpx.hh**:

```
1.  class Cmpx {
2.    // instance variables
3.    double re;
4.    double im;
5.    // constructors
6.    Cmpx(void);                    // "default" constructor
7.    Cmpx(std::string);
8.    Cmpx(double rp, double ip);
9.    Cmpx(const Cmpx& that);        // "copy" constructor (later)
10.   // methods
11.   Cmpx plus(Cmpx that);
12.   Cmpx times(Cmpx that);
13.   std::string to_string();
14. };
```

IMPLEMENTATION FILE: CMPX.CC

- Here are the key method definitions within **Cmpx.cc**:

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  Cmpx z {rp,ip};
  return z;
}
Cmpx Cmpx::times(Cmpx that) {
  double rp = this->re*that.re - this->im*that.im;
  double ip = this->im*that.re + this->re*that.im;
  Cmpx z {rp,ip};
  return z;
}
std::string Cmpx::to_string() { // basic version
  return std::to_string(this->re)+"+"+std::to_string(this->im)+"i";
}
```

IMPLEMENTATION FILE: CMPX.CC

- Here those are again, but without explicitly using the **`this`** pointer:

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = re + that.re;
  double ip = im + that.im;
  Cmpx z {rp,ip};
  return z;
}
Cmpx Cmpx::times(Cmpx that) {
  double rp = re*that.re - im*that.im;
  double ip = im*that.re + re*that.im;
  Cmpx z {rp,ip};
  return z;
}
std::string Cmpx::to_string() { // basic version
  return std::to_string(re) + "+" + std::to_string(im) + "i";
}
```

IMPLEMENTATION FILE: CMPX.CC

- Here are the constructor definitions within **Cmpx.cc**:

```
Cmpx::Cmpx(void) {
  this->re = 0.0;
  this->im = 0.0;
}
Cmpx::Cmpx(double rp, double ip) {
  this->re = rp;
  this->im = ip;
}
Cmpx::Cmpx(std::string s) {
  parseCmpx(s,this->re,this->im);
}
Cmpx::Cmpx(const Cmpx& that) {
  this->re = that.re;
  this->im = that.im;
}
```

IMPLEMENTATION FILE: CMPX.CC

- Here are the constructors without use of **`this`**:

```
Cmpx::Cmpx(void) {
  re = 0.0;
  im = 0.0;
}
Cmpx::Cmpx(double rp, double ip) {
  re = rp;
  im = ip;
}
Cmpx::Cmpx(std::string s) {
  parseCmpx(s,re,im);
}
Cmpx::Cmpx(const Cmpx& that) {
  re = that.re;
  im = that.im;
}
```

IMPLEMENTATION FILE: CMPX.CC

- Here are the constructors using *initializers*:

```
Cmpx::Cmpx(void) :
  re {0.0}, im {0.0}
{ }

Cmpx::Cmpx(double rp, double ip) :
  re {rp}, im {ip}
{ }

Cmpx::Cmpx(std::string s) {
  parseCmpx(s,re,im);
}

Cmpx::Cmpx(const Cmpx& that) :
  re {that.re}, im {that.im}
{ }
```

IMPLEMENTATION FILE: CMPX.CC

- Here are the constructors using *initializers*, having one use aother.

```
Cmpx::Cmpx(double rp, double ip) :
  re {r}, im {i}
{ }

Cmpx::Cmpx(void) :
  Cmpx {0.0, 0.0}
{ }

Cmpx::Cmpx(std::string s) {
  parseCmpx(s,re,im);
}

Cmpx::Cmpx(const Cmpx& that) :
  re {that.re}, im {that.im}
{ }
```

CLIENT CODE FILE: TEST_CMPX.CC

- Here is use of the {string} constructor:

```
#include <iostream>
#include "Cmpx.hh"
int main() {
  Cmpx z1 {"6.7 + 2.0i"}; // Uses a std::string argument.
  Cmpx z2 {"6.7 - 2.0i"};

  Cmpx sum = z1.plus(z2);
  cout << "The sum of " << z1.to_string();
  cout << " and " << z2.to_string();
  cout << " is " << sum.to_string();
  cout << "." << endl;

  Cmpx product = z1.times(z2);
  cout << "Their product is " << product.to_string() << endl;
}
```

CLIENT CODE FILE: TEST_CMPX.CC

- Here is the {double,double} constructor:

```
#include <iostream>
#include "Cmpx.hh"
int main() {
  Cmpx z1 {6.7,2.0};      // Uses constructor that takes two doubles.
  Cmpx z2 {6.7,-2.0};

  Cmpx sum = z1.plus(z2);
  cout << "The sum of " << z1.to_string();
  cout << " and " << z2.to_string();
  cout << " is " << sum.to_string();
  cout << "." << endl;

  Cmpx product = z1.times(z2);
  cout << "Their product is " << product.to_string() << endl;
}
```

NOTE ON C++ LANGUAGE VERSIONS

- The syntax of initializers within constructors, and also the initializer lists in client calls to constructors, were introduced later in C++.
- Need to compile with an extra flag on the command line:

```
g++ -std=c++11 -o test_cmpx test_cmpx.cc Cmpx.cc
```

CLIENT CODE SYNTAX: FIELD ACCESS AND METHOD INVOCATION

Syntax to access the instance variable of an object:

object-expression **.** *instance-variable-name*

- Examples: `that.re` `(z1.plus(z2)).im`

Syntax to invoke a method on an object instance:

object-expression **.** *method-name* **(** *argument-expressions* **)**

- Examples: `z1.plus(z2)` `sum.to_string()` `shape.draw(BLUE,0.5,1.6)`
  - NOTE: the address of the object of *object-expression* will be **`this`** when the method's code runs (it will be a pointer to that objects's struct)

BUT MORE IS GOING ON HERE

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {6.7,2.0};
6.    Cmpx z2;
7.    z2.re = 6.7;
8.    z2.im = -2.0;
9.
10.   Cmpx sum = z1.plus(z2);
11.   cout << "The sum of " << z1.to_string();
12.   cout << " and " << z2.to_string();
13.   cout << " is " << sum.to_string();
14.   cout << "." << endl;
15.
16.   Cmpx product = z1.times(z2);
17.   cout << "Their product is " << product.to_string() << endl;
18.
19.   return 0;
20. }
```

BUT MORE IS GOING ON HERE... CONSTRUCTORS

```
1.  #include <iostream>
2.  #include "Cmpx.hh"
3.
4.  int main() {
5.    Cmpx z1 {"6.7 + 2.0i"};
6.    Cmpx z2 = Cmpx(6.7,-2.0);
7.
8.    Cmpx sum = z1.plus(z2);
9.    cout << "The sum of " << z1.to_string();
10.   cout << " and " << z2.to_string();
11.   cout << " is " << sum.to_string();
12.   cout << "." << endl;
13.
14.   Cmpx product = z1.times(z2);
15.   cout << "Their product is " << product.to_string() << endl;
16.
17.   return 0;
18. }
```

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name **;**

class-name variable-name **(** *constructor-arguments* **);**

class-name variable-name **{** *constructor-arguments* **};**

class-name variable-name **=** *class-name* **(** *constructor-arguments* **};**

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
```

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name **;**

class-name variable-name **(** *constructor-arguments* **);**

class-name variable-name **{** *constructor-arguments* **};**

class-name variable-name **=** *class-name* **(** *constructor-arguments* **};**

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
```

- Each one calls a *constructor* that initializes the object's struct data.

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name **;**

class-name variable-name **(** *constructor-arguments* **);**

class-name variable-name **{** *constructor-arguments* **};**

class-name variable-name **=** *class-name* **(** *constructor-arguments* **};**

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
```

- The bottom three each call a constructor that takes two double parameters.

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name **;**

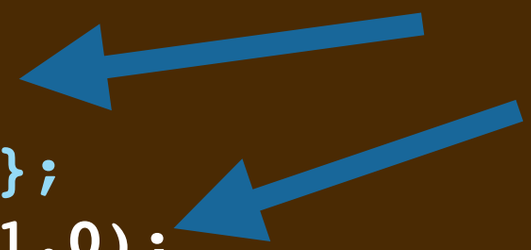
class-name variable-name **(** *constructor-arguments* **);**

class-name variable-name **{** *constructor-arguments* **};**

class-name variable-name **=** *class-name* **(** *constructor-arguments* **};**

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
```



- The parenthesized ones use early C++ standards' syntax.

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name **;**

class-name variable-name **(** *constructor-arguments* **);**

class-name variable-name **{** *constructor-arguments* **};**

class-name variable-name **=** *class-name* **(** *constructor-arguments* **};**

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
```

- The **`z2`** one uses *initializer list* syntax introduced in C++11.

CLIENT CODE SYNTAX: STACK VARIABLE OBJECT INSTANTIATION

Syntax to create an object (variable) allocated on the stack:

class-name variable-name ;

class-name variable-name (*constructor-arguments*) ;

class-name variable-name { *constructor-arguments* } ;

class-name variable-name = *class-name* (*constructor-arguments*) ;

class-name variable-name = *class-name* { *constructor-arguments* } ;

Examples in client code:

```
Cmpx z;
Cmpx z1(6.07,2.0);
Cmpx z2 {6.07,-2.0};
Cmpx i = Cmpx(0.0,1.0);
Cmpx i = Cmpx {0.0,1.0};
```

- Initializer list syntax is encouraged by the language inventor, B. Stroustrup.

SYNTAX OF A CLASS DEFINITION

Syntax of a class definition:

`class` *class-name* **`{`** *declarations-and-signatures* **`};`**

➡NOTE: normally in a header file named "*class-name*`.hh`"

Example:

```
class Cmpx {
  double re;
  double im;
  Cmpx();
  Cmpx(double rp, double ip);
  Cmpx plus(Cmpx that);
  std::string to_string();
};
```

SYNTAX OF A CLASS DEFINITION

Syntax of a class definition:

`class` *class-name* **`{`** *declarations-and-signatures* **`};`**

➡NOTE: normally in a header file named "*class-name*`.hh`"

Example:

```
class Cmpx {
  double re;
  double im;
  Cmpx();
  Cmpx(double rp, double ip);
  Cmpx plus(Cmpx that);
  std::string to_string();
};
```

instance variable declarations

SYNTAX OF A CLASS DEFINITION

Syntax of a class definition:

`class` *class-name* **`{`** *instance-variable-and-method-declarations* **`};`**

➡NOTE: normally in a header file named "*class-name*`.hh`"

Example:

```
class Cmpx {
  double re;
  double im;
  Cmpx();
  Cmpx(double rp, double ip);
  Cmpx plus(Cmpx that);
  std::string to_string();
};
```

constructor signatures

SYNTAX OF A CLASS DEFINITION

Syntax of a class definition:

`class` *class-name* **`{`***instance-variable-and-method-declarations* **`};`**

➡NOTE: normally in a header file named "*class-name*`.hh`"

Example:

```
class Cmpx {
  double re;
  double im;
  Cmpx();
  Cmpx(double rp, double ip);
  Cmpx plus(Cmpx that);
  std::string to_string();
};
```

method signatures

SPECIFICATION FILE THAT PREVENTS "DOUBLE INCLUSION"

```
1.  #ifndef __CMPX_HH
2.  #define __CMPX_HH
3.  class Cmpx {
4.    // instance variables
5.    double re;
6.    double im;
7.    // constructors
8.    Cmpx(void);
9.    Cmpx(std::string);
10.   Cmpx(double rp, double ip);
11.   Cmpx(const Cmpx& that);
12.   // methods
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   std::string to_string();
16. };
17. #endif
```

SYNTAX OF A CLASS DEFINITION: INSTANCE VARIABLES

Syntax of a class definition:

`class` *class-name* **`{`** *instance-variable-and-method-declarations* **`};`**

Syntax of an instance variable declaration:

type instance-variable-name **`;`**

- Example:

```
class Cmpx {
  double re;
  double im;
  ...
};
```

  - ➡NOTE: looks just like the **`struct`** syntax.

SYNTAX OF A METHOD SIGNATURE

return-type method-name **(** *parameter-declarations* **);**

- Example:

```
class Cmpx {
  ...
  Cmpx plus(Cmpx that);
  std::string to_string();
};
```

➡Unlike Python, there is no explicit receiver parameter (**`self`**).

➡We'll see that there is an implicit **`this`** which acts like Python's **`self`**

SYNTAX OF A CONSTRUCTOR SIGNATURE

class-name **(** *parameter-declarations* **);**

- Example:

```
class Cmpx {
  ...
  Cmpx();
  Cmpx(double rp, double ip);
  ...
};
```

➡ Like a method signature but

- ✦ no return type
- ✦ named after the class.

➡ Used when the client introduces a stack object (variable) or allocates one on the heap with **`new`**.

DEFAULT CONSTRUCTOR

The signature with no parameters is the *default constructor* declaration.

class-name `();`

- Example:

```
class Cmpx {
  ...
  Cmpx();
  Cmpx(double rp, double ip);
  ...
};
```

SYNTAX OF A METHOD DEFINITIONS

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

statements

}

- Examples:

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
std::string Cmpx::to_string() {
  std::string s1 = std::to_string(this->re);
  std::string s2 = std::to_string(this->im);
  return ss1 + "+" + s2 + "i";
}
```

➡ Normally in an implementation file named "*class-name*`.cc`"

SYNTAX OF A METHOD DEFINITIONS

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

statements

}

- Examples:

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
std::string Cmpx::to_string() {
  std::string s1 = std::to_string(this->re);
  std::string s2 = std::to_string(this->im);
  return ss1 + "+" + s2 + "i";
}
```

➡ Looks like a function definition put in a namespace.

SYNTAX OF A METHOD DEFINITION

return-type *class-name* **::** *method-name* **(** *parameter-declarations* **)** **{**

    statements

}

- Examples:

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
std::string Cmpx::to_string() {
  std::string s1 = std::to_string(this->re);
  std::string s2 = std::to_string(this->im);
  return ss1 + "+" + s2 + "i";
}
```

SYNTAX OF A METHOD DEFINITION

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

statements

}

- Example:

```
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
```

- Note: a method is a client of its own class.
- It might use dot notation to access components, too.
- It might use its own constructors.

SYNTAX OF A METHOD DEFINITION

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

    statements

}

- Example:

```
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
```

- Note: a method is a client of its own class.
- It might use dot notation to access components, too.
- It might use its own constructors.

SYNTAX OF A METHOD DEFINITION

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

    statements

}

- Example:

```
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
```

- Note: a method is a client of its own class.
- It might use dot notation to access components, too.
- It might use its own constructors.

RECEIVER ACCESS

return-type class-name **::** *method-name* **(** *parameter-declarations* **)** **{**

statements that access the **this** *pointer*

}

- Example:

```
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}
```

- It needs access to the receiving object (recall: **self** in Python).
- Method has access to a(n implicitly defined) pointer variable **this**.
- (It would be declared as **Cmpx* this;** to be used here.)

POINTER FIELD ACCESS AND METHOD INVOCATION

Syntax to access the instance variable of an object:

pointer-to-instance **->** *instance-variable-name*

- Examples: `this->re` `D->size`

Syntax to invoke a method on an object instance:

pointer-to-instance **->** *method-name* **(** *argument-expressions* **)**

- Examples: `q->dequeue()` `this->times(that)`
  - ➡NOTE: these are equivalent to the dereference notation

    (*pointer-to-instance* **->** *method-name* **(** *argument-expressions* **)**

POINTER FIELD ACCESS AND METHOD INVOCATION (CONT'D)

These are equivalent to the *dereference* notation that uses *****:

(***** *pointer-to-instance* **)** **.** *instance-variable-name*

(***** *pointer-to-instance* **)** **.** *method-name* **(** *argument-expressions* **)**

- Examples:

```
(*this).re      (*D).size
(*q).dequeue()  (*this).times(that)
```

USE OF THE THIS POINTER

- It turns out that you don't need to use **`this`** to access the receiver's instance variables...

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}

std::string Cmpx::to_string() {
  std::string s1 = std::to_string(this->re);
  std::string s2 = std::to_string(this->im);
  return ss1 + "+" + s2 + "i";
}
```

USE OF THE THIS POINTER

- It turns out that you don't need to use **`this`** to access the receiver's instance variables...

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = this->re + that.re;
  double ip = this->im + that.im;
  return Cmpx(rp,ip);
}

std::string Cmpx::to_string() {
  std::string s1 = std::to_string(this->re);
  std::string s2 = std::to_string(this->im);
  return ss1 + "+" + s2 + "i";
}
```

USE OF THE THIS POINTER

- It turns out that you don't need to use **`this`** to access the receiver's instance variables...

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double rp = re + that.re;
  double ip = im + that.im;
  return Cmpx(rp,ip);
}

std::string Cmpx::to_string() {
  std::string s1 = std::to_string(re);
  std::string s2 = std::to_string(im);
  return ss1 + "+" + s2 + "i";
}
```

INVOKING METHODS IN METHODS

- Here we define complex number division using "helper" methods:

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

INVOKING METHODS IN METHODS

- Here we define complex number division:

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

- Several methods use **`conjugate`**.

INVOKING METHODS IN METHODS

- Here we define complex number division:

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

- The **`reciprocal`** method uses **`modulus2`**.

INVOKING METHODS IN METHODS

- Here we define complex number division:

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

- The division method **`over`** uses **`reciprocal`**.

INVOKING METHODS IN METHODS

- We reference the receiver **`this`** several times.

```
Cmpx Cmpx::conjugate() {
  return Cmpx {this->re,-this->im};
}
double Cmpx::modulus2() {
  return this->times(this->conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return this->conjugate().times(1.0 / this->modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return this->times(that.reciprocal());
}
```

INVOKING METHODS IN METHODS

- Here instead we touch the receiver's fields and invoke its methods...

```
Cmpx Cmpx::conjugate() {
  return Cmpx {re,-im};
}
double Cmpx::modulus2() {
  return times(conjugate()).re;
}
Cmpx Cmpx::reciprocal() {
  return conjugate().times(1.0 / modulus2());
}
Cmpx Cmpx::over(Cmpx that) {
  return times(that.reciprocal());
}
```

CONSTRUCTOR IMPLEMENTATION

The role of a constructor is to initialize a new object instance's components:

```
Cmpx::Cmpx(void) {
  this->re = 0.0;
  this->im = 0.0;
}
Cmpx::Cmpx(double rp, double ip) {
  this->re = rp;
  this->im = ip;
}
Cmpx::Cmpx(std::string s) {
  parseCmpx(s,this->re,this->im); // note: passed by reference
}
Cmpx::Cmpx(const Cmpx& that) {
  this->re = that.re;
  this->im = that.im;
}
```

CONSTRUCTOR IMPLEMENTATION WITHOUT THIS

```
Cmpx::Cmpx(void) {
  re = 0.0;
  im = 0.0;
}
Cmpx::Cmpx(double rp, double ip) {
  re = rp;
  im = ip;
}
Cmpx::Cmpx(std::string s) {
  parseCmpx(s,re,im);
}
Cmpx::Cmpx(const Cmpx& that) {
  re = that.re;
  im = that.im;
}
```

CONSTRUCTOR INITIALIZER LISTS

- Here are the constructors using *constructor initializer list* syntax:

```
Cmpx::Cmpx(double rp, double ip) :
  re {rp}, im {ip}
{ }

Cmpx::Cmpx(void) :
  Cmpx {0.0, 0.0}
{ }

Cmpx::Cmpx(std::string s) {
  parseCmpx(s,re,im);
}

Cmpx::Cmpx(const Cmpx& that) :
  re {that.re}, im {that.im}
{ }
```

CONTROLLING FIELD/METHOD ACCESS

- Might not want clients to use helper methods.
- Might not want clients to directly access instance variables.
- Why?
  -

CONTROLLING FIELD/METHOD ACCESS

- Might not want clients to use helper methods.
- Might not want clients to directly access instance variables.
- Why?
  - Should isolate the underlying implementation.
  - That way it can be changed by the programmer later.

CONTROLLING FIELD/METHOD ACCESS

- Might not want clients to use helper methods.
- Might not want clients to directly access instance variables.
- Why?
  - Should isolate the underlying implementation.
  - That way it can be changed by the programmer later.

C++ allows field/method access control with public and private keywords.

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:            // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:             // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:            // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:             // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that); // Uses reciprocal.
16.   std::string to_string();
17. };
```

CONTROLLING FIELD/METHOD ACCESS W/ GETTERS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    double getReal();
10.   double getImag();  // "Getters" for access to fields.
11.   Cmpx(void);
12.   Cmpx(std::string);
13.   Cmpx(double rp, double ip);
14.   Cmpx(const Cmpx& that);
15.   Cmpx plus(Cmpx that);
16.   Cmpx times(Cmpx that);
17.   Cmpx over(Cmpx that); // Uses reciprocal.
18.   std::string to_string();
19. };
```

CONTROLLING FIELD/METHOD ACCESS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void); // Uses conjugate and modulus2.
8.  public:           // Can be invoked by clients.
9.    double getReal();
10.   double getImag();  // "Getters" for access to fields.
11.   Cmpx(void);
12.   Cmpx(std::string);
13.   Cmpx(double rp, double ip);
14.   Cmpx(const Cmpx& that);
15.   Cmpx plus(Cmpx that);
16.   Cmpx times(Cmpx that);
17.   Cmpx over(Cmpx that); // Uses reciprocal.
18.   std::string to_string();
19. };
```

GIVING FIELD/METHOD TO FRIENDS

```
1.  class Cmpx {
2.  private:          // Can only be accessed/invoked by methods.
3.    double re;
4.    double im;
5.    Cmpx conjugate(void);
6.    double modulus2(void);
7.    Cmpx reciprocal(void);
8.  public:           // Can be invoked by clients.
9.    Cmpx(void);
10.   Cmpx(std::string);
11.   Cmpx(double rp, double ip);
12.   Cmpx(const Cmpx& that);
13.   Cmpx plus(Cmpx that);
14.   Cmpx times(Cmpx that);
15.   Cmpx over(Cmpx that);
16.   std::string to_string();
17.   friend Cmpx sum(Cmpx z1, Cmpx z2);
18.   friend Cmpx quotient(Cmpx z1, Cmpx z2);
19. };
```

FRIEND FUNCTIONS

- Friends of a class can access private fields and invoke private methods.

```
1.  class Cmpx {
2.  private:
3.    double re;
4.    double im;
5.    ...
6.    Cmpx reciprocal(void);
7.  public:
8.    ...
9.    friend Cmpx sum(Cmpx z1, Cmpx z2);
10.   friend Cmpx quotient(Cmpx z1, Cmpx z2);
11. };
```

- Function definitions (usually defined within "Cmpx.cc"):

```
Cmpx sum(Cmpx z1, Cmpx z2) {
  return Cmpx {z1.re + z2.re, z1.im + z2.im};
}
Cmpx quotient(Cmpx z1, Cmpx z2) {
  return z1.times(z2.reciprocal());
}
```

CLASS MEMBERS

We can associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  private:
3.    double re;
4.    double im;
5.    static const double kEpsilon;
6.    static void parse(std::string s, double &rp, double &ip);
7.  public:
8.    Cmpx(void);
9.    Cmpx(std::string);
10.   Cmpx(double rp, double ip);
11.   Cmpx(const Cmpx& that);
12.   Cmpx plus(Cmpx that);
13.   std::string to_string();
14.   static const Cmpx I;
15.   static Cmpx product(Cmpx z1, Cmpx z2);
16. };
```

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  private:
3.     double re;
4.     double im;
5.     static const double kEpsilon;
6.     static void parse(std::string s, double &rp, double &ip);
7.  public:
8.     Cmpx(void);
9.     Cmpx(std::string);
10.    Cmpx(double rp, double ip);
11.    Cmpx(const Cmpx& that);
12.    Cmpx plus(Cmpx that);
13.    std::string to_string();
14.    static const Cmpx I;
15.    static Cmpx product(Cmpx z1, Cmpx z2);
16. };
```

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  ...
3.    static const double kEpsilon;
4.    static void parse(std::string s, double &rp, double &ip);
5.  ...
6.    static const Cmpx I;
7.    static Cmpx product(Cmpx z1, Cmpx z2);
8.  };
```

- The keyword static distinguishes them from *instance* variables/methods.
- They are called "class variables" (const in this case) and"class methods."
- In the case of variables, there is only one defined, shared by all instances.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

STATIC MEMBERS

We associate values and functions with the class, rather than instances:

```
1.  class Cmpx {
2.  ...
3.    static const double kEpsilon;
4.    static void parse(std::string s, double &rp, double &ip);
5.  ...
6.    static const Cmpx I;
7.    static Cmpx product(Cmpx z1, Cmpx z2);
8.  };
```

- The keyword static distinguishes these variables/methods.
- They are called "static variables" and "static methods."
- In the case of variables, they are shared by all instances.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

i.e. not dynamic, or "at run time"

IMPLEMENTING CLASS MEMBERS

```
1.  const double Cmpx::kEpsilon = 0.000001;
2.
3.  const Cmpx Cmpx::I {0.0,1.0};
4.
5.  bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7.  }
8.  Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define these class members, we ***do not*** label them as static.

- Each of their declared names start with the class name prefix.

IMPLEMENTING CLASS MEMBERS

```
1.  const double Cmpx::kEpsilon = 0.000001;
2.
3.  const Cmpx Cmpx::I {0.0,1.0};
4.
5.  bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7.  }
8.  Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define these class members, we ***do not*** label them as static.

- Each of their declared names start with the class name prefix.
- Class method code won't access **this**; there is no particular receiver.

USING CLASS MEMBERS

- Within the class implementation code:

```
Cmpx::Cmpx(std::string) {
  parse(s,this->re,this->im);
}
std::string Cmpx::to_string() {
  if (std::abs(im) < kEpsilon) {
    return std::to_string(re);
  } else if (std::abs(re) < kEpsilon) {
    return std::to_string(im) + "i";
  } else {
   if (im < 0.0) {
      return std::to_string(re)+std::to_string(im) + "i";
    } else {
      return std::to_string(re)+"+"+std::to_string(im) + "i";
    }
  }
}
```

- Within the client's code (if part of the class's public interface):

```
Cmpx rotate(Cmpx z) {
  return Cmpx::product(z, Cmpx::I)
}
```