LOGISTICS

- Office hours:
  - Mon/Wed/Fri 2:55-3:10, 4-4:15, questions before and after lecture.
  - Tue/Thu 1:30-2, 3:30-4:30. Will share Zoom link for regular "meeting."
    - ➡ Will manage a Zoom "waiting room" for these.
  - Also, by appointment. Can also email or Slack.
- Reminder: Homework 08 (MIPS assembly) due Wednesday before lecture.

MULTIPLICATION USING SHIFTING

- Here is the solution to my challenge from Lecture 08-3:

```
# $t1 and $t2 contain the mulltiplicands x and y
multiply:
        li      $t0, 0          # product = 0
multiply_loop:
        beqz    $t2, report     # if y == 0 goto report
        andi    $t3, $t2, 1     # bit = y & 1
        beqz    $t3, skip       # if bit == 0 goto skip
        addu    $t0, $t0, $t1   # sum += x
skip:
        sll     $t1, $t1, 1     # x *= 2
        sra     $t2, $t2, 1     # y /= 2
        b       multiply_loop
report:
```

- This can be found in **samples/multiply-shift.asm**

FUNCTION CALL CONVENTIONS EXAMPLE

- I never gave the complete code for my example illustrating MIPS function calling conventions.
  - ➡ It can be found in **samples/four-digits.asm**
- Some notes:
  - Both `main` and `four_digits` set up and take down a stack frame.
  - Neither `two_digits` nor `times100` bother making a frame.
    - ➡ They are both "leaf procedures," making no function calls.
    - ➡ (If they used callee-saved registers, they would each save the caller's. Neither do.)
  - The `times100` uses only v0 and a0, solution to another Lecture 08-3 exercise.

OBJECT-ORIENTATION IN C++

LECTURE 09-1

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, COPY CONSTRUCTORS, OPERATOR OVERLOADING

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

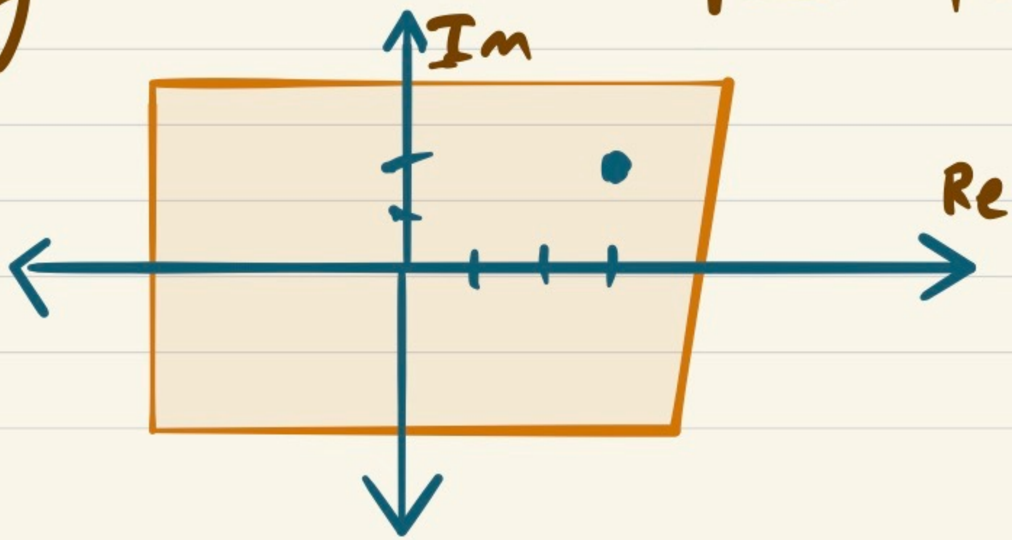
- especially physics & engrg

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:

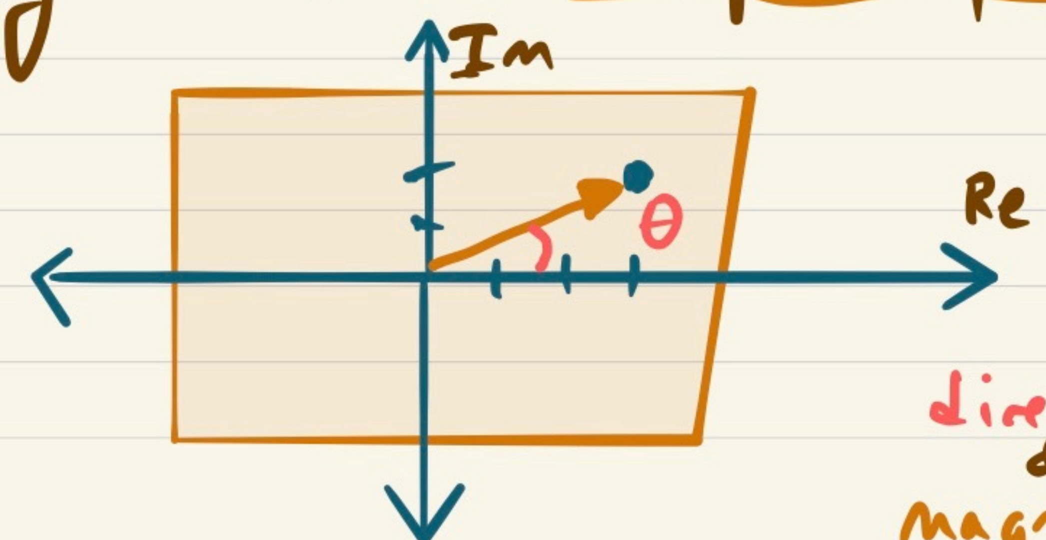


A QUICK OVERVIEW OF COMPLEX NUMBERS

Can be thought of as a pair of numbers

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

living in the complex plane:



direction θ
&
magnitude

Can treat algebraically...

- have an addition operation $+$
- and also a multiplication operation $\cdot$

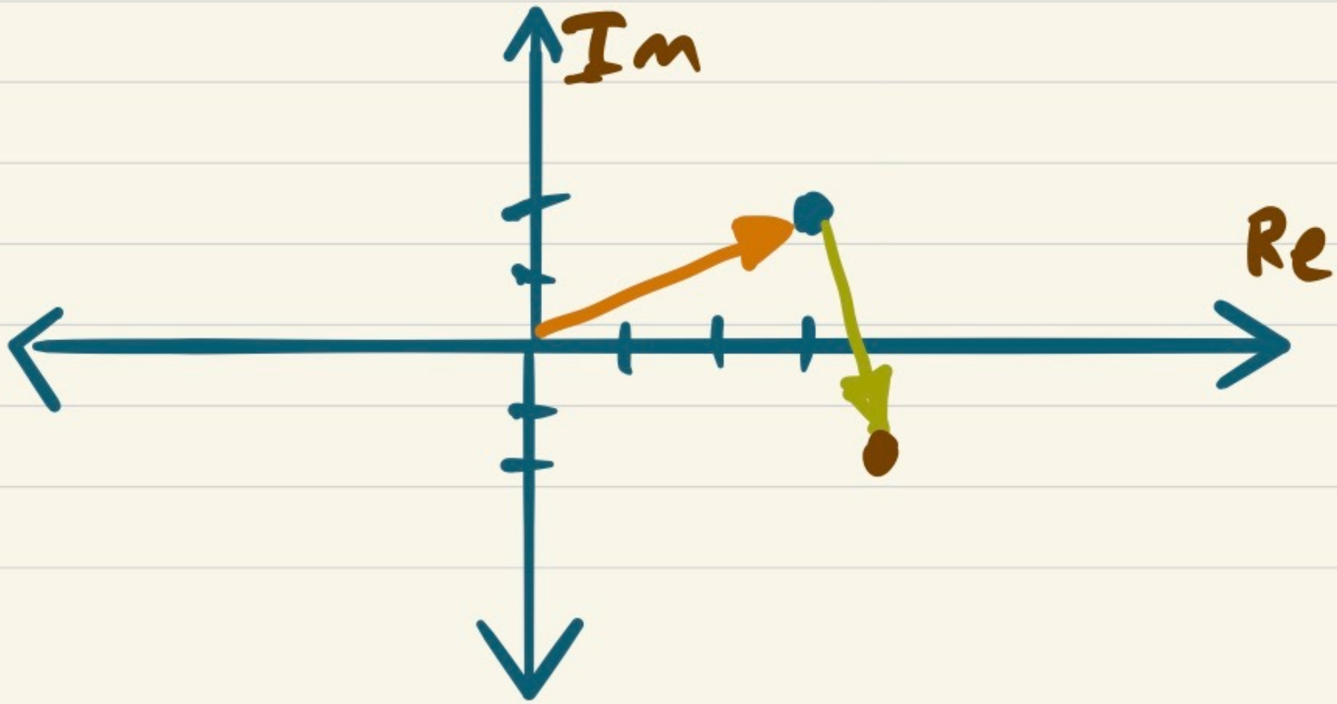
let $z_1 = a + bi$

$z_2 = c + di$

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

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

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

And $z_1 \cdot z_2$ is given by

$$(a+bi)\cdot(c+di) \quad \text{F.O.I.L.} \quad -1$$

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

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

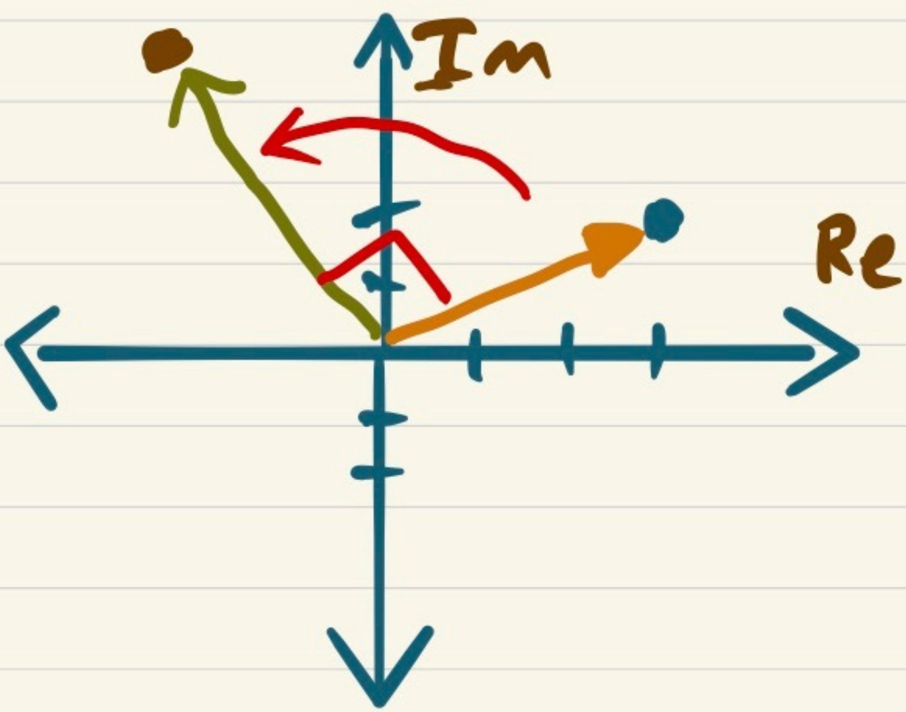
And $z_1 \cdot z_2$ is given by

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

$$= \underline{ac} + \underline{a\cdot di} + \underline{b\cdot ci} + \underline{b\cdot d}\cdot \overset{-1}{i^2}$$

$$= \underbrace{(ac-bd)}_{\text{real part}} + \underbrace{(ad+bc)}_{\text{imaginary part}}\, i$$

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 z1, cmpx z2);
  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 plus(cmpx this, cmpx that) {
  cmpx z;
  z.re = this.re + that.re;
  z.im = this.im + that.im;
  return z;
}
cmpx times(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 also write functions to "build" complex number "objects" as structs:

```
namespace cmpx {

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

  cmpx build(void) {
    cmpz z;
    z.re = 0.0;
    z.im = 0.0;
    return z;
  }
  cmpx build(double r, double i) {
    cmpx z;
    z.re = r;
    z.im = i;
    return z;
  }
  cmpx build(std::string s) {
    cmpx z;
    parse(s,z.re,z.im);
    return z;
  }
}
```

IMPLEMENTATION FILE: CMPX.CC

- Different version of the build functions using initializer lists:

```
namespace cmpx {

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

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

  cmpx build(double r, double i) {
    cmpx z {r,i};
    return z;
  }

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

CLIENT CODE

```
#include <iostream>
#include "cmpx.hh"

int main() {
  cmpx::cmpx z1 = cmpx::build(6.7,2.0);
  cmpx::cmpx z2 = cmpx::build(6.7,-2.0);

  cmpx::cmpx sum = cmpx::sum(z1,z2);
  cout << "The sum of " << cmpx::to_string(z1);
  cout << " and " << cmpx::to_string(z2);
  cout << " is " << cmpx::to_string(sum);
  cout << "." << endl;

  cmpx::cmpx product = cmpx::product(z1,z2);
  cout << "Their product is " << cmpx::to_string(product) << endl;

  return 0;
}
```

CLIENT CODE USING INITIALIZER LISTS

```cpp
#include <iostream>
#include "cmpx.hh"

int main() {
  cmpx::cmpx z1 {6.7,  2.0};
  cmpx::cmpx z2 {6.7, -2.0};

  cmpx::cmpx sum = cmpx::sum(z1,z2);
  cout << "The sum of " << cmpx::to_string(z1);
  cout << " and " << cmpx::to_string(z2);
  cout << " is " << cmpx::to_string(sum);
  cout << "." << endl;

  cmpx::cmpx product = cmpx::product(z1,z2);
  cout << "Their product is " << cmpx::to_string(product) << endl;

  return 0;
}
```

C-LIKE OBJECT CODING

```
#include <iostream>
#include "cmpx.hh"

int main() {
  cmpx::cmpx z1 = cmpx::build("6.7+2.0i");
  cmpx::cmpx z2 = cmpx::build("6.7-2.0i");

  cmpx::cmpx sum = cmpx::sum(z1,z2);
  cout << "The sum of " << cmpx::to_string(z1);
  cout << " and " << cmpx::to_string(z2);
  cout << " is " << cmpx::to_string(sum);
  cout << "." << endl;

  cmpx::cmpx product = cmpx::product(z1,z2);
  cout << "Their product is " << cmpx::to_string(product) << endl;

  return 0;
}
```

OUR ASPIRATIONS FOR OBJECT-ORIENTATION

- Consider this C++ program instead:

```
#include <iostream>
#include "Cmpx.hh"

int main() {
  Cmpx z1 {"6.7 + 2.0i"};
  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;

  return 0;
}
```

SUPER-ASPIRATIONAL OBJECT-ORIENTATION

- Or consider this C++ program:

```
#include <iostream>
#include "Cmpx.hh"

int main() {
  Cmpx z1 {"6.7 + 2.0i"};
  Cmpx z2 {"6.7 - 2.0i"};

  cout << "The sum of " << z1;
  cout << " and " << z2;
  cout << " is " << z1+z2;
  cout << "." << endl;

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

  return 0;
}
```

OUR ASPIRATIONS FOR OBJECT-ORIENTATION

```
#include <iostream>
#include "Cmpx.hh"

int main() {
  Cmpx z1 {"6.7 + 2.0i"};
  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;
  Cmpz z1p = product.over(z2);
  cout << "Dividing out the 2nd to obtain the 1st: " << z1p.to_string() << endl;
  Cmpz z2p = product.over(z1);
  cout << "Dividing out the 1st to obtain the 2nd: " << z2p.to_string() << endl;
  Cmpx i {0.0,1.0};
  cout << "Rotating 1st by 90 degrees: " << z1.times(i).to_string() << endl;
  cout << "Rotating 2nd by 90 degrees: " << z2.times(i).to_string() << endl;
  return 0;
}
```

SUPER-ASPIRATIONAL OBJECT-ORIENTATION

```
#include <iostream>
#include "Cmpx.hh"

int main() {
  Cmpx z1 {"6.7 + 2.0i"};
  Cmpx z2 {"6.7 - 2.0i"};

  cout << "The sum of " << z1;
  cout << " and " << z2;
  cout << " is " << z1+z2;
  cout << "." << endl;

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

  cout << "Dividing out the 2nd to obtain the 1st: " << product/z2 << endl;
  cout << "Dividing out the 1st to obtain the 2nd: " << product/z1 << endl;

  Cmpx i {0.0,1.0};
  cout << "Rotating 1st by 90 degrees: " << z1*i << endl;
  cout << "Rotating 2nd by 90 degrees: " << z2*i << endl;

  return 0;
}
```

OBJECT-ORIENTATION IN C++

- GOAL: cover the key O-O syntax of C++ to reach these two code examples.
- We'll work to reach the code of our first aspirations by the end of today.
- We'll work to reach our second aspirations Wednesday (probably).
- Outline: explain the code in **samples/Cmpx**
  - **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 O-O 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.
- Objects instances are passed *by value*. But can pass pointers, or *by reference*.
- **`__init__`** replaced by (possibly several) *constructors*, including a *default*.
- Need *destructors* because there is no garbage collector.
- Can limit access with **`public`** versus **`private`**. Allow **`friend`**s.
- Can *overload* methods. Can define **`operator`** like **`+`**, and others.

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

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

```
class Cmpx {
  // instance variables
  double re;
  double im;
  // constructors
  Cmpx(void);                    // "default" constructor
  Cmpx(std::string);
  Cmpx(double re, double im);
  Cmpx(const Cmpx& that);        // "copy" constructor (later)
  // methods
  Cmpx plus(Cmpx that);
  Cmpx times(Cmpx that);
  std::string to_string();
};
```

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

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

```
class Cmpx {
  // instance variables
  double re;
  double im;
  // constructors
  Cmpx(void);                  // "default" constructor
  Cmpx(std::string);
  Cmpx(double re, double im);
  Cmpx(const Cmpx& that);      // "copy" constructor (later)
  // methods
  Cmpx plus(Cmpx that);
  Cmpx times(Cmpx that);
  std::string to_string();
};
```

IMPLEMENTATION FILE: CMPX.CC

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

```
#include "Cmpx.hh"
...
Cmpx Cmpx::plus(Cmpx that) {
  double r = this->re + that.re;
  double i = this->im + that.im;
  Cmpx z(r,i);
  return z;
}
Cmpx Cmpx::times(Cmpx that) {
  double r = this->re*that.re - this->im*that.im;
  double i = this->im*that.re + this->re*that.im;
  Cmpx z(r,i);
  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 r = re + that.re;
  double i = im + that.im;
  Cmpx z {r,i};
  return z;
}
Cmpx Cmpx::times(Cmpx that) {
  double r = re*that.re - im*that.im;
  double i = im*that.re + re*that.im;
  Cmpx z {r,i};
  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 **samples/Cmpx/Cmpx.cc**:

```
Cmpx::Cmpx(void) {
  this->re = 0.0;
  this->im = 0.0;
}
Cmpx::Cmpx(double r, double i) {
  this->re = r;
  this->im = i;
}
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 r, double i) {
  re = r;
  im = i;
}
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 the initializer syntax*:

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

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

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 constructor with 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;
}
```

CLIENT CODE FILE: TEST_CMPX.CC

- Here are two uses of the default constructor:

```
#include <iostream>
#include "Cmpx.hh"
int main() {
  Cmpx z1 {};       // Uses default constructor explicitly.
  cin >> z1.re;     // Accesses each instance variable.
  cin >> z1.im;     //
  Cmpx z2;          // Also uses default constructor.
  z2.re = 6.67;
  z2.im = 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;
}
```

CONTROLLING FIELD/METHOD ACCESS

```
class Cmpx {
private:          // Can only be accessed/invoked by methods.
  double re;
  double im;
  Cmpx conjugate(void);
  double modulus2(void);
  Cmpx reciprocal(void); // Uses conjugate and modulus2.
public:           // Can be invoked by clients.
  double getReal();
  double getImag();  // "Getters" for access to fields.
  Cmpx(void);
  Cmpx(std::string);
  Cmpx(double re, double im);
  Cmpx(const Cmpx& that);
  Cmpx plus(Cmpx that);
  Cmpx times(Cmpx that);
  Cmpx over(Cmpx that); // Uses reciprocal.
  std::string to_string();
};
```

NOTE ON C++ LANGUAGE VERSIONS

- The syntax of initializers within constructors, and also the initializer list 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
```

NEXT LECTURE

- An example of a "container class" Queue.
- Destructors.
- Heap-allocating objects using new; deallocation with delete.
- Defining operators like +, * for Cmpx.
- Overloading operators like << and >> as friends of Cmpx.
- Homework 09.
- Exercise: add subtraction and negation methods to Cmpx.