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:

```
1.  # $t1 and $t2 contain the mulltiplicands x and y
2.  multiply:
3.          li      $t0, 0          # product = 0
4.  multiply_loop:
5.          beqz    $t2, report     # if y == 0 goto report
6.          andi    $t3, $t2, 1     # bit = y & 1
7.          beqz    $t3, skip       # if bit == 0 goto skip
8.          addu    $t0, $t0, $t1   # sum += x
9.  skip:
10.         sll     $t1, $t1, 1     # x *= 2
11.         sra     $t2, $t2, 1     # y /= 2
12.         b       multiply_loop
13. 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 09-3

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)
- MISSING: DESTRUCTORS, COPY CONSTRUCTORS, OPERATOR OVERLOADING

OVERVIEW OF COMPLEX NUMBERS

IMPLEMENTATION FILE: CMPX.CC

- Here functions we can use to "build" complex number "objects" as structs:

```
namespace cmpx {

  void parse(std::string, double& rp, souble& 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

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

```
namespace cmpx {

cmpx cmpx::plus(cmpx this, cmpx that) {
  cmpx z;
  z.re = this.re + that.re;
  z.im = this.im + that.im;
  return z;
}
cmpx 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 cmpx::to_string(cmpx z) {
  return std::to_string(z.re) + "+" + std::to_string(z.im) + "i";
}
```

IMPLEMENTATION FILE: CMPX.CC

- Different version of the build functions using initializer lists:

```
namespace cmpx {

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

  cmpx build(void) {
    cmpz 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

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx z1 = cmpx::build(6.7,2.0);
6.    cmpx z2 = cmpx::build(6.7,-2.0);
7.    cmpx sum = cmpx::sum(z1,z2);
8.    cout << "The sum of " << cmpx::to_string(z1);
9.    cout << " and " << cmpx::to_string(z2);
10.   cout << " is " << cmpx::to_string(sum);
11.   cout << "." << endl;
12.   cmpx product = cmpx::product(z1,z2);
13.   cout << "Their product is " << cmpx::to_string(product) << endl;
14.   cmpz z1p = cmpx::quotient(product,z2);
15.   cmpz z2p = cmpx::quotient(product,z1);
16.   std::string s1 = cmpx::to_string(z1p);
17.   std::string s1 = cmpx::to_string(z1p);
18.   cout << "Dividing out the 2nd to obtain the 1st: " << s1p << endl;
19.   cout << "Dividing out the 1st to obtain the 2nd: " << s2p << endl;
20.   return 0;
21. }
```

C-LIKE OBJECT CODING

- Consider this C++ program:

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx z1 {6.7,  2.0};
6.    cmpx z2 {6.7, -2.0};
7.    cmpx sum = cmpx::sum(z1,z2);
8.    cout << "The sum of " << cmpx::to_string(z1);
9.    cout << " and " << cmpx::to_string(z2);
10.   cout << " is " << cmpx::to_string(sum);
11.   cout << "." << endl;
12.   cmpx product = cmpx::product(z1,z2);
13.   cout << "Their product is " << cmpx::to_string(product) << endl;
14.   cmpz z1p = cmpx::quotient(product,z2);
15.   cmpz z2p = cmpx::quotient(product,z1);
16.   std::string s1 = cmpx::to_string(z1p);
17.   std::string s1 = cmpx::to_string(z1p);
18.   cout << "Dividing out the 2nd to obtain the 1st: " << s1p << endl;
19.   cout << "Dividing out the 1st to obtain the 2nd: " << s2p << endl;
20.   return 0;
21. }
```

C-LIKE OBJECT CODING

- Consider this C++ program:

```
1.  #include <iostream>
2.  #include "cmpx.hh"
3.
4.  int main() {
5.    cmpx z1 = cmpx::parse("6.7+2.0i");
6.    cmpx z2 = cmpx::parse("6.7-2.0i");
7.    cmpx sum = cmpx::sum(z1,z2);
8.    cout << "The sum of " << cmpx::to_string(z1);
9.    cout << " and " << cmpx::to_string(z2);
10.   cout << " is " << cmpx::to_string(sum);
11.   cout << "." << endl;
12.   cmpx product = cmpx::product(z1,z2);
13.   cout << "Their product is " << cmpx::to_string(product) << endl;
14.   cmpz z1p = cmpx::quotient(product,z2);
15.   cmpz z2p = cmpx::quotient(product,z1);
16.   std::string s1 = cmpx::to_string(z1p);
17.   std::string s1 = cmpx::to_string(z1p);
18.   cout << "Dividing out the 2nd to obtain the 1st: " << s1p << endl;
19.   cout << "Dividing out the 1st to obtain the 2nd: " << s2p << endl;
20.   return 0;
21. }
```

OUR ASPIRATIONS FOR OBJECT-ORIENTATION

- 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.    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. }
```

SUPER-ASPIRATIONAL OBJECT-ORIENTATION

- 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.   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 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:

```
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 re, double im);
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) samples/Cmpx/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 re, double im);
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 **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 a simple version of the to_string method:

```
#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;
  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;
}
```

CLIENT CODE FILE: TEST_CMPX.CC

- Here is a simple version of the to_string method:

```
#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;
  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;
}
```

CLIENT CODE FILE: TEST_CMPX.CC

- Here is a simple version of the to_string method:

```
#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.
  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;
}
```

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.