

OBJECT-ORIENTATION IN C++

LECTURE 09-2

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN:

- ▶ REVIEW OF SYNTAX
 - CLIENT, SPEC'N, IMPL'N SYNTAX
- ▶ THE **THIS** POINTER
- ▶ CONSTRUCTOR SYNTAX
- ▶ RESTRICTING ACCESS, GRANTING ACCESS
- ▶ CLASS MEMBERS

NOTE: Homework 09 will be assigned *Friday*.

CLIENT OF AN OBJECT CLASS

```
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. }
```

CLIENT OF AN OBJECT CLASS

```
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. }
```

CLIENT OF AN OBJECT CLASS

```
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. }
```

SPECIFICATION FILE: 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 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);                // "default" constructor
9.     Cmpx(std::string);
10.    Cmpx(double re, double im);
11.    Cmpx(const Cmpx& that);      // "copy" constructor (later)
12.    // methods
13.    Cmpx plus(Cmpx that);
14.    Cmpx times(Cmpx that);
15.    std::string to_string();
16. };
17. #endif
```

CLIENT CODE, AGAIN

```
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. }
```


OBJECTS ARE LIKE STRUCTS

```
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. }
```

OBJECTS ARE LIKE STRUCTS BUT WITH ATTACHED FUNCTIONS

```
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. }
```

OBJECTS ARE LIKE STRUCTS BUT WITH METHODS

```
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. }
```

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 object'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. }
```

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);
```

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.

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.

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) ;

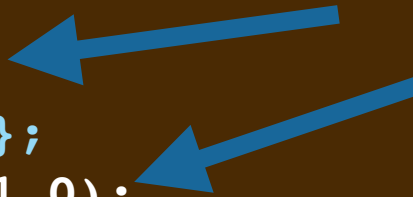
Examples in client code:

```
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.

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.

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 ) ;
```



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

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
```

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 top one (implicitly) calls the *default constructor*. It takes no parameters.

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 z ( ) ;
```

```
Cmpx z { } ;
```

```
Cmpx z = Cmpx ( ) ;
```

- ▶ The others just added above do the same using the syntax variants.


OBJECT INSTANTIATION EXPRESSION

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

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 last line is actually a combination of three things:
 - An *anonymous* object instance made using a construction expression
 - A default construction of the stack struct storage for **i**.
 - A *copy assignment* of the RHS one into the storage for **i** on the LHS.

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*.h" or "*class-name*.hpp"

Example:

```
class Cmpx {  
    double re;  
    double im;  
    Cmpx();  
    Cmpx(double rp, double ip);  
    Cmpx sum(Cmpx z1, Cmpx z2);  
    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 sum(Cmpx z1, Cmpx z2);  
    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 sum(Cmpx z1, Cmpx z2);  
    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 sum(Cmpx z1, Cmpx z2);  
    std::string to_string();  
};
```

 *method signatures*

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 sum(Cmpx z1, Cmpx z2);  
    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

Syntax of a constructor signature:

class-name (*parameter-declarations*) ;

► Example:

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

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

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 s1 + "+" + 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 s1 + "+" + 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 s1 + "+" + 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 s1 + "+" + 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 s1 + "+" + 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 = r;  
    this->im = i;  
}  
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 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;  
}
```

CONSTRUCTOR INITIALIZER LISTS

► Here are the constructors using *constructor initializer list* 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}  
{ }
```

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 re, double im);
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 re, double im);
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 re, double im);
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 re, double im);
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 re, double im);
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 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 re, double im);
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.

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

3. static const Cmpx Cmpx::I {0.0,1.0};

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define what these members mean, we label them as **static**.

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

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

3. static const Cmpx Cmpx::I {0.0,1.0};

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define what these members mean, we 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.

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

3. static const Cmpx Cmpx::I {0.0,1.0};

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define what these members mean, we 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.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

IMPLEMENTING STATIC MEMBERS

```
1. static const double Cmpx::kEpsilon = 0.000001;

2.

3. static const Cmpx Cmpx::I {0.0,1.0};

4.

5. static bool Cmpx::parse(string s, double& rp, double& ip) {
6.     ...
7. }
8. static Cmpx Cmpx::product(Cmpx z1, Cmpx z2) {
9.     ...
10. }
```

When we define what the static members are, we declare them as static.

- Each of their definitions is *i.e. not dynamic, or "at run time"*.
 - Class method code is not associated with a particular receiver.
- NOTE: "static" is carried from C meaning "can lay out at compile time."

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:

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

EXERCISE (FOR FRIDAY)

- ▶ The README.md file for this lecture describes an exercise to develop a rational number class:
 - define a Rational class in Rational.hh
 - define constructors, addition, and multiplication in Rational.cc
 - define a test_rational.cc to test your code
- ▶ The exercise is *optional* and replaces lab.

FRIDAY'S PLAN

- ▶ MY SOLUTION FOR RATIONAL
- ▶ (OVERLOADING) OPERATORS (LIKE +, *, <<)
- ▶ USING **new** TO HEAP-ALLOCATE OBJECTS
- ▶ CONTAINER CLASS
 - STACK EXAMPLE
- ▶ DESTRUCTORS AND THEIR IMPLICIT INVOCATION
- ▶ PASSING BY REFERENCE
- ▶ CONST METHODS

NOTE: Homework 09 will be assigned then.