EECS 280 – Lecture 7

Abstract Data Types in C

1/31/2022

Abstraction

Interface vs. Implementation

- Procedural abstraction separates **what** a function does from **how** it works.

```
// EFFECTS: extracts one column of data from a tab
//  separated values file (.tsv)
//  Prints errors to stdout and exits with non-zero
//  status on errors
std::vector<double> extract_column(
  std::string filename, std::string column_name);
```

- Data abstraction separates **what** an **Abstract Data Type (ADT)** does from **how** it works.

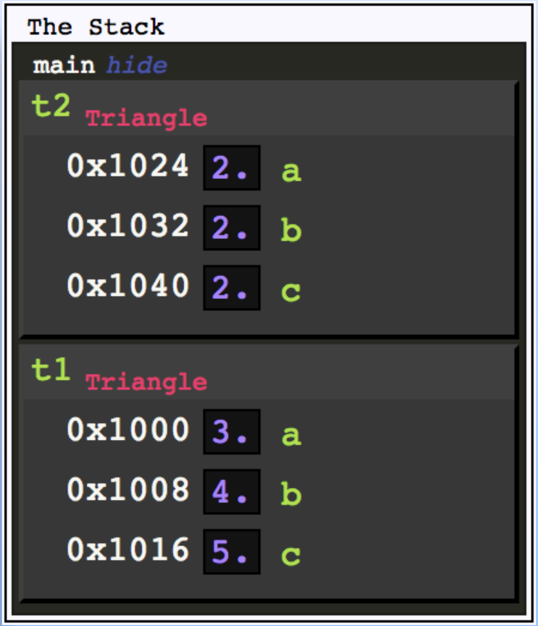
```
string str1 = "hello ";
string str2 = "jello";
cout << str1 + str2 << endl; // Prints "hello jello"
```

1 Contrast this to C-style strings where you pretty much have to know all the details of how they work or you get tripped up.

C-Style ADTs: Data Representation

- Let's say we want to represent **triangles**.
- First, pick a **data representation**:
  - Three side lengths a, b, c as **members** of a **struct**.
- This is an **implementation detail**.

```
struct Triangle {
  double a;
  double b;
  double c;
};

int main() {
  Triangle t1 = { 3, 4, 5 };
  Triangle t2 = { 2, 2, 2 };
}
```



C-Style ADTs: Interface Functions

- Define **functions** for `Triangle` **behaviors**.
- These determine the **interface** for a triangle.

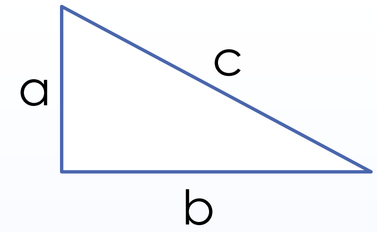
```
struct Triangle {
  double a, b, c;
};

double Triangle_perimeter(const Triangle *tri) {
  return tri->a + tri->b + tri->c;
}

int main() {
  Triangle t1 = { 3, 4, 5 };
  cout << Triangle_perimeter(&t1) << endl;
}
```

The code in here is part of the implementation.

The first parameter is a pointer to the `Triangle` `struct` we want to work with.

When you call the function, pass it the address of the Triangle you want to work with.

C-Style ADTs



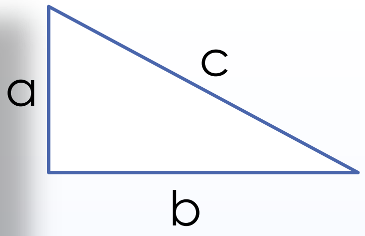
- We use **pass-by-pointer** so that we can work with the **original object**.

```
struct Triangle {
  double a, b, c;
};

double Triangle_perimeter(const Triangle *tri) {
  return tri->a + tri->b + tri->c;
}

int main() {
  Triangle t1 = { 3, 4, 5 };
  cout << Triangle_perimeter(&t1) << endl;
}
```

Memory

```
struct Triangle {
  double a, b, c;
};

int main() {
  Triangle t1 = { 2, 2, 2 };
  Triangle_scale(&t1, 2); // Scale by x2
  cout << Triangle_perimeter(&t1) << endl; // prints 12
}
```

Let's say we want to add a function to scale triangles by a given factor.




```
void Triangle_scale(const Triangle *tri,
                    double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

```
void Triangle_scale(Triangle tri,
                    double s) {
  tri.a *= s;
  tri.b *= s;
  tri.c *= s;
}
```

```
void Triangle_scale(Triangle *tri,
                    double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

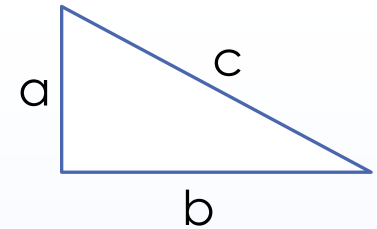
```
void Triangle_scale(Triangle *tri,
                    double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

```
void Triangle_scale(double s) {
  t1.a *= s;
  t1.b *= s;
  t1.c *= s;
}
```

<u>Question</u>

Which of these `Triangle_scale` functions are written correctly?

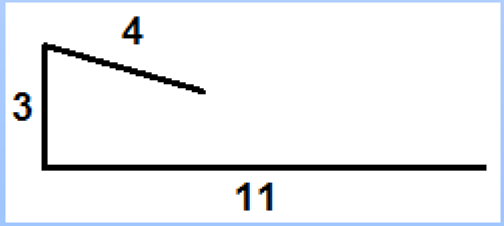
C-Style ADTs (structs)



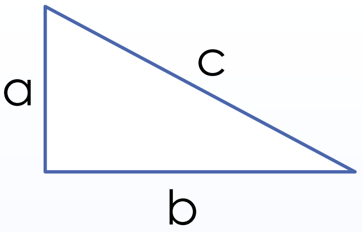
- There are some issues with the way we're initializing the triangle.
- What's wrong with this code?

```
int main() {
  Triangle t1 = { 3, 4, 11 };
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;

}
```

We have no check on the values used to initialize the `Triangle` member variables.



C-Style ADTs (structs)



- Define an initializer function

```
struct Triangle {
  double a, b, c;
};

void Triangle_init(Triangle *tri, double a_in,
                   double b_in, double c_in) {
  // TODO: Check the a, b, c, values we get
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;
}
```

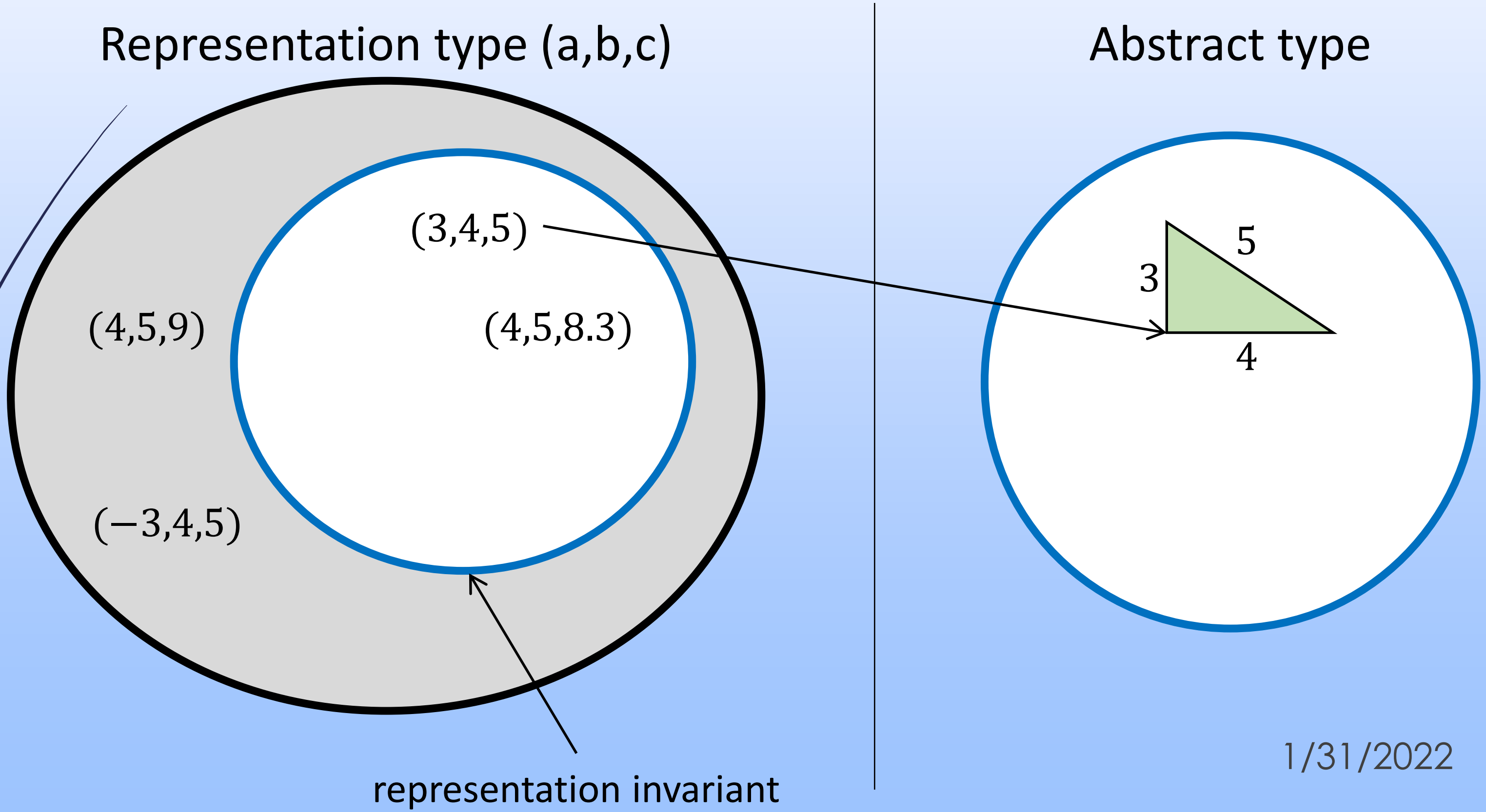
Representation Invariants

- A problem for compound types…
  - Some combinations of member values don't make sense together.
- We use **representation invariants** to express the conditions for a **valid** compound object.
- For Triangle:

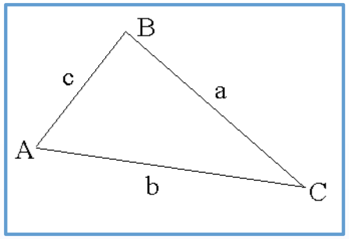
Positive Edge Lengths	Triangle Inequality
`0 < a`	`a + b > c`
`0 < b`	`a + c > b`
`0 < c`	`b + c > a`

Representation vs. Abstraction

- We pick some way to represent our abstract concept with data. (e.g. 3 doubles)
- Only some possible values in the representation are **valid** (meaningful in the abstraction).



C-Style ADTs (structs)

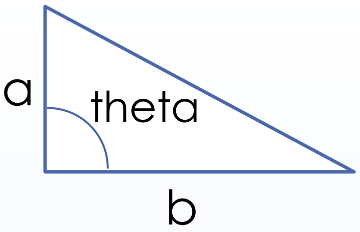


- Check invariants using assert.

```
struct Triangle {
  double a, b, c;
};

void Triangle_init(Triangle *tri, double a_in,
                         double b_in, double c_in) {
  assert(0 < a_in && 0 < b_in && 0 < c_in);
  assert(a_in + b_in > c_in  && a_in + c_in > b_in &&
         b_in + c_in > a_in);
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 11);
}
```

This will now cause a failed assertion.

C-Style ADTs (structs)



- Other representations are possible.
  - Different representation invariants!

```
struct Triangle {
  double a;
  double b;
  double theta;
};

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;
}
```

Different implementation.

Same interface.

Respect the Interface!

- What's wrong with this code?

```
int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);

  // Print out the perimeter (BAD)
  cout << t1.a + t1.b + t1.c << endl;
}
```

```
struct Triangle {
  double a;
  double b;
  double theta;
};
```

The existence of a member called c is an implementation detail, not part of the interface!

Plain Old Data

- A simple compound type can be defined as *Plain Old Data (POD)*, where the interface and implementation are the same.

```
struct Pixel {
  int r; // red
  int g; // green
  int b; // blue
};

int main() {
  Pixel p = { 255, 0, 0 };
  cout << p.r << " "
       << p.g << " "
       << p.b << endl;
}
```

(255,0,0)	(0,255,0)	(0,0,255)
(0,0,0)	(255,255,255)	(100,100,100)
(101,151,183)	(124,63,63)	(163,73,164)

Composing ADTs

- One ADT might be a member of another.

```
struct Professor {
  int age;
  Triangle favTriangle;
};
```

Always remember to initialize member ADTs, either by using an _init function or as a copy of a pre-existing ADT.

```
void Professor_init(Professor *prof, int age, int side) {
  prof->age = age;
  Triangle_init(&prof->favTriangle, side, side, side);
}

void Professor_init(Professor *prof, int age,
                    const Triangle &favTriangle) {
  prof->age = age;
  prof->favTriangle = favTriangle;
}
```

Abstraction Layers

- ADTs can be composed to for multiple layers of abstraction.

Image "what".

	0	1	2	3	4
0	(0,0,0)	(0,0,0)	(255,255,250)	(0,0,0)	(0,0,0)
1	(255,255,250)	(126,66,0)	(126,66,0)	(126,66,0)	(255,255,250)
2	(126,66,0)	(0,0,0)	(255,219,183)	(0,0,0)	(126,66,0)
3	(255,219,183)	(255,219,183)	(0,0,0)	(255,219,183)	(255,219,183)
4	(255,219,183)	(0,0,0)	(134,0,0)	(0,0,0)	(255,219,183)

Image.cpp

Image "how", using Matrix "what".

	0	1	2	3	4
0	0	0	255	0	0
1	255	126	126	126	255
2	126	0	255	0	126
3	255	255	0	255	255
4	255	0	134	0	255

	0	1	2	3	4
0	0	0	255	0	0
1	255	66	66	66	255
2	66	0	219	0	66
3	219	219	0	219	219
4	219	0	0	0	219

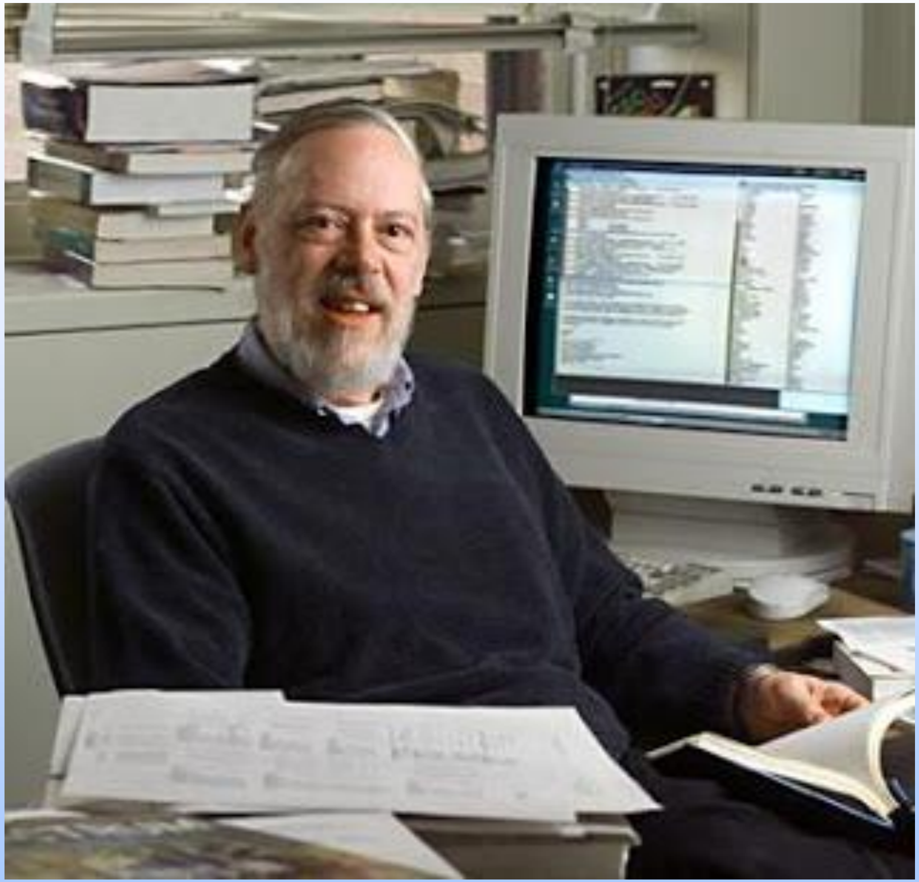
	0	1	2	3	4
0	0	0	250	0	0
1	250	0	0	0	250
2	0	0	183	0	0
3	183	183	0	183	183
4	183	0	0	0	183

Matrix.cpp

Matrix "how".

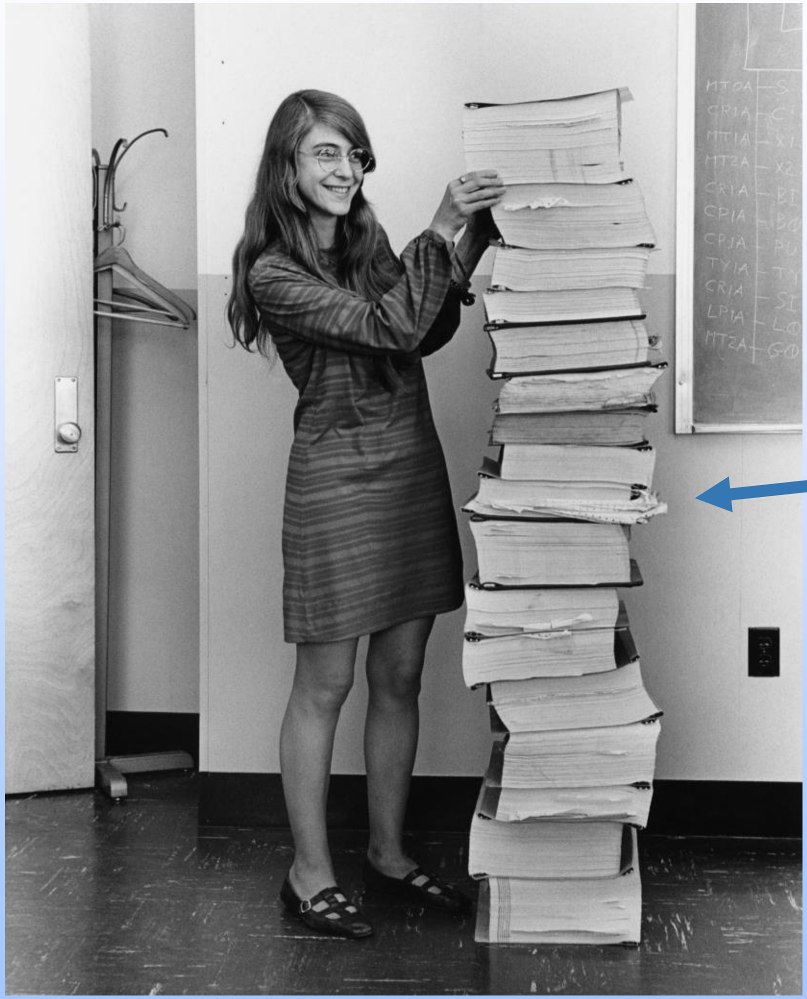
0	0	255	0	0	255	126	126	126	255	126	0	255	0	126	255	255	0	255	255	255	0	134	0	255
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

Dennis Ritchie



Creator of the C Programming Language

Margaret Hamilton



This giant stack is the printed code for the Apollo Project's Navigation System!

Lead Developer for Flight Software
NASA's Apollo Program

Break Time!

We’ll start again in one minute.

Review: Types of Testing

- **Unit testing**
  - One piece at a time (e.g., a function)
  - Find and fix bugs early! Less work.
    - Test smaller, less complex, easier to understand units.
    - You just wrote the code: easier to debug
- **System testing**
  - Entire project (code base)
  - Do this *after* unit testing
- **Regression testing**
  - Automatically run all unit and system tests after a code change

Kinds of Test Cases

Consider test cases for the `Matrix_at` function from project 2…

```
// REQUIRES: mat points to a valid Matrix
//           0 <= row && row < Matrix_height(mat)
//           0 <= column && column < Matrix_width(mat)
// EFFECTS:  Returns a pointer to the element in the Matrix
//           at the given row and column.
int* Matrix_at(Matrix* mat, int row, int column);
```

Don't write these.

Type Prohibited	`ASSERT_EQUAL(*Matrix_at("cat", 2, 2), 42)`
REQUIRES Prohibited	`ASSERT_EQUAL(*Matrix_at(` $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$ `, 1, -1), 42)`
Simple	`ASSERT_EQUAL(*Matrix_at(` $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$ `, 1, 1), 5)`
(Edge) Special	`ASSERT_EQUAL(*Matrix_at(` $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$ `, 2, 2), 9)`
Stress	`Matrix_init(big, 400, 400); Matrix_fill(big, 1);` `ASSERT_TRUE(Matrix_equal(big,` $\begin{bmatrix}1 & \cdots & 1\\ \vdots & \ddots & \vdots\\1 & \cdots & 1\end{bmatrix}$ `));`

Not needed for P2.

Simple Test

```
// Fills a 3x5 Matrix with a value and checks that
// Matrix_at returns that value for each element.
TEST(test_fill_basic) {
  Matrix *mat = new Matrix;
  const int width = 3;
  const int height = 5;
  const int value = 42;
  Matrix_init(mat, width, height);
  Matrix_fill(mat, value);
  for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      ASSERT_EQUAL(*Matrix_at(mat, row, col), value);
    }
  }
  delete mat;
}
```

Create `Matrix` object in dynamic memory[1]. The result is a pointer to the new object.

Delete `Matrix` object when we no longer need it.

[1]We create `Matrix` and `Image` objects in dynamic memory because the default stack size on most of our machines is too small to store them on the stack.

Bad Edge Test

```
// Places the maximum value at a corner of the
// matrix and tests that Matrix_max finds it.
TEST(edge_test_max) {
  Matrix *mat = new Matrix;
  const int width = 3;
  const int height = 5;

  Matrix_init(mat, width, height);
  for (int i = 0; i < width * height; ++i) {
    mat->data[i] = i;
  }

  mat->data[14] = 99;

  ASSERT_EQUAL(Matrix_max(mat), 99);
  delete mat;
}
```

Breaks the Matrix interface.

Respect the Interface!

- What's wrong with this code?

```
for (int i = 0; i < width * height; ++i) {
  mat->data[i] = i;
}
mat->data[14] = 99
```

- The existence of a member called `data` and its layout is an implementation detail, not part of the interface!

- Instead, use the interface functions, **even when testing**.

```
*Matrix_at(mat, 4, 2) = 99;
```

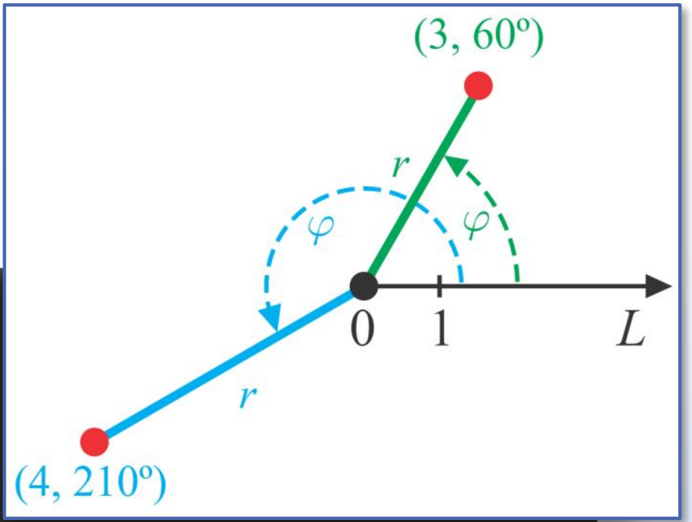
Good Edge Test

```
// Places the maximum value at a corner of the
// matrix and tests that Matrix_max finds it.
TEST(edge_test_max) {
  Matrix *mat = new Matrix;
  const int width = 3;
  const int height = 5;
  const int max_value = 99;
  Matrix_init(mat, width, height);
  for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      *Matrix_at(mat, row, col) = row * width + col;
    }
  }
  *Matrix_at(mat, 4, 2) = max_value;
  ASSERT_EQUAL(Matrix_max(mat), max_value);
  delete mat;
}
```

Review: Test-Driven Development

1. First, write tests for the desired behavior.
2. Then write implementations with the goal of passing those tests.
3. Go to step 1.
   - You may have thought of more tests while writing the implementation.
   - There should be lots of back and forth between testing and implementing!
   - If you find a bug, make sure you have a test case that exposes the bug.

Testing an ADT




```
struct Polar {
  double r;
  double phi;
};

void Polar_init(Polar* p, double radius,
                double angle);
double Polar_radius(const Polar* p);
double Polar_angle(const Polar* p);

TEST(test_basic) {
  Polar p;
  Polar_init(&p, 5, 45);

  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}
```

struct for data representation.

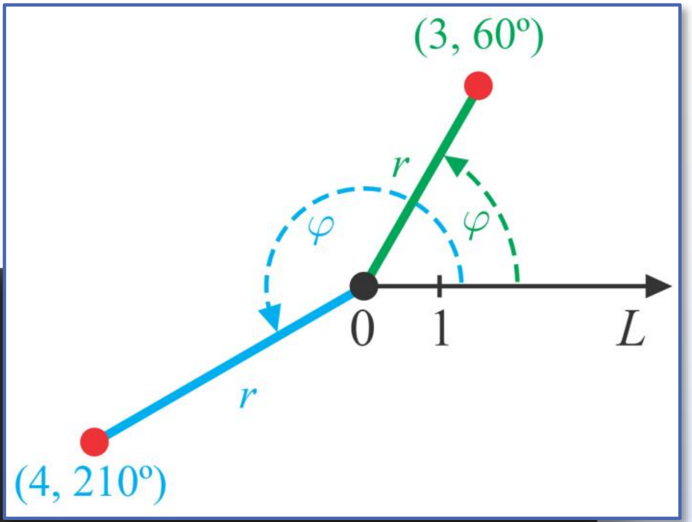
Initializer Function

Polar Functions

Test cases for Polar. Write these before implementing!

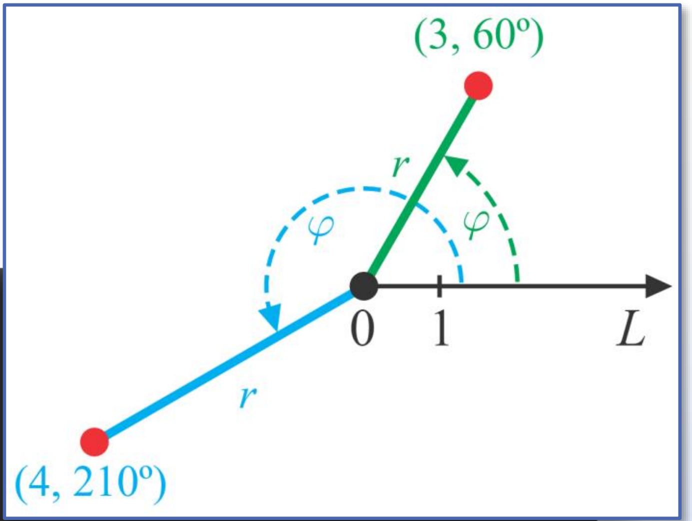
Testing an ADT

```
struct Polar {
  double r;
  double phi;
};

void Polar_init(Polar* p, double radius,
                double angle) {
  p->r = radius;
  p->phi = angle;
}
double Polar_radius(const Polar* p) {
  return p->r;
}
double Polar_angle(const Polar* p) {
  return p->phi;
}
```



Then, make an initial attempt at writing implementations.

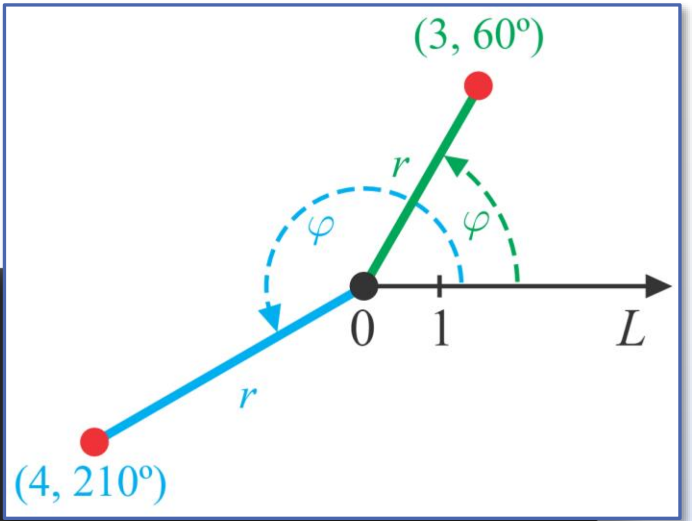
Testing an ADT




```
struct Polar {
  double r;
  double phi;
};

void Polar_init(Polar* p, double radius,
                double angle);
double Polar_radius(const Polar* p);
double Polar_angle(const Polar* p);

TEST(test_basic) {
  Polar p;
  Polar_init(&p, 5, 45);

  ASSERT_EQUAL(Polar_radius(&p), 5); // OK
  ASSERT_EQUAL(Polar_angle(&p), 45); // OK
}
```

Finally, run tests to check if the implementation works.

Testing an ADT

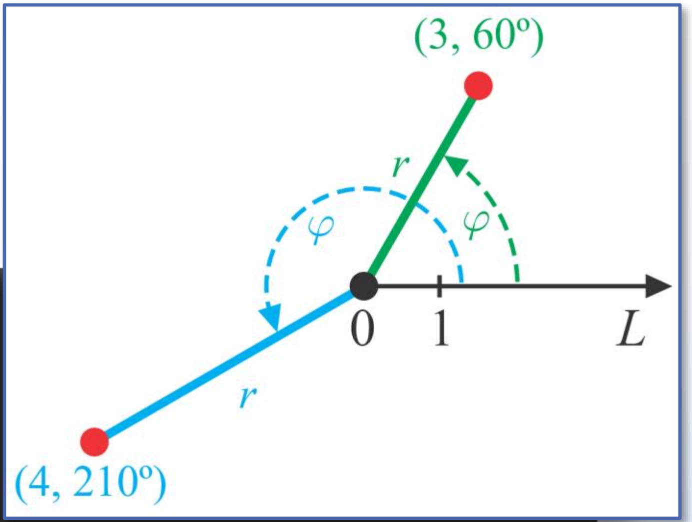



```
struct Polar {
  double r;
  double phi;
};
// INVARIANTS
// 0 <= radius
// 0 <= angle < 360

TEST(test_invariants) {
  Polar p;
  Polar_init(&p, -5, 225);
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);

  Polar_init(&p, 5, 405);
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}
```

Wait! Not done yet!
What didn't we think of?

If you "discover" a new bug or something you haven't accounted for, FIRST write new test cases to check for it!

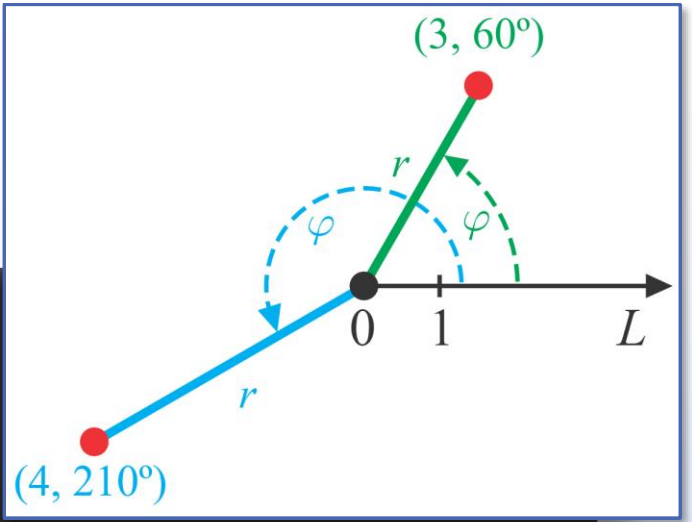
Testing an ADT




```
struct Polar {
  double r;
  double phi;
};

void Polar_init(Polar* p, double radius,
                double angle) {
  p->r = abs(radius);
  p->phi = angle;
  if (radius < 0) {
    p->phi = p->phi + 180;
  }
}
```

Then, modify your implementation with the goal of passing all the tests (including the new ones).

Testing an ADT

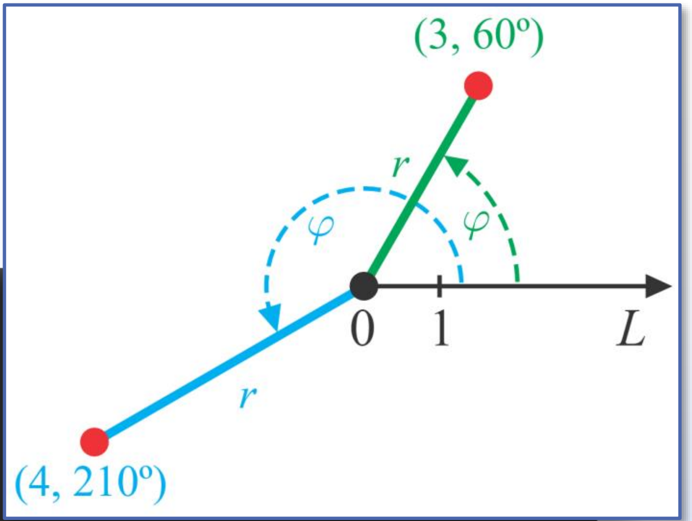



```
struct Polar {
  double r;
  double phi;
};
// INVARIANTS
// 0 <= radius
// 0 <= angle < 360

int test_invariants() {
  Polar p;
  Polar_init(&p, -5, 225);
  ASSERT_EQUAL(Polar_radius(&p), 5); // OK
  ASSERT_EQUAL(Polar_angle(&p), 45); // FAIL

  Polar_init(&p, 5, 405);
  ASSERT_EQUAL(Polar_radius(&p), 5);
  ASSERT_EQUAL(Polar_angle(&p), 45);
}
```

Finally, run tests to check if the implementation works.

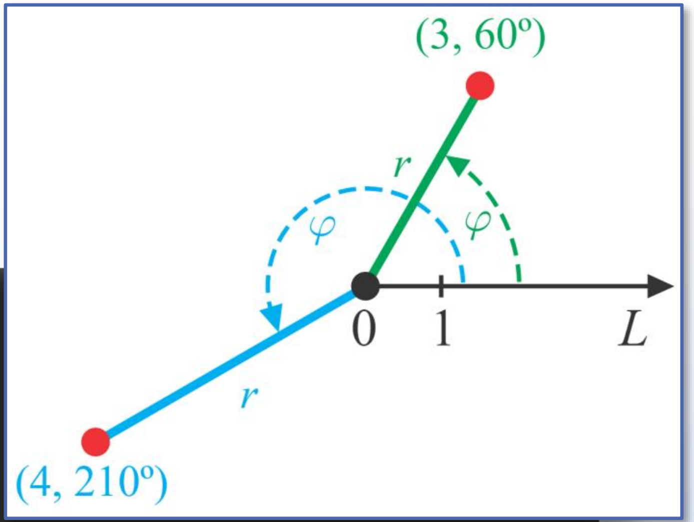
Testing an ADT

```
struct Polar {
  double r;
  double phi;
};

void Polar_init(Polar* p, double radius,
                double angle) {
  p->r = abs(radius);
  p->phi = angle;
  if (radius < 0) {
    p->phi = fmod(p->phi + 180, 360);
  }
}
```



Oops. We forgot to mod by 360. Our tests caught this bug!

Testing an ADT




```
struct Polar {
  double r;
  double phi;
};
// INVARIANTS
// 0 <= radius
// 0 <= angle < 360

int test_invariants() {
  Polar p;
  Polar_init(&p, -5, 225);
  ASSERT_EQUAL(Polar_radius(&p), 5);  // OK
  ASSERT_EQUAL(Polar_angle(&p), 45);  // OK

  Polar_init(&p, 5, 405);
  ASSERT_EQUAL(Polar_radius(&p), 5);  // OK
  ASSERT_EQUAL(Polar_angle(&p), 45);  // FAIL
}
```

Finally, run tests to check if the implementation works.

Another bug! Back to implementing again…

"Guard against Murphy, not Machiavelli![1]

- Do: Try to write test cases to catch bugs that people would *realistically* make.

```
Matrix_init(mat, 2, 2); Matrix_fill_border(mat, 1);
```

$$\texttt{ASSERT\_TRUE(Matrix\_equal(mat, } \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \texttt{));}$$

"Tricky because it's all border. Could expose a bug."

- Don't: Try to write test cases to catch bugs introduced by a devious coder.

$$\texttt{ASSERT\_EQUAL(*Matrix\_at(} \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} \texttt{, 42, 42), 1)}$$

"Tricky because maybe the element at (42, 42) secretly doesn't work."

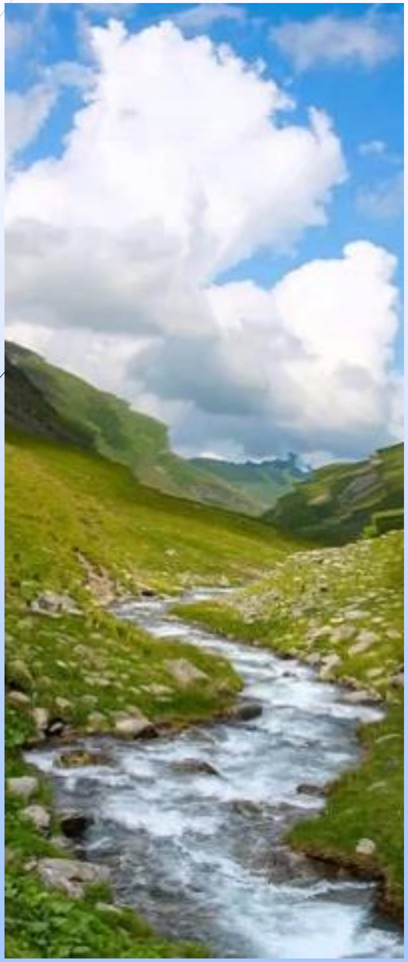
1 Attributed to Damian Conway

“Thorough testing with “small” test cases is sufficient to find most bugs within a system.”

The Small Scope Hypothesis

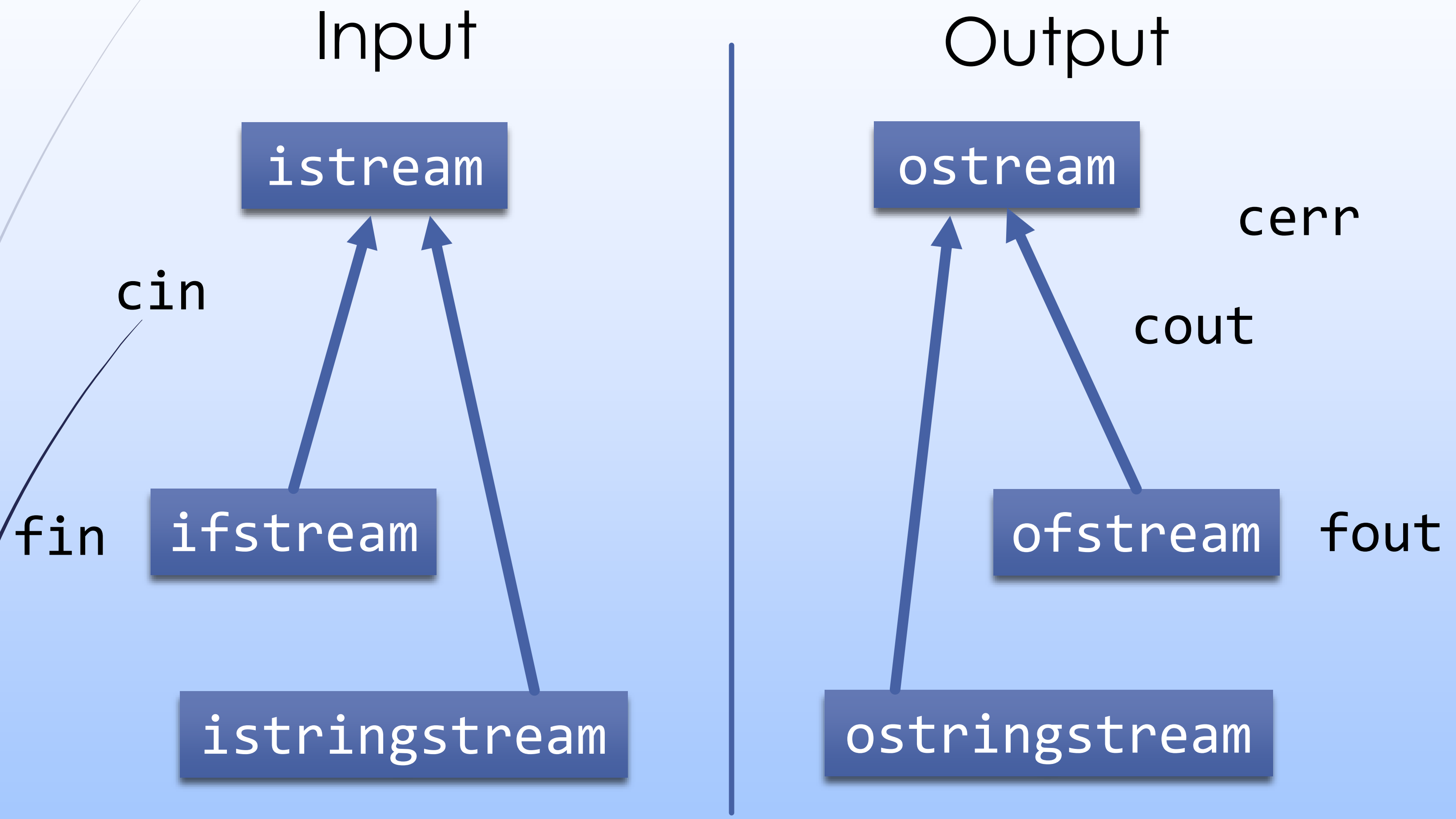
- Think about what makes two test cases **meaningfully different** for the function’s behavior.
  - Beyond a small size, just making test cases bigger doesn’t make them meaningfully different.
  - Testing with a 4x4 `Matrix` is just as good as a 5x5 `Matrix`

C++ Streams



- A stream acts as an abstraction over…
  - A **source** from which we can read data as **input**.
  - A **sink** to which we can write data as **output**.
  - Support character-based I/O from the terminal, files, etc.
- We've already been using a variety of streams:
  - `cout`, `cin`, file I/O, etc.

Different Kinds of Streams



stringstreams: Big Idea

Stringstreams and Testing

- `istringstream`
  - An input stream that uses a `string` as its source.
  - Useful for simulating stream input from a "hardcoded" `string`.

```
TEST(test_image_basic) {
  // A hardcoded PPM image
  string input = "P3\n2 2\n255\n255 0 0 0 255 0 \n";
  input += "0 0 255 255 255 255 \n";

  // Use istringstream for simulated input
  istringstream ss_input(input);
  Image *img = new Image;
  Image_init(img, ss_input);

  ASSERT_EQUAL(Image_width(img), 2);
  Pixel red = { 255, 0, 0 };
  ASSERT_TRUE(Pixel_equal(Image_get_pixel(img, 0, 0), red));
  delete img;
}
```

Can pass an `istringstream` where an `istream` is expected.

Stringstreams and Testing

- ostringstream
  - An output stream that writes into a string.
  - Useful for capturing output as a string that can be checked for correctness.

```
TEST(test_matrix_basic) {
  Matrix *mat = new Matrix;
  Matrix_init(mat, 3, 3);
  Matrix_fill(mat, 0);
  Matrix_fill_border(mat, 1);

  // Hardcoded correct output
  string output_correct = "3 3\n1 1 1 \n1 0 1 \n1 1 1 \n";

  // Capture output in ostringstream
  ostringstream ss_output;
  Matrix_print(mat, ss_output);
  ASSERT_EQUAL(ss_output.str(), output_correct);
  delete mat;
}
```

Can pass an ostringstream where an ostream is expected.

Stream Output (Reference)

- To write output into a stream, use the **insertion** operator (<<).
- The behavior is specific to the type of value inserted into the stream.

```
char c;
cout << c;
```

Writes a single character into the stream.

```
string s;
cout << s;
```

Writes the characters from the `string` into the stream.

```
double d;
cout << d;
```

Writes the `double` value formatted in floating point notation.

```
char *cstr;
cout << cstr;
```

Assumes it's pointing to a cstring. Prints out characters until `'\0'` is found.

Stream Input (Reference)

- To read input from a stream, use the **extraction** operator (>>).
- The behavior is specific to the type of object you are extracting into.

```
char c;
cin >> c;
```

Reads in a single character.

```
string s;
cin >> s;
```

Reads in one "word", delimited by whitespace.

```
int i;
cin >> i;
```

Attempts to parse the next characters from the stream as an integer value.

```
double d;
cin >> d;
```

Attempts to parse the next characters from the stream as a floating point value.

How do << and >> work? (Reference)

- The << and >> operators are binary operators.
  - LHS is a stream object.
  - For <<, the RHS is a value to insert into the stream.
  - For >>, the RHS is an object to store the value extracted out of the stream.

- The << and >> operators have two parts:
  - Side effect: Reading/Writing
  - Evaluation: Turns back into the stream on the LHS.
    - This allows chaining of several read/write operations.

```
cout << "The number is: " << num << "!";
          cout            << num << "!";
                    cout         << "!";
                           cout;
```