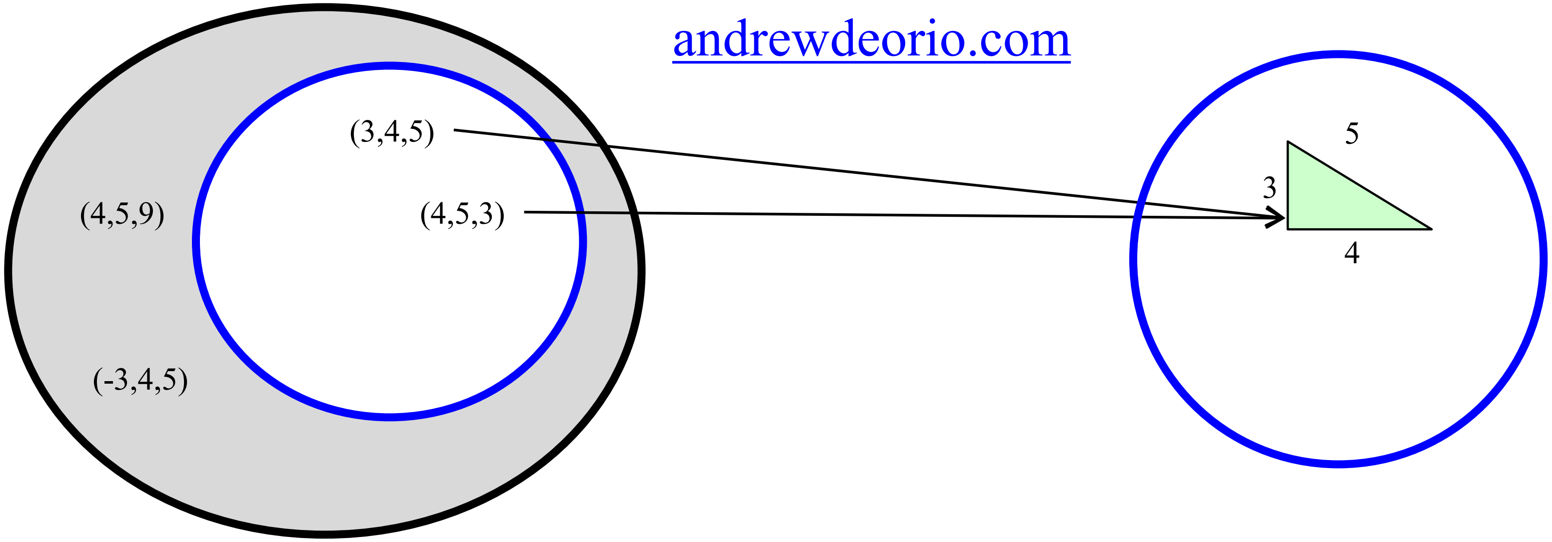
Data Abstraction

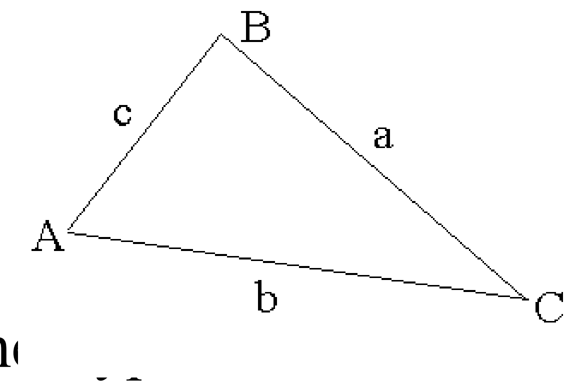
Andrew DeOrio

awdeorio@umich.edu

andrewdeorio.com



Review: compound types

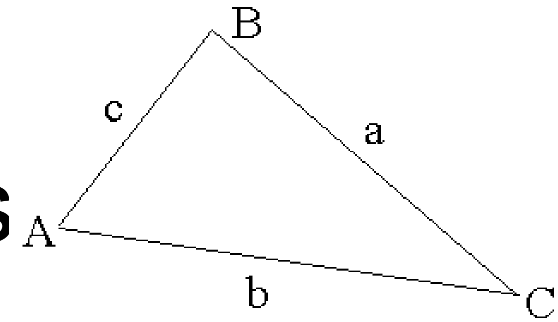


- A compound type “binds together” several oth
new type
- In C++, we can create a compound type using a `class`

```
class Triangle {
public:
  double a, b, c; //edge lengths
};
```

- `a`, `b`, and `c` are called *member data*

Review: member functions




```
class Triangle {
public:
  double a, b, c; //edge lengths
  double area() { //compute area
    double s = (a+b+c)/2;
    double a = sqrt(s*(s-a)*(s-b)*(s-c));
    return a;
  }
};
```

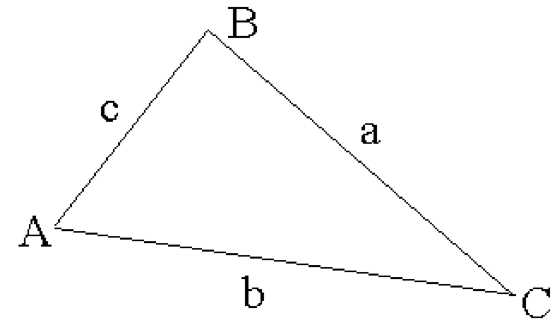
Heron's formula

$$s = \frac{a+b+c}{2}$$

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

- In addition to data, a `class` can contain *member functions*
- Because member functions are within the same scope as member data, they have access to the member data directly

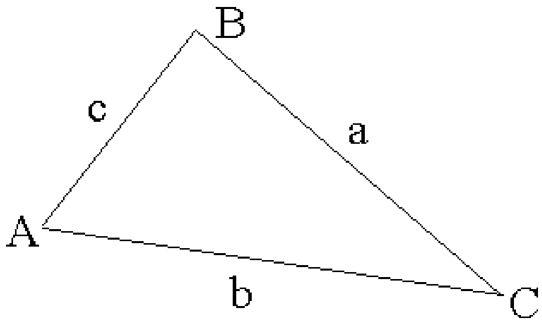
Review: constructors




```
class Triangle {
public:
  double a, b, c; //edge lengths
  double area() {/*...*/}
  Triangle(double a_in, double b_in, double c_in) {
    a = a_in;
    b = b_in;
    c = c_in;
  }
};
```

- Member data can be initialized using a constructor

Review: using classes




```
class Triangle {/*...*/};
int main() {
  Triangle t(3,4,5);
  cout << t.area() << "\n";
}
```

- Users of a class can call member functions

```
$ g++ test.cpp
$ ./a.out
area = 6
```

Abstraction in computer programs

- **Procedural abstraction** lets us separate *what* a procedure does from *how* it is implemented
- In C++, we use functions to implement procedural abstraction
- For example:

```
//returns n!, requires that n >= 0
int factorial(int n);
```

```
int factorial (int n) {
  if (n == 0) return 1;
  return n*factorial(n-1);
}
```

```
int factorial(int n) {
  int result = 1;
  while (n != 0) {
    result *= n;
    n -= 1;
  }
  return result;
}
```

Abstraction in computer programs

- **Data abstraction** lets us separate *what* a type is (and what it can do) from *how* the type is implemented

- In C++, we use a `class` to implement data abstraction
  - We can create an Abstract Data Type (ADT) using a `class`

- ADTs let us model complex phenomena
  - More complex than built-in data types like `int`, `double`, etc.

- ADTs make programs easier to maintain and modify
  - You can change the implementation and no users of the type can tell

Creating our ADT

- Let's build on our triangle compound data type to make it an Abstract Data Type
- We will write an abstract description of values and operations
  - *What* the data type does, but not *how*

`Triangle.h`

- Next, we will implement the ADT
  - *How* the data type works

`Triangle.cpp`

- Finally we will use our new ADT

`Graphics.cpp`

Creating our ADT

- What if we have two programmers?

- Alice and Bob agree on an abstraction

`Triangle.h`

- Alice codes `Triangle.cpp`
  - Implements ADT

`Triangle.cpp`

- Bob codes `Graphics.cpp`
  - Uses ADT

`Graphics.cpp`

Cartoon credit: xkcd.com

Triangle.h

Triangle ADT

```
class Triangle {
  //a geometric representation of a triangle
  //...
};
```

- Put only the class declaration (no implementations) in the file `Triangle.h`
- This file contains the abstraction
- We will add operations to
  - Create a triangle
  - Print some information (helpful for debugging)
  - Calculate the area

Triangle.h

Triangle ADT

```
class Triangle {
  //a geometric representation of a triangle
 public:
  //Creates a triangle from edge lengths
  //Requires that a, b, c are non-negative and
  //form a triangle
  Triangle(double a_in, double b_in, double c_in);
};
```

- A constructor initializes member variables when a `Triangle` is created

`Triangle.h`

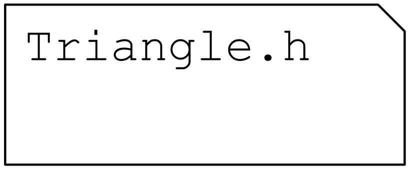
Triangle ADT

```
class Triangle {
  //a geometric representation of a triangle
 public:
  Triangle(double a_in, double b_in, double c_in);

  //Prints edge lengths
  void print();

  //Returns triangle area
  double area();
};
```

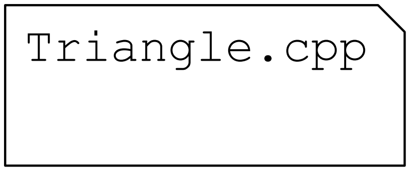
- Add member functions to print edges and compute area

What vs. How

- We now have an abstract description of values and operations
  - *What* the data type does



- Now, we need to implement this ADT
  - *How* the data type works
  - Member variables go in `Triangle.h`
  - Member function implementations go in `Triangle.cpp`



```
Triangle.h
```

Triangle ADT

```
class Triangle {
  //a geometric representation of a triangle
 public:
  Triangle(double a_in, double b_in,double c_in);
  void print();
  double area();

  //edges are non-negative and form a triangle
  double a, b, c;
};
```

- Add member variables

Triangle.cpp

Triangle ADT

- Function implementations go in `Triangle.cpp`
- Users of the Triangle ADT *do not need to see this file!*
- `#include "Triangle.h"` tells the compiler to "copy-paste" `Triangle.h` at the top of this file

```
#include "Triangle.h”
```

Triangle.cpp

Triangle ADT

```
Triangle::Triangle(double a_in, double b_in, double c_in){
  a = a_in;
  b = b_in;
  c = c_in;
}
```

- Constructor initializes member variables when a triangle is created
- `::` is the scope resolution operator
  - Tells compiler this is a *member function* inside the `Triangle class`'s scope

Representation invariant

```
Triangle::Triangle(double a_in, double b_in, double c_in){
  a = a_in;
  b = b_in;
  c = c_in;
}
```

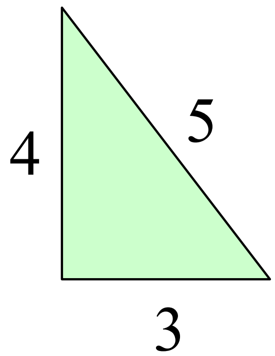
Triangle.cpp

- Recall that member variables are a class's representation
- The description of how a member variable should behave is the *representation invariant*

```
class Triangle {
  //...
  //edges are non-negative and form a triangle
  double a, b, c;
};
```

Triangle.h

Representation invariant

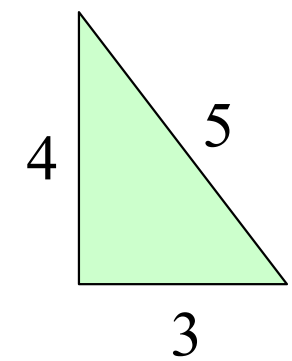


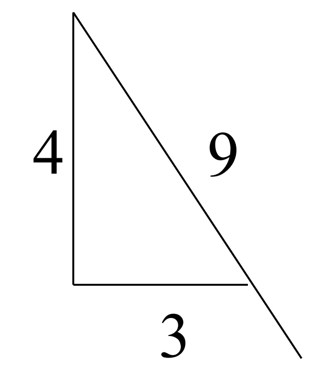

```
#include "Triangle.h"
int main() {
  Triangle t(3,4,5);

  // later in the program ...
  t.c = 9;
}
```

- What is wrong with this code?

The problem with public

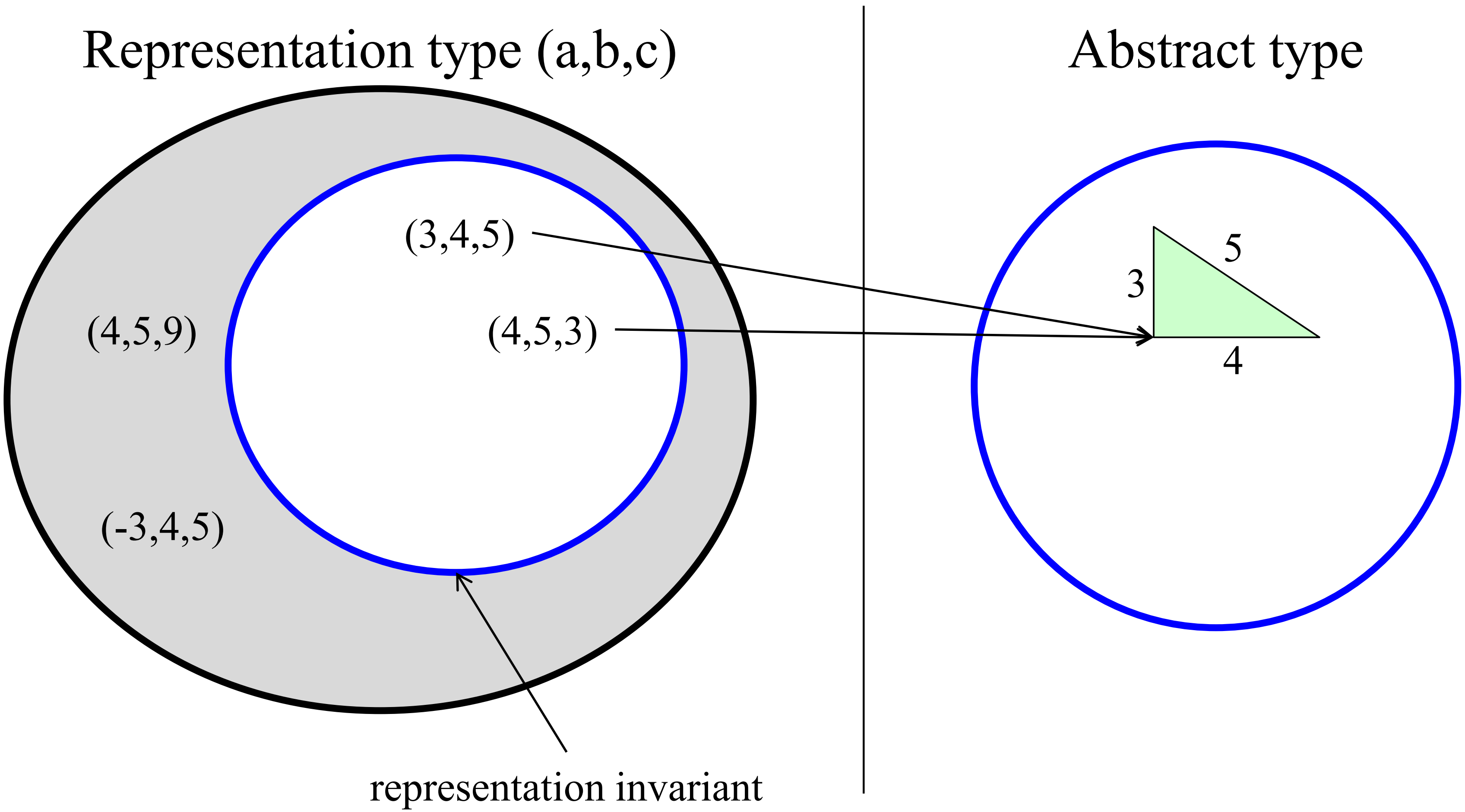
```
#include "Triangle.h"
int main() {
  Triangle t(3,4,5);

  // later in the program ...
  t.c = 9;
}
```



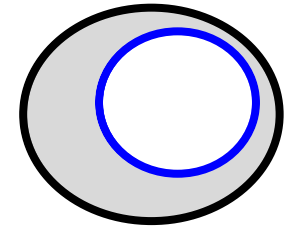


- Problem: `class`'s internal representation of a triangle is no longer a triangle!
- We have violated the *representation invariant*

Representation invariant



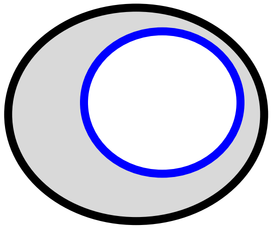
Solution: private



```
class Triangle {
  //...
 private:
  //edges are non-negative and form a triangle
  double a, b, c;
};
```

- An ADT's member variables should be `private`
- This is an aspect of *information hiding*

```
int main() {
  Triangle t(3,4,5);
  t.c = 9; //compiler error
}
```



Representation invariant

```
#include "Triangle.h"
int main() {
  Triangle t(3,4,9);
  //...
}
```

- Same problem, this time in the constructor!

Recall assert()

- `assert()` is a programmer's friend for debugging
- Does nothing if *`expression`* is true
- Exits and prints an error message if *`expression`* is false
- We can *assert* that the representation invariant is true

Triangle.cpp

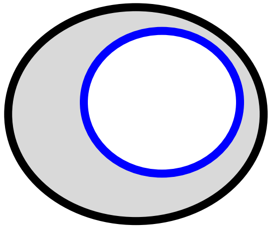
```
#include <cassert>
Triangle::Triangle(double a_in, double b_in, double c_in){
  a = a_in;
  b = b_in;
  c = c_in;
  assert(expression);
}
```

Exercise

- Write an assertion that checks the representation invariant
  - Edges are non-negative and form a triangle
- Recall: the sum of the lengths of any two sides of a triangle always exceeds the length of the third side

`Triangle.cpp`

```
#include <cassert>
Triangle::Triangle(double a_in, double b_in, double c_in){
  a = a_in;
  b = b_in;
  c = c_in;
  assert(expression);
}
```



Solution

- Write an assertion that checks the representation invariant
  - Edges are non-negative and form a triangle
- Recall: the sum of the lengths of any two sides of a triangle always exceeds the length of the third side

Triangle.cpp

```
#include <cassert>
Triangle::Triangle(double a_in, double b_in, double c_in){
  a = a_in;  b = b_in;  c = c_in;
  assert(
          (a + b > c) &&
          (a + c > b) &&
          (b + c > a)
         );
}
```

Triangle.cpp

Triangle ADT

```
//...
double Triangle::area() {
  double s = (a + b + c) / 2;
  return sqrt(s*(s-a)*(s-b)*(s-c));
}

void Triangle::print() {
  cout << "a=" << a << " b=" << b << " c=" << c
       << "\n";
}
```

- Implementations for `area()` and `print()`

Using our ADT

- We now have an abstract description of values and operations
  - *What* the data type does, but not *how*

`Triangle.h`

- We have an implementation of this ADT
  - *How* the data type works

`Triangle.cpp`

- Now, let's use our new ADT

`Graphics.cpp`

Cartoon credit: xkcd.com

Graphics.cpp

A use for triangles

- In computer graphics, 3D surfaces can be modeled using connected triangles, called a triangle mesh
- Let’s calculate the area of this surface

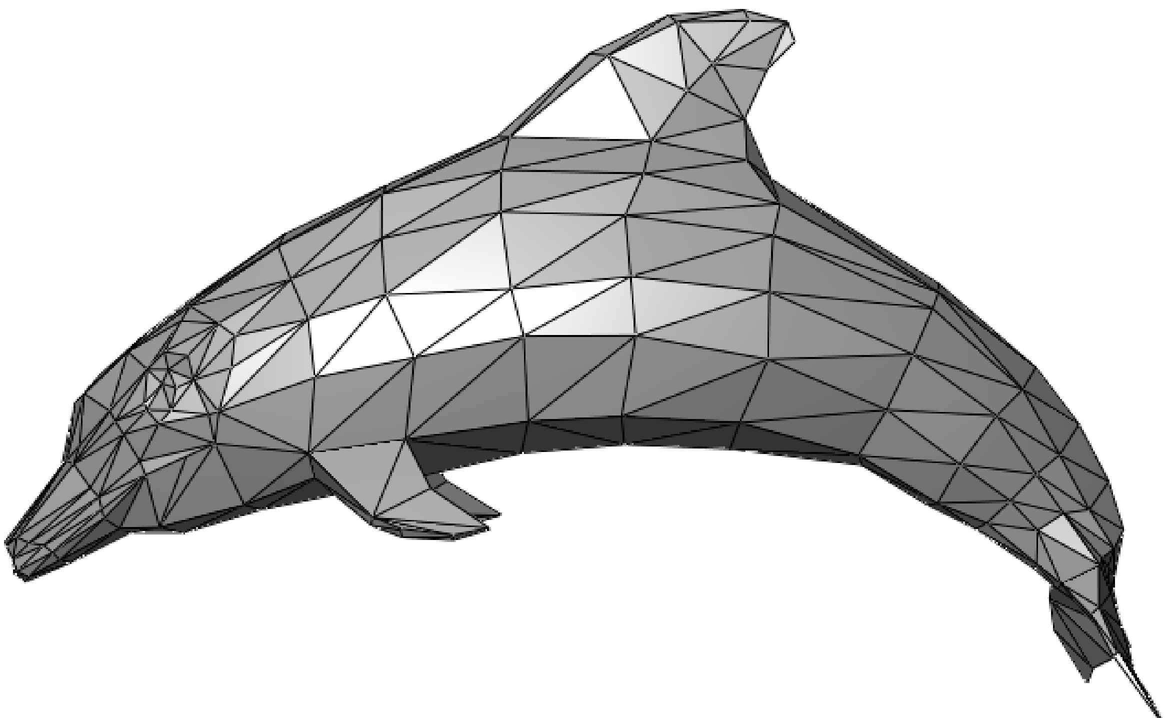


Image: wikipedia.org

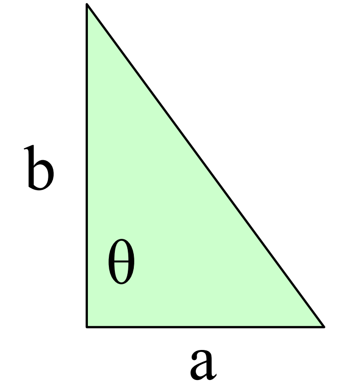
Graphics.cpp

Triangle ADT

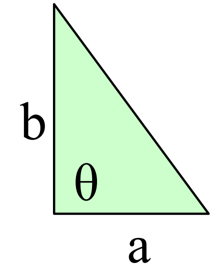
```
#include "Triangle.h"
int main() {
  const int SIZE = 3;
  Triangle mesh[SIZE];
  // fill with triangles ...

  double area = 0;
  for (int i=0; i<SIZE; ++i) {
    area += mesh[i].area();
  }
  cout << "total area = " << area << "\n";
}
```

```
$ g++ Graphics.cpp Triangle.cpp
$ ./a.out
total area = 22.3196
```

Exercise



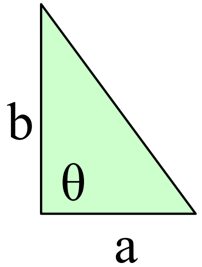
- There is more than one way to represent a triangle
- Let's change our representation from 3 edges to 2 edges and an angle: `a`, `b`, and `theta`
- Do we need to change *what* our ADT does?
- Do we need to change *how* our ADT does it?
- Do we need to change anything in `Triangle.h`? What?
- Do we need to change anything in `Triangle.cpp`? What?
- Do we need to change anything in `Graphics.cpp`? What?
- Will Alice, Bob or both need to change their code?



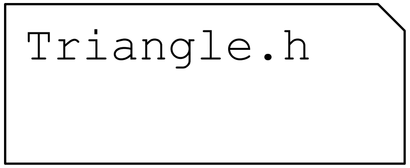
Solution

- Do we need to change *what* our ADT does?
  - No, don't touch `public` function inputs or outputs
- Do we need to change *how* our ADT does it?
  - Yes, because internal representation is different now

Solution

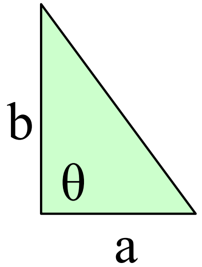


- Do we need to change anything in `Triangle.h`? What?
- Yes. Only the `private` member variables




```
class Triangle {
  //...
 private:
  //edges a and b are separated by angle theta
  //and form a triangle
  double a, b;  //edges
  double theta; //angle
};
```

Solution

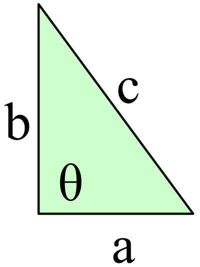


- Do we need to change anything in `Triangle.cpp`? What?
- Yes. The function implementations change.

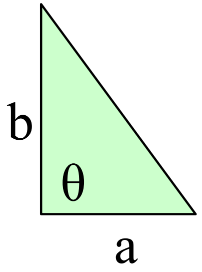
```
Triangle::Triangle(double a_in, double b_in, double c_in) {
  a = a_in;
  b = b_in;
  assert(/*...*/);
  theta = acos((a*a + b*b - c_in*c_in) / (2*a*b));
}
```

Law of cosigns

$$\Theta = \arccos\left(\frac{a^2 + b^2 - c^2}{2ab}\right)$$



Solution

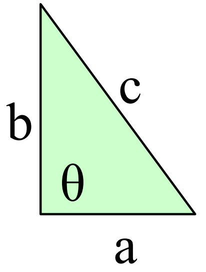


- Do we need to change anything in `Triangle.cpp`? What?
- Yes. The function implementations change.

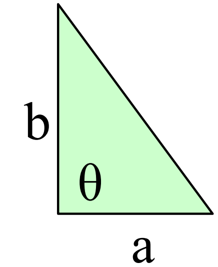
```
Triangle::print() {
  double c = sqrt(a*a + b*b + 2*a*b*cos(theta));
  cout << "a=" << a << " b=" << b << " c=" << c
       << "\n";
}
```

<u>Law of cosigns</u>

$$c = \sqrt{a^2 + b^2 - 2ab \cos \Theta}$$

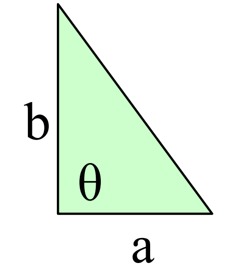


Abstraction exercise



- Do we need to change anything in `Graphics.cpp`? What?
  - No! That's the cool part ☺
- Will Alice, Bob or both need to change their code? Just Alice.


```
int main() {
  //...

  double area = 0;
  for (int i=0; i<SIZE; ++i) {
    area += mesh[i].area();
  }
  cout << "total area = " << area << "\n";
}
```



The power of abstraction

- We changed the implementation, but not the abstraction
  - Modified `private` member variables
  - Modified `public` function implementations
- We changed *how* the abstract data type worked
- We did not change *what* the abstract data type did
- Because the abstraction remained the same, our old code that used the abstract data type still worked
- This is especially important when you have many people working on one project
- This is a big benefit of ADTs!

Building on ADTs

- The next few lectures will build on abstract data types
- Subtypes and Polymorphism
  - Using C++ derived types (AKA inheritance) to create hierarchies of Abstract Data Types
- Interfaces
  - Using C++ pure virtual functions to omit the member variables
- Container Abstract Data Types