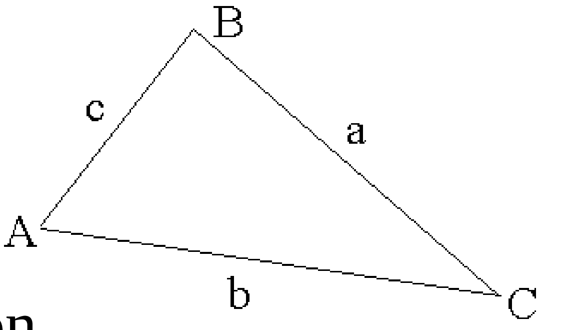
Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Compound Types

Representing a triangle

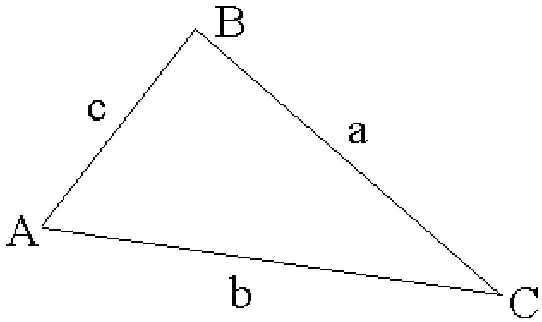


- A triangle can be represented by a combination of vertex angles and edge lengths
- Heron's formula lets us calculate the area of a triangle using only its edge lengths

$$s = \frac{a+b+c}{2} \qquad T = \sqrt{s(s-a)(s-b)(s-c)}$$

- Let's write a function calculate the area of a triangle

Representing a triangle



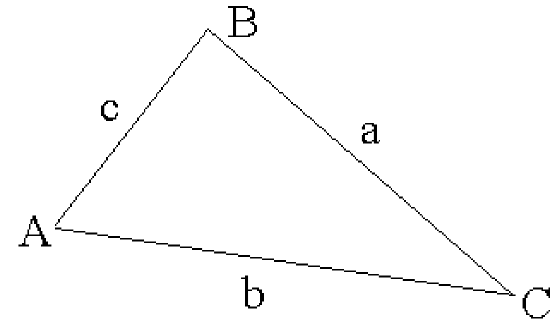

```
#include <cmath> //for sqrt()
double Triangle_area(double a, double b, double c) {
  double s = (a + b + c) / 2;
  double area = sqrt(s*(s-a)*(s-b)*(s-c));
  return area;
}

int main() {
  double a=3, b=4, c=5; // triangle edges
  cout << "area = " << Triangle_area(a, b, c) << endl;
}
```

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

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

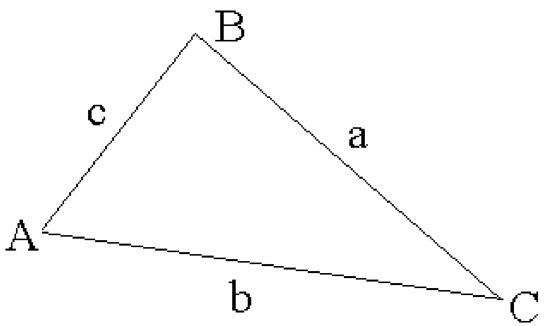
```
$ ./a.out
area = 6
```

Multiple triangles



- What if we have more than one triangle?
- For example, 3-4-5, 5-12-13, 8-15-17
- Let's use 3 arrays: one for all the a edges, another for the b edges and a third for the c edges

Multiple triangles

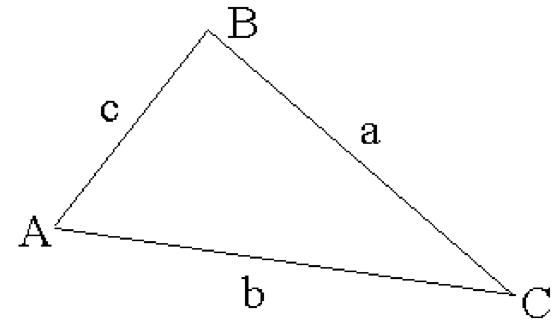


- Three triangles: 3-4-5, 5-12-13, 8-15-17

```
const int SIZE = 3;
double a[SIZE] = {3,  5,  8};
double b[SIZE] = {4, 12, 15};
double c[SIZE] = {5, 13, 17};

for (int i=0; i<SIZE; ++i)
  cout << "area = " << Triangle_area(a[i],b[i],c[i])
       << endl;
```

```
$ ./a.out
area = 6
area = 30
area = 60
```

Compound type




```
const int SIZE = 3;
double a[SIZE] = {3,  5,  8};
double b[SIZE] = {4, 12, 15};
double c[SIZE] = {5, 13, 17};
```

- Problem: it is not clear that `a[i]` is related to `b[i]` and `c[i]`
- Instead of three separate arrays, what we really want is a type that can “bind together” several other types into one “meta-type”.
- This is called a compound type

Compound type

- **Kinds of objects in C++**
- **Atomic**
  - Also known as **primitive**
  - `int`, `double`, `char`, etc.
  - Pointer types.

- **Arrays** (homogeneous)
  - A *contiguous* sequence of objects of the same type

- **Compound** (heterogeneous)
  - A compound object made up of **member** subobjects
  - The members and their types are defined by a **struct** or **class**

Introducing struct

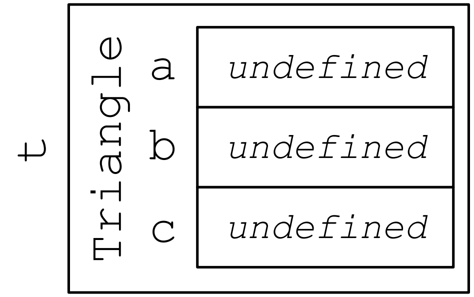
- C++ supports a compound type: the `struct`
- This `struct` contains a triangle’s edge lengths:

```
struct Triangle {
  double a,b,c; //edge lengths
};
```

- This statement declares a new type `Triangle`, but does not define any variables of that type
- By C++ convention, we name new types with a capital letter:
  - `struct Triangle { /*...*/ };`

Introducing struct

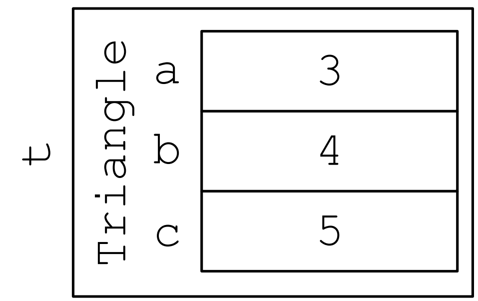
```
struct Triangle {
  double a,b,c; //edge lengths
};

int main() {
  Triangle t;
}
```



- This code define a new variable of the `Triangle` type
- In memory, we get storage for each member inside the struct
- The values of `a`, `b` and `c` are undefined

Initializing a struct

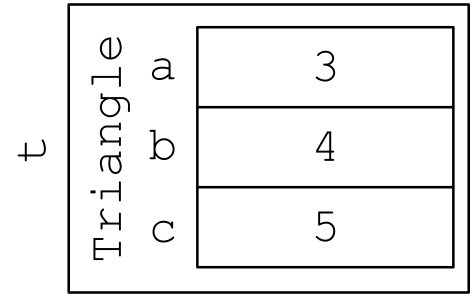
```
struct Triangle {
  double a,b,c; //edge lengths
};

int main() {
  Triangle t;
  t.a = 3;
  t.b = 4;
  t.c = 5;
}
```



- Initialize each member one at a time

Initializing a struct

```
struct Triangle {
  double a,b,c; //edge lengths
};

int main() {
  Triangle t = {3,4,5};
}
```

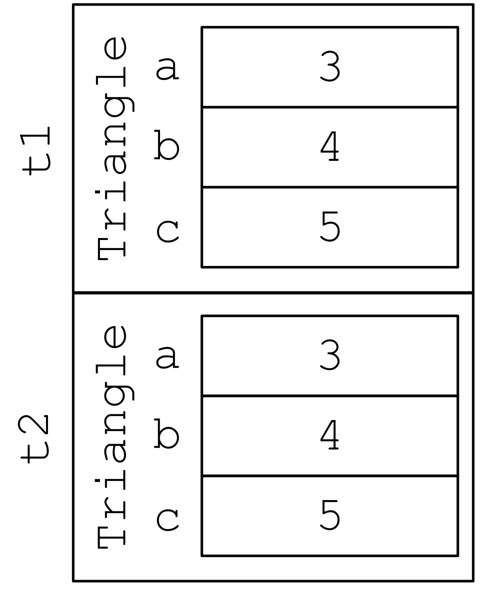


- Initialize all members at once

Assigning a struct

```
struct Triangle {
  double a,b,c; //edge lengths
};

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

- Assign one `struct` to another



Passing a struct to a function

- Like other types, structs can be passed as function arguments
- Structs are passed by value, unlike arrays

```
//EFFECTS: computes the area of Triangle t
double Triangle_area(Triangle t);
```

- Exercise: write this function using Heron's formula:

$$s = \frac{a+b+c}{2} \qquad T = \sqrt{s(s-a)(s-b)(s-c)}$$

Passing a struct to a function

```
//EFFECTS: computes the area of Triangle t
double Triangle_area(Triangle t) {
  double s = (t.a + t.b + t.c) / 2;
  double area = sqrt(s *(s-t.a)*(s-t.b)*(s-t.c));
  return area;
}
```

Passing a struct to a function

```
struct Triangle {
  double a,b,c; //edge lengths
};

double Triangle_area(Triangle t) {
  //...
}

int main() {
  Triangle t = {3,4,5};
  cout << "area = " << Triangle_area(t) << endl;
}
```

```
$ ./a.out
area = 6
```

Passing a struct to a function

- Problem: structs can become very large, and pass-by-value is inefficient!
- Solution 1: pass by reference

```
//EFFECTS: computes the area of Triangle t
double Triangle_area(Triangle &t);
```

- Solution 2: pass by pointer

```
//EFFECTS: computes the area of Triangle t
double Triangle_area(Triangle *t);
```

- Problem: the function could mistakenly change the contents of the *caller's* copy of `Triangle t`

Recall const

- Problem: the function could mistakenly change the contents of the *caller's* copy of `Triangle t`
- Solution: **`const`**

```
double Triangle_area(const Triangle &t);
double Triangle_area(const Triangle *t);
```

- `const` is a type-qualifier, which adds a property
- `const` means "the compiler won't let you modify this variable"
- Best of both worlds compared to pass-by-value
  - No expensive copy
  - Safety guarantee that the function can't change caller's variable

Exercise: pointer-to-struct

- There are two ways to access the data inside a pointer-to-struct

```
double Triangle_area(Triangle *t) {
   (*t).a + ... //the hard way with pointers
   *t.a + ...   //the wrong way with pointers
   t->a + ...   //the easy way with pointers
}
```

- Exercise: write a new version of `Triangle_area()` that passes a `Triangle` by pointer, and promises not to modify the pointed-to object

Exercise: pointer-to-struct

- Exercise: write a new version of `Triangle_area()` that passes a `Triangle` by pointer, and promises not to modify the pointed-to object

```
double Triangle_area(const Triangle *t) {
  double s = (t->a + t->b + t->c) / 2;
  double area = sqrt(
    s * (s - t->a) * (s - t->b) * (s - t->c)
  );
  return area;
}
```

Composable data types

- So far, we have seen several mechanisms used to extend the basic data types (`int`, `char`, etc.)
  - Arrays
    ```
    int a[3] = {1,2,3};
    ```
  - Pointers
    ```
    int *p = a;
    ```
  - References
    ```
    int &a_ref = a[0];
    ```
  - Structs
    ```
    struct x {
      int  i;
      char c;
    };
    ```

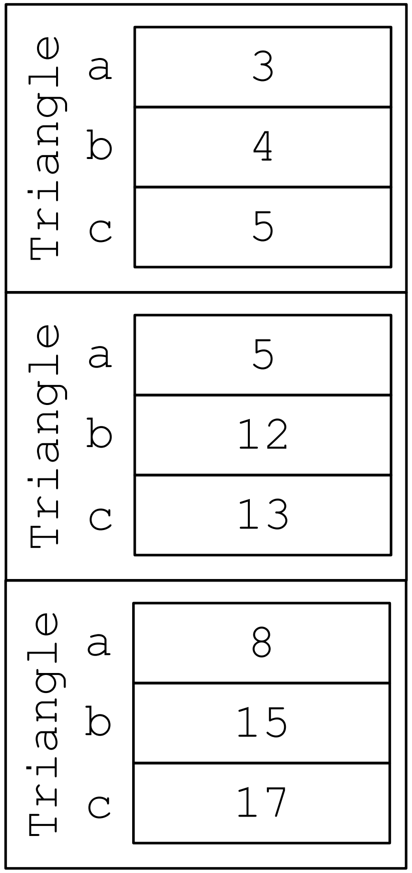
Types are composable.

Once you have declared a type struct x, that type can be used to define a pointer to struct x, an array of struct x, or even a struct that contains an element of struct x.

Arrays of structs

- Let’s compose structs and arrays

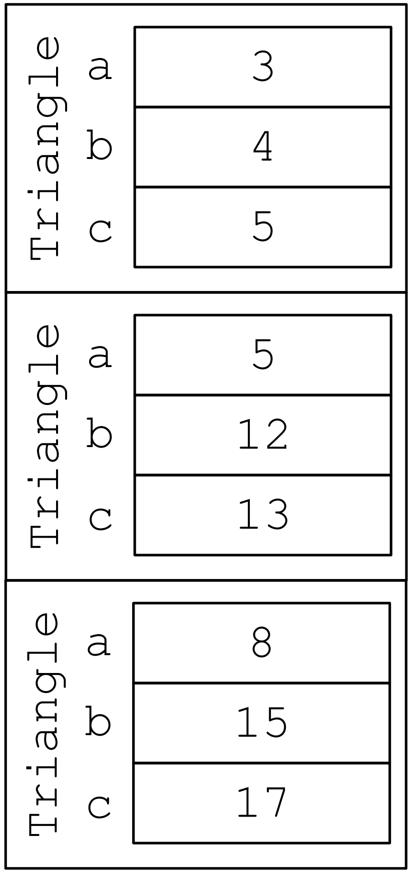
```
const int SIZE = 3;
Triangle triangles[SIZE];
triangles[0].a=3;
triangles[0].b=4;
triangles[0].c=5;
triangles[1].a=5;
triangles[1].b=12;
triangles[1].c=13;
triangles[2].a=8;
triangles[2].b=15;
triangles[2].c=17;
```



Exercise: arrays of structs

- Call `Triangle_area()` on each `Triangle` in the array using *traversal by pointer*

```
const int SIZE = 3;
Triangle triangles[SIZE];
triangles[0].a=3;
triangles[0].b=4;
triangles[0].c=5;
triangles[1].a=5;
triangles[1].b=12;
triangles[1].c=13;
triangles[2].a=8;
triangles[2].b=15;
triangles[2].c=17;
```



Exercise: arrays of structs

```
const int SIZE = 3;
Triangle triangles[SIZE];
//initialize triangles...
```

`++t` moves the pointer to the next struct

```
for (Triangle *t=triangles; t<triangles+SIZE; ++t)
  cout << "area = " << Triangle_area(t) << endl;
```

```
$ ./a.out
area = 6
area = 30
area = 60
```

struct vs. array

struct

- `struct Triangle{/*…*/};`
- Stores group of data
- Heterogeneous
  - Different types
- Access by name
- Default pass-by-value
- Creates a custom type

array

- `double edges[3];`
- Stores group of data
- Homogeneous
  - All the same type
- Access by index
- Default pass-by-pointer
- Does not create a new type

const variables

```
const int SIZE = 3;
Triangle triangles[SIZE];

// initialize ...

for (Triangle *t=triangles; t<triangles+SIZE; ++t)
  cout << "area = " << Triangle_area(t) << endl;
```

This is a good use of `const`, because we reuse the value in multiple places.

const variables

```
#define SIZE 3
Triangle triangles[SIZE];

// initialize ...

for (Triangle *t=triangles; t<triangles+SIZE; ++t)
  cout << "area = " << Triangle_area(t) << endl;
```

This works, but is bad style. It has no type!

const variables

```
const int SIZE = 4;
Triangle triangles[SIZE];

// initialize ...

for (Triangle *t=triangles; t<triangles+SIZE; ++t)
  cout << "area = " << Triangle_area(t) << endl;
```

Add another Triangle?
No problem. Just change one piece of code

On to classes!

struct

- Heterogeneous aggregate data type
- **C style**
- **Contains only data**
- **Undefined by default**
- **All data is accessible**

class

- Heterogeneous aggregate data type
- **C++ style**
- **Contains data and functions**
- **Constructors can be used to initialize**
- **Control of data access**

Introducing classes

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

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

- A `class` contains both data *and* functions
- These are called *member data* and *member functions*

Member data and 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;
  }
};
```

- Because member functions are within the same scope as member data, they have access to the member data directly

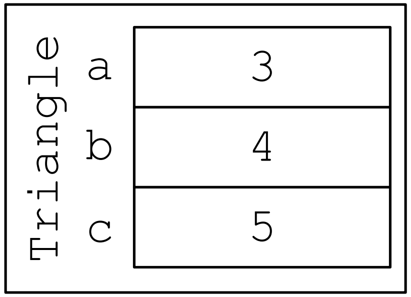
Using classes

```
class Triangle {
public:
  double a,b,c;
  double area() {/*compute area*/}
};
int main() {
  Triangle t;
  t.a=3; t.b=4; t.c=5;
  cout << "area = " << t.area() << endl;
}
```

- `public` means members are accessible from outside the `class`
- From outside scope, access `class` members just like a `struct`

```
$ ./a.out
area = 6
```

Using classes




```
class Triangle {
public:
  double a,b,c;
  double area() {/*compute area*/}
};
int main() {
  Triangle t;
  t.a=3; t.b=4; t.c=5;
  cout << "area = " << t.area() << endl;
}
```

- In memory, we get storage for each member inside the `class`, but *not* the functions. This is like a `struct`.
- Why? Because functions are the same for all `Triangle` objects

Using classes

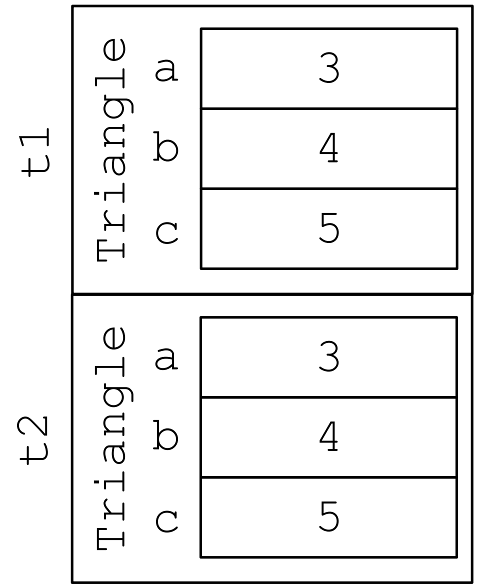
```
int main() {
  Triangle t;
  t.a=3; t.b=4; t.c=5;
  cout << "area = " << t.area() << endl;
}
```

- Calling a member function is just like evaluating a function:

1. Evaluate the arguments
2. Create a stack frame
3. Pass the arguments
4. Execute the function
5. Replace the function call with the result
6. Destroy the frame

Using classes

```
class Triangle {
public:
  double a,b,c;
  double area() {/*compute area*/}
};
int main() {
  Triangle t1;
  t1.a=3; t1.b=4; t1.c=5;
  Triangle t2 = t1;
}
```



- Just like a `struct`, we can copy a `class` object

Exercise: array of classes

```
struct Triangle {/*...*/};
int main() {
  const int SIZE = 3;
  Triangle triangles[SIZE];
  // initialize ...

  for (Triangle *t=triangles; t<triangles+SIZE; ++t)
    cout << "area = " << Triangle_area(t) << endl;
}
```

- Just like a `struct`, we can create arrays of `class` objects
- Exercise: rewrite the **bold code** to use C++ classes instead of structs

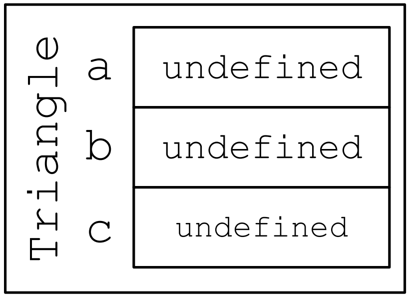
Exercise: array of classes

```
class Triangle {/*...*/};
int main() {
  const int SIZE = 3;
  Triangle triangles[SIZE];
  // initialize ...

  for (Triangle *t=triangles; t<triangles+SIZE; ++t)
    cout << "area = " << t->area() << endl;
}
```

- Call a member function on a pointer-to-class just like accessing data inside a pointer-to-struct

Initialization problem

```
class Triangle {
public:
  double a,b,c;
  double area() {/*compute area*/}
};

int main() {
  Triangle t;
  // forget to initialize
  t.area(); //garbage!
}
```



- Just like a struct, values are undefined for a new class object
- Uninitialized values are common source of bugs

Initialization solution: constructors

```
class Triangle {
public:
  double a,b,c;
  double area() {/*compute area*/}
  Triangle() { a=0; b=0; c=0; }
};
```

- A constructor is a member function that has the same name as the class
- A constructor runs automatically when a new object is defined
- A constructor is typically used to initialize member variables

Anatomy of a constructor

```
class Triangle {
public:
  double a,b,c;
  double area() {/*...*/}
  Triangle() { a=0; b=0; c=0; }
};
```

Same name as class

No return value

Initializes member variables

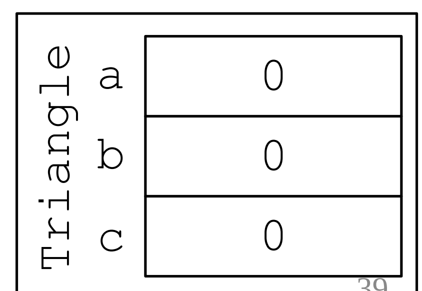
No input: “default constructor”

```
int main() {
  Triangle t;
  t.area();
}
```

```
$ ./a.out
area = 0
```

Constructor executes automatically

New object is initialized!



Constructor exercise

- What does this code print? Why?

```
int main() {
  Triangle t1, t2;
  cout <<  t1.a << t1.b << t1.c << endl;
  cout <<  t2.a << t2.b << t2.c << endl;
}
```

Constructor exercise

- What does this code print? Why?

```
int main() {
  Triangle t1, t2;
  cout <<  t1.a << t1.b << t1.c << endl;
  cout <<  t2.a << t2.b << t2.c << endl;
}
```

```
$ ./a.out
000
000
```

The default constructor initializes a, b, and c.

Another initialization problem

```
class Triangle {
public:
  double a,b,c;
  double area() {/*...*/}
  Triangle() { a=0; b=0; c=0; }
};
```

- Problem 1: a Triangle with 0 length edges doesn't make much sense
- Problem 2: Initializing lengths is still a pain

```
Triangle t;
t.a=3, t.b=4, t.c=5;
```

Another initialization solution

```
class Triangle {
public:
  double a,b,c;
  double area() {/*...*/}
  Triangle() { a=0; b=0; c=0; }
  Triangle(double a_in, double b_in, double c_in)
  { a=a_in; b=b_in; c=c_in; }
};
```

- Add a second constructor
- Constructors can take arguments, just like functions

Function overloading

```
class Triangle {
  //...
  Triangle() {/*...*/}
  Triangle(double a_in, double b_in, double c_in)
  {/*...*/}
};
```

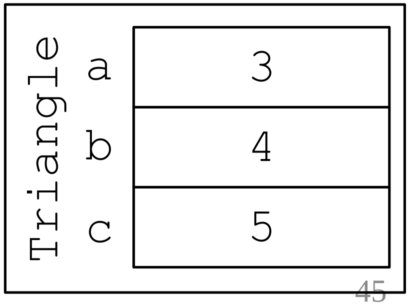
- This is called *function overloading:* two different functions with the same name, but different prototypes
- Since the compiler knows the argument types, it can select the correct constructor when a new object is created

Another initialization solution

```
class Triangle {
public:
  double a,b,c;
  double area() {/*...*/}
  Triangle() { a=0; b=0; c=0; }
  Triangle(double a_in, double b_in, double c_in)
  { a=a_in; b=b_in; c=c_in; }
};
int main() {
  Triangle t = Triangle(3,4,5);
  t.area();
}
```

New constructor runs

```
$ ./a.out
area = 6
```



Another initialization solution

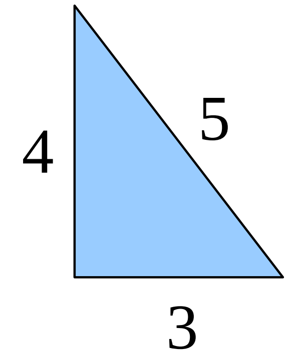
```
class Triangle {
public:
  double a,b,c;
  double area() {/*...*/}
  Triangle() { a=0; b=0; c=0; }
  Triangle(double a_in, double b_in, double c_in)
  { a=a_in; b=b_in; c=c_in; }
};
int main() {
  Triangle t = Triangle(3,4,5);
  Triangle t(3,4,5);
  t.area();
}
```

Alternate syntax

```
$ ./a.out
area = 6
```

Triangle		
	a	3
	b	4
	c	5

The problem with public




```
class Triangle {/*...*/};
int main() {
  Triangle t(3,4,5);

  // later in the code ...
  t.c = 9;
  cout << "area = " << t.area() << endl;
}
```

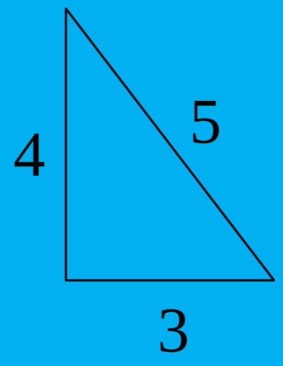
- What is wrong with this code?

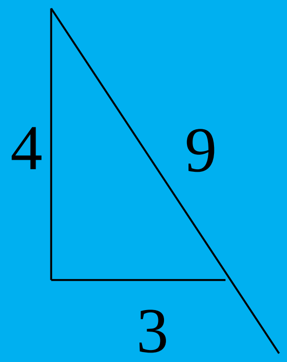
The problem with public

```
class Triangle {/*...*/};
int main() {
  Triangle t(3,4,5);

  // later in the code ...
  t.c = 9; //bad!
  cout << "area = " << t.area() << endl;
}
```

```
$ ./a.out
area = -nan
```

- Problem: `class's` internal representation of a triangle is no longer a triangle!





Solution: private member variables

```
class Triangle {
private:
  double a,b,c;
public:
  double area() {
    double s = (a + b + c) / 2;
    double area = sqrt(s*(s-a)*(s-b)*(s-c));
  }
  Triangle() { a=0; b=0; c=0; }
  Triangle(double a_in, double b_in, double c_in)
  { a=a_in; b=b_in; c=c_in; }
};
```

member variables are `private` to class

member functions are `public` to outside

Member functions *can* access `private` members

Solution: private member variables

```
class Triangle {
private:
  double a,b,c;
public:
 double area() {/*...*/}
  Triangle() {/*...*/}
  Triangle(double a_in, double b_in, d
  {/*...*/}
};
int main() {
  Triangle t(3,4,5);
  t.c = 9; //compiler error
  cout << "area = " << t.area() << endl;
}
```

Non-members *cannot* access `private` members

Compiler helps catch programming error

Non-members *can* access `public` members

public vs. private

- By default, every member of a class is `private`
- A private member is visible only to other members of this class.
- The `public` keyword is used to signify that everything after it is visible to anyone who sees the class declaration, not just members of this class.
- Usually, we make member variables `private`
- `public` member variables often indicate a bad design

get and set functions

- For convenience, some classes include functions to `get` and `set` member functions
- A `get` function is a `public` function that returns a copy of a `private` member variable

```
class Triangle {
  //...
 public:
  //EFFECTS: returns edge a, b, c
  double get_a() { return a; }
  double get_b() { return b; }
  double get_c() { return c; }
};
```

get and set functions

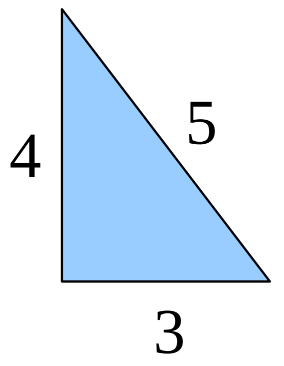
- A `set` function is a `public` function that modifies a `private` member variable

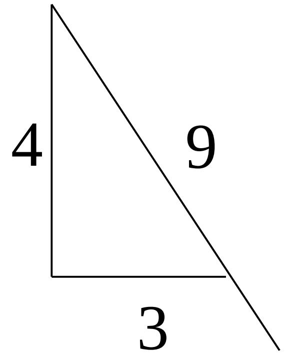
```
class Triangle {
  //...
 public:
  //REQUIRES: a,b,c are non-negative and form a
  //          triangle
  //MODIFIES: a, b, c
  //EFFECTS: sets length of edge a, b, c
  void set_a(double a_in);
  void set_b(double b_in);
  void set_c(double c_in);
};
```

get and set functions

- `set` functions allow you to run extra code when a member variable changes, for example:

```
class Triangle {
  //...
 public:
  void set_a(double a_in) {
    a = a_in;
    // add a check to make sure a,b,c still
    // form a triangle
  }
};
```





struct vs. class

struct

- Heterogeneous aggregate data type
- **C style**
- **Contains only data**
- **Undefined by default**
- **All data is accessible**

class

- Heterogeneous aggregate data type
- **C++ style**
- **Contains data and functions**
- **Constructors can be used to initialize**
- **Control of data access**

struct vs. class

class

- Contains data and functions
  - Member variables and member functions
- Constructors can be used to initialize
  - Constructors run automatically
- Control of data access
  - `public` vs. `private`

Exercise

- Write a class called `Rectangle`
- Include appropriate member variables
- Include a default constructor
- Include a second constructor to initialize member variables
- Write one `set()` function that sets all member variables
- Include an `area()` function
- Use `public` and `private` to expose member functions and hide member variables
- Write a main function that defines a `Rectangle` variable of size 2x4, and then calls the `area()` function

Exercise

```
class Rectangle {
  double a,b; //private by default
public:
  Rectangle() { a=0; b=0; }
  Rectangle(double a_in, double b_in)
  { a=a_in; b=b_in; }
  void set(double a_in, double b_in)
  { a=a_in; b=b_in; }
  double area() { return a * b; }
  void print()
  { cout << "a=" << a << " b=" << b << endl; }
};
```

Exercise

```
class Rectangle { //private parts omitted
public:
  Rectangle() {/*...*/}
  Rectangle(double a_in, double b_in) {/*...*/}
  void set(double a_in, double b_in) {/*...*/}
  double area() {/*...*/}
  void print() {/*...*/}
};

int main() {
  Rectangle r(2,4);
  r.print();
  cout << "area = " << r.area() << endl;
}
```

```
$ ./a.out
a=2 b=4
area = 8
```