Slides by Andrew DeOrio

EECS 280

Programming and Introductory Data Structures

Subtypes and Subclasses

Triangle.h

Review: Triangle ADT

```
class Triangle {
  //...
 public:
  Triangle();
  Triangle(double a_in, double b_in, double c_in);
  double area() const;
  void print() const;
 private:
   //edges are non-negative and form a triangle
   double a, b, c;
};
```

- Member functions and member variables

Review: get and set functions

- A `get` function is a `public` function that returns a `private` member variable

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

Review: get and set functions

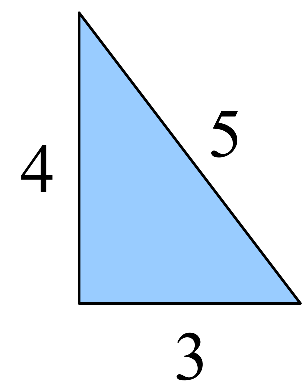
- A `set` function is a `public` function that modifies a `private` member variable

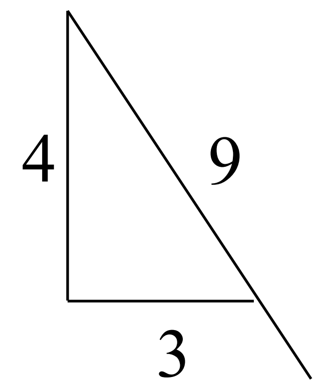
```
class Triangle {
  //...
 public:
  //REQUIRES: a,b,c are non-negative and form a
  //          triangle
  //MODIFIES: a, b, c
  //EFFECTS: sets length of edge a, b, c
  void set_a(double a_in);
  void set_b(double b_in);
  void set_c(double c_in);
};
```

get and set functions

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

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





Review: Triangle ADT

- In computer graphics, 3D surfaces can be modeled using connected triangles, called a triangle mesh

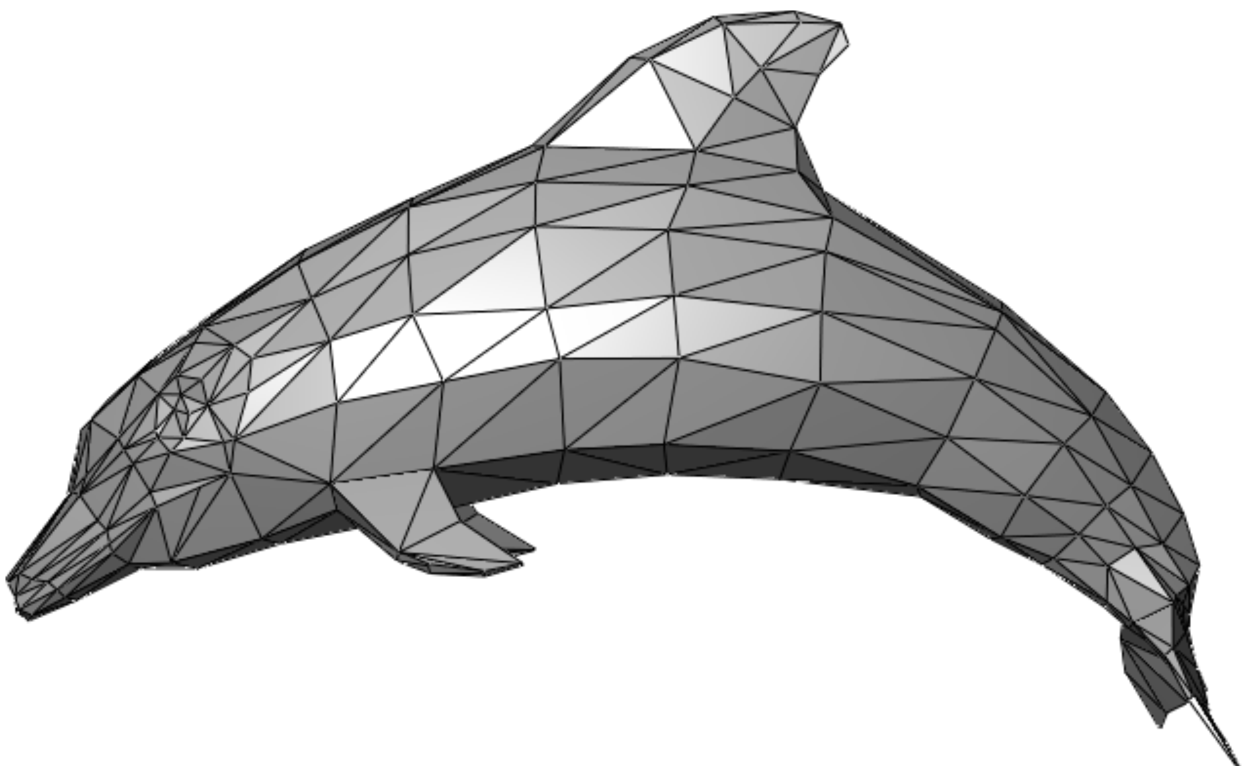


Image: wikipedia.org

Graphics.cpp

Review: Triangle ADT

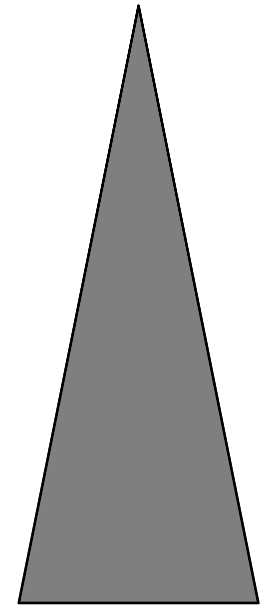
```
#include "Triangle.h"
int main() {
  const int SIZE = 3;
  Triangle mesh[SIZE];
  // fill with triangles ...

  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
```

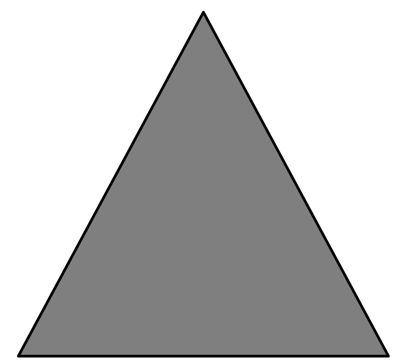
Different Kinds of Triangles

- So far, we have represented general triangles
- Let's add some more types of triangles, like Isosceles and Equilateral
- We can represent this kind of relationship between two C++ classes with a *derived class*
- Other terms for a derived class (type) are *inherited class* or *subclass*

- Let's create derived classes for an isosceles triangle and an equilateral triangle



Isosceles



Equilateral

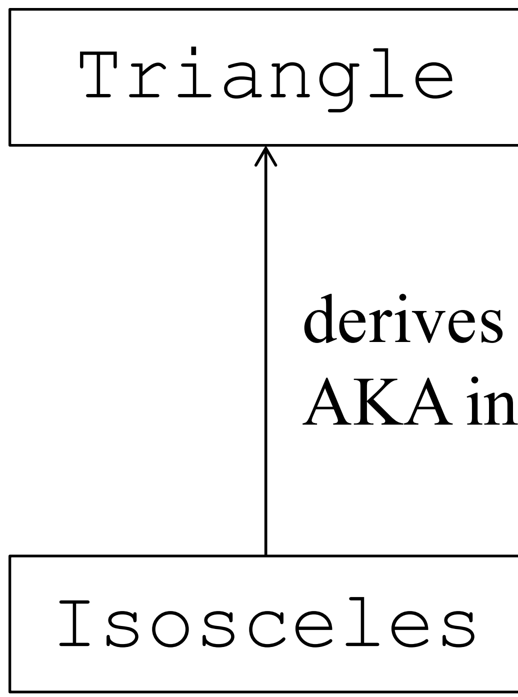
Derived classes

```
class Isosceles : public Triangle {
  //OVERVIEW: a geometric representation of an
  //isosceles triangle with edge a representing
  //the base, and b=c the legs
  //...
};
```

- This creates a new type called `Isosceles` that contains all of the `Triangle` member functions and member data
- Think of this as an "*is a*" relationship
  an `Isosceles` *is a* `Triangle`

Class hierarchy

- Derivation is often represented by a graph, where each vertex is a class, and each edge shows derivation



base class
AKA parent class
AKA superclass

derives from
AKA inherits from

derived class
AKA child class
AKA inherited class
AKA subclass

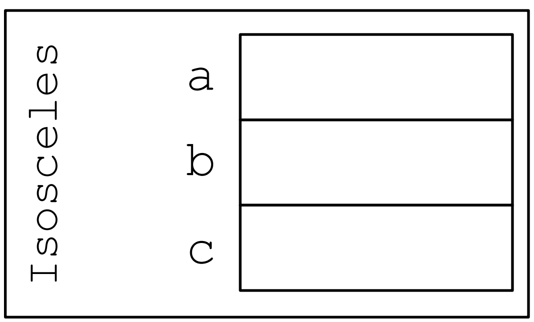
Using derived classes

```
class Triangle {
 private:
  double a,b,c;
  //...
```

- Now we can define an `Isosceles` object

```
int main() {
  Isosceles i;
}
```

- Because member variables are inherited, the compiler allocates memory for each one



Using derived classes

- We call member functions just like we did with the base class
- Because member functions are inherited, we do not need to rewrite them (copy paste avoided!)

```
int main() {
  Isosceles i;
  i.set_a(1);
  i.set_b(11);
  i.set_c(11);
  i.print();
  cout << "area=" << i.area() << endl;
}
```

```
$ ./a.out
a=1 b=11 c=11
area=5.49432
```

Adding member variables

- In addition to the inherited member variables, we can add extra member variables
- Let's add extra member variables to `Isosceles` to store the `base` and `leg` edge lengths

```
class Isosceles : public Triangle {
 private:
  double base, leg; //new member variables
};
```

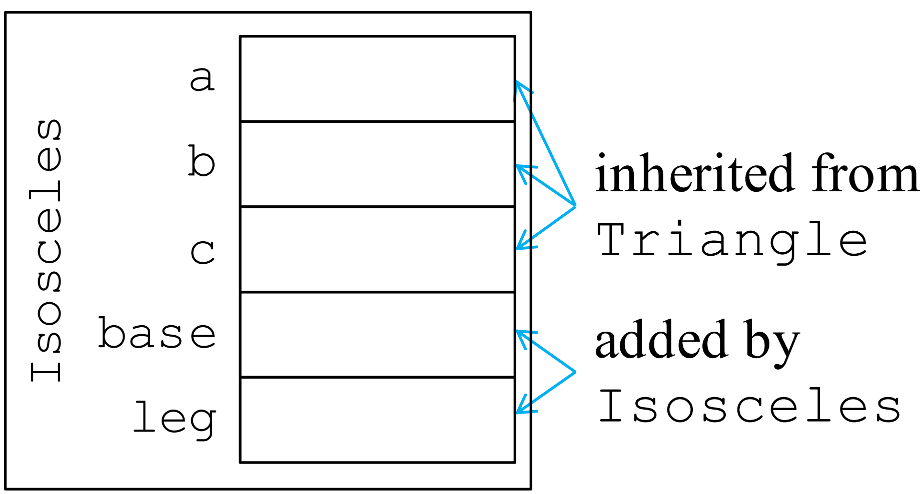
Adding member varia

```
class Triangle {
 private:
  double a, b, c;
};
```

- Now, we get memory for the member variables inherited from `Triangle`, plus the two added member variables

```
class Isosceles : public
Triangle {
 private:
  double base, leg;
};
```

```
int main() {
  Isosceles i;
}
```



Adding member functions

- In our example, this seems wasteful, since we already have `a`, `b` and `c` to store the edge lengths
- Instead, let's add member functions to change the base and legs

```
class Isosceles : public Triangle {
 private:
  double base, leg;

 public:
  //EFFECTS: sets base (edge a)
  void set_base(double base);

  //EFFECTS: sets legs (edges b and c)
  void set_leg(double leg);
};
```

Exercise: implement these two functions

Adding member functions

- Solution 1: set the member variables directly

```
class Isosceles : public Triangle {
 public:
  //EFFECTS: sets base (edge a)
  void set_base(double base) {
    a = base;
  }

  //EFFECTS: sets legs (edges b and c)
  void set_leg(double leg) {
    b = c = leg;
  }
};
```

Adding member functions

```
class Triangle {
  //...
 private:
  double a, b, c;
};
```

- Problem: **a**, **b** and **c** are private members of `Triangle`, and derived classes cannot access `private` member variables of a base class

```
class Isosceles : public Triangle {
 public:
  //EFFECTS: sets base (edge a)
  void set_base(double base) {
    a = base; //compile error
  }

  //EFFECTS: sets legs (edges b and c)
  void set_leg(double leg) {
    b = c = leg; //compile error
  }
};
```

Adding member functions

- Solution 2: use `set_*()` functions inherited from `Triangle`

```
class Isosceles : public Triangle {
 public:
  //EFFECTS: sets base (edge a)
  void set_base(double base) {
    set_a(base);
  }

  //EFFECTS: sets legs (edges b and c)
  void set_leg(double leg) {
    set_b(leg);
    set_c(leg);
  }
};
```

Adding member functions

- Now, we can call our new `Isosceles` member functions, in addition to the inherited member functions

```
int main() {
  Isosceles i;
  i.set_base(1); //additional member function
  i.set_leg(11); //additional member function
  i.print();     //inherited member function
}
```

```
$ ./a.out
a=1 b=11 c=11
```

Derived class constructors

- Constructors are *not* inherited, so let's add one

```
class Isosceles : public Triangle {
  //...
  //EFFECTS: creates a zero size Isosceles triangle
  Isosceles();
};
```

Exercise: derived class ctors

- What is wrong with these implementations?
- Hint: think like a compiler, think about efficiency

```
Isosceles() {
  a = b = c = 0;
}
```

```
Isosceles() {
  set_a(0);
  set_b(0);
  set_c(0);
}
```

```
Isosceles() {
  Triangle();
}
```

Derived class constructors

```
class Isosceles : public Triangle {
  //...
  //EFFECTS: creates a zero size Isosceles triangle
  Isosceles() {}
};
```

- Solution: do nothing!
- Constructors run automatically, starting with the base class

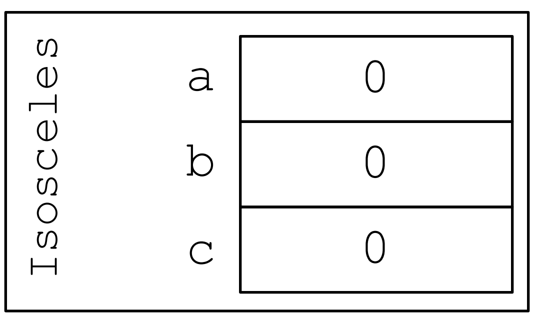
Derived class constructors

```
int main() {
  Isosceles i;
}
```

```
Triangle()
  : a(0), b(0), c(0) {}
```

```
Isosceles() {}
```

- First, `Triangle` constructor runs
- Second, `Isosceles` constructor runs
- In the end, we get an initialized chunk of memory like this:



Derived class constructors

- Next, let's add a custom constructor to set the base and legs

```
class Isosceles : public Triangle {
  //...
  //REQUIRES: base and leg are non-negative and
  //          form an isosceles triangle
  //EFFECTS: creates an Isosceles triangle with
  //         given edge lengths
  Isosceles(double base, double leg);
};
```

Derived class constructors

- Next, let's add a custom constructor to set the base and legs

```
class Isosceles : public Triangle {
  //...
  //REQUIRES: base and leg are non-negative and
  //           form an isosceles triangle
  //EFFECTS: creates an Isosceles triangle with
  //          given edge lengths
  Isosceles(double base, double leg)
   : Triangle(base, leg, leg) {}
};
```

- Solution: reuse constructor from base class
- Initializer lists are the *only* way to call a base class constructor from a derived class constructor

Constructors: common pitfall


```
class Isosceles : public Triangle {
  //...
  Isosceles(double base, double leg) {
    Triangle(base, leg, leg);//bad
  }
};
```

- Pitfall: calling the base class constructor *inside* the derived class constructor, but *outside* the initializer list
- This creates a new, anonymous `Triangle` object, which is a local variable without a name inside the `Isosceles` constructor
- Usually not what you intended!

Exercise: constructors

```
int main {
  Isosceles i(0.9, 8);
}
```

- Which constructors run?
- Specify the exact constructors, arguments, and order

Member functions from base class

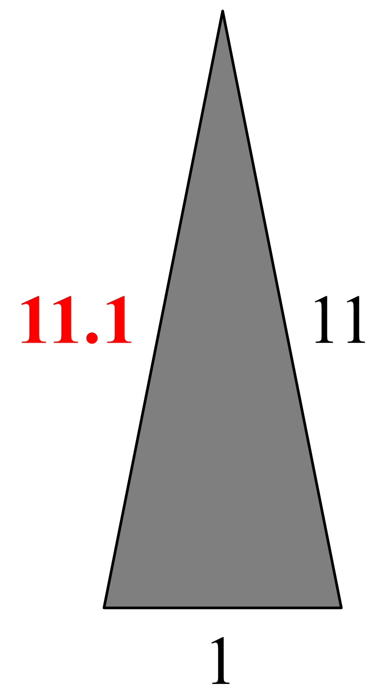
- What is wrong with this code?

```
Isosceles i(1,11);
i.print();
i.set_b(11.1);
i.print();
```

Member functions from base class

- What is wrong with this code?

```
Isosceles i(1,11);
i.print();
i.set_b(11.1);
i.print();
```



- Problem:
- `i` is no longer an isosceles triangle!

Representation invariant

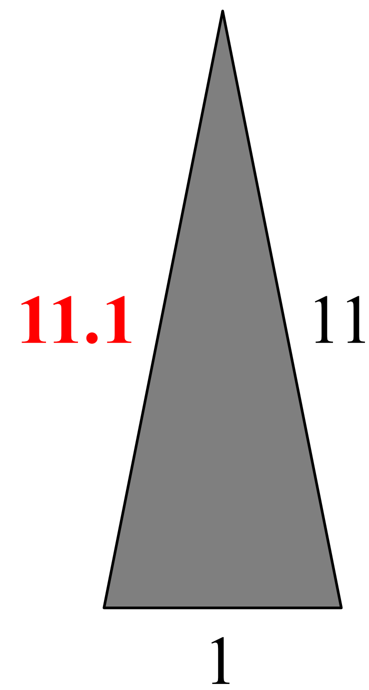
- The `Isosceles` *representation invariant* has been broken
- The *representation invariant* constrains the member variables
- You can think of the representation invariant as a sanity-check for the class

- `Triangle` invariant: `a`, `b`, and `c` form a triangle
  - Long edge is less than the sum of both short edges
- `Isosceles` invariant: `base` and 2 `leg` edges form an isosceles triangle
  - Base is less than sum of two legs
  - Legs are equal

Member functions from base class

- What is wrong with this code?

```
Isosceles i(1,11);
i.print();
i.set_b(11.1);
i.print();
```



- Solution: change `set_b()` implementation only in `Isosceles` derived class
- This is called a function *override*

Override vs. Overload

- A function *override* is where a derived class has a function with the same name and prototype as the parent

```
Triangle::set_b(double b_in);
Isosceles::set_b(double b_in);
```

- A function *overload* is where a single class has two different functions with the same name, but different prototypes

```
Triangle::Triangle();
Triangle::Triangle(double a_in,
                   double b_in,
                   double c_in);
```

Overriding member functions

- Override `set_b()` and `set_c()` to set both legs of the triangle, maintaining the `Isosceles` representation invariant

```
class Isosceles : public Triangle {
  //...
  //REQUIRES: a, b, c, are non-negative and form
  // an isosceles triangle
  //MODIFIES: b, c
  //EFFECTS: set edge lengths
  void set_b(double b_in) { b = c = b_in; }
  void set_c(double c_in) { b = c = c_in; }
};
```

Overriding member funct

```
class Triangle {
  //...
 private:
  double a, b, c;
};
```

- Problem: `Isosceles` can't modify **`a`**, **`b`** or **`c`**, because they are `private` members of `Triangle`
- We have seen this before
- Bad solution: `protected` members

```
class Isosceles : public Triangle {
  //...
  void set_b(double b_in) { b = c = b_in; }
  void set_c(double c_in) { b = c = c_in; }
};
```

protected members

```
class Triangle {
 public:
  //member functions ...

 protected:
  //edge lengths represent a triangle
  double a, b, c;
};
```

- `protected` members can be seen by all members of this class and any derived classes

public vs. private vs. protected

- `public`
  - Any code inside the class (member functions) or outside the class can access `public` members
- `private`
  - Only code inside the class (member functions) can access `private` members
- `protected`
  - Code inside the class (member functions) as well as derived classes (member functions of inherited classes) can access `protected` members

protected members

```
class Triangle {
  //...
 protected:
  double a, b, c;
};
```

```
class Isosceles : public Triangle {
  //...
  void set_b(double b_in) { b = c = b_in; }
  void set_c(double c_in) { b = c = c_in; }
};
```

- Now, `Isosceles` member functions can modify `a`, `b` and `c` because they are `protected` member variables of `Triangle`, and `Isosceles` is derived from `Triangle`

Overriding member functions

- When a class overrides a function, the function in the derived class is called, not the base class

```
Isosceles i(1,11);
i.print();
i.set_b(11.1);
i.print();
```

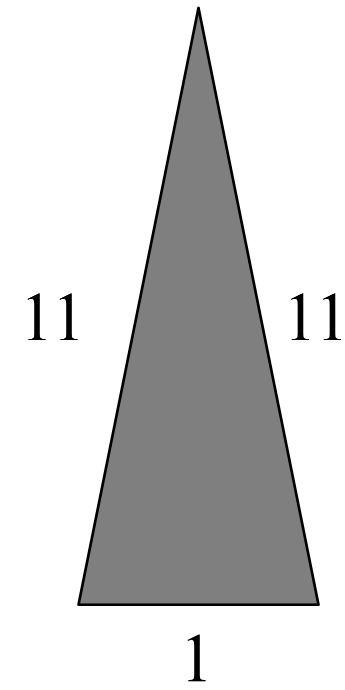
```
$ ./a.out
a=1 b=11 c=11
a=1 b=11.1 c=11.1
```

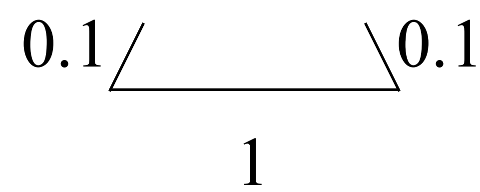
`Isosceles::set_b()` runs, not `Triangle::set_b()`
This is what we want ☺

Problems with protected

```
Isosceles i(1,11);
i.set_b(0.1);
```

Problem: we no longer have a triangle!

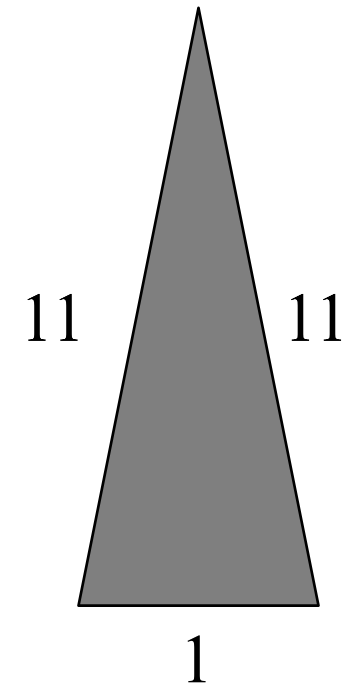




Problems with protected

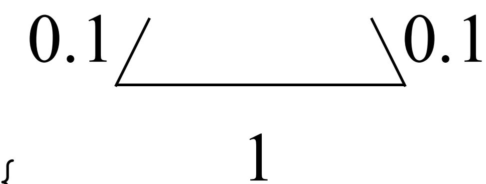
```
Isosceles i(1,11);
i.set_b(0.1);
```

Problem: we no longer have a triangle!



- `Triangle::set_b()` checks the new dimensions

```
void Triangle::set_b(double b_in) {
  b = b_in;
  //check if new dimensions form a triangle
}
```



- `Isosceles::set_b()` does not

```
void Isosceles::set_b(double b_in) {
  b = c = b_in;
}
```

Problems with protected

- Solution: forget protected member variables

```
class Triangle {
 private: ~~protected:~~
  double a, b, c;
};
```

- Let's just reuse `Triangle::set_b()`, which already checks if the new dimensions form a triangle

```
class Isosceles : public Triangle {
  //...
  void set_b(double b_in) {
    Triangle::set_b(b_in);
    Triangle::set_c(b_in);
  }
};
```

Scope resolution operator (`::`)

- Use the scope resolution operator (`::`) to call the `set_b()` and `set_c()` functions inherited from `Triangle` instead of `Isosceles`

```
class Isosceles : public Triangle {
  //...
  void set_b(double b_in) {
    Triangle::set_b(b_in);
    Triangle::set_c(b_in);
  }
};
```

Digression (for correctness)

- This code could break if we did this:

```
int main() {
  Isosceles i(1,11);
  i.set_b(1000); //set_b() check fails, before set_c() can run
}
```

- Solution: add another set function and use it:

```
class Triangle {
  //...
  void set(double a_in, double b_in, double c_in) {
    a = a_in; b = b_in; c = c_in;
    // check that edges make a proper triangle
  }
};
class Isosceles : public Triangle {
  //...
  void set_b(double b_in) {
    Triangle::set(get_a(), b_in, b_in);
  }
};
```

Subtypes: Introduction

- `Isosceles` has a special property: any code that expects a `Triangle` will work correctly with an `Isosceles` object

- Put another way: we can replace any `Triangle` object in a program with an `Isosceles` object and the program will still work

Liskov Substitution Principle

- For a derived type to also be a subtype, code written to correctly use the supertype must still be correct if it uses the subtype

Graphics.cpp

```
Isosceles 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";
```

Liskov Substitution Principle

Barbara Liskov, MIT

- If S is a *subtype* of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program (correctness)

  In other words:

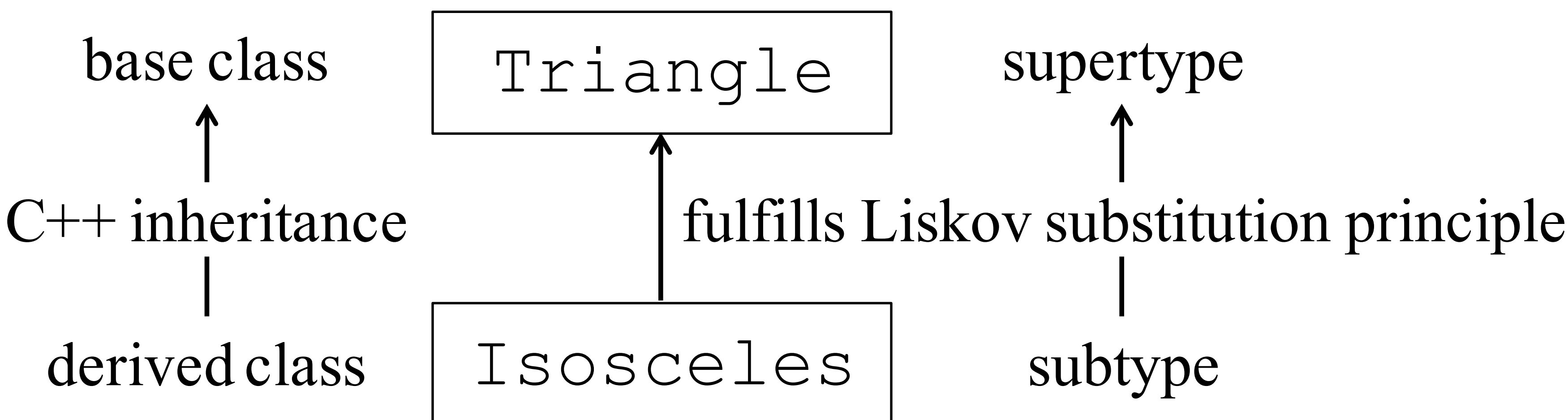
  For any instance where an object of type T is expected, an object of type S can be supplied without changing the correctness of the original computation

- This is called the *Liskov substitution principle*

Liskov Substitution Principle

- In C++, subtypes can be created with derived classes
- However, not all derived types (classes) are subtypes!
- In our Graphics example, `Isosceles` is a *derived type* (class) because it inherits from `Triangle`

  ```
  class Isosceles : public Triangle { //...
  ```
- `Isosceles` is also a *subtype* because it fulfills the Liskov Substitution Principle



Liskov Substitution Principle

- For a derived type to also be a subtype, code written to correctly use the supertype must still be correct if it uses the subtype
- This is true of our Graphics example
- It is helpful to remember that `Isosceles` *is a* `Triangle`, so we can use an `Isosceles` in place of a `Triangle`

Liskov Substitution Principle

- For a derived type to also be a subtype, code written to correctly use the supertype must still be correct if it uses the subtype

<u>Graphics.cpp</u>

```
Isosceles 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";
```

Liskov Substitution Principle

- Will this really get the correct answer if we use isosceles triangles instead of general triangles?
- Yes. In this example, we define “desirable properties of the program” (correctness) as “computes the area”.

Graphics.cpp

```
Isosceles ~~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";
```

How to create a subtype

- With Abstract Data Types, there are three ways to create a subtype from a derived type

1. Weaken the precondition of one or more operations
2. Strengthen the postcondition of one or more operations
3. Add one or more operations

How to create a subtype

- #1 and #2 apply to overridden functions

1. Weaken the precondition of one or more operations
    - The overridden member function must require no more of the caller than the old method did, but it can require less
2. Strengthen the postcondition of one or more operations
    - The overridden member function must do everything the old function did, but it is allowed to do more as well

- Think of this as doing *more with less*

Weaken precondition

1. Weaken the precondition of one or more operations.
   - The overridden member function must require no more of the caller than the old method did, but it can require less
- The preconditions of a method are formed by two things:
  - Its argument type signature
  - The REQUIRES clause

```
//REQUIRES: b_in is non-negative and forms a
//  triangle with existing edges
//MODIFIES: b, c
//EFFECTS: sets edges b and c
void Isosceles::set_b(double b_in) {
   Triangle::set(get_a(), b_in, b_in);
}
```

Weaken precondition

- We can weaken the preconditions by requiring less
- For example, allowing negative inputs
  - Take absolute value of any negative input

```
//REQUIRES: b_in ~~is non-negative and~~ forms a
//  triangle with existing edges
//MODIFIES: b, c
//EFFECTS: sets edges b and c
void Isosceles::set_b(double b_in) {
  b_in = abs(b_in);
  Triangle::set(get_a(), b_in, b_in);
}
```

Strengthen postcondition

2. Strengthen the postcondition of one or more operations
   - The overridden member function must do everything the old function did, but it is allowed to do more as well
- The postconditions of a method are formed by two things:
  - Its return type signature
  - The EFFECTS clause

```
//REQUIRES: b_in is non-negative and forms a
//  triangle with existing edges
//MODIFIES: this
//EFFECTS: sets edges b and c
void Isosceles::set_b(double b_in) {
  Triangle::set(get_a(), b_in, b_in);
}
```

Strengthen postcondition

- We can strengthen the EFFECTS clause by promising everything we used to, plus extra
- For example, `Isosceles` overrode the `Triangle::set_b()` function, to not only set `b`, but also `c`

```
void Triangle::set_b(double b_in) {
  b = b_in;
}

void Isosceles::set_b(double b_in) {
  // will do everything Triangle did (i.e. set b),
  // plus more (i.e. set c)
  Triangle::set(get_a(), b_in, b_in);
}
```

Add an operation

- The final way of creating a subtype is to add a member function
- Any code expecting only the old function will still see all of them, so the new function won't break old code
- For example:

```
class Isosceles : public Triangle {
public:
  //...
  void set_base(double base) {/*...*/}
  void set_leg(double leg)   {/*...*/}
};
```

Exercise: equilateral triangle

- Code an `Equilateral` class representing an equilateral triangle
- Declare a derived class
- Draw the new class hierarchy
- Write two constructors: one default, one with input
- Override any necessary member functions
- Make sure your new type fulfills the Liskov Substitution Principle