CS106L Lecture 7:

Classes 🕋

Fall 2023

Fabio Ibanez, Haven Whitney

Attendance

bit.ly/classesF23

Announcement!

- [Apply to section lead](#)

- Section leading is one of the most rewarding things we've done at Stanford – it's how we're here!

- PLEASE, ask us questions about it :)

- App is due Oct. 19th (Thursday), if you're in CS106B the deadline is Nov. 4th (Saturday).

Plan

1. Introduction to classes

2. Container adapters

3. Inheritance

Why classes?

- One of the premises of the entire C++ language was the lack of object-oriented-programming (OOP) in C.

Why classes?

- One of the premises of the entire C++ language was the lack of object-oriented-programming (OOP) in C.

- Classes are user-defined types that allow a user to **_encapsulate_** data and functionality using member variables and member functions

You've got no class

What is object-oriented-programming?

- Object-oriented-programming is centered around **objects**

What is object-oriented-programming?

- Object-oriented-programming is centered around **objects**
- Focuses on design and implementation of classes!
- Classes are the **user-defined types** that can be declared as an object!

Containers are classes defined in the STL! 🥳

Comparing 'struct' and 'class'

classes *containing a sequence of objects of various types, a set of functions for manipulating these objects, and a set of restrictions on the access of these objects and function;*

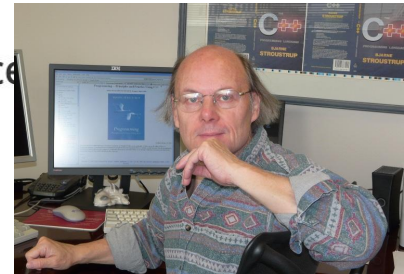structures *which are classes without access restrictions;*

Bjarne Stroustrup, The C++ Programming Language – Reference Manual, §4.4 Derived types

Comparing 'struct' and 'class'

> classes *containing a sequence of objects of various types, a set of functions for manipulating these objects, and a set of restrictions on the access of these objects and function;*
>
> structures *which are classes without access restrictions;*
>
> Bjarne Stroustrup, The C++ Programming Language – Reference Manual, §4.4 Derived types

Recall the 'struct'

```cpp
struct Student {
    std::string name; /// these are fields!
    std::string state;
    int age;
};

Student s;
s.name = "Fabio";
s.state = "CA";
s.age = 20;
```

Recall the `struct`

```cpp
struct Student {
    std::string name; /// these are fields!
    std::string state;
    int age;
};

Student s;
s.name = "Fabio";
s.state = "CA";
s.age = 20;
```

All these fields are public, i.e. can be changed by the user

Recall the `struct`

```
struct Student {
    std::string name; /// these are fields!
    std::string state;
    int age;
};

Student s;
s.name = "Fabio";
s.state = "CA";
s.age = 20;
s.age = -2345; /// 💀?
```

All these fields are public, i.e. can be changed by the user

Recall the `struct`
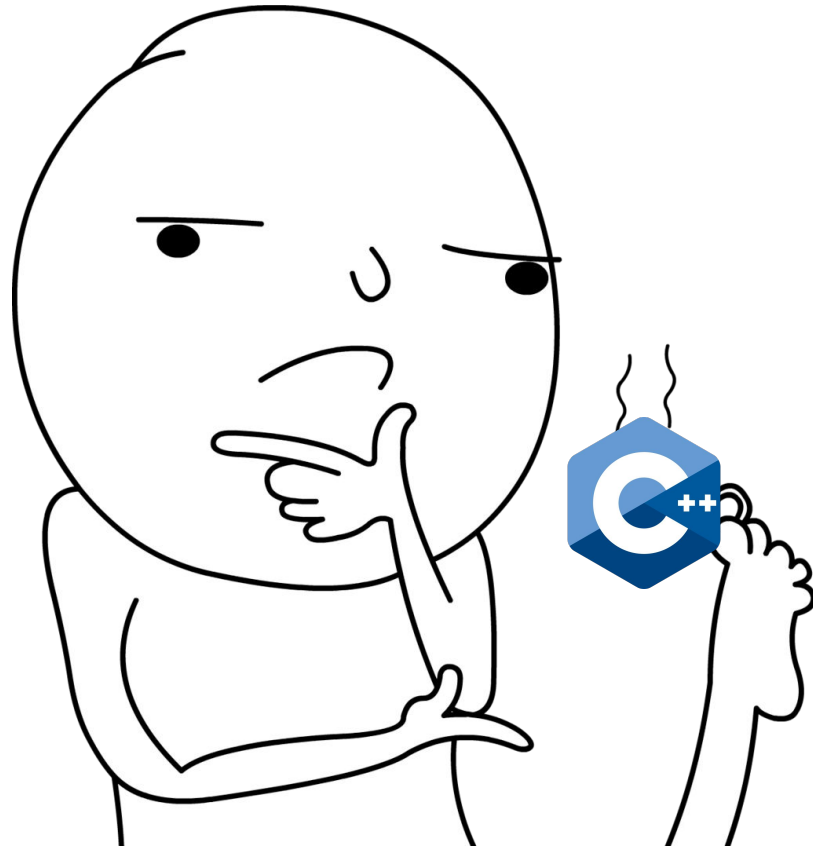
```cpp
struct Student {
    std::string name; /// these are fields!
    std::string state;
    int age;
};


Student s;
s.name = "Fabio";
s.state = "CA";
s.age = 20;
s.age = -2345; /// 💀?
```

All these fields are public, i.e. can be changed by the user

Because of this, we can't enforce certain behaviors in structs, like avoiding a negative age.

What questions do we have?

As you might have guessed

```
class className {
private:
```



```
public:
```



```
}
```

Classes have `public` and `private` sections!

User can access the **`public`**

User is restricted from `private`

A backpack

A backpack

Struct

Class

Enjoy

Let's make a `Student` class based on our struct!

Header File (**.h**) vs Source Files (**.cpp**)

Header File

- Are used to define the interface of a class
- Typically contain:
  - Function prototypes
  - Variable declarations
  - Class definitions
  - Type definitions
  - Macros and constants
  - Template definitions

Header File (**.h**) vs Source Files (**.cpp**)

Header File

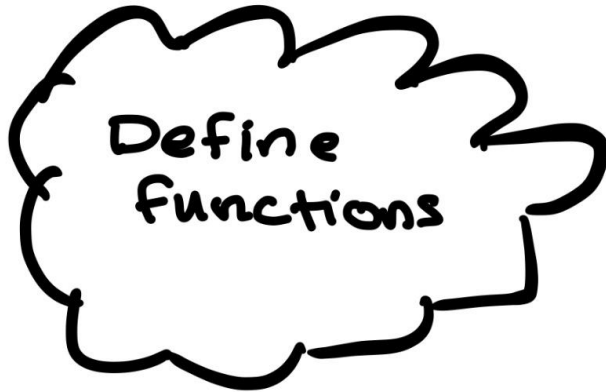- Are used to define the interface of a class
- Typically contain:
    - Function prototypes
    - Variable declarations
    - Class definitions
    - Type definitions
    - Macros and constants
    - Template definitions

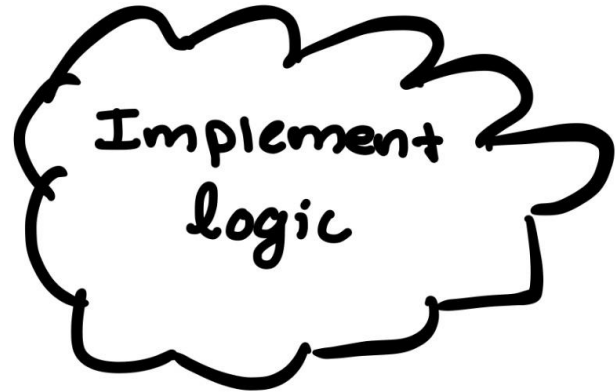## Source File

- Are used to define the implementations of the functions and classes declared in the header file
- Typically contain:
    - Function implementations
    - Executable code
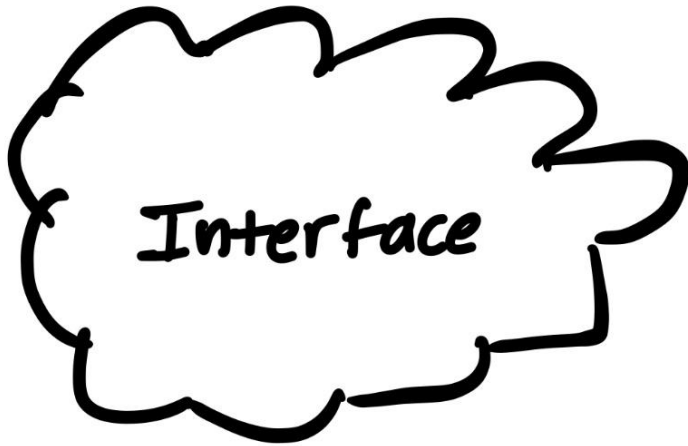
Header File (**.h**) vs Source Files (**.cpp**)
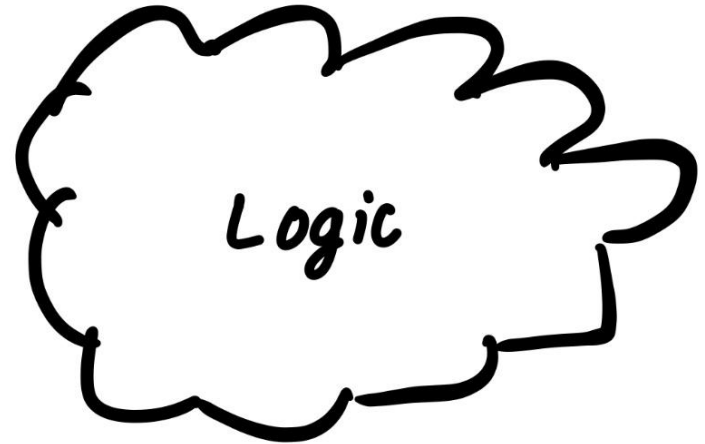

Header Files — Define functions

Source Files — Implement logic

Header File (**.h**) vs Source Files (**.cpp**)

Header Files

Interface

Source Files

Logic

```c
        // Declare an integer vector register with __cs149_vec_int
   28
   29   #define __cs149_vec_int    __cs149_vec<int>
   30
   31   //***********************
   32   //* Function Definition *
   33   //***********************
   34
   35   // Return a mask initialized to 1 in the first N lanes and 0 in the others
   36   __cs149_mask _cs149_init_ones(int first = VECTOR_WIDTH);
   37
   38   // Return the inverse of maska
   39   __cs149_mask _cs149_mask_not(__cs149_mask &maska);
   40
   41   // Return (maska | maskb)
   42   __cs149_mask _cs149_mask_or(__cs149_mask &maska, __cs149_mask &maskb);
   43
   44   // Return (maska & maskb)
   45   __cs149_mask _cs149_mask_and(__cs149_mask &maska, __cs149_mask &maskb);
   46
   47   // Count the number of 1s in maska
   48   int _cs149_cntbits(__cs149_mask &maska);
   49
   50   // Set register to value if vector lane is active
   51   //  otherwise keep the old value
   52   void _cs149_vset_float(__cs149_vec_float &vecResult, float value, __cs149_mask &mask);
   53   void _cs149_vset_int(__cs149_vec_int &vecResult, int value, __cs149_mask &mask);
   54   // For user's convenience, returns a vector register with all lanes initialized to value
   55   __cs149_vec_float _cs149_vset_float(float value);
   56   __cs149_vec_int _cs149_vset_int(int value);
   57
```

Class design

1. A constructor
2. Private member functions/variables
3. Public member functions (interface for a user)
4. Destructor

Constructor

- The constructor initializes the state of newly created objects

Constructor

- The constructor initializes the state of newly created objects
- For our `Student` class what do our objects need?

Constructor

- The constructor initializes the state of newly created objects
- For our `Student` class what do our objects need?

```
s.name = "Fabio";

s.state = "CA";

s.age = 20;
```

Constructor

.h file

```
class Student {
private:

    ?


public:

    ?


}
```

Constructor

.h file

```cpp
class Student {
private:
    std::string name;
    std::string state;
    int age;

public:
    /// constructor for our student
    Student(std::string name, std::string state, int age);
}
```

Constructor

.h file

```cpp
class Student {
private:
    std::string name;
    std::string state;
    int age;

public:
    /// constructor for our student
    Student(std::string name, std::string state, int age);
    /// method to get name, state, and age, respectively
    std::string getName();
    Std::string getState();
    int getAge();
}
```

Parameterized Constructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    name = name;
    state = state;
    age = age;
}
```

Parameterized Constructor

.cpp file (implementation)

```cpp
#include "Student.h"

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    name = name;
    state = state;
    age = age;
}
```

Remember namespaces, like `std::`

Parameterized Constructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    name = name;
    state = state;
    age = age;
}
```

Remember namespaces, like `std::`

In our `.cpp` file we need to use our class as our namespace when defining our member functions

What questions do we have?

Parameterized Constructor
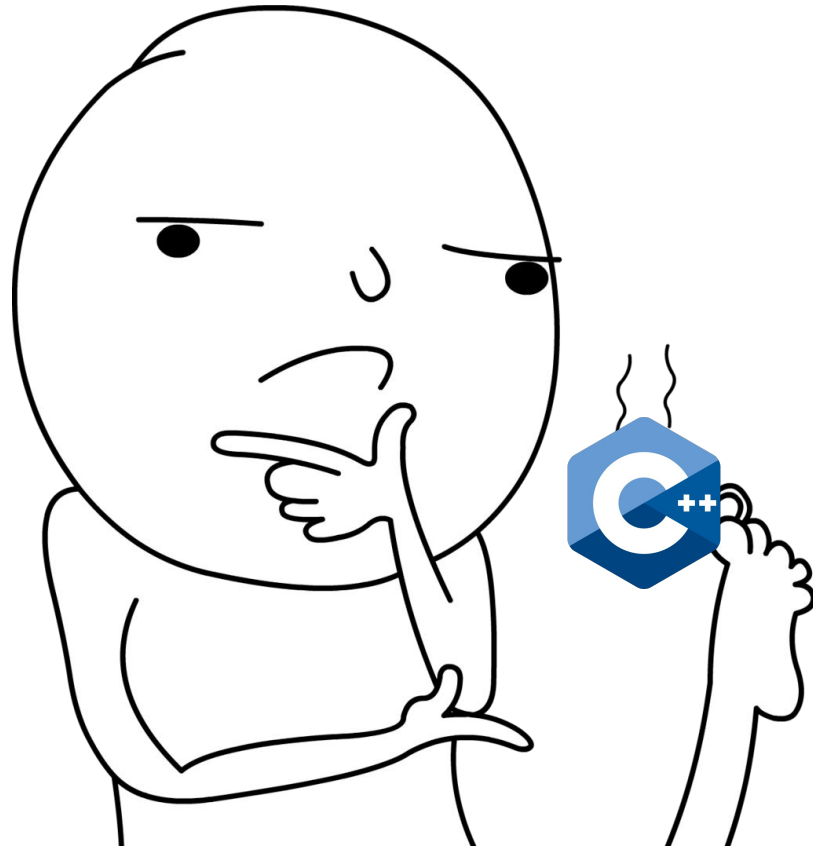
.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    name = name;
    state = state;
    age = age;
}
```

Does anyone see a problem here?

Parameterized Constructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    name = name;
    state = state;
    age = age;
}
```

Does anyone see a problem here?

Our .h definition

.h file

```cpp
#include <string>
class Student {
private:
    std::string name;
    std::string state;
    int age;

public:
    /// constructor for our student
    Student(std::string name, std::string state, int age);
    /// method to get name, state, and age, respectively
    std::string getName();
    Std::string getState();
    int getAge();
}
```

Use the **`this`** keyword

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) {
    this->name = name;
    this->state = state;
    this->age = age;
}
```

Use this **`this`** keyword to disambiguate which 'name' you're referring to.

List initialization constructor (C++11)

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student(std::string name, std::string state, int age) name{name}, state{state},
age{age} {}
```

> Recall, uniform initialization,
> this is similar but not quite!

Default constructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// implement constructor
Student::Student() {
    name = "John";
    state = "Appleseed";
    age = 18;
}
```

If we call our constructor without parameters we can set default ones!

Constructor Overload

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

/// default constructor
Student::Student() {
    name = "John Appleseed";
    state = "CA";
    age = 18;
}
/// parameterized constructor
Student::Student(std::string name, std::string state, int age) {
    this->name = name;
    this->state = state;
    this->age = age;
}
```

Our compilers will know which one we want to use based on the inputs!

Back to our class definition

.h file

```cpp
class Student {
private:
    std::string name;
    std::string state;
    int age;

public:
    /// constructor for our student
    Student(std::string name, std::string state, int age);
    /// method to get name, state, and age, respectively
    std::string getName();
    std::string getState();
    int getAge();
}
```

Let's implement them

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

std::string Student::getName() {

}


std::string Student::getState() {

}


int Student::getAge() {

}
```

Implemented members

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

std::string Student::getName() {
    return this->name;
}


std::string Student::getState() {
    return this->state;
}


int Student::getAge() {
    return this->age;
}
```

The destructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

Student::~Student() {
    /// free/deallocate any data here
}
```

The destructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

Student::~Student() {
    /// free/deallocate any data here
}
```

In our student class we are not dynamically allocating any data by using the **new** keyword

The destructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

Student::~Student() {
    /// free/deallocate any data here
}
```

Nonetheless destructors are an important part of an object's lifecycle.

The destructor

.cpp file (implementation)

```cpp
#include "Student.h"
#include <string>

Student::~Student() {
    /// free/deallocate any data here

    delete [] my_array; /// for illustration
}
```

The destructor is not explicitly called, it is automatically called when an object goes out of scope

Some other cool class stuff

Type aliasing - allows you to create synonymous identifiers for types

Some other cool class stuff

Type aliasing - allows you to create synonymous identifiers for types
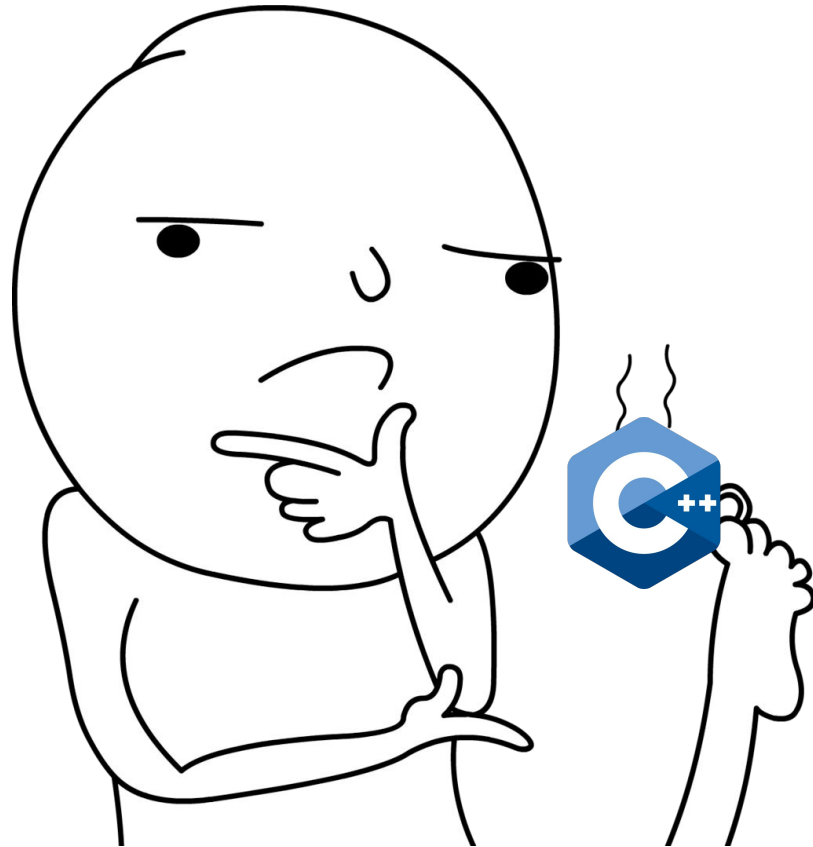
Wut? 😪

Back to our class definition

.h file

```cpp
class Student {
Private:
    /// An example of type aliasing
    using String = std::string;
    String name;
    String state;
    int age;

public:
    /// constructor for our student
    Student(String name, String state, int age);
    /// method to get name, state, and age, respectively
    String getName();
    String getState();
    int getAge();
}
```

What questions do we have?

Taking a look at the student class

Replit Link

Plan

1. ~~Introduction to classes~~

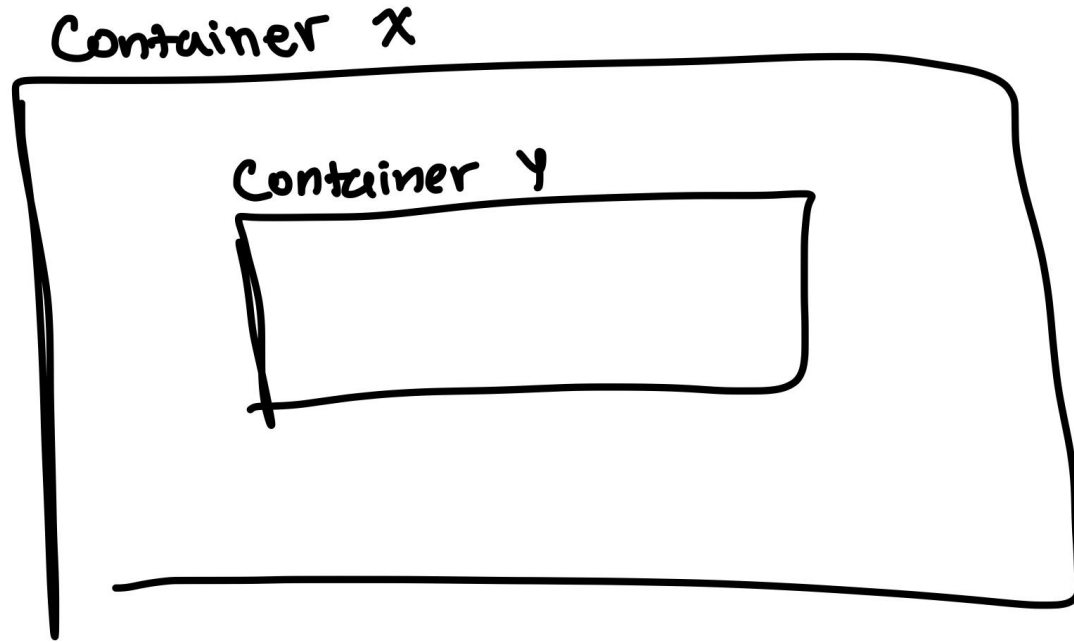2. Container adapters

3. Inheritance

Surprise!

All containers in the STL are ⭐***classes***⭐

Surprise (AGAIN)!

All containers in the STL are ⭐ ***classes*** ⭐
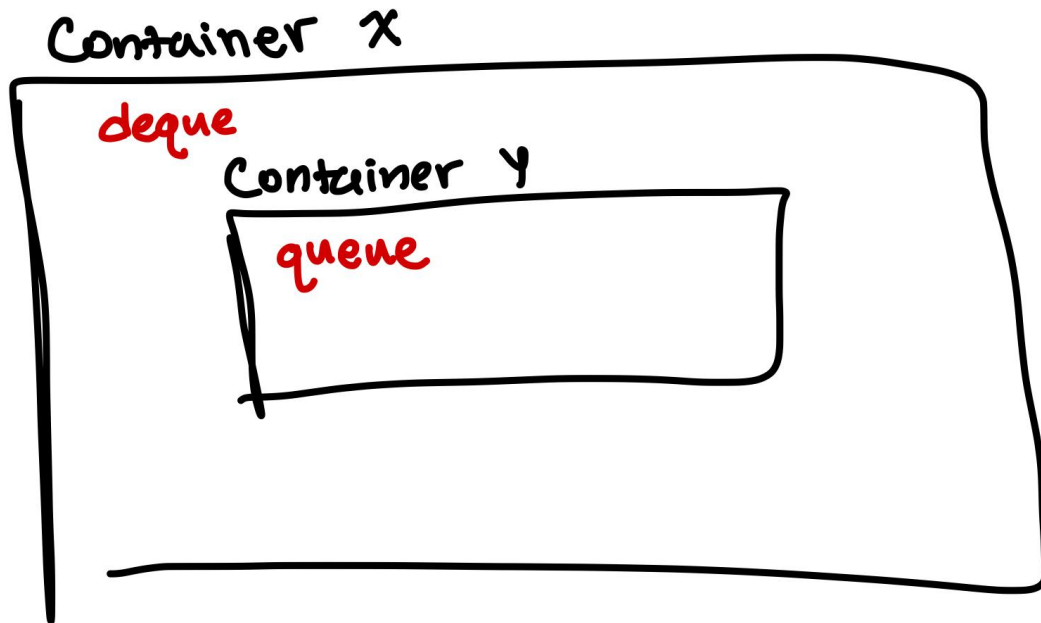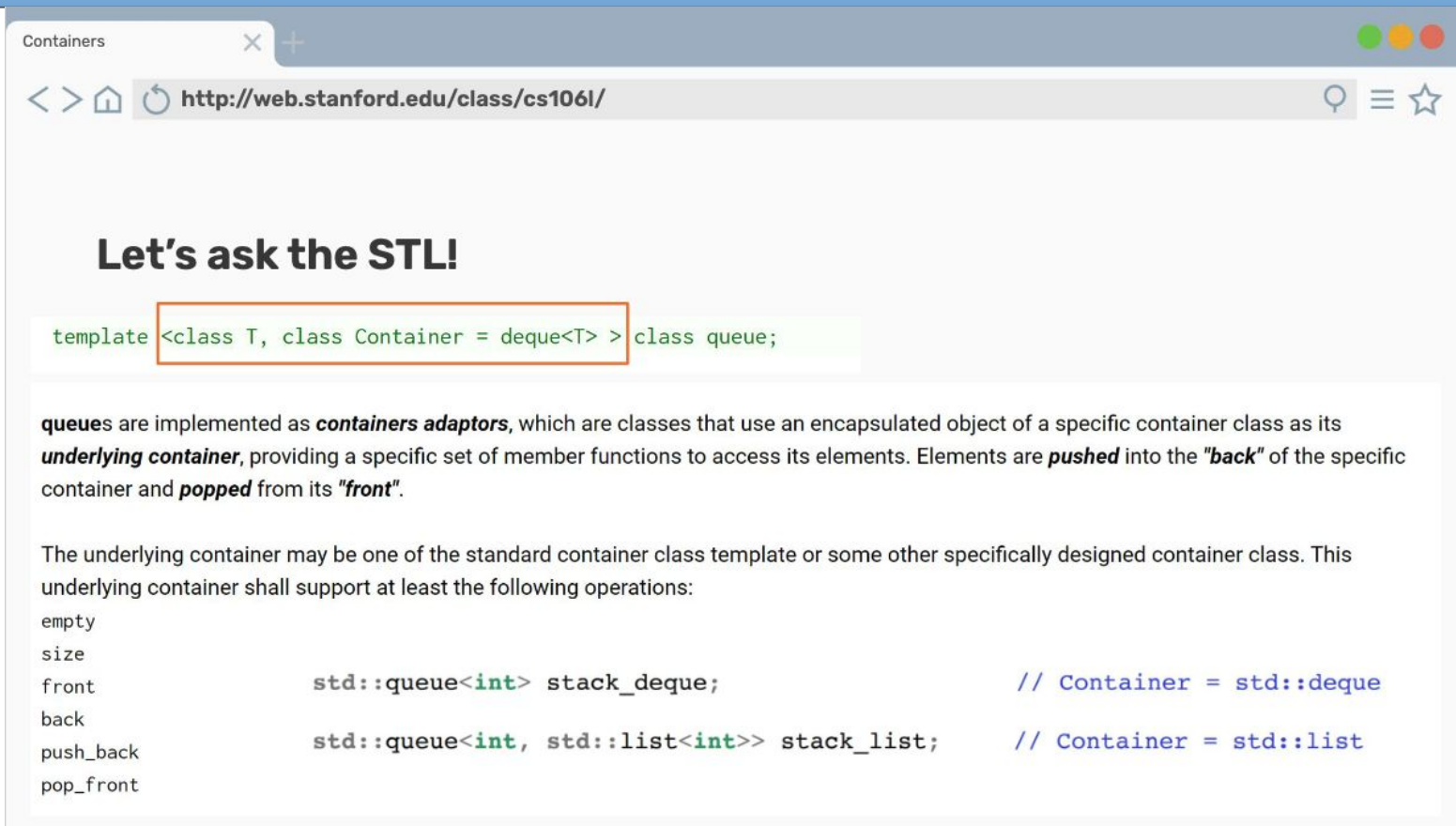
Container Adapters

Container Adapters

```
template <class T, class Container = deque<T> > class queue;
```
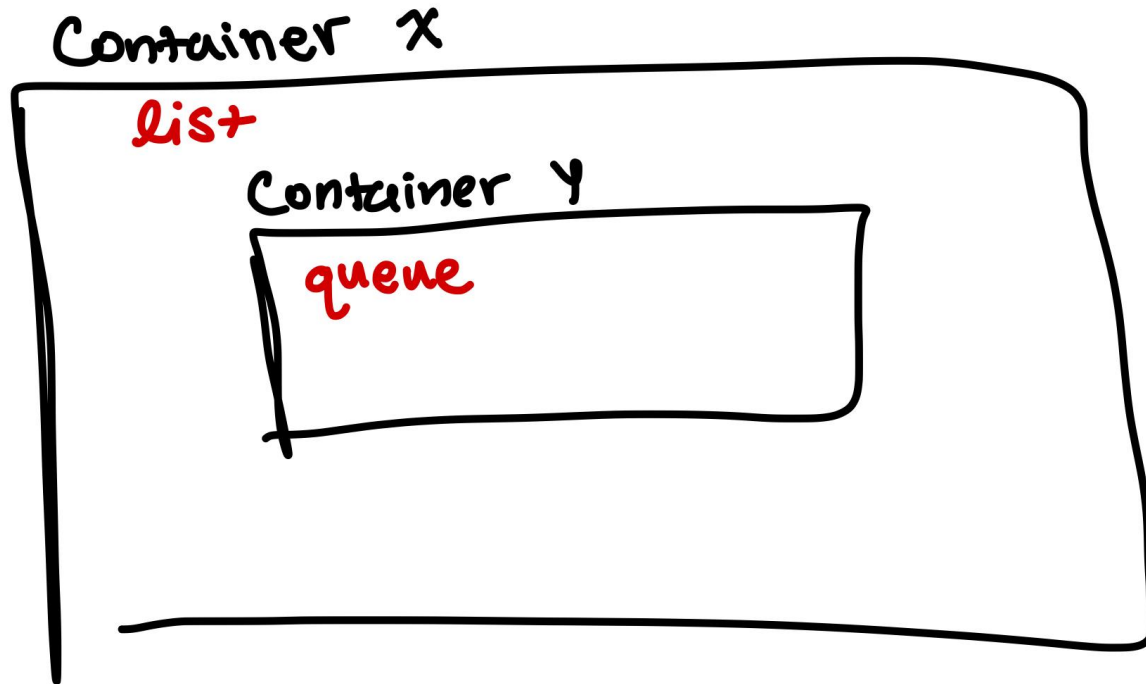
From last week

Containers

http://web.stanford.edu/class/cs106l/

Let's ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as **containers adaptors,** which are classes that use an encapsulated object of a specific container class as its **underlying container,** providing a specific set of member functions to access its elements. Elements are **pushed** into the **"back"** of the specific container and **popped** from its **"front"**.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

```
empty
size
front                std::queue<int> stack_deque;              // Container = std::deque
back
push_back             std::queue<int, std::list<int>> stack_list;   // Container = std::list
pop_front
```

Container Adapters

```cpp
std::queue<int, std::list<int>> stack_list;   // Container = std::list
```

Container X
list

Container Y
queue
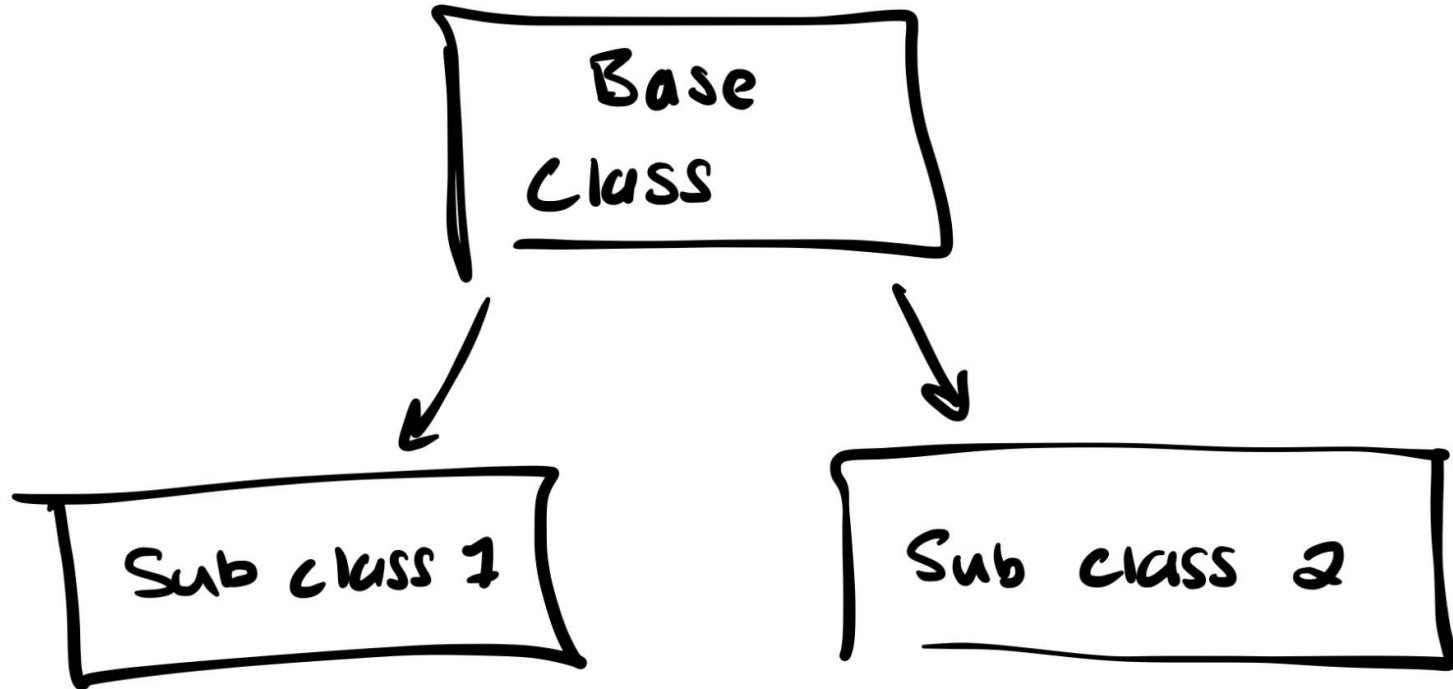
What questions do we have?

Plan

1. ~~Introduction to classes~~

2. ~~Container adapters~~

3. Inheritance

(Class) Inheritance

(Class) Inheritance
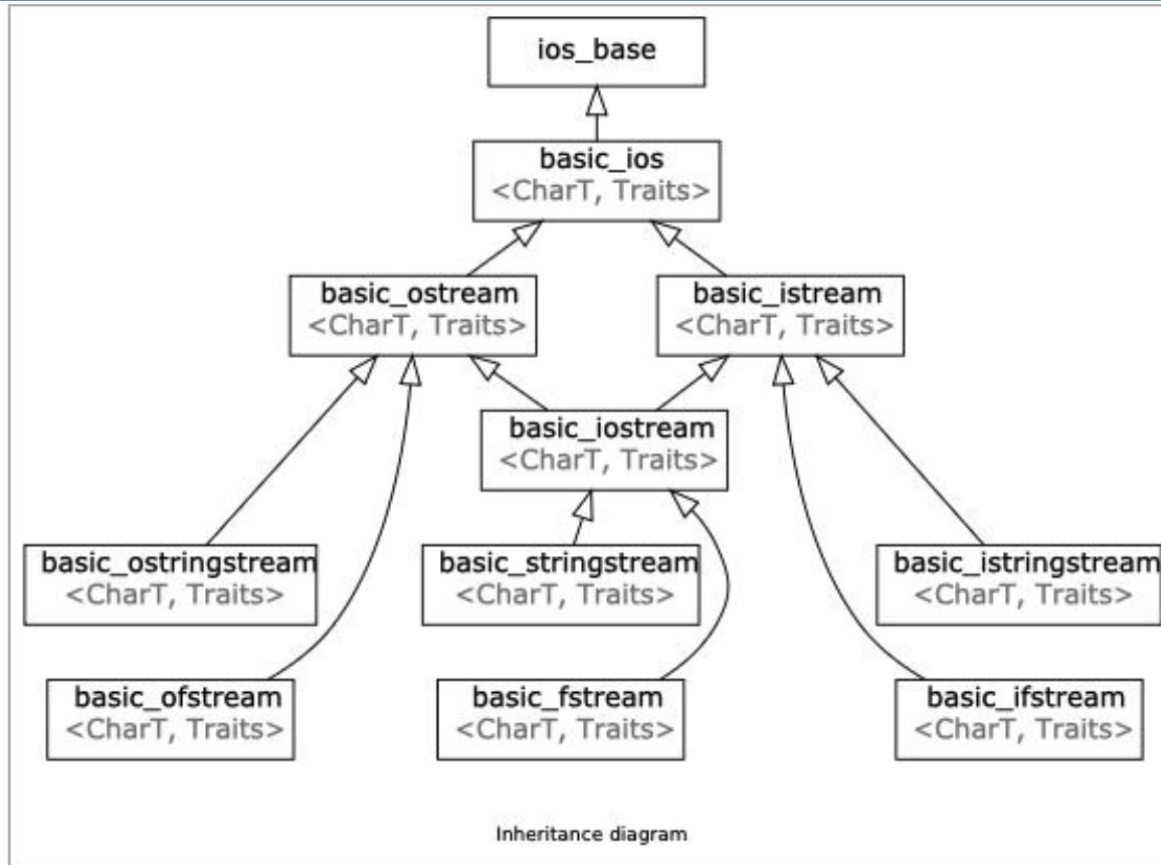
Inheritance diagram
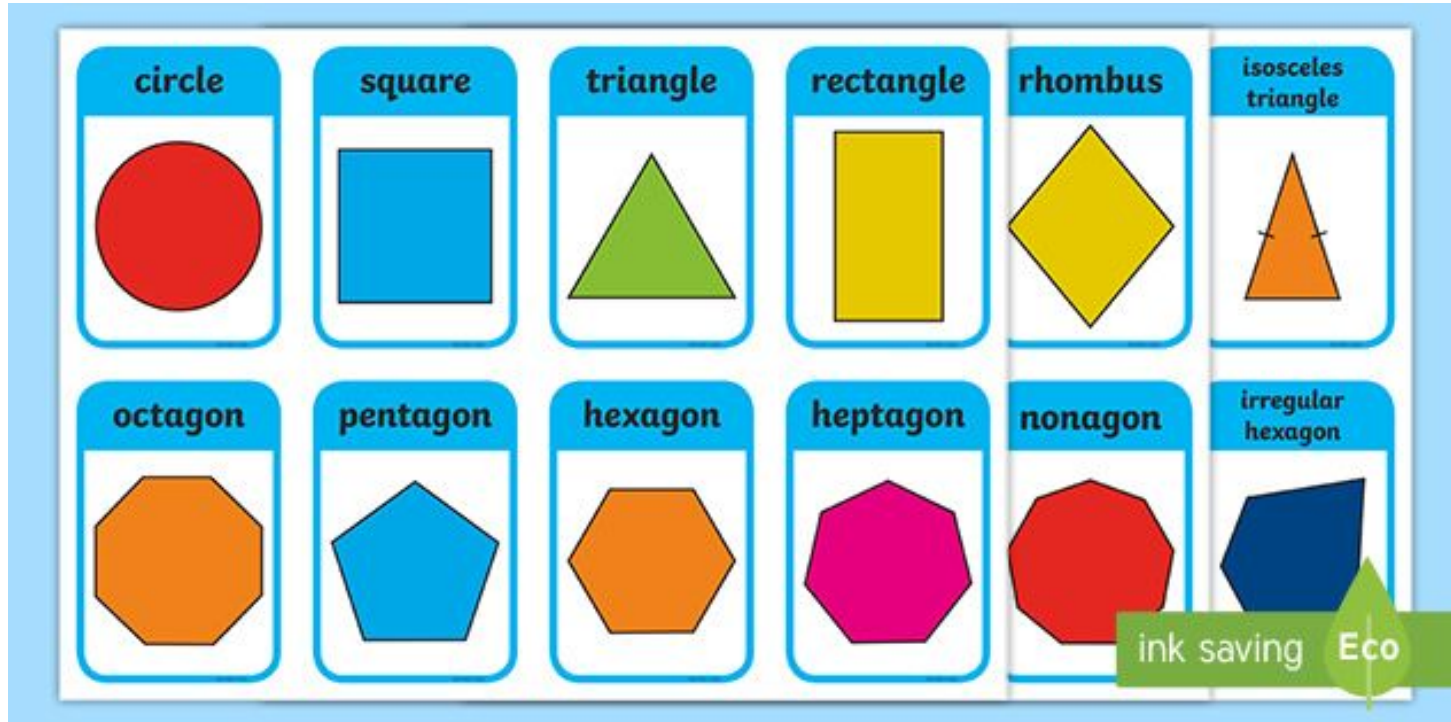
Inheritance

Why inheritance?

- **Polymorphism**: Different objects might need to have the same interface (we'll see this in just a second)

Inheritance

Why inheritance?
- **Polymorphism**: Different objects might need to have the same interface (we'll see this in just a second)

- **Extensibility:** Inheritance allows you to extend a class by creating a subclass with specific properties

So what is inheritance in practice?

So what is inheritance in practice?

What if we had a `shape` class, what do we think we would include?
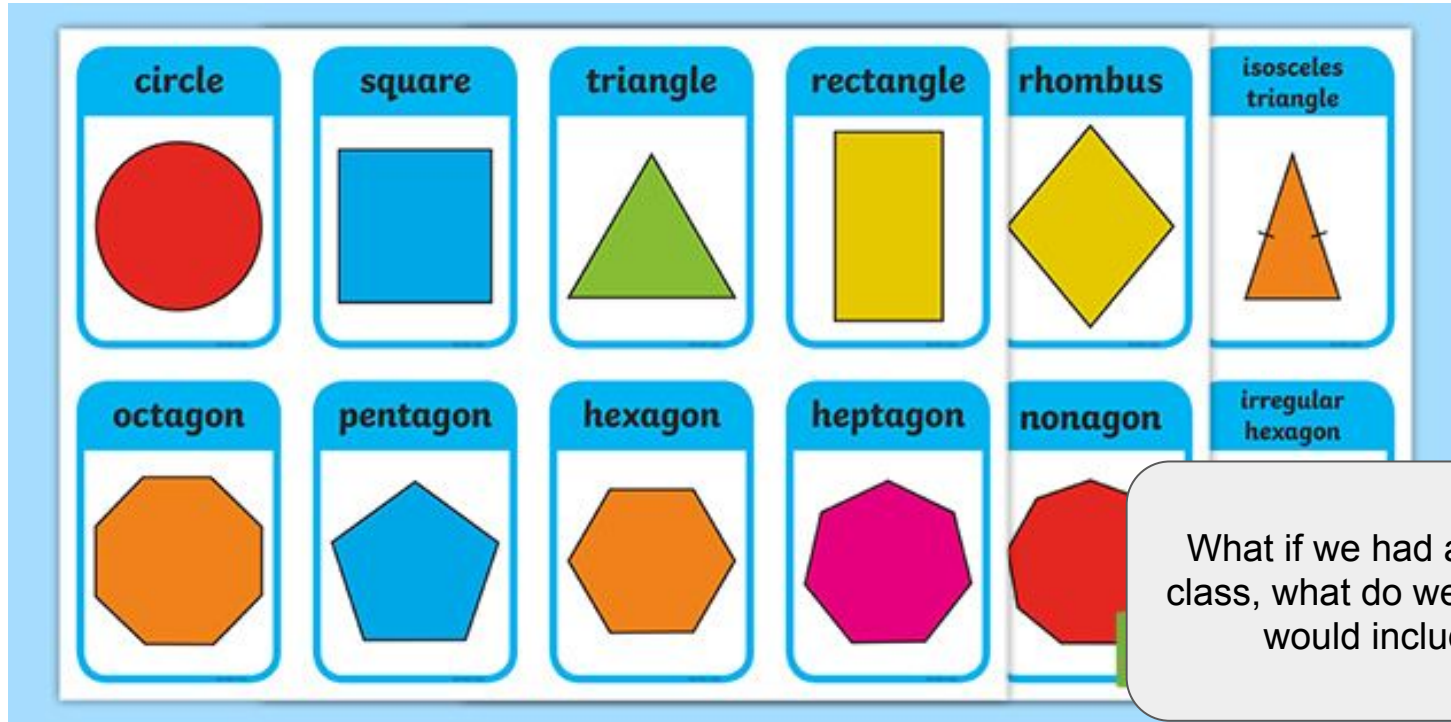
Shapes have

1. Area

Shapes have

1. Area

2. Radius? Or height? Or Width?

Shapes have

1. Area

2. Radius? Or height? Or Width?

3. Anything else?

Back to our class definition

.h file

```
class Shape {
public:
    virtual double area() const = 0;
};
```

This is a virtual function, meaning that it is instantiated in the base class but overwritten in the subclass.

(Polymorphism)

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};


class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius(radius) {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

Let's break this down step by step

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};


class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius(radius) {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

Here we declare the `Circle` class which inherits from the `Shape` class

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};


class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius(radius) {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

This is a virtual function we declare in our base class, `Shape`

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};


class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius{radius} {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

Here we have our constructor using list initialization construction

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};

class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius{radius} {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

Here we are overwriting the base class function `area()` for a circle

Back to our class definition

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};

class Circle : public Shape {
public:
    /// constructor
    Circle(double radius): _radius{radius} {};
    double area() const {
        return 3.14 * _radius * _radius;
    }
private:
    double _radius;
};
```

> Another pro of inheritance is the ***encapsulation*** of class variables.

Another one!

.h file

```cpp
class Shape {
public:
    virtual double area() const = 0;
};
. . . .
class Rectangle: public Shape {
public:
    /// constructor
    Rectangle(double height, double width): _height{height}, _width{width}
{};
    double area() const {
        return _width * _height;
    }
private:
    double _width, _height;
};
```

Shape subclasses!

.h file

```cpp
class Rectangle: public Shape {
public:
    /// constructor
    Rectangle(double height, double
width): _height{height},
_width{width} {};

    double area() const {
        return _width * _height;
    }
private:
    double _width, _height;
};
```

```cpp
class Circle : public Shape {
public:
    /// constructor
    Circle(double radius):
_radius{radius} {};
    double area() const {
        return 3.14 * _radius *
_radius;
    }
private:
    double _radius;
};
```

What questions do we have?
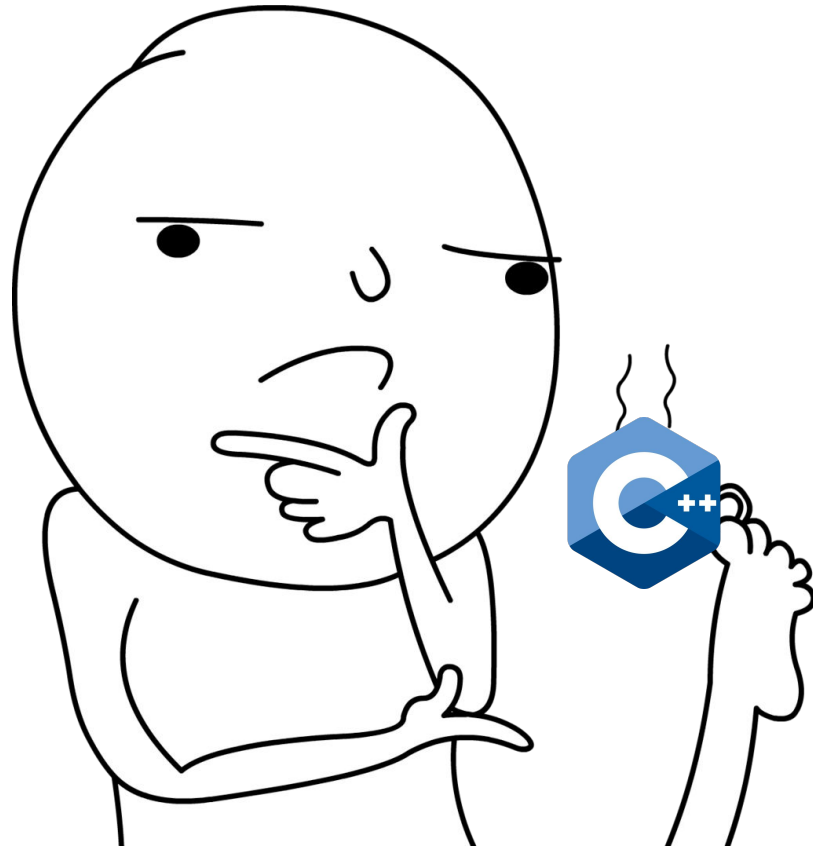
Subclasses vs Container Adapter

-   These are not to be confused

-   **Subclasses** inherit from base class functionality

-   **Container adapters** provide the interface for several classes and act as a template parameter.

Subclasses vs Container Adapter

- These are not to be confused

- **Subclasses** inherit from base class functionality

- **Container adapters** provide the interface for several classes and act as a **template parameter**.

We'll talk all about these on Thursday!

Lets implement a vector class for `ints`!

Let's write some code!

🤖