CS106L Lecture 7:
Classes

Thomas Poimenidis, Rachel Fernandez

Attendance

Today's Agenda

1. Classes
2. Inheritance

Why classes?

- C has no **objects**
- No way of **encapsulating** data and the functions that operate on that data
- No ability to have object-oriented programming (OOP) design patterns

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!

Surprise!

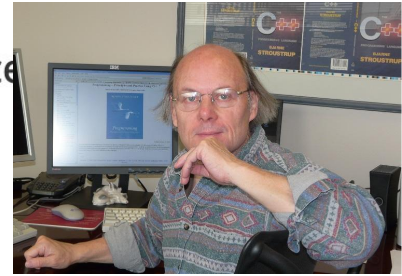
Containers are classes defined in the STL! 🥳

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**’

```
struct StanfordID {
    std::string name; // these are fields!
    std::string sunet;
    int idNumber;
};

StanfordID s;
s.name = “Thomas Poimenidis”;
s.sunet = “tpoimen”;
s.idNumber = 01243425;
```

Recall the ‘struct’

```
struct StanfordID {
    std::string name; // these are fields!
    std::string sunet;
    int idNumber;
};

StanfordID s;
s.name = “Thomas Poimenidis”;
s.sunet = “tpoimen”;
s.idNumber = 01243425;
```

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

Recall the ‘**struct**’

```
struct StanfordID {
    std::string name; // these are fields!
    std::string sunet;
    int idNumber;
};

StanfordID s;
s.name = “Thomas Poimenidis”;
s.sunet = “tpoimen”;
s.idNumber = 01243425;
s.idNumber = -123451234512345; // 💀?
```

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

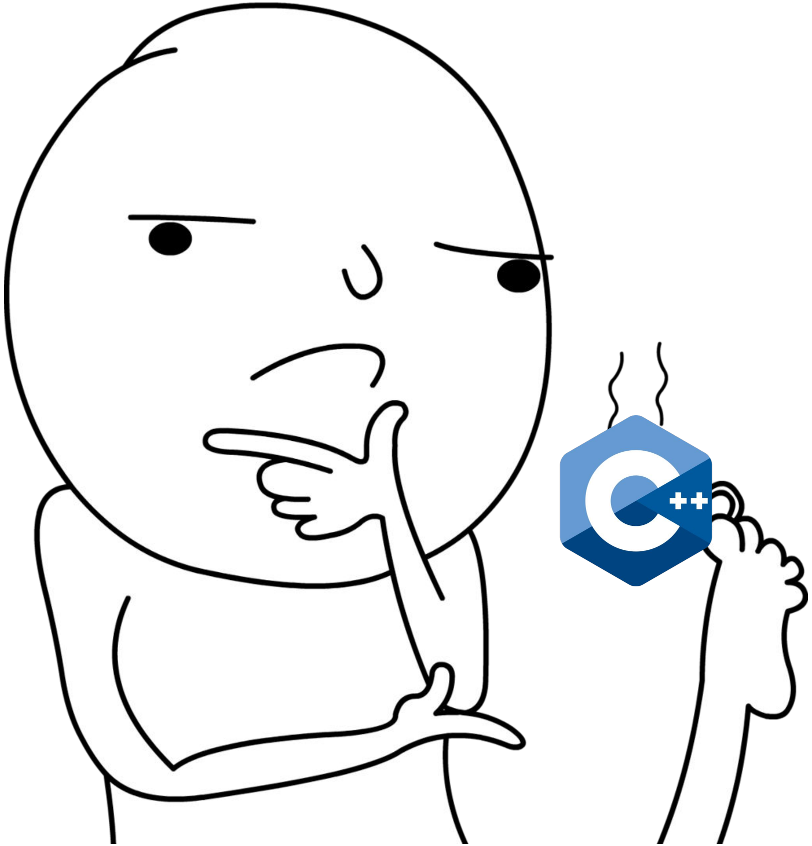
Recall the ‘**struct**’

```cpp
struct StanfordID {
    std::string name; // these are fields!
    std::string sunet;
    int idNumber;
};

StanfordID s;
s.name = “Thomas Poimenidis”;
s.sunet = “tpoimen”;
s.idNumber = 01243425;
s.idNumber = -12345123451234
```

There are no direct access controls while using structs

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**

```
class ClassName {
private:
```

```
public:
```

```
}
```

Classes have **public** and **private** sections!

A user can access the **public** stuff

User is restricted from `private`

```
class ClassName {
private:
```

```
public:
```

```
}
```

Classes have **`public`** and **`private`** sections!

A user can access the **`public`** stuff

But is **restricted** from accessing the private stuff

A backpack

A backpack

Struct

Class

Daily Meme

Let’s make a `StanfordID` class based on our struct!

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

	Header File (.h)	**Source File (.cpp)**
Purpose	Defines the interface	Implements class functions
Contains	Function prototypes, class declarations, type definitions, macros, constants	Function implementations, executable code
Access	This is shared across source files	Is compiled into an object file
Example	`void someFunction();`	`void someFunction(){...};`

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

	Header File (.h)	**Source File (.cpp)**
Purpose	Defines the interface	Implements class functions
Contains	Function prototypes, class declarations, type definitions, macros, constants	Function implementations, executable code
Access	This is shared across source files	Is compiled into an object file
Example	`void someFunction();`	`void someFunction() {...};`

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

	Header File (.h)	**Source File (.cpp)**
Purpose	Defines the interface	Implements class functions
Contains	Function prototypes, class declarations, type definitions, macros, constants	Function implementations, executable code
Access	This is shared across source files	Is compiled into an object file
Example	`void someFunction();`	`void someFunction() {...};`

C CS149intrin.h ×

prog2_vecintrin > C CS149intrin.h > ...

```
// Declare an integer vector register with __cs149_vec_int
#define __cs149_vec_int   __cs149_vec<int>

//***********************
//* Function Definition *
//***********************

// Return a mask initialized to 1 in the first N lanes and 0 in the others
__cs149_mask _cs149_init_ones(int first = VECTOR_WIDTH);

// Return the inverse of maska
__cs149_mask _cs149_mask_not(__cs149_mask &maska);

// Return (maska | maskb)
__cs149_mask _cs149_mask_or(__cs149_mask &maska, __cs149_mask &maskb);

// Return (maska & maskb)
__cs149_mask _cs149_mask_and(__cs149_mask &maska, __cs149_mask &maskb);

// Count the number of 1s in maska
int _cs149_cntbits(__cs149_mask &maska);

// Set register to value if vector lane is active
//  otherwise keep the old value
void _cs149_vset_float(__cs149_vec_float &vecResult, float value, __cs149_mask &mask);
void _cs149_vset_int(__cs149_vec_int &vecResult, int value, __cs149_mask &mask);
// For user's convenience, returns a vector register with all lanes initialized to value
__cs149_vec_float _cs149_vset_float(float value);
__cs149_vec_int _cs149_vset_int(int value);
```

Class design

- A constructor
- Private member functions/variables
- Public member functions (interface for a user)
- Destructor

Constructor

- The constructor initializes the state of newly created objects

Constructor

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

Constructor

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

```
s.name = “Thomas Poimenidis”;

s.sunet = “tpoimen”;

s.idNumber = 01243425;
```

Constructor

.h file

```
class StanfordID {
private:
    ?

public:
    ?
};
```

Constructor

.h file

```
class StanfordID {
private:
    std::string name;
    std::string sunet;
    int idNumber;

public:
    // constructor for our StudentID
    StanfordID(std::string name, std::string sunet, int idNumber);
}
```

Constructor

.h file

```
class StanfordID {
private:
    std::string name;
    std::string sunet;
    int idNumber;

public:
    // constructor for our StudentID
    StanfordID(std::string name, std::string sunet, int idNumber);
}
```

The syntax for the constructor is just the name of the class

Constructor

.h file

```
class StanfordID {
private:
    std::string name;
    std::string sunet;
    int idNumber;

public:
    // constructor for our StudentID
    StanfordID(std::string name, std::string sunet, int idNumber);
    // method to get name, sunet, and idNumber, respectively
    std::string getName();
    std::string getSunet();
    int getID();
}
```

Parameterized Constructor

!!.cpp file!! (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    idNumber = idNumber;
}
```

Parameterized Constructor

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    idNumber = idNumber;
}
```

Remember namespaces, like **std::**

Parameterized Constructor

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    idNumber = idNumber;
}
```

Remember namespaces, like **std::**

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

Parameterized Constructor

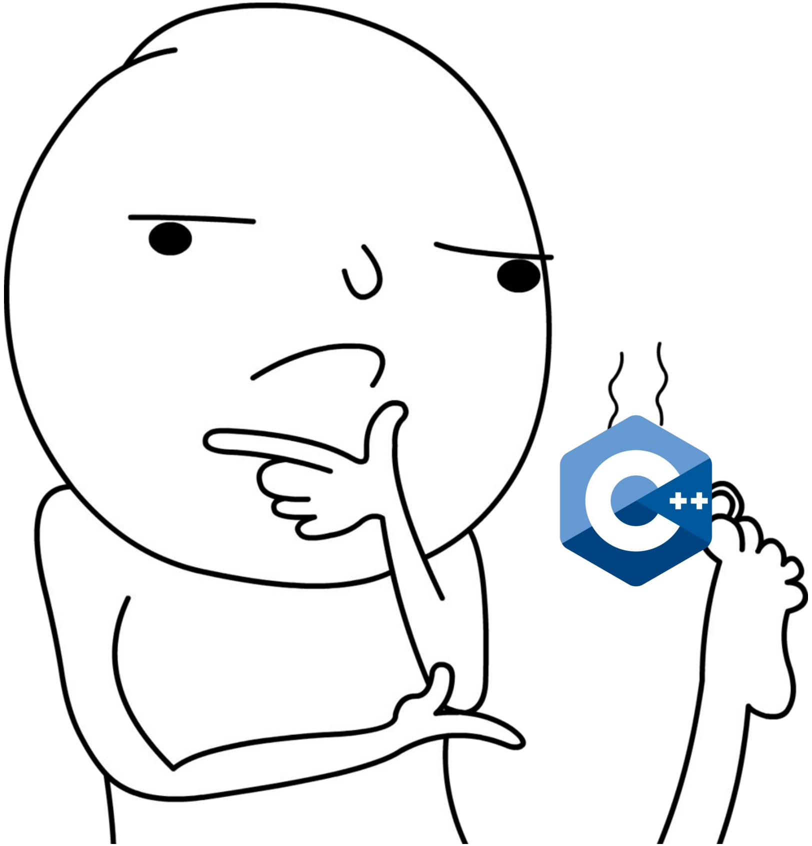
.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    if ( idNumber > 0 ) idNumber = idNumber;
}
```

We can now also enforce checks on the values that we initialize or modify our members to!

What questions do we have?



Parameterized Constructor

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    if ( idNumber > 0 ) idNumber = idNumber;
}
```

Does anyone see a problem here?

Parameterized Constructor

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    name = name;
    sunet = sunet;
    if ( idNumber > 0 ) idNumber = idNumber;
}
```

Does anyone see a problem here?

Our .h definition

.h file

```
#include <string>
class StanfordID {
private:
    std::string name;
    std::string sunet;
    int idNumber;

public:
    // constructor for our student
    StanfordID(std::string name, std::string sunet, int idNumber);
    // method to get name, sunet, and idNumber, respectively
    std::string getName();
    std::string getSunet();
    int getID();
}
```

Use the **this** keyword

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    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)

```
#include "StanfordID.h"
#include <string>

// list initialization constructor
StanfordID::StanfordID(std::string name, std::string sunet, int idNumber):
name{name}, sunet{sunet}, idNumber{idNumber} {};
```

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

Default constructor

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

// default constructor
StanfordID::StanfordID() {
    name = “John Appleseed”;
    sunet = “jappleseed”;
    idNumber = 00000001;
}
```

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

Constructor Overload

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

// default constructor
StanfordID::StanfordID() {
    name = “John Appleseed”;
    sunet = “jappleseed”;
    idNumber = 00000001;
}
// parameterized constructor
StanfordID::StanfordID(std::string name, std::string sunet, int idNumber) {
    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

```
class StanfordID {
private:
    std::string name;
    std::string sunet;
    int idNumber;

public:
    /// constructor for our student
    StanfordID(std::string name, std::string sunet, int idNumber);
    /// method to get name, sunet, and ID, respectively
    std::string getName();
    std::string getSunet();
    int getID();
}
```

Let's implement them

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

std::string StanfordID::getName() {

}

std::string StanfordID::getSunet() {

}

int StanfordID::getID() {

}
```

Implemented members

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

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

std::string StanfordID::getSunet() {
    return this->sunet;
}

int StanfordID::getID() {
    return this->idNumber;
}
```

Implemented members (setter functions)

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

void StanfordID::setName(std::string name) {
    this->name = name;
}

void StanfordID::setSunet(std::string sunet) {
    this->sunet = sunet;
}
void StanfordID::setID(int idNumber) {
    if (idNumber >= 0){
        this->idNumber = idNumber;
    }
}
```

The destructor

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

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

The destructor

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

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

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

The destructor

.cpp file (implementation)

```
#include “StanfordID.h”
#include <string>

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

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

The destructor

.cpp file (implementation)

```
#include "StanfordID.h"
#include <string>

StanfordID::~StanfordID() {
    // 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

```
template <typename T>
class vector {
  using iterator = T*;

  // Implementation details...
};
```

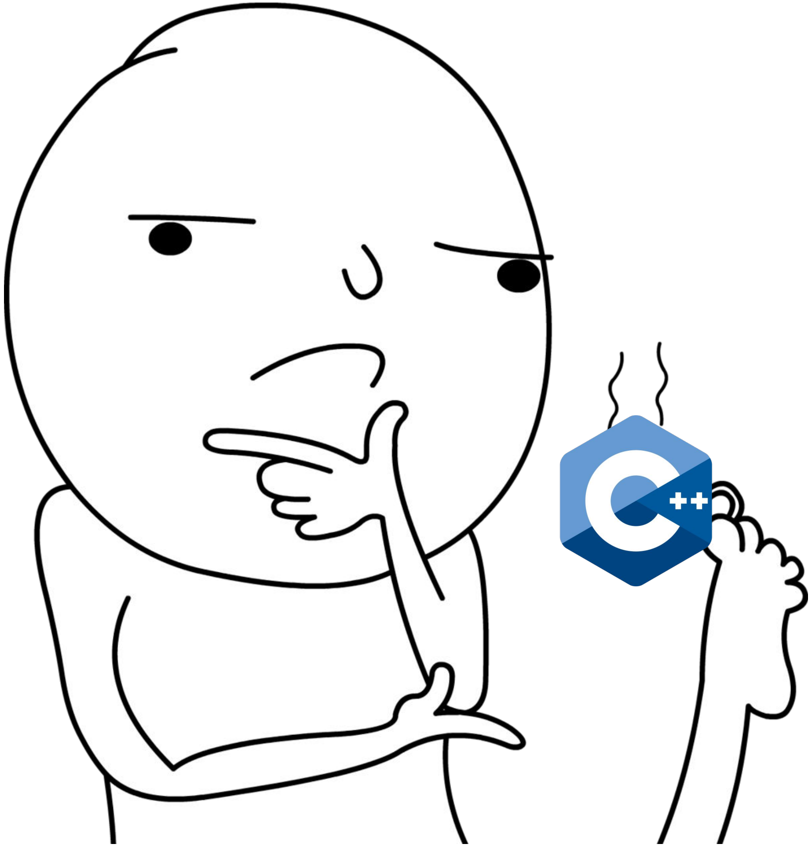
Back to our class definition

.h file

```
class StanfordID {
private:
    // An example of type aliasing
    using String = std::string;
    String name;
    String sunet;
    int idNumber;

public:
    // constructor for our student
    StanfordID(String name, String sunet, int idNumber);
    // method to get name, state, and age, respectively
    String getName();
    String getSunet();
    int getID();
}
```

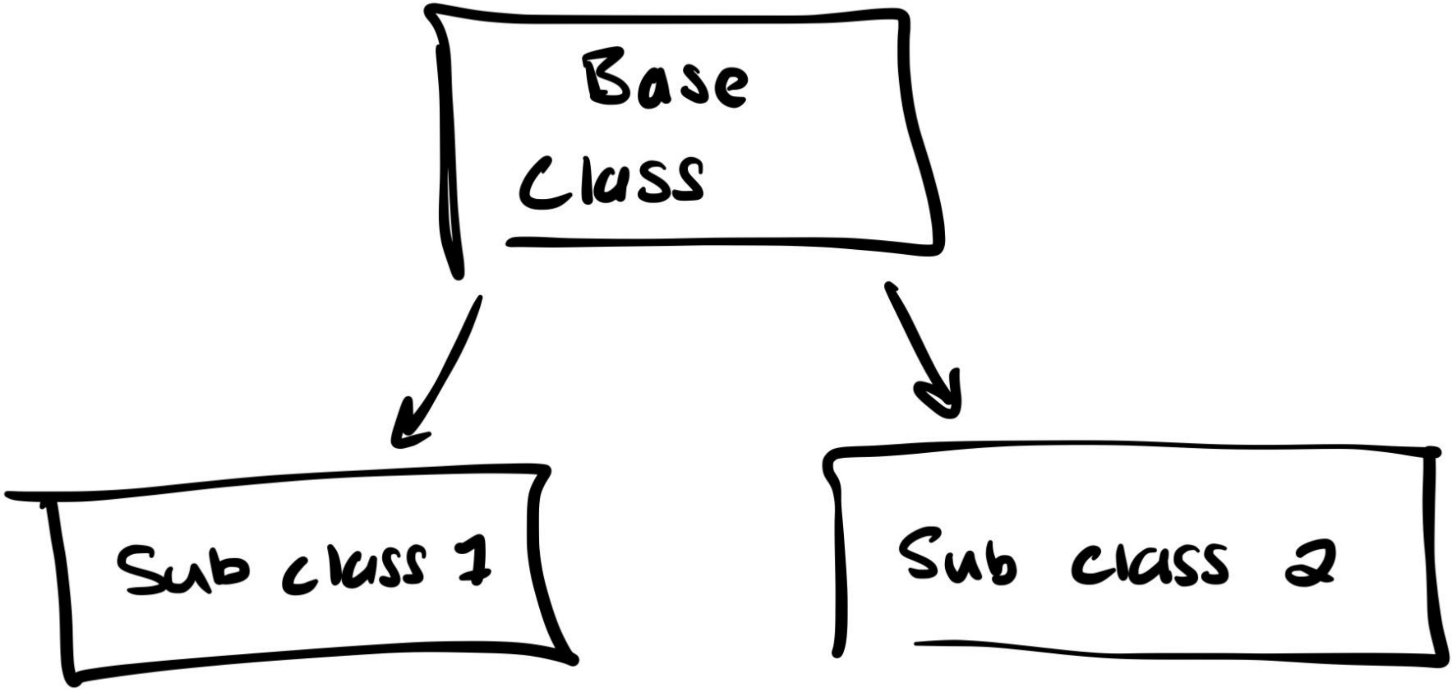
What questions do we have?



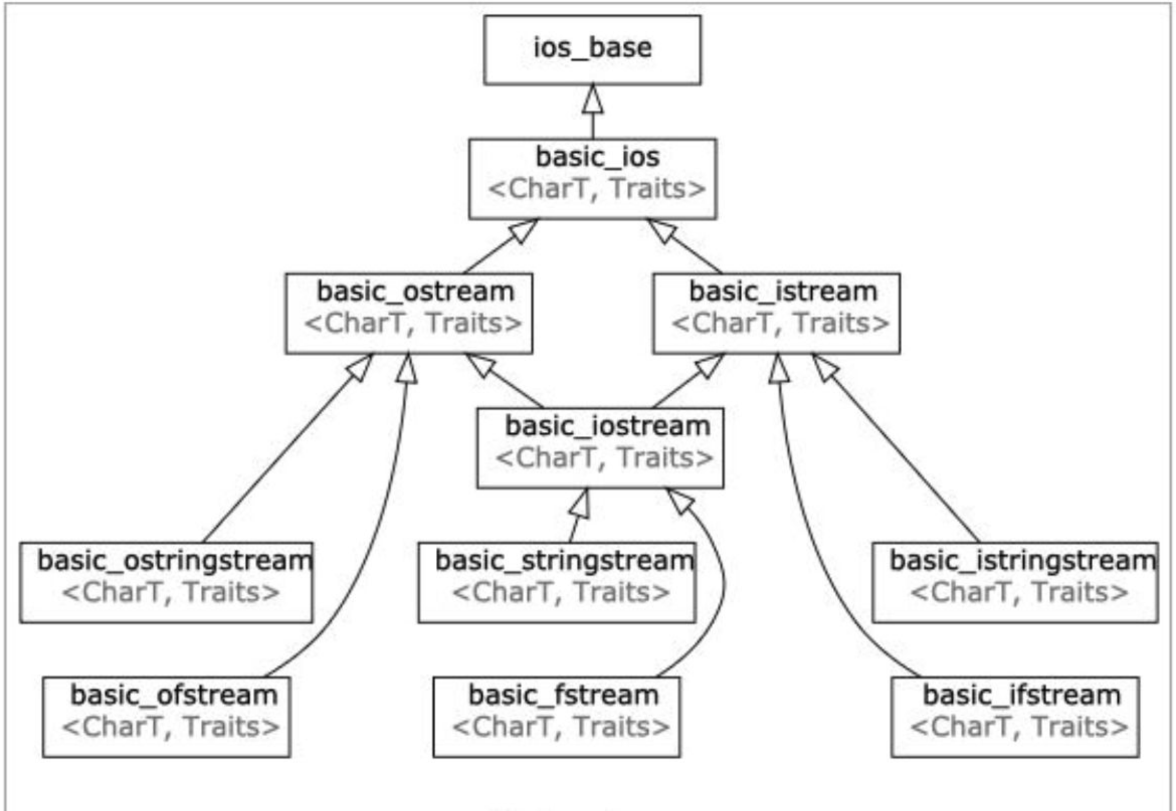
Plan

1. Classes
2. **Inheritance**

(Class) Inheritance



Circling back to this diagram



Inheritance diagram

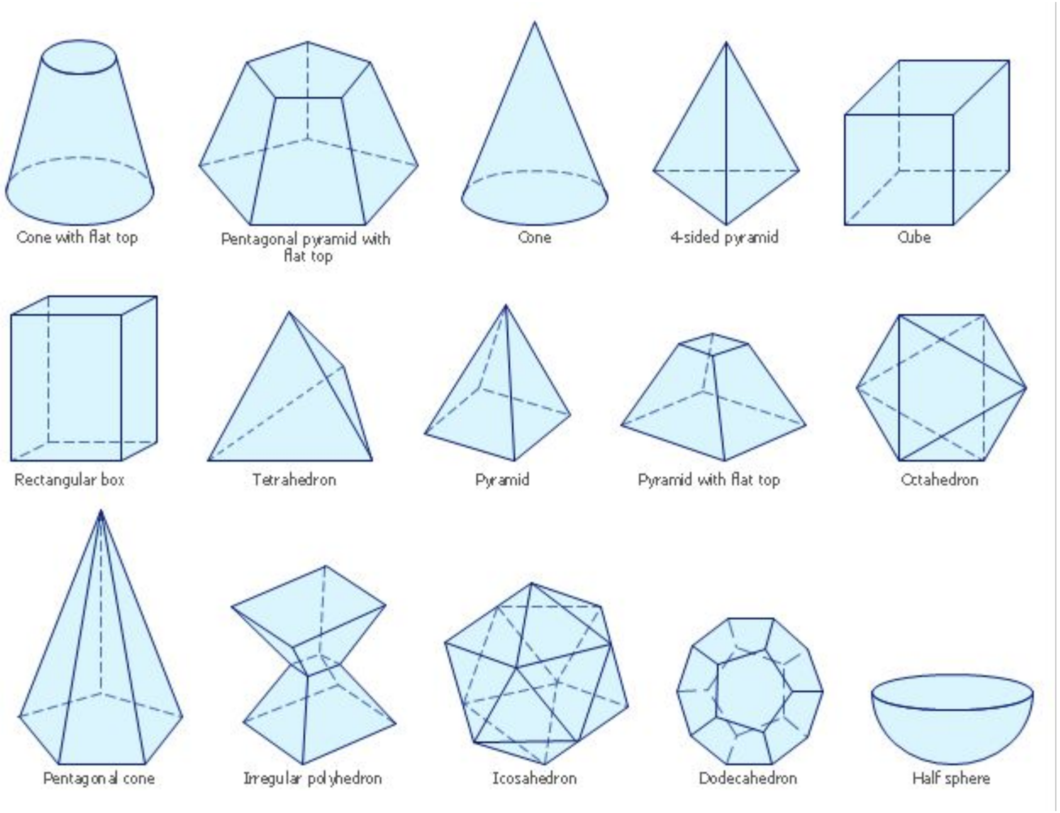
Inheritance

- **Dynamic Polymorphism**: Different types of objects may need the same interface

Inheritance

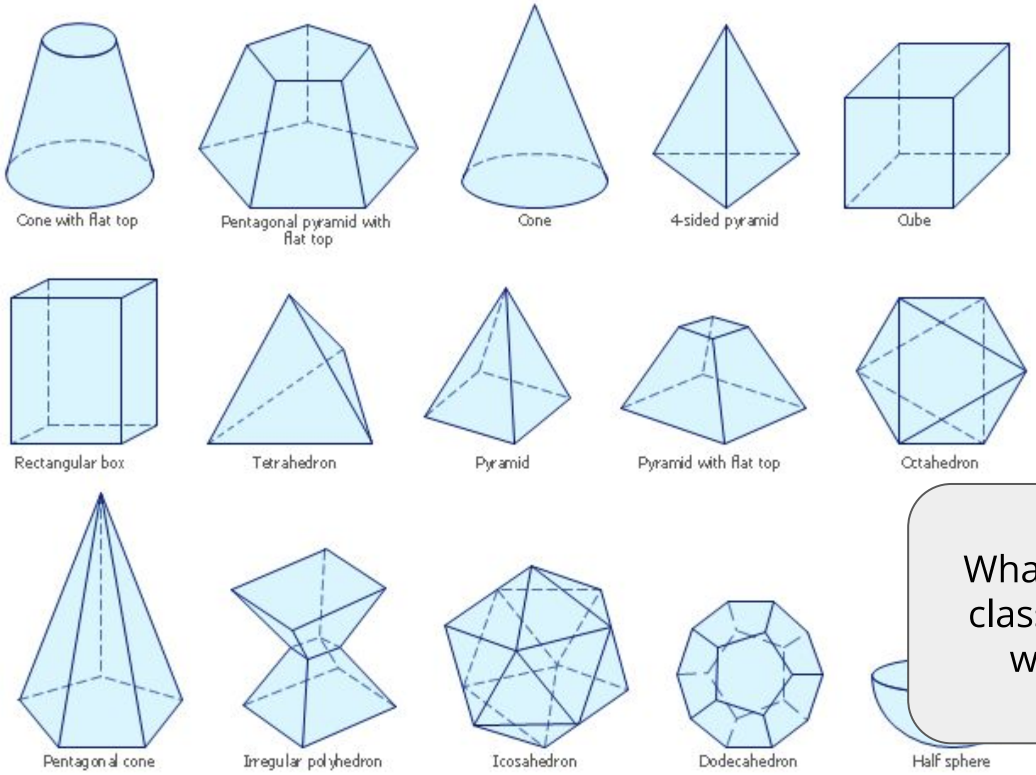
- **Dynamic Polymorphism**: Different types of objects may need the same interface
- **Extensibility:** Inheritance allows you to extend a class by creating a subclass with specific properties

Inheritance in practice



[source]

Inheritance in practice



What if we had a **shape** class, what do we think we would include?

[source]

Shapes have

- Area

Shapes have

- Area
- Radius? Or height? Or Width?

Shapes have

- Area
- Radius? Or height? Or Width?
- Anything else?

Shape class definition

.h file

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

Pure virtual function: it is instantiated in the base class but overwritten in the subclass.

(Dynamic Polymorphism)

`Circle` class definition

.h file

```
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

Circle class definition

.h file

```
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

`Circle` class definition

`.h file`

```
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 pure virtual function we declare in our base class, **`Shape`.**

`Circle` class definition

.h file

```
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;
};
```

constructor using list initialization construction

`Circle` class definition

.h file

```
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

`Circle` class definition

.h file

```
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.

Rectangle class definition

.h file

```
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 subclass definitions

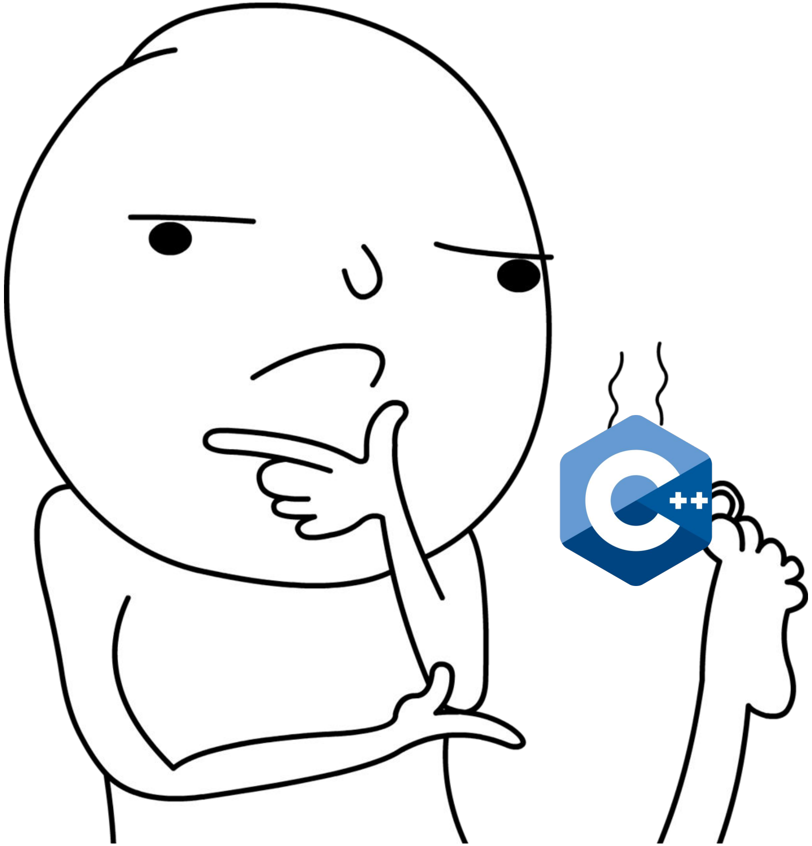
.h file

```
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;
};
```

```
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?



Types of inheritance

Type	**`public`**
Example	**`class B: public A {...}`**
Public Members	Are public in the derived class
Protected Members	Protected in the derived class
Private Members	Not accessible in derived class

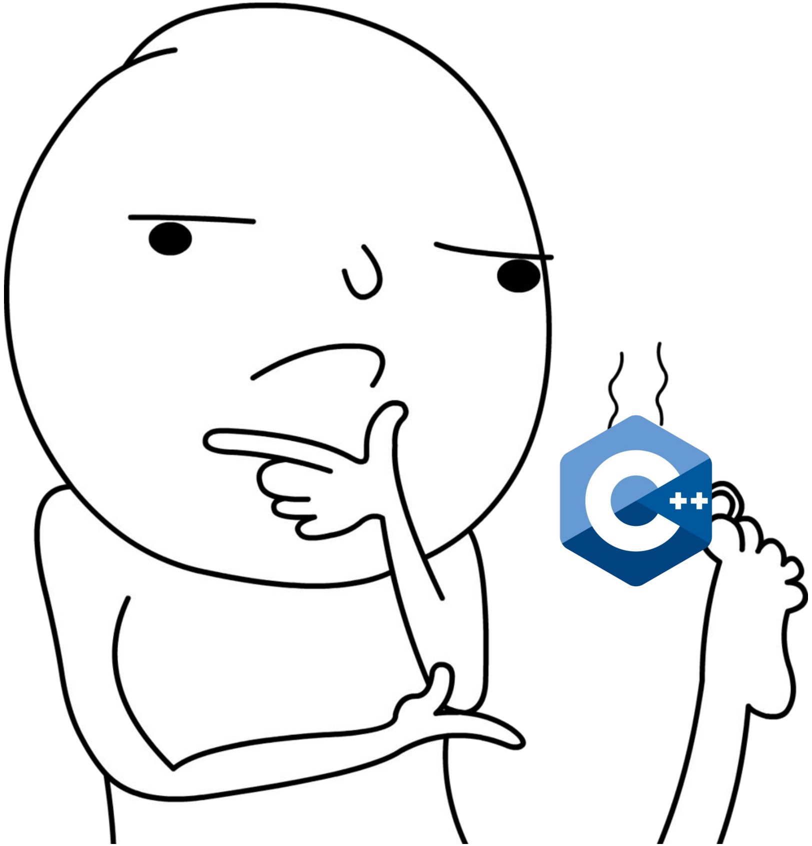
Types of inheritance

Type	`public`	`protected`
Example	`class B: public A {...}`	`class B: protected A {...}`
Public Members	Are public in the derived class	Protected in the derived class
Protected Members	Protected in the derived class	Protected in the derived class
Private Members	Not accessible in derived class	Not accessible in derived class

Types of inheritance

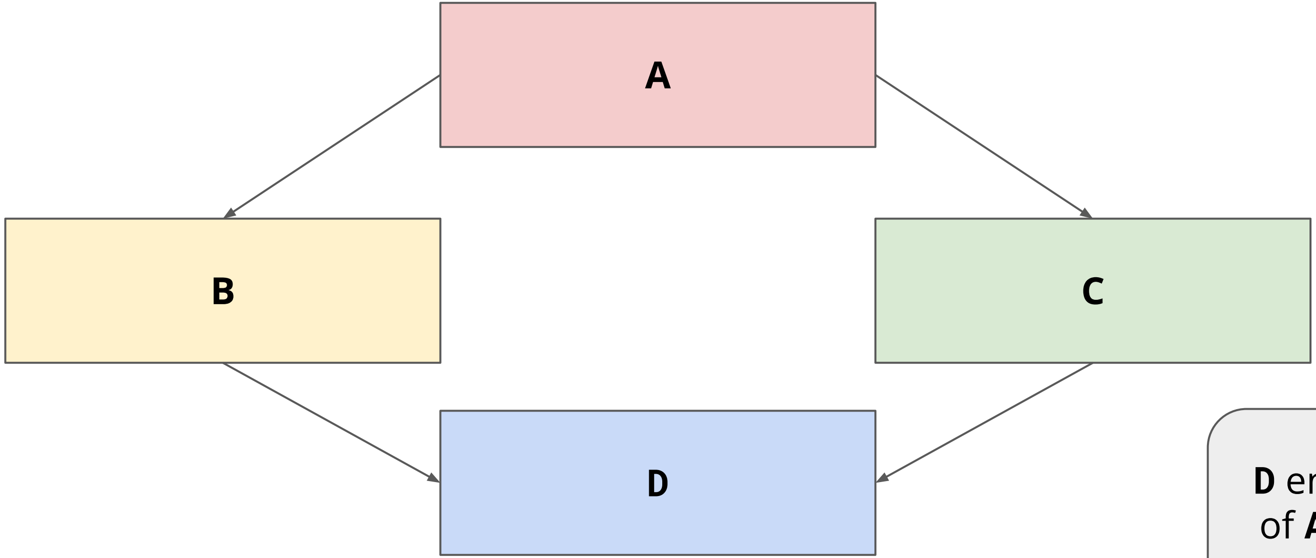
Type	`public`	`protected`	`private`
Example	`class B: public A {...}`	`class B: protected A {...}`	`class B: private A {...}`
Public Members	Are public in the derived class	Protected in the derived class	Privated in the derived class
Protected Members	Protected in the derived class	Protected in the derived class	Private in the derived class
Private Members	Not accessible in derived class	Not accessible in derived class	Not accessible in derived class

What questions do we have?



The Diamond Problem

Since both **B** and **C** inherit from **A**, they each call the constructor of **A**.



D ends up with two copies of **A**. One from **B** another from **C**

A,B,C,D classes

.h file

```
class A {
public:
    A();
    void hello() {
        // print “hello from A”
    }
}
```

```
class C : public A {
public:
    C();
}
```

```
class B : public A {
public:
    B();
}
```

```
class D
: public B, public C {
public:
    D();
}
```

Which hello() does D call?

```
D obj {};

obj.B::hello()

obj.C::hello()

obj.hello()
```

Which hello() does D call?

```
D obj {};

obj.B::hello()  // call B's hello method

obj.C::hello()

obj.hello()
```

Which hello() does D call?

```
D obj {};

obj.B::hello()  // call B’s hello method

obj.C::hello() // call C’s hello method

obj.hello()
```

Which hello() does D call?

```
D obj {};

obj.B::hello()  // call B’s hello method

obj.C::hello() // call C’s hello method

obj.hello() // whose method do I call ???
```

The Diamond Problem

The way to fix this is to make **B** and **C** inherit from **A** in a **virtual way**.

Virtual inheritance means that a derived class, in this case **D**, should only have a single instance of base classes, in this case **A**.

Solution? Inherit Virtually!

.h file

```
class C : virtual public A {
public:
     C();
}
```

```
class B : virtual public A {
public:
     B();
}
```

This creates a shared instance of **A** between **B** and **C**!

Fixed!

```
D obj {};

obj.B::hello()  // call B's hello method

obj.C::hello() // call C's hello method

obj.hello() // no longer ambiguous :)
```

Recap

1. Classes allow you to encapsulate functionality and data with access protections

2. Inheritance allows us to design powerful and versatile abstractions that can help us model complex relationships in code.

3. These concepts are tricky – this lecture *really* highlights the power of C++