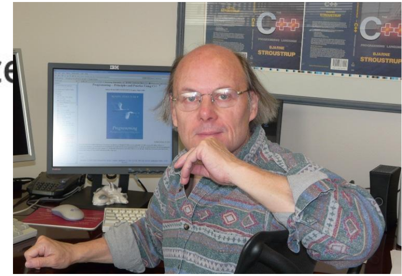
Comparing ‘struct’ and ‘class’

> 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 StudentID {
    std::string name; // these are fields!
    std::string sunet;
    int idNumber;
};

Student s;
s.name = “Fabio Ibanez”;
s.sunet = “fabioi”;
s.idNumber = 01243425;
s.idNumber = -123451234512345; // 💀?
```

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

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

Recall from Jacob’s lecture!

```
struct StanfordID {
  string name;        // These are called fields
  string sunet;       // Each has a name and type
  int idNumber;
};

StanfordID id;         // Access fields with ‘.’
id.name = "Jacob Roberts-Baca";
id.sunet = "jtrb";
id.idNumber = 6504417;
```

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() {...};`

Constructor

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

```
s.name = “Fabio Ibanez”;

s.sunet = “fabioi”;

s.idNumber = 01243425;
```

Constructor

.h file

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

public:
    // constructor for our StudentID
    StudentID(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 “StudentID.h”
#include <string>

StudentID::StudentID(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!

Parameterized Constructor

.cpp file (implementation)

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

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

Does anyone see a problem here?

Use the **this** keyword

.cpp file (implementation)

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

StudentID::StudentID(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 “StudentID.h”
#include <string>

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

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

Constructor Overload

.cpp file (implementation)

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

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

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

Implemented members

.cpp file (implementation)

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

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

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

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

Implemented members (setter functions)

.cpp file (implementation)

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

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

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

The destructor

.cpp file (implementation)

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

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

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

The destructor

.cpp file (implementation)

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

StudentID::~StudentID() {
    /// 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 StudentID {
private:
    // An example of type aliasing
    using String = std::string;
    String name;
    String sunet;
    int idNumber;

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

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

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;
};
```

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

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;
};
```

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

Person class

.h file

```
class Person {
protected:
    std::string name;

public:
    Person(const std::string& name) : name(name) {}
    std::string getName();
}
```

Student class

.h file

```
class Student : public Person {
protected:
    std::string idNumber;
    std::string major;
    std::string advisor;
    uint16_t year;
public:
    Student(const std::string& name, …);
    std::string getIdNumber() const;
    std::string getMajor() const;
    uint16_t getYear() const;
    void setYear(uint16_t year);
    void setMajor(const std::string& major);
    std::string getAdvisor() const;
    void setAdvisor(const std::string& advisor);
};
```

The constructor has all of the protected members. For the sake of space I've omitted them here.

Employee class

.h file

```
class Employee : public Person {
protected:
    double salary;
public:
    Employee(const std::string& name);
    virtual std::string getRole() const = 0;
    virtual double getSalary() const = 0;
    virtual void setSalary() const = 0;
    virtual ~Employee() = default;
};
```

SectionLeader class

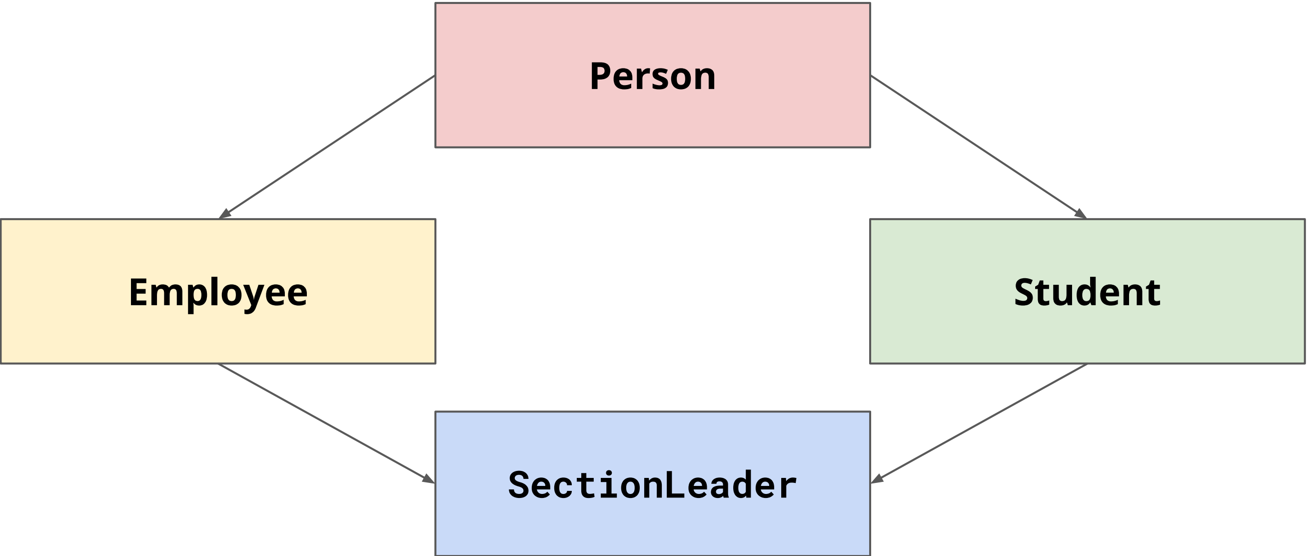
.h file

```
class SectionLeader : public Person, public Employee {
protected:
    std::string section;
    std::string course;
    std::vector<std::string> students;
public:
    Student(const std::string& name, …);
    std::string getSection() const;
    std::string getCourse() const;
    void addStudent(const std::string& student);
    void removeStudent(const std::string& student);
    std::vector<std::string> getStudents() const;
    std::string getRole() const override;
    double getSalary() const override;
    void setSalary(double salary) override;
};
```

And the destructor `~SectionLeader()`

The Diamond Problem

Since both **`Student`** and **`Employee`** inherit from **`Person`**, they each call the constructor of **`Person`**.



`SectionLeader` ends up with two copies of **`Person`**. One from **Student** another from **Employee**

The Diamond Problem

The way to fix this is to make **`Employee`** and **`Student`** inherit from **`Person`** in a **virtual way**.

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

Student class

.h file

```
class Student : public virtual Person {
protected:
    std::string idNumber;
    std::string major;
    std::string advisor;
    uint16_t year;
public:
    Student(const std::string& name, …);
    std::string getIdNumber() const;
    std::string getMajor() const;
    uint16_t getYear() const;
    void setYear(uint16_t year);
    void setMajor(const std::string& major);
    std::string getAdvisor() const;
    void setAdvisor(const std::string& advisor);
};
```

Employee class

.h file

```
class Employee : public virtual Person {
protected:
    double salary;
public:
    Employee(const std::string& name);
    virtual std::string getRole() const = 0;
    virtual double getSalary() const = 0;
    virtual void setSalary() const = 0;
    virtual ~Employee() = default;
};
```

The Diamond Problem

The way to fix this is to make **`Employee`** and **`Student`** inherit from **`Person`** in a **virtual way**.

Inheritance virtually just means that a derived class, in this case **`SectionLeader`**, should only have a single instance of base class **`Person`**.

❗ **This requires the derived class to initialize the base class!** ❗