09-25-2025

Classes

- A class is a type.
- It describes a set of objects that:
  - Share some properties.
  - Meet some requirements.
- Adding properties or requirements:
  - Constrains the type.
  - Fewer objects have the properties or requirements.
    - Example: There are many students, but fewer **CSE3150** students specifically.
    - Called **subclasses**.

Student Class Example

```cpp
#pragma once
#include <string>
#include <vector>

struct Student {
    std::string name_;
    double mt_, final_;
    std::vector<double> homeworks_;
};
```

Style Guidelines

- Classes (and types in general) start with uppercase.
- Methods (and functions in general) start with lowercase.
- Underscores:
  - Generally, don't use **leading underscores** (reserved in C++).
  - Use **trailing underscores** to indicate private attributes.

Objects

- Objects are **instances of a class**.
  - Members of the set of objects denoted by the class.
- When a program starts:
  - It has a lot of classes.
  - Often, no instances yet.
    - You can declare classes at the start of runtime with `static`, but generally frowned upon.
  - Instances are generally created at runtime.

Smart Pointers

- Raw pointers (`Student* s = new Student();`) require manual `delete`.
- Smart pointers automatically manage memory and help avoid leaks.
- Three types of smart pointers.

`std::unique_ptr`

- Represents sole ownership of a resource.
- Cannot be copied, only moved.
- Automatically deletes the resource when it goes out of scope.

```cpp
#include <memory>

auto up = std::make_unique<Student>();
// auto up2 = up; // ERROR: cannot copy
auto up2 = std::move(up); // Ownership transferred
```

`std::shared_ptr`

- Shared ownership: multiple pointers can manage the same object.
- Internally uses reference counting.
- Deletes the object **only** when the last owner goes out of scope.
- Very similar to Python's memory management (minus garbage collection).

```cpp
#include <memory>

auto sp1 = std::make_shared<Student>();
auto sp2 = sp1; // Both share ownership
// Resource destroyed only when both sp1 and sp2 go out of scope
```

Cycles with `shared_ptr`

- Two objects that reference each other with `shared_ptr` can cause a memory leak.
- Reference count never goes to zero.
- In Python, this is still garbage collected, but slowly.

`std::unique_ptr` and `std::shared_ptr` Comparison

Feature | `unique_ptr` | `shared_ptr` |

|||–| | Ownership | Single | Multiple | | Copyable? | No (only move) | Yes | | Overhead | None | Reference counting (minimal) | | Use Case | Strict Ownership | Shared Resources |

Best Practices

- Prefer `std::make_unique` or `std::make_shared` over raw `new`.
- Default to `unique_ptr` unless shared ownership is truly needed.

Other Smart Pointers

`std::weak_ptr<T>`

- Non-owning observer of a `shared_ptr`.
- Used to prevent cyclic references.

Fixing cycles with `weak_ptr`

- Two objects that reference each other with `shared_ptr` can cause a memory leak.
- Solution: Use `std::weak_ptr` for one of the links.
- Breaks the cycle because it does not increment the reference count.
- In Python, similar functionality comes from `weakref.proxy`.

Student Class Example Continued

`student.h`

```cpp
#pragma once
#include <string>
#include <vector>

struct Student {
    std::string name_;
    double mt_, final_;
    std::vector<double> homeworks_;
};
```

student.cpp

```cpp
#include "student.h"
#include <memory>
#include <iostream>

using std::cout, std::endl;

int main() {
    Student s; // Stored on stack

    // Modifying attributes of Student instance stored on stack
    s.name_ = "John";
    s.mt_ = 80;
    s.final_ = 90;
    s.homeworks_.push_back(65);
    s.homeworks_.push_back(95);

    std::shared_ptr<Student> sp = std::make_shared<Student>(); // Stored on heap; or use auto keyword
    // Could also be unique_ptr; can copy into a unique_ptr but not copy a unique_ptr to another var

    *sp = s;
    cout << "s.name_ = " << s.name_ << endl;
    cout << "sp->name_ = " << sp->name_ << endl; // Remember arrow syntax for pointer attribute from C
    // Name will be the same

    cout << "&s.name_ = " << &s.name_ << endl;
    cout << "&s.name_ = " << s->name_ << endl;
    // Copying name from stack onto heap, will have different addresses

    s.name_ = "Bert";
    cout << "&s.name_ = " << &s.name_ << endl;
    cout << "&s.name_ = " << s->name_ << endl;
    // Will have same addresses as above respectively after name_ change
}
```

Going OO

- We must combine both state and behavior and have strong cohesion between both
- Must couple state and behavior
  - We must create abstractions for the Student classes
- Separation between interface and implementation
  - Like with a queue ADT, it can be implemented under the hood with a list or singly linked list or doubly linked list and the users of the class don't care they can still enqueue and dequeue like

normal

student.h

```h
#pragma once
#include <string>
#include <vector>
#include <istream>
#include <ostream>

struct Student {
    std::string name_;
    double mt_, final_;
    std::vector<double> homeworks_;

    // Don't need to define functions in class, can just define stubs like below and define the impleme
    void read(std::istream& is);
    void print(std::ostream& os);
};
```

student.cpp

```cpp
#include "student.h"
#include <memory>
#include <iostream>

using std::cout, std::endl;

void Student::read(std::istream& is) { // Implementation of function stub defined here
    is >> name_ >> mt_ >> final_; // Note: Don't need to use self. or this. to access attributes
    int num_homeworks = 0;
    is >> num_homeworks;

    for (int i = 0; i < num_homeworks; i++) {
        int v;
        is >> v;
        homeworks_.push_back(v);
    }
}

void Student::print(std::ostream& os) {
    os << "Name: " << name_ << "[" << mt_ << "," << final_ << "]" << endl;

    for(int v : homeworks_) {
        os << "\t" << v << endl;
    }
}

int main() {
    Student s; // Stored on stack

    // Modifying attributes of Student instance stored on stack
    s.name_ = "John";
    s.mt_ = 80;
    s.final_ = 90;
    s.homeworks_.push_back(65);
```

```
    s.homeworks_.push_back(95);

    std::shared_ptr<Student> sp = std::make_shared<Student>(); // Stored on heap; or use auto keyword
    // Could also be unique_ptr; can copy into a unique_ptr but not copy a unique_ptr to another var

    *sp = s;

    return 0;
}
```

Method Overloading

- C++ supports overloaidng methods, which is several definitions with:
  - Same name
  - Different number of arguments
  - Different types of arguments
- C++ disambiguates the call and selects the right method
- NOT the same as method overriding (which is related to inheritance)

Class Lifetime

Birth: Constructor

- Idea:
  - Go a memory pool (heap or stack) and grab memory for class
    - Either grab memory from stack (atomatic)
    - Or use keyword `new` to grab memory from heap
  - Then initialize
  - Execute a method when creating an object
  - Can oberload constructors (multiple ways to initialize)
  - Multiple kinds of constructors
    - Default, Customer, Copy Move

Default Constructor `student.h`

```
#pragma once
#include <string>
#include <vector>
#include <istream>
#include <ostream>

struct Student {
    std::string name_;
    double mt_, final_;
    std::vector<double> homeworks_;

    // Defualt constructor; remember to define all variables that are not objects with their own defaul
    Student() {
        mt_ = final_ = 0;
    }
    // std::string and std::vector have their own default constructors, we do not need to do it here

    // Constructor overloading
    Student(const std::string& s) {
```

```
        mt_ = final_ = 0;
        name_ = s;
    }

    // Initializer list; allows for increased performance
    Student(const std::string& s): name_(s), mt_(0), final_(0) {}

    void read(std::istream& is);
    void print(std::ostream& os);
};
```

Copy Constructor `student.h`

```
#pragma once
#include <string>
#include <vector>
#include <istream>
#include <ostream>

struct Student {
    std::string name_;
    double mt_, final_;
    std::vector<double> homeworks_;

    // Copy Constructor: Takes in a second student, and builds a new student
    Student(const Student& s2) {
        name_ = s2.name_;
        mt_ = s2.mt_;
        final_ = s2.final_;

        for (double d : s2.homeworks_) {
            homeworks_.push_back(d);
        }
    }

    void read(std::istream& is);
    void print(std::ostream& os);
};
```