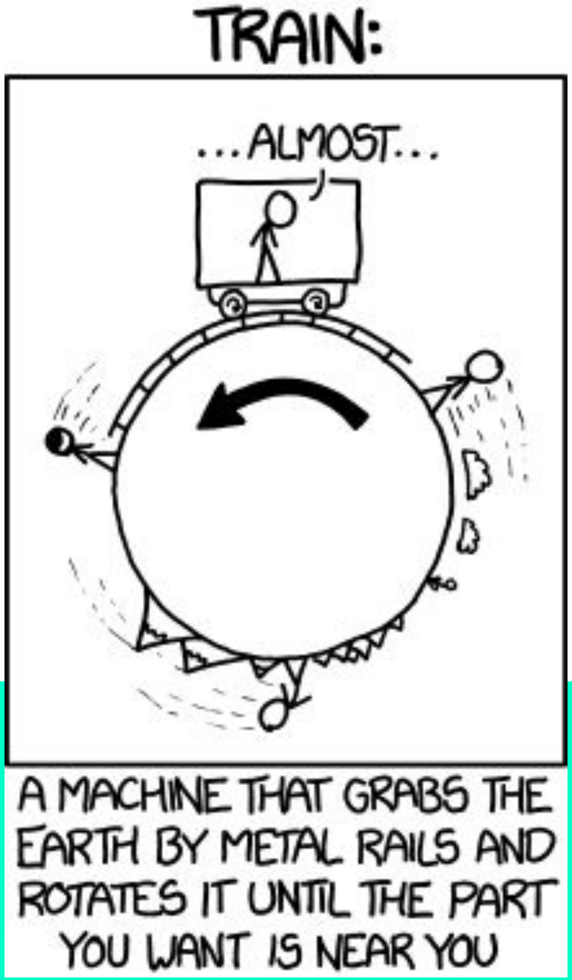

TRAIN:
...ALMOST...
A MACHINE THAT GRABS THE EARTH BY METAL RAILS AND ROTATES IT UNTIL THE PART YOU WANT IS NEAR YOU

Lab 4 - Abstract Data Types

ReadMe

Reminders

- Lab 4 due Sunday, October 7th at 8pm on Canvas
- Project 2 due TONIGHT at 8pm
  - DO NOT WAIT UNTIL 8 TO SUBMIT

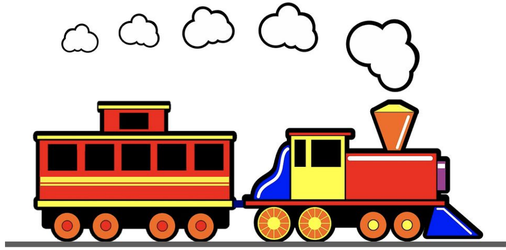
Agenda

- Review: ADTs, structs, classes
- Worksheet
- Debugging Tutorial
- Do lab / ask me questions

Let's talk about trains*.

*Train example stolen from Jon Juett.

ADTs



- Help us organize things in an intuitive way
- C - style idea: it is intuitive to us to think about objects as possessing properties:
  - “A train has cars, wheels, and a station.”
- C++ - style idea: it is intuitive to us to think about objects as possessing properties, but also as having associated actions:
  - “A train has cars, wheels, and a station. But it also runs, burns coal, leaves the station, and whistles.”
- The key difference between C-style ADT’s and C++ style ADT’s is that C++ style ADT’s surround the idea that “objects don’t just have properties, they should also have associated actions.”

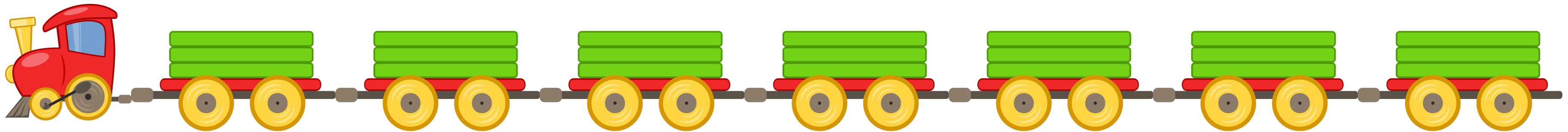
Structs vs. Classes

Structs

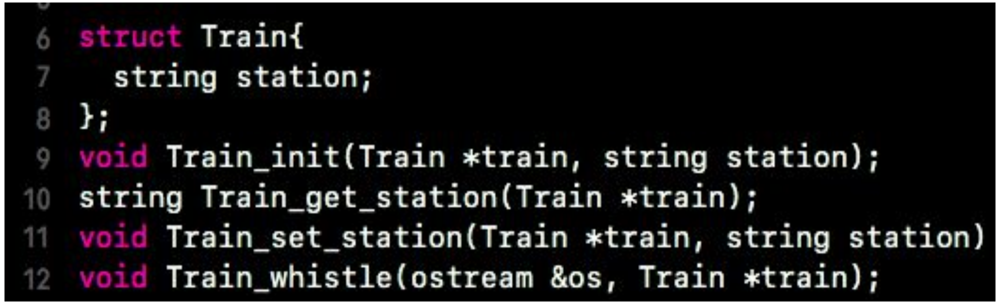
- C style
- Only data -- use as Plain Old Data structures
- No constructors -- start off with uninitialized values and then use init functions
- Member variable/ function access is public by default

Classes

- C++ style
- Data & functions -- use with data, member functions, inheritance, etc.
- Constructors & destructors
- Member variable/ function access is private by default



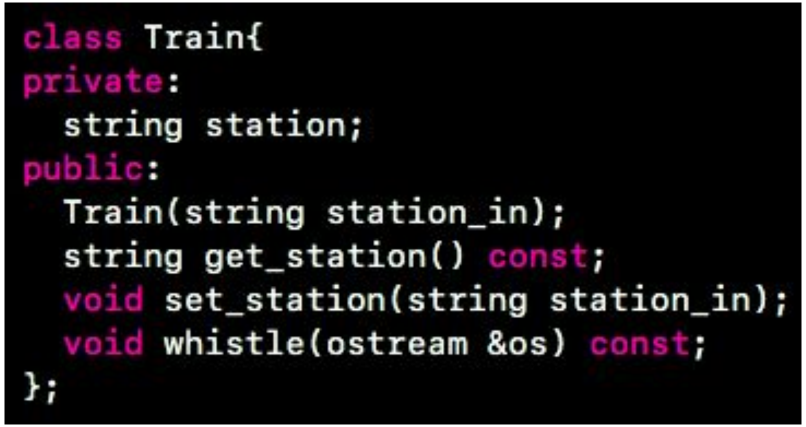
Structs vs. Classes



```
struct Train{
  string station;
};
void Train_init(Train *train, string station);
string Train_get_station(Train *train);
void Train_set_station(Train *train, string station)
void Train_whistle(ostream &os, Train *train);
```

C-style / struct ADT -- notice how the data (station name) is inside the struct, but the associated functions are separate pieces. Also notice how we have to pass a pointer to the train into these functions.

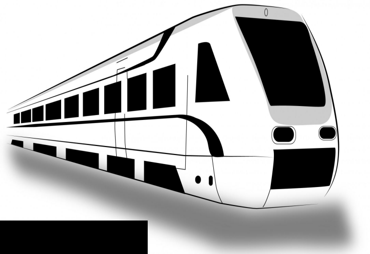
*Notice, however, that we can do pretty much everything with this object that we can with the class type object shown at the right. Just in different ways!



```
class Train{
private:
  string station;
public:
  Train(string station_in);
  string get_station() const;
  void set_station(string station_in);
  void whistle(ostream &os) const;
};
```

C++ style / class ADT -- notice how both data and functions are declared inside the class. Notice how we don’t need to pass any train pointer into these functions. That’s because functions in the train class already have a pointer to the train-- the “this” pointer!

Structs vs. Classes -- Initialization



Pass a pointer to the train you want to initialize

```
struct Train{
  string station;
};

void Train_init(Train *train, string station) {
    train->station = station;
    cout << "Hello train!" << endl;
}
```

No scope resolution operator

No member initializer list

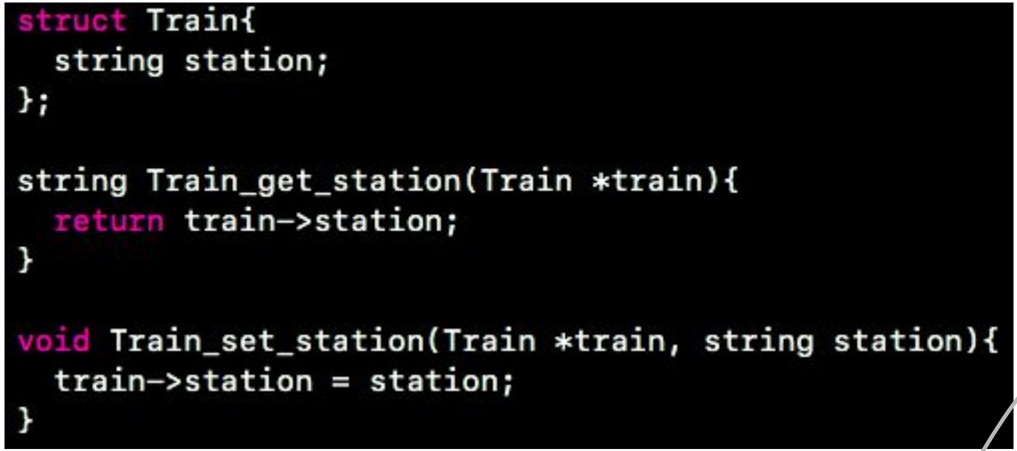
```
class Train{
private:
  string station;
public:
  Train(string station_in);
  string get_station() const;
  void set_station(string station_in);
  void whistle(ostream &os) const;
};
```

Use the :: operator to define a function outside the class

```
Train::Train(string station_in) : station(station_in) {
      cout << "Hello train!" << endl;
  }
```

Use member initializer list and/or write code in function body

Structs vs. Classes -- Getters and Setters

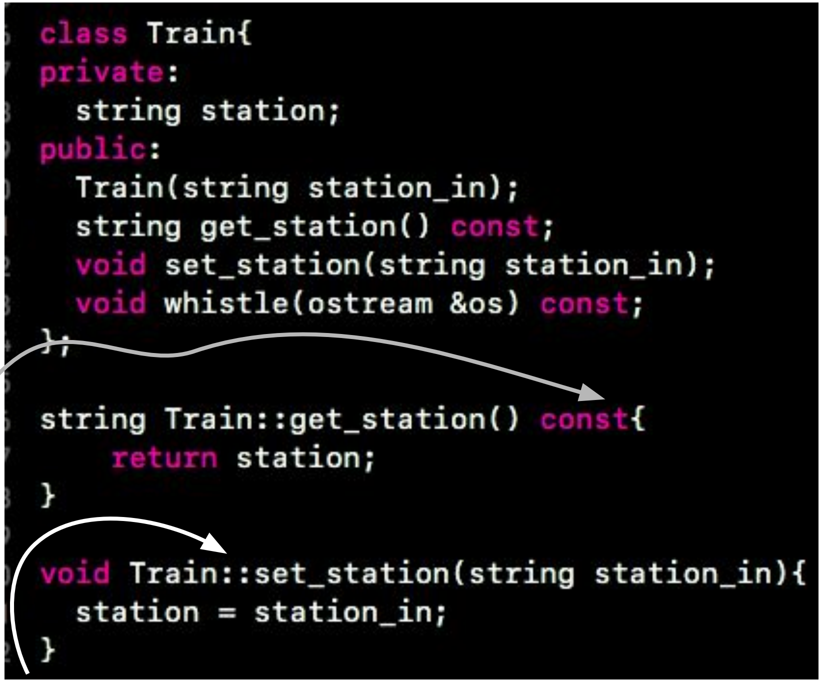


```
struct Train{
  string station;
};

string Train_get_station(Train *train){
  return train->station;
}

void Train_set_station(Train *train, string station){
  train->station = station;
}
```

No scope resolution needed

Note: in both cases, we are using getter and setter functions to help us separate interface from implementation.

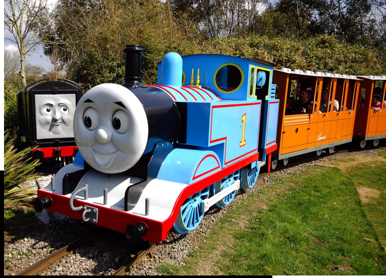
This "const" prevents get_station() from changing any member variables

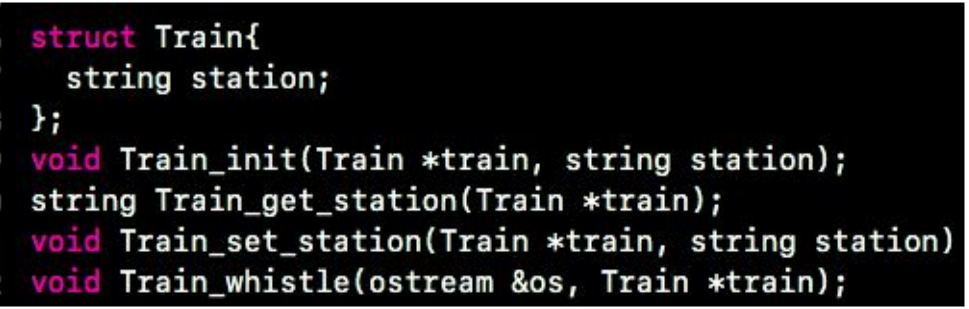


```
class Train{
private:
  string station;
public:
  Train(string station_in);
  string get_station() const;
  void set_station(string station_in);
  void whistle(ostream &os) const;
};

string Train::get_station() const{
    return station;
}

void Train::set_station(string station_in){
  station = station_in;
}
```

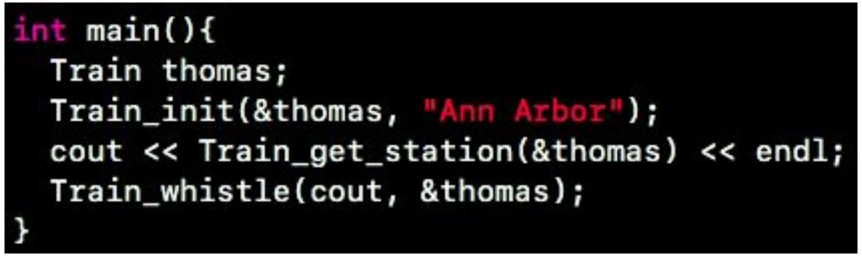
Scope resolution operator

Structs vs. Classes -- Creating & Using Objects



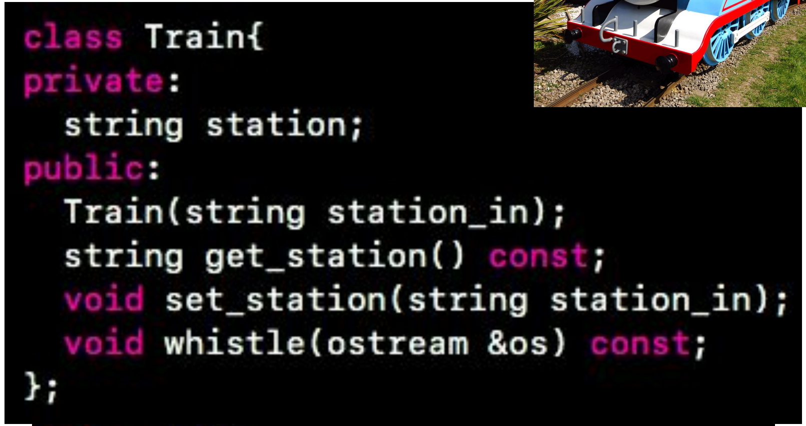


```
struct Train{
  string station;
};
void Train_init(Train *train, string station);
string Train_get_station(Train *train);
void Train_set_station(Train *train, string station)
void Train_whistle(ostream &os, Train *train);
```

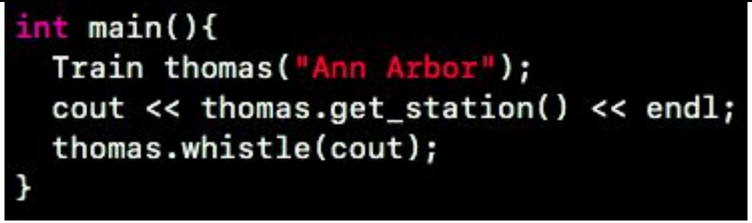


```
int main(){
  Train thomas;
  Train_init(&thomas, "Ann Arbor");
  cout << Train_get_station(&thomas) << endl;
  Train_whistle(cout, &thomas);
}
```

We have to create a new train object, and then initialize it with the init function. When we call the train functions, we have to pass in the train's address.



```
class Train{
private:
  string station;
public:
  Train(string station_in);
  string get_station() const;
  void set_station(string station_in);
  void whistle(ostream &os) const;
};
```

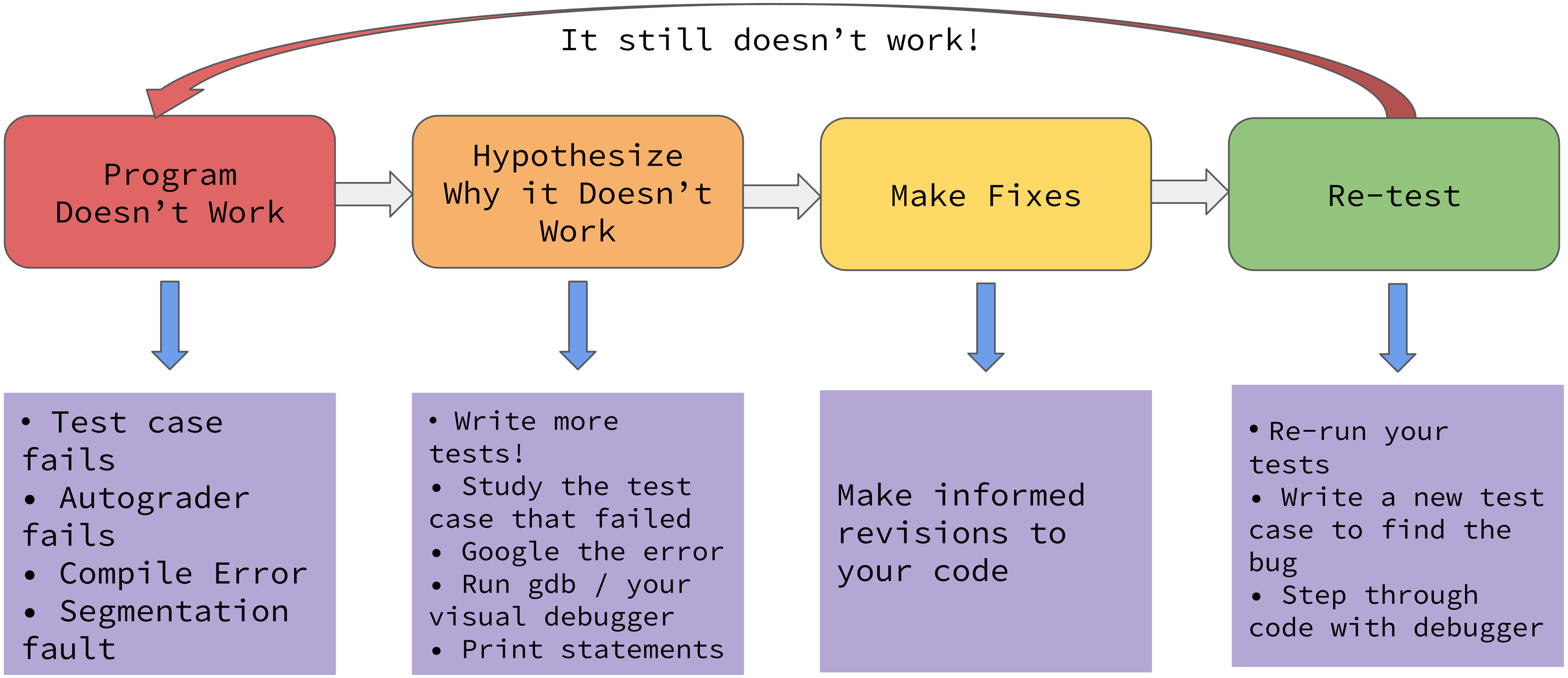


```
int main(){
  Train thomas("Ann Arbor");
  cout << thomas.get_station() << endl;
  thomas.whistle(cout);
}
```

We can create & initialize the train object all at once. We use the "dot" operator to call member functions. We do not have to pass in the address.

Worksheet!

The Essence of Debugging



- Test case fails
- Autograder fails
- Compile Error
- Segmentation fault

- Write more tests!
- Study the test case that failed
- Google the error
- Run gdb / your visual debugger
- Print statements

Make informed revisions to your code

- Re-run your tests
- Write a new test case to find the bug
- Step through code with debugger

Time for Lab Assignment!