ENGR 101 – Chapter 18

structs

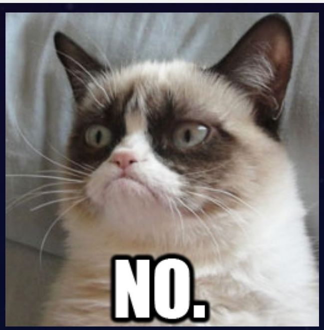
3/14/21

Modeling Real-World Objects in Code

- Let's model the autonomous rovers used to explore the dark side of Proxima b.

```
int rover1Type;
string rover1Id;
double rover1Charge;

int rover2Type;
string rover2Id;
double rover2Charge;

int rover3Type;
string rover3Id;
double rover3Charge;

...
```

Is this a good approach?



This proliferation of variables will quickly become unmanageable.

Using vectors

- Vectors can store sequences of objects, which helps a bit…

```
vector<int> roverTypes;
vector<string> roverIds;
vector<double> roverCharges;
```

We no longer need an arbitrarily large number of variables. The vectors just grow to accommodate new rovers.

- However, each attribute of the rovers' information is still stored as a separate variable. This makes code awkward.

```
double doSomethingWithRovers(vector<int> &types,
                             vector<string> &ids,
                             vector<double> &charges);
```

For example, in a function to work with rovers, we still have to pass all attributes separately!

A Rover Type

- Wouldn't it be great if C++ had a type for a rover?

```
// Create a Rover variable
Rover myRover;

// Access attributes of the rover using the dot
cout << myRover.charge << endl;

// Store several of them in a vector
vector<Rover> fleet;
```

- Of course, C++ doesn't have this type…

  …but we can create our own!

Defining structs

- A **struct definition** creates a new **compound type**.

It’s common to start custom type names with a capital letter.

Comments can add more description on the purpose and intended values of each member.

```
struct Rover {
  int type;       // either 1, 2, or 3
  string id;      // 4 alphanumeric characters
  double charge;  // % of charge, between 0 and 1
};
```

Don’t forget this semicolon!

These are **member declarations**. They indicate which kinds of pieces the `struct` is made from.

The struct definition goes at the top level of your code, not inside any function.

- Afterward, you can now declare variables of that type.

```
Rover rover; // creates a rover object
```

Name of type

Name of variable

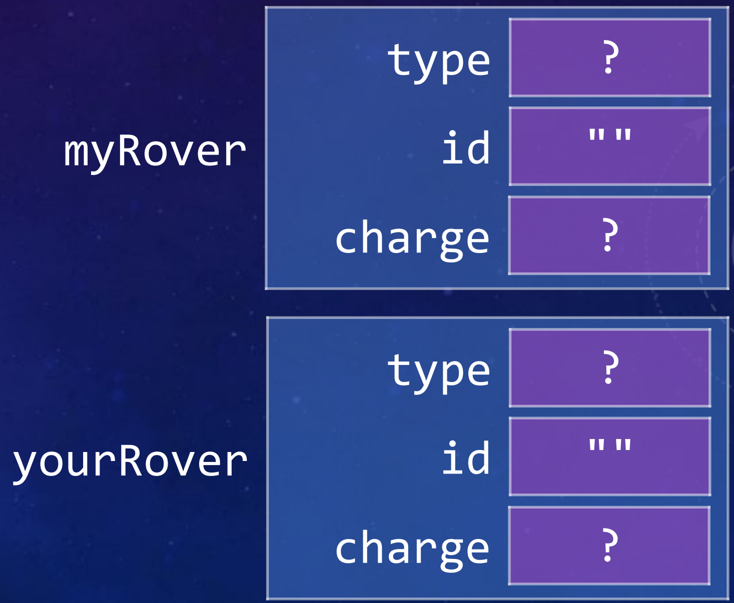
Member Variables and Memory

- `structs` are **compound** data types.
  - They are composed of several **member variables** of various types.

```
struct Rover {
  int type;
  string id;
  double charge;
};
```

- In memory, a compound object requires space to store each of its member variables.

```
Rover myRover;
Rover yourRover;
```

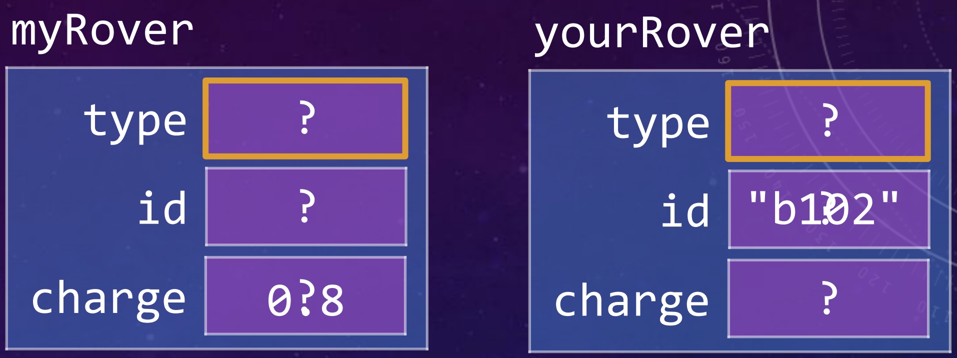


  - Member variables are not initialized by default.[1]

[1] `string` members are an exception and default to `""`.

Member Access with the Dot Operator

```
struct Rover {
  int type;
  string id;
  double charge;
};

Rover myRover;
Rover yourRover;
```



- Use the dot operator to access a member variable.
  - This allows working with a single piece of the overall `struct` object.

```
// Use an individual member as the target of an assignment
myRover.charge = 0.8;
yourRover.id = "b102";

// Read the values of members to use in expressions
if (myRover.type == yourRover.type) {
  ...
}
```

Initializing structs

- A special syntax can be used to initialize structs.
  - Specify an initial value for each member inside curly braces:

```
struct Rover {
  int type;
  string id;
  double charge;
};
```

```
Rover myRover = {1, "a238", 0.8};
Rover yourRover = {3, "b102",
0.37};

yourRover = {2, "b103", 0.9};
```

Error! This syntax can not be used for assignment later on. It only works on the same line as the declaration.[1]

myRover

type	1
id	"a238"
charge	0.8

yourRover

type	3
id	"b102"
charge	0.37

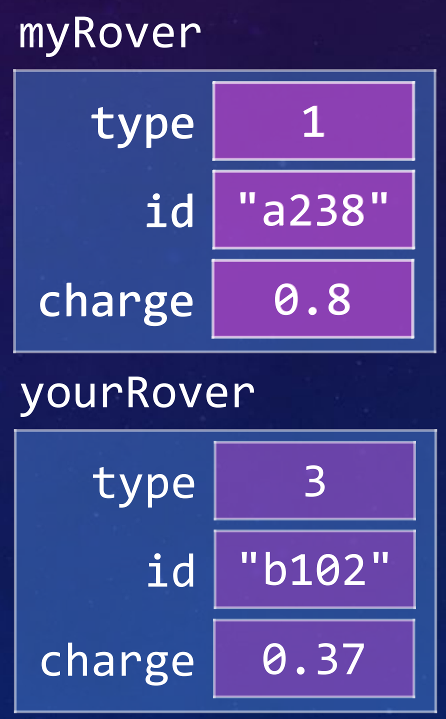
[1] Actually, it may or may not work, depending on the version of C++.

Copying structs

- Variables of the same `struct` type can copied to each other.
  - The built-in behavior is a straightforward **member-by-member** copy.

```
struct Rover {
  int type;
  string id;
  double charge;
};
```

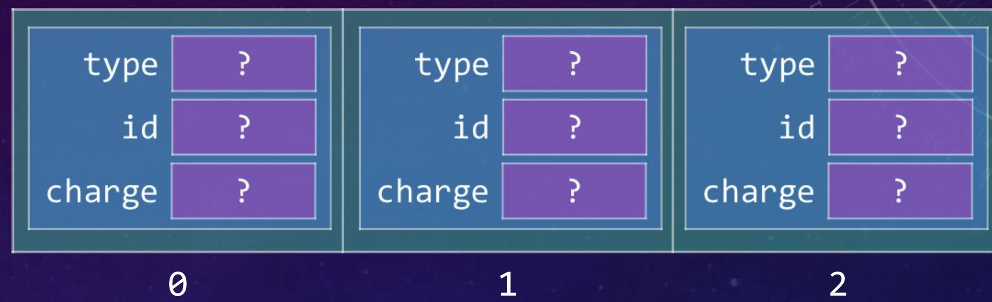
```
Rover myRover = {1, "a238", 0.8};
Rover yourRover = {3, "b102",
0.37};

yourRover = myRover;
```



vectors of structs

- Let's model the rovers with a vector of Rover structs.

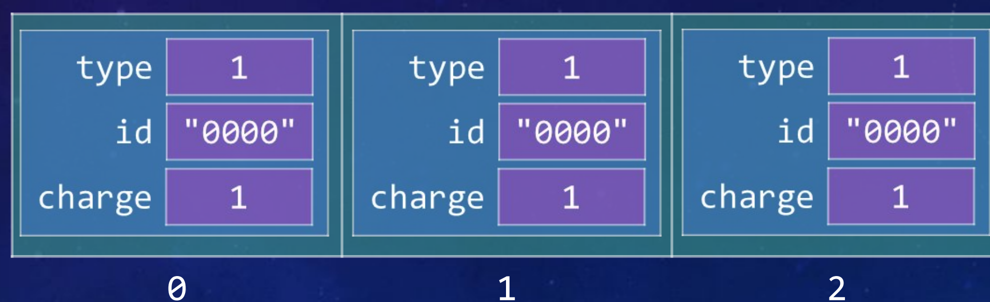
```
struct Rover {
  int type;
  string id;
  double charge;
};
```




```
vector<Rover> fleet(3);
```

- If there is a default value, you can initialize them like this:

```
struct Rover {
  int type;
  string id;
  double charge;
};
```




```
Rover defaultRover = {1, "0000", 1};
vector<Rover> fleet(3,
defaultRover);
```

vectors of structs

- We could also read information about the fleet of rovers from a file. Let's write a function to do this:

```
1 a283 0.6
2 a294 0.1
2 a110 0.5
3 b102 0.3
...
```

Recall: General File I/O Pattern

- Generally, it's good practice to keep input and output (I/O) processes separate from computation processes.
- Your program design should reflect this -- usually your functions will do *either* I/O *or* computation[1].
- A general pattern for file I/O follows the spellchecker example.

```
void loadWords(vector<string> &vec, istream &is) {
  string word;
  while (is >> word) {
    vec.push_back(word);
  }
}
```

```
int main() {
  vector<string> dictionary;
  ifstream fin("dictionary.txt");
  loadWords(dictionary, fin);
}
```

GENERAL PATTERN

1. The primary data structure (i.e. the `vector`) lives in `main`.
2. Open the file stream in `main`.
3. Pass the stream and data structure into a function by reference.
4. The function reads data from the stream into the data structure.

[1]Of course, there are always exceptions. ¯_(ツ)_/¯

vectors of structs

```
1 a283 0.6
2 a294 0.1
2 a110 0.5
3 b102 0.3
...
```

- We could also read information about the fleet of rovers from a file. Let's write a function to do this:

```
struct Rover {
  int type;
  string id;
  double charge;
};

void loadRovers(vector<Rover> &fleet, istream &is) {
  Rover rover;
  while(is >> rover.type >> rover.id >> rover.charge) {
    fleet.push_back(rover);
  }
}


int main() {
  vector<Rover> fleet;
  ifstream roversInput("rover_data.txt");
  loadRovers(fleet, roversInput);
  roversInput.close();
}
```

The struct definition needs to come first (and outside any functions, including `main`) or the compiler will complain.

Pass the vector by reference so we can fill it!

Use the dot expression here to specify the member as the target of the read operation.

The order of the read operations matches the order of information on each line of the input file.

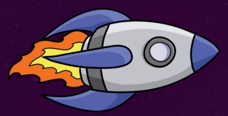
Printing a struct

- The built-in << operator won't work on our custom types[1].

```
struct Rover {
  int type;
  string id;
  double charge;
};

// Write the printRover function here



int main() {
  Rover myRover = {1, "a238", 0.8};
  cout << myRover << endl;
}
```

Error! The compiler doesn't know how to print out a Rover data type -- it only knows how to print basic types.

[1] You can specify behavior for << (and other operators) for custom types by defining special operator overload functions. Check out the online documentation if you're interested!



5 min

Exercise: Printing a struct

- The built-in << operator won’t work on our custom types.
- Instead, write a function to print out one of our Rovers to an output stream (e.g. cout or a file):

```
struct Rover {
  int type;
  string id;
  double charge;
};

// Write the printRover function here


int main() {
  Rover myRover = {1, "a238", 0.8};
  printRover(myRover, cout);
  cout << endl; // Make formatting nice
}
```

printRover.cpp

We've left the function header for you to write. Think carefully about what parameter and return types you need.

Solution: Printing a struct

```
struct Rover {
  int type;
  string id;
  double charge;
};

void printRover(const Rover &rover, ostream &output ) {

  output << "Type " << rover.type;
  output << " Rover #" << rover.id;
  output << " (" << (100 * rover.charge) << "%)";

}

int main() {
  Rover myRover = {1, "a238", 0.8};
  printRover(myRover, cout);
  cout << endl;
}
```

Pass by const reference for efficiency and safety.

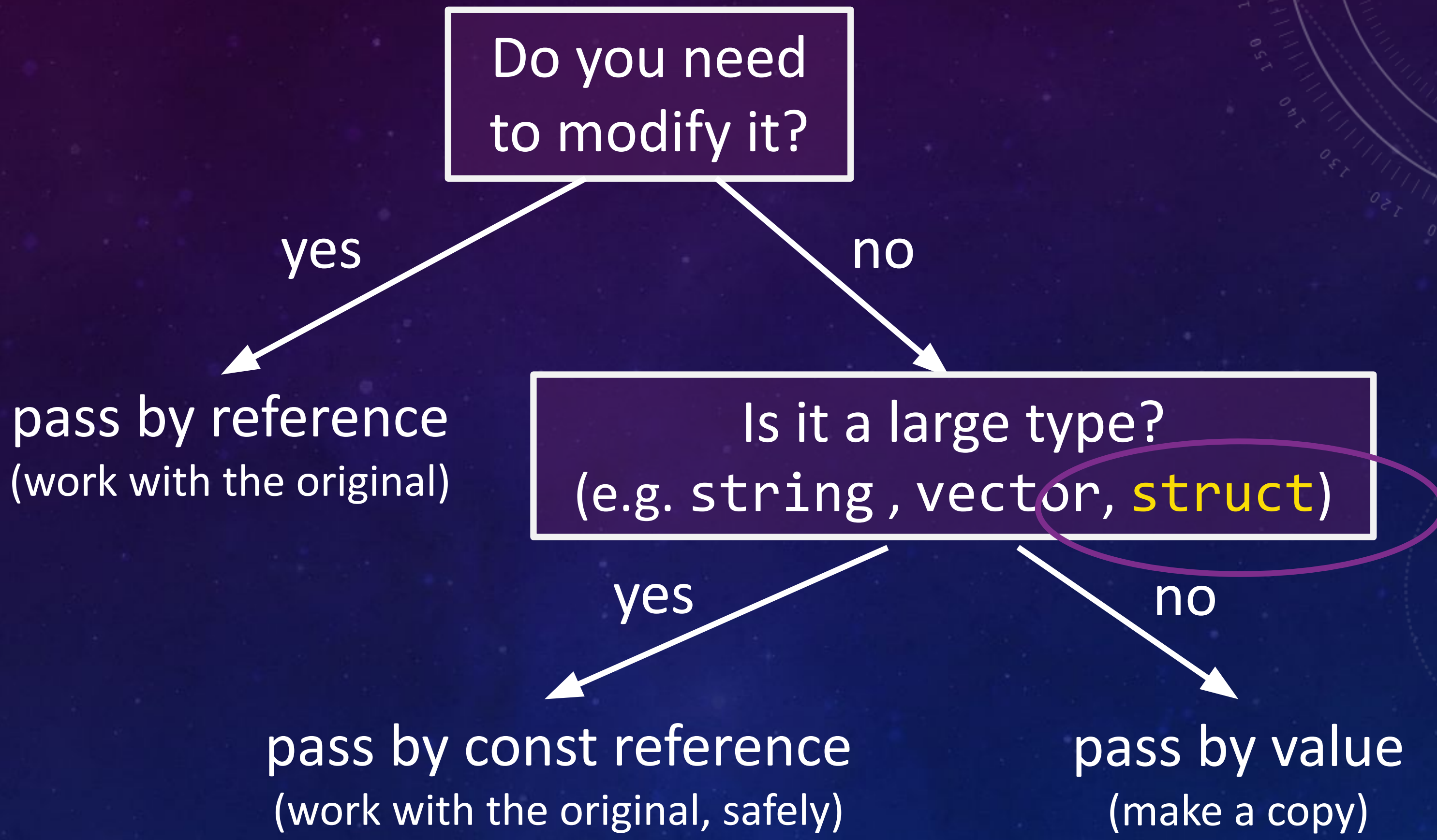
printRover.cpp

Careful not to confuse Rover (the type) with rover (the variable name)!

use `ostream` type so this function can print to any output stream (e.g. a file or cout)

```
Type 1 Rover #a238 (80%)
```

Recall: Parameter Passing



Common Compiler Errors with structs

```
struct Rover {
  int type;
  string id;
  double charge;
}

// Write the printRover function here
void printRover(const Rover &rover, ostream &output ) {
  output << "Type " << rover.type;
  output << " Rover #" << rover.ID;
  output << " (" << (100 * rover.charge) << "%)";
}

int main() {
  Rover myRover = {1, "a238", 0.8};
  int someInteger;
  cout << someInteger.charge;
}
```

Look directly above! The compiler got off track due to the missing semicolon.

Compiler gives mysterious error on this line…

“error: struct Rover has no member named 'ID'”

“error: request for member 'charge' in 'someInteger', which is of non-class type 'int'“

Selecting Rovers for a Mission

- Let's select a set of rovers to conduct a mission – we would like to collect soil samples from the dark side of the planet.
- First, we need to add some members to the `struct`:
  - The cargo capacity of each rover: an `int`
  - Whether or not the rover has been selected for the mission: a `bool`

```
struct Rover {
  int type;          // either 1, 2, or 3
  string id;         // 4 alphanumeric characters
  double charge;     // % of charge, between 0 and 1
  int capacity;      // cargo capacity in kilograms
  bool isSelected;   // has it been selected for the mission?
};
```

Updating the print Function

```
struct Rover {
  int type;
  string id;
  double charge;
  int capacity;
  bool isSelected;
};

// Write the printRover function here
void printRover(const Rover &rover, ostream &output ) {
  output << "Type " << rover.type;
  output << " Rover #" << rover.id;
  output << " (" << (100 * rover.charge) << "%) ";
  output << " carrying " << rover.capacity << "kg. ";
}

int main() {
  Rover myRover = {1, "a238", 0.8, 200, false};
  printRover(myRover, cout);
}
```

A nice feature of structs is that we can often modify their member variables without having to change the interface of functions that work with them. We still just pass in a Rover object here.

Add a statement to print the capacity of the rover.

Example: Loading Rover Data from a File

```
// Loads rovers from the specified file into the fleet
// vector. Each rover will have its type, id, capacity,
// and charge set according to the information in the file,
// and their isSelected member will also be set to false.
void loadRovers(vector<Rover> &rovers, istream &is) {
  Rover rover;
  rover.isSelected = false;
  while(is >> rover.type >> rover.id
          >> rover.capacity >> rover.charge) {
    rovers.push_back(rover);
  }
}

int main() {
  vector<Rover> fleet;
  ifstream roversInput("rover_data.txt");
  loadRovers(fleet, roversInput);
  roversInput.close();
}
```

Rover properties are grouped together in a struct, so we only need to pass one vector!

Assume the data is in some orderly format.

rover_data.txt

```
1 a238 200 0.6
1 a239 200 0.2
1 b102 200 0.4
2 a294 300 0.1
2 a110 300 0.5
2 a287 300 0.3
3 b102 400 0.3
3 c321 400 0.7
...
```

Selecting Rovers for a Mission

- Equipment:
  - A fleet of rovers, each at some % of full charge
  - A rover must be fully charged before departing on a mission.
  - A battery at the base camp that can provide 2 total “units of charge”
- Example:

rover	charge	charge needed (1-charge)
A	0.2	0.8
B	0.5	0.5
C	0.3	0.7
D	0.8	0.2

= 2.0 “units of charge”

so, we can take these 3 rovers on the mission, but not this one because our battery is out of “charge”

Selecting Rovers for a Mission

- Problem[1]: Find the set of rovers with the **greatest capacity**, subject to our charge constraint (maximum 2 “units” of charge).
- Idea: A rover with a high ratio of capacity vs. needed charge is best.
  - Let's wrap this up in a helper function.

```
double desirability(const Rover &rover) {
  return rover.capacity / (1 - rover.charge);
}
```

- Testing: Can you think of any test cases where this breaks?
  - It breaks when the charge is already 100% due to a divide by zero.
  - Let's fix this…

[1] This is a specific example of a “Knapsack Problem”

Selecting Rovers for a Mission

- Problem[1]: Find the set of rovers with the **greatest capacity**, subject to our charge constraint (maximum 2 “units” of charge).
- Idea: A rover with a high ratio of capacity vs. needed charge is best.
  - Let's wrap this up in a helper function.

```
double desirability(const Rover &rover) {
  // SPECIAL CASE
  if (rover.charge > 0.9) {
    return rover.capacity / 0.1;
  }

  // REGULAR CASE
  return rover.capacity / (1 - rover.charge);
}
```

Simplifying assumption: Any rover with > 0.9 charge is equivalent in terms of desirability for our decision.

This is a specific example of a “Knapsack Problem”