Today – Lecture #2

1. Topic #1 Slides
2. Designing using classes
3. Example Code

Reminders

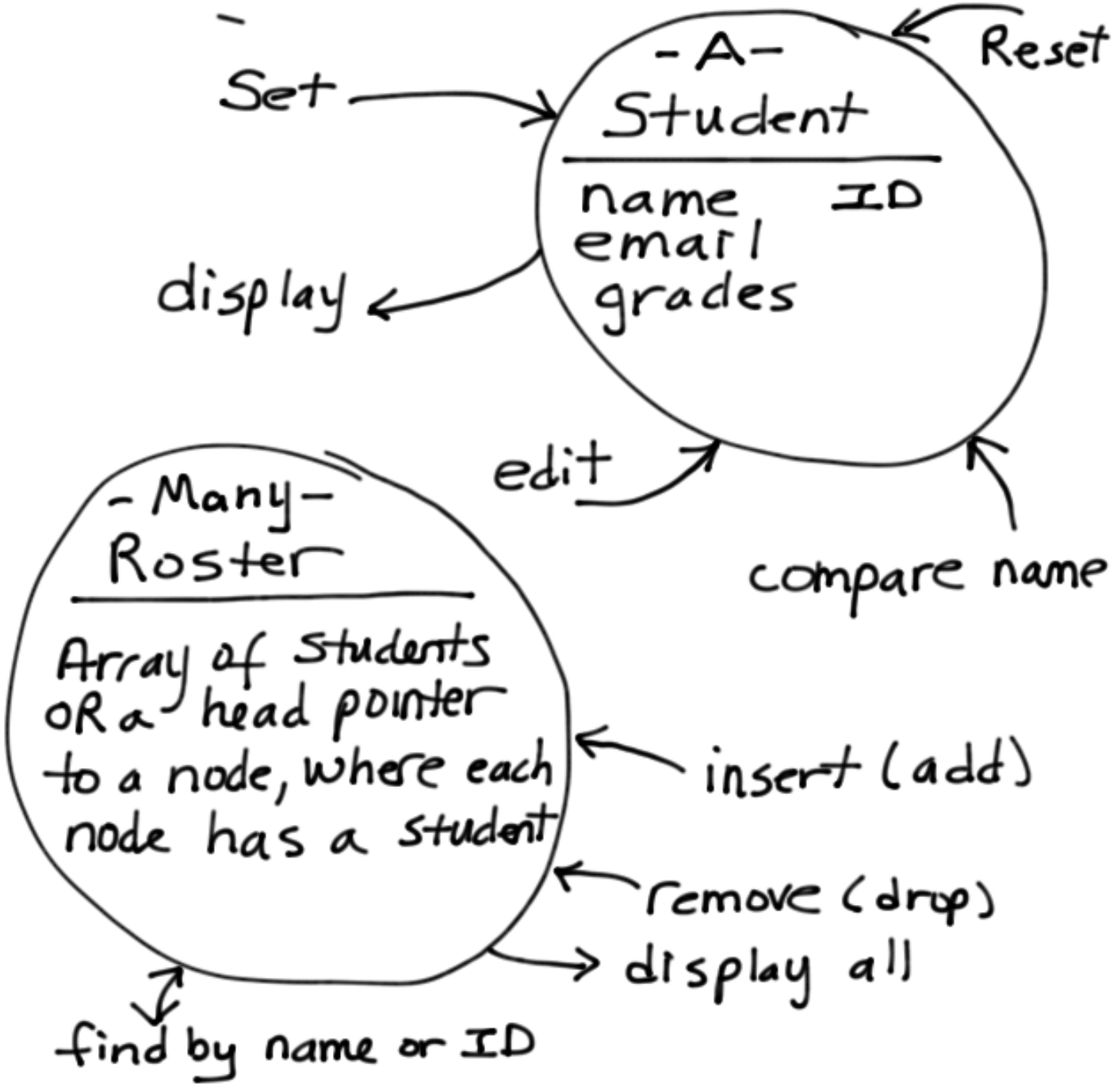
* Login to D2L at least twice a week
* Start designing your class(es) for program #1

* watch 2 lectures every week
* Time to use unix this term
  - NOT Dev C++ or other PC compilers

Rules for Programs

1. No global Variables
2. No Statically allocated arrays in your class ADT implementations
3. Always have at least 1 .h file and 2 .cpp files. Separate the ADT code from the application code
4. NEVER have an ADT:
   a. prompt
   b. read
   c. send out error messages
5. NEVER have "nodes" as arguments to public member functions
6. Use Arrays of characters — not the string class

Create a Student Roster ADT



Alternate Design

Struct
- Groups the name, ID, grades together

Add → Roster ← Drop

Find

display all

Now all member functions will manage not only the list of names but also the name, ID, grades for each student.

Sample .h code

```
class Roster
{ public:
    Roster();
    ~Roster();
    int add (student &);
    int reset (char ID [];
    int find (char ID [], student & found);
    int display();
    int drop(char ID [];
    //etc.
  private:
    node * head;
};
```

Avoid long argument lists

```
int add (char name[],
         char ID [],
         float grades [],
         char email [ ]);
```

Could be a struct or a class

So... if student was a struct...

Let's look at what add would be like:

```
int Roster::add(student &s)
{
  if(!head) //empty lis-
  {  head = new node;
     head->data.name =
         new char[strlen(s.name)+1];
     strcpy(head->data.name, s.name);
     head->data.ID = new char[strlen(s.ID)+1];
     strcpy(head->data.ID, s.ID);
     // and on and on for the
     // email and all grades
```

Why?

```
struct node
{  student data;
   node *next;
};
```

And

```
struct student
{  char name[41];
   char ID[10];
   char email[131];
   float grades[10];
};
```

NO WORK IS being done. These just group data together

But, if Student was a class... with member functions managing handling the data:

```
int Roster:: add (student & s)
{   if (!head)
    {
        head = new node;
        head->data.set(s);
        //Done!
```

```
int student :: set (student &s)
{  name = new char [strlen(s.name)+1];
   strcpy(name, s.name);
   email = new char [strlen(s.email)+1];
   strcpy (email, s.email);
   //and so on. Much easier
   //to add more fields without
   // affecting the entire list.
```

Reminders

When to use → vs • when accessing members:

object • member

Direct Member Access Operator

object of a class or struct

pointer → member

Indirect member access operator

pointer to a class object or struct

head → data . name

Student object

pointer to a node struct

Application

1. not graded
2. can correspond with user
3. Passes info to the class (ADT) member functions
4. Doesnt use complex data structures

ADT

1. THIS is graded
2. No correspondence with the USER
3. MUST supply success/failure information back to the application
4. Manages the data structures so the application doesn't need to
5. Allows the Application to create many instances of the ADT, as needed with minimal overhead

4/5

Sample Application

```
Roster cs162, cs163, cs202;
```

Three objects of the ADT class are created

```
Student a_student;
a_student.set(name, ID);
```

already prompt and read in the data

```
cs163.add(a_student);
```

```
cs163.display();
```

Just displays cs163 roster not cs202 or others

OR,

```
Roster classes[12];
```

my rosters for all year!

```
classes[i].add(~~);
```