As you arrive:

1. Start up your computer and plug it in
2. ***Log into Angel*** and go to CSSE 120
3. Do the ***Attendance Widget*** – the PIN is on the board
4. Go to the course ***Schedule Page***
5. Open the ***Slides*** for today if you wish
6. Check out today's project: `Session24_Geometry`

Plus in-class time working on these concepts AND practicing previous concepts, continued as homework.

Structures, Preamble

- Declaring structure types
- Using structure types
- Using typedef, #Define

C Modules

- Defining and using header files
- Function prototypes/signatures
- Multiple .c files

Preamble: #define and typedef

- C allows us to define our own constants and type names to help make code more readable

```
#define TERMS 3
#define FALL 0
#define WINTER 1
#define SPRING 2

typedef int coinValue;
coinValue quarter = 25, dime = 10;
```

For more on these topics:

typedef: Kochan, p. 325-327, or
www.cprogramming.com/tutorial/typedef.html

#define: Kochan, p. 299-303, or
www.cprogramming.com/tutorial/cpreprocessor.html

- How could we make our own *Boolean* type? Answer:

```
typedef int boolean
#define TRUE 1
#define FALSE 0
```

Q1-3

Structures

- No objects or dictionaries in C. Structures (structs) are the closest thing that C has to offer.
- Two ways of grouping data in C:
  - **Array**: group several data elements of the **same type.**
    - Access individual elements by *position* : `student[i]`
  - **Structure**: group of related data
    - Data in struct may be of different types
    - Conceptually like dictionaries, syntax like objects
    - Access individual elements by *name*: `student.gpa`
      - Not `student["gpa"]`

Structure variable, where the structure has a field called *gpa*

Q4-5

struct syntax

- struct <optional_tag_name> {
  <type_1> <fieldname_1> ;
  <type_2> <fieldname_2> ;
  . . .
  <type_n> <fieldname_n> ;
  };
- This says that each variable of this **struct** type has all these fields, with the specified types
- But structs are best declared in conjunction with **typedef**, as on on next slide…

Example: Student struct type

- Declare the type:

```
typedef struct {
    int year;
    double gpa;
} Student;
```

There are other ways to declare structure types, but this is by far the *best* way. Follow its notation carefully.

Note that it just declares the **Student type**. It does NOT make a Student or Student variable.

- Make and print a student's info:

```
Student s;

s.gpa = 3.4;
s.year = 2010;
printf("Year %d  GPA %4.2f\n", s.year, s.gpa);
```

Declares ***s*** to be of type ***Student*** and allocates space (an ***int*** and a ***double***) for ***s***.

Initializes the fields of ***s***.

Accesses the fields of ***s***. Note the dot notation for assignment and access.

Q6

Define a **Point** struct type together

- Make a new C Project called ***PointModule***
  - **File ~ New ~ C Project,** then choose **Hello World ANSI C Project**
- Expand the *PointModule* project and find the **PointModule.c** file beneath the **src** folder. Rename this **PointModule.c** file to **main.c**
  - (it will help avoid confusion later)
- Within **main.c** create a **typedef** for a **Point** structure
  - After the **#include's**, but before the definition of *main*
  - Two fields, named **x** and **y**
  - Make both **x** and **y** have type ***int***
  - Follow the pattern from the previous slide, but do a **Point** structure (not a Student).

```
typedef struct {
    int year;
    double gpa;
} Student;
```

Declare, initialize and access a Point variable

- In **main:**
  - Delete the line the wizard included that prints "Hello World"
  - Delete the *void* the wizard put in `int main(void)`
  - Declare a variable of type **Point**
  - Initialize its two fields to (say) 3 and 4
  - Print its two fields

Follow the pattern we saw on a previous slide:

```
Student s;

s.gpa = 3.4;
s.year = 2010;

printf("Year %d  GPA %4.2f\n", s.year, s.gpa);
```

That's a struct

- That's an easy introduction to using *typedef* with *struct*
- Let's make some fancier ways to initialize a *struct*

Three ways to initialize a struct variable

```
typedef struct {
    int year;
    double gpa;
} Student;
```

#1

```
Student juan;
juan.year = 2008;
juan.gpa = 3.2;
```

#2

```
Student juan = {2008, 3.2};
```

(Only allowed when declaring and initializing variable together in a single statement. Not recommended, since if the order of the fields changes, this statement breaks.)

#3

```
Student makeStudent(int year, double gpa) {
    Student student;
    student.year = year;
    student.gpa = gpa;
    return student;
}

Student juan = makeStudent(2008, 3.2);
```

Define a function that constructs a Student and returns it

Call the constructor, in *main* or elsewhere

makePoint

- Write a **`makePoint`** function:

  `Point makePoint(int xx, int yy)`

  It receives two **`int`** parameters and returns a **`Point`**

- From within the **`main`** function:
  - Declare a **`Point`** called (say) **`myPoint2`**
  - Call **`makePoint`** and store the result into **`myPoint2`**
  - Print the values of the returned Point's two fields (x and y)

```
Student makeStudent(int year, double gpa) {
    Student student;
    student.year = year;
    student.gpa = gpa;
    return student;
}

Student juan = makeStudent(2008, 3.2);
```

Follow the pattern #3 from the previous slide, repeated here

C Modules

- Grouping code into separate files for the purposes of organization, reusability, and extensibility
- **Header files**
  - .h file extension
  - Typically, .c files will #include your header file
  - For publicly available functions, types, #defines, etc.
- Source files
  - **.c file extension**
  - The actual C code implementations of functions, etc.
  - Needs to #include .h files to use functions that are not written in this file

Making Modules in C

- The **.c** and **.h** file with the same name are called collectively a **module**
- Our example:
  - PointOperations.c
  - PointOperations.h
- Let's create this module together in Eclipse
  - Right-click ***src*** folder, then **New → Header File**
    - Call the file **PointOperations.h**
  - Right-click ***src*** folder, then **New → Source file**
    - Call the file **PointOperations.c**

Move your code

- Publicly available content goes into **.h** files
- Private content and code implementations go into **.c** files
- Move into **PointOperations.h**
  - The code that defines the `Point` structure
  - The prototype for `makePoint`
    - Put these (and all other code in the .h file) *between* the `#ifndef` and `#endif`:

```
#ifndef POINTOPERATIONS_H_
#define POINTOPERATIONS_H_
        YOUR STUFF HERE
#endif /* POINTOPERATIONS_H_ */
```

The compiler automatically knows that the implementation of the function is within the .c file of this module.

Any .c file that has `#include "PointOperations.h"` can now call that function – it's publicly available.

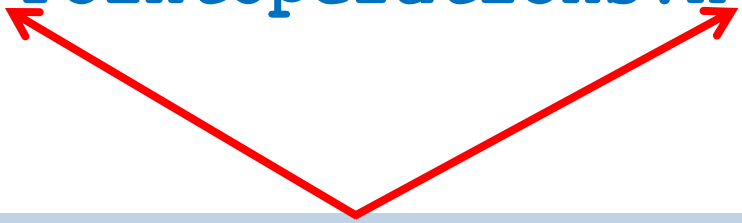
- Move into **PointOperations.c**
  - The `makePoint` function definition

Q7-8

Adding the wiring

- **main.c** and **PointOperations.c** need to know about **PointOperations.h**
  - Both need the *Point* structure definition
  - main needs the prototype for *makePoint*
- Add #include's into both files, like this:

```
#include "PointOperations.h"
```



Note the double quotes, not angle brackets as we have been using.

Angle brackets tell the compiler to look in the place where system files are kept. Double quotes tell the compiler to look in our project itself.

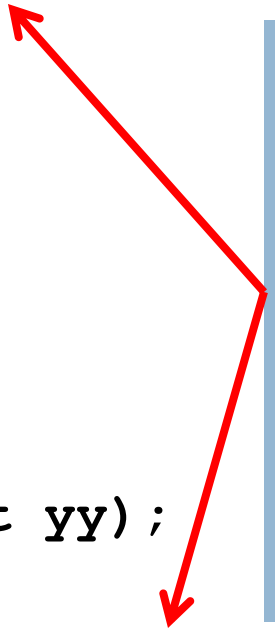
Summary – PointOperations.h

```
#ifndef POINTOPERATIONS_H_
#define POINTOPERATIONS_H_

typedef struct {
    int x;
    int y;
} Point;

Point makePoint(int xx, int yy);

#endif /* POINTOPERATIONS_H_ */
```



This “include guard” ensures that the code in this file is processed only ONCE, even if many .c files #include it. Put an include guard in all your .h files, as a matter of standard practice.

Summary – PointOperations.c

```
#include "PointOperations.h"

Point makePoint(int xx, int yy) {
  Point result;

  result.x = xx;
  result.y = yy;

  return result;
}
```

Summary – main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "PointOperations.h"

int main(void) {
  Point myPoint = makePoint(3,5);
  printf("myPoint.x = %i  myPoint.y = %i\n",
         myPoint.x,
         myPoint.y);

  return EXIT_SUCCESS;
}
```

Try it out

- Save all 3 files, build (**Project → Build Project**) and run
  - Ctrl Shift S, Ctrl B, Ctrl F11 (some keyboard short cuts)
- Works exactly like it did before but using modules!
  - Refactoring code always feels a little odd
  - So much effort for no visible difference
  - A modular approach is much more extensible
    - In software engineering, extensibility is a system design principle where the implementation takes into consideration future growth.

Extended in class example

- Next we're going to do an extented example using structs, typedef, and modules
- If you get stuck during any part, RAISE YOUR HAND and get a TA to help you stay caught up
- There will be a bunch of parts, so getting behind early works out BADLY
- Make sure each works before moving on
- Raise your hand if you have trouble with weird build errors (it happens!)

Geometry Operations

- To make sure everyone is together checkout the project **Session24_Geometry**
- Look at the code and try running the program
- If at ANY point you get a '*Binaries not found*' error
  - File ~ Save All [it will be grayed out if all is already saved]
  - Project ~ Clean [cleans the project and rebuilds it]
  - Examine your console window – if errors remain, fix them
  - Run (Ctrl F11) to run code

The Goal

- Sit back and we'll talk about what this code WILL do
- Look in the Tasks window for TODO instructions
  - Close other projects so that their TODOs don't show up
  - For example, close the That's Perfect project

Files

- Testing your modules code
  - main.c
- Point Operations module
  - PointOperations.h
  - PointOperations.c
- Line Segment Operations module
  - LineSegmentOperations.h
  - LineSegmentOperations.c

Main

- Used to test your modules
- Things it already does
  - tests Point operations:
    - Creates a Point (using `makePoint`)
    - Gets a Point from the console (using `getPointFromConsole`)
    - Prints the Points (using `printPoint`)
    - Call a `calculateDistance` function and prints the returned distance
- Things you'll add
  - Test code for LineSegment operations
    - After you define a LineSegment structure and write functions that operate on it

Two Modules

- Functions in the **PointOperations** module:

```
Point makePoint(int newX, int newY);
double calculateDistance(Point point1, Point point2);
void printPoint(Point point);
Point getPointFromConsole();
```

- Functions in the **LineSegmentOperations** module:

```
LineSegment makeLineSegment(Point oneEndPoint,
                            Point otherEndPoint);
void printLineSegment(LineSegment line);
double calculateLength(LineSegment line);
```

Implement LineSegmentOperations.h

- For this .h file, you need (as usual):
  - Structure definitions relevant to functions of this module
  - Prototypes of functions defined in this module
  - #include statements as needed for the prototypes
- Finish your quiz, then do the TODO's in *Session24_Geometry* project
  - Do them in order: 0, 1, 2, …
  - They are SCATTERED throughout the files. After TODO 0, begin in **`LineSegmentOperations.h`**

File ~ Save All
Project ~ Clean

Often helpful! Fix errors as you proceed!

Examine your console window – if errors remain, fix them
Run (Ctrl F11) to run code

A C Program in Multiple Files

- You should have checked out the ***Session24_RectangleStructs*** project from SVN.
- A large program can be organized by separating it into multiple files.
- Notice the three source files:
  - **rectangle.h** contains the struct definitions and function signatures used by the other files.
  - **rectangle.c** contains the definitions of the functions that comprise operations on point and Rectangle objects.
  - **Session24_RectangleStructs.c** contains a main function to test the various functions of the rectangle module.
- Both of the **.c** files must include the **.h** file.