

STRUCTS, TYPEDEF, #DEFINE, AND USING C MODULES

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
```

For more info, see Kochan,
p. 299-303 (#define),
p. 325-327 (typedef)

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

How could we make our own boolean type?

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]**
 - Similar to Python's list, but more low-level
 - ▣ **Structure:** group of related data
 - Data in structure may be of different types
 - Conceptually like dictionaries, syntax like objects
 - Access individual elements by *name*: **endPoint.x**
 - Not endPoint["X"]

Structure variable,
where the structure
has a field called x

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;
```

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

```
s.gpa = 3.4;
```


```
s.year = 2010;
```

Initializes the fields of *s*.

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

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

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***
 - ▣ Include a prototype for it, as usual
- From within the **main** function:
 - ▣ Declare a Point called *p2*
 - ▣ Call **makePoint**
 - ▣ Store the result into *p2*
 - ▣ Print the values of *p2*'s **x** and **y**

Follow the pattern
#3 from the previous
slide.

Solution (try it on your own first)

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


Point makePoint(int xx, int yy);

int main() {
    Point p, p2;

    p.x = 3;
    p.y = 4;
    printf("%i %i\n", p.x, p.y);

    p2 = makePoint(8, 5);
    printf("%i %i\n", p2.x, p2.y);

    return EXIT_SUCCESS;
}
```



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

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

    return result;
}
```

C Modules

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

Making Modules

- 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

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.

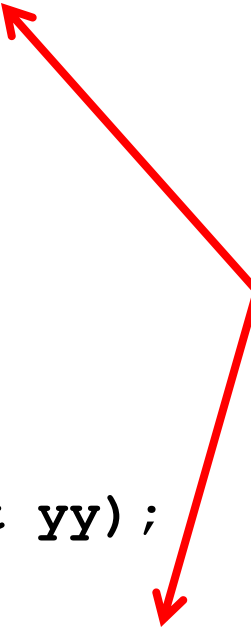
Summary – PointOperations.h

```
#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() {
```

```
    Point p, p2;
```

```
    p.x = 3;
```

```
    p.y = 4;
```

```
    printf("%i %i\n", p.x, p.y);
```

```
}
```

```
p2 = makePoint(8, 5);
```

```
printf("%i %i\n", p2.x, p2.y);
```

```
return EXIT_SUCCESS;
```



Try it out

- Save all 3 files, build (**Project → Build Project**) and run
- 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 extended example using structures 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!)
- In this example, you will implement a LineSegment structure
 - ▣ Do the remaining quiz questions now

Structures1 – Geometry Operations

- Checkout the project

25-Structures1

- It does NOT compile yet (you'll fix that shortly)
- Overview of this exercise:
 - ▣ We give you *main* and a *PointOperations* module that can do operations on points
 - Similar to the one you just developed
 - ▣ You develop a *LineSegmentOperations* module that can do operations on line segments
 - ▣ We'll begin together (via the next several slides), then you will do the remaining TODO's at your own pace

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
 - Gets a Point from the console
 - Prints the Points
 - Call a distance function
 - Prints the 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 xx, int yy);
```

```
void printPoint(Point point);
```

```
double calculateDistance(Point pt1, Point pt2);
```

- Functions in the **LineSegmentOperations** module:

```
LineSegment makeLineSegment(Point pt1, Point pt2);
```

```
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
- Do TODO's #1, 2 and 3 in **LineSegmentOperations.h**:

// TODO #1: Put the typedef for a structure type called LineSegment

// TODO #2: Put prototypes for the three functions that you will implement.

// See their descriptions in the comment at the top of this file.

// TODO #3: Put the #include into this file that you need, as a result
// of the makeLineSegment function that you added a prototype for.

Fix the fields in LineSegmentOperations.c

- Do TODO #4: In the **printLineSegment** function in **LineSegmentOperations.c**, change the field names to the names that YOU used in YOUR LineSegment structure type.

Run the project

- Save, compile and run the project.
 - ▣ It will tell you the distance between a point that you input and (0, 0), but it gets the answer wrong (you'll fix it)
 - ▣ You enter your point's coordinates as two numbers with a comma in between, as suggested by the prompt
 - ▣ If you have compile errors, something went wrong in the preceding steps: be quick to get help as needed.
 - ▣ If you have trouble running the code, try the usual tricks:
 - Save the file (perhaps introducing a trivial change like a space first)
 - Select the project and do **Project → Build Project**
 - Sometimes a **Project → Clean** helps
 - If you have a **Release** folder in your project, get help getting rid of it
- Then do the remaining TODO's, in order.

TODO #5 – implement the *calculateDistance* function

- Work through the TODO's, in numbered order
 - ▣ TODOs #1, then #2, etc
 - ▣ The next slides offer help, but move ahead at your own pace
- For TODO #4 in **pointOperations.c**, notice that the “stub” that we supplied for **calculateDistance** always returns 0.0
 - ▣ $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- Remember **math.h**?
- For practice try to use `pow` (even though less efficient)

```
double pow(double x, double y);  
// x raised to power y
```

From a list of C's standard libraries at:

<http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html>

TODO #6 & 7: Implement and test makeLineSegment

- TODO #6: Implement makeLineSegment in LineSegmentOperations.c
- TODO #7: Uncomment the lines in main that test your makeLineSegment (and the existing printLineSegment) functions.
 - ▣ You will also need to add an appropriate #include, since now *main* uses the LineSegment module.
 - ▣ Run the code to see that it works correctly so far. Fix errors as needed.

TODO #8 and 9: Implement and test the `calculateLength` function

- TODO #8: Implement **`calculateLength`** in **`LineSegmentOperations.c`**
 - ▣ IMPORTANT: Use **`calculateDistance`** from your `PointOperations` module!
- TODO #9: Uncomment the lines in `main` that test your `makeLineSegment` (and the existing `printLineSegment`) functions.
 - ▣ Run the code to confirm that it works correctly. Fix errors as needed.
- That's it for 25-Structures1 – good job!
- Rest of class: Checkout and begin working on
25-Structures2
 - ▣ Read the description in `homework25` before doing this exercise!
 - ▣ Finish for homework