STRUCTS, TYPEDEF, #DEFINE, AND USING C MODULES

CSSE 120—Rose Hulman Institute of Technology

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

How could we make our own bool 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]**
  - **Structure**: group of related data
    - Data in struct may be of different types
    - Conceptually like dictionaries, syntax like objects
    - Access individual elements by *name*: **endPoint.x**
      - Not endPoint["X"]

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;
```

- Make and print a student's info:

```
Student myStudent;
myStudent.gpa = 3.4;
myStudent.year = 2010;)
printf("[%s  %d  %4.2lf]\n",s.year,s.gpa);
```

Hands on working together

- Let's define a **Point** struct type together
- Make a new C Project called "PointModule"
  - (Hello World ANSI C Project)
  - Rename file PointModule.c to main.c
    - (it will help avoid confusion later)
  - Within **main.c** create a **typedef** for a Point struct
    - Two fields, named **x** and **y**
    - Make both **x** and **y** have type int
    - See code on next slide

Together let's make a Point type

Type this in after the #includes but before main

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

Together let's make a Point

Type this in within main

```
int main(void) {
  Point myPoint;
  myPoint.x = 3;
  myPoint.y = 4;
  printf("myPoint.x = %d  myPoint.y = %d\n"
      ,myPoint.x,myPoint.y);
  return EXIT_SUCCESS;
}
```

That's a struct

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

Initializing a struct

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

Shorter:

Student juan = {2008, 3.2};

(Only allowed when declaring and initializing variable together in a single statement.)

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

```
typedef struct {
        char *name;
        int year;
        double gpa;
} Student;
```

makePoint

- Write code for makePoint:
  - `Point makePoint(int xx, int yy)`
  - It receives two int parameters and returns a Point

- From within the **`main`** function:
  - Call **`makePoint`**
  - Store the result into a new Point called myPoint2
  - **`print`** the values of **`x`** and **`y`**

[Hidden] Solution

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

Point makePoint(int xx, int yy) {
        Point result;
        result.x = xx;
        result.y = yy;
        return result;
}

int main(void) {
  Point myPoint2 = makePoint(3,5);
  printf("myPoint2.x = %d  myPoint2.y = %d\n",myPoint2.x,myPoint2.y);
  return EXIT_SUCCESS;
}
```

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 publically 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 → New → Header File
    - Call the file PointOperations.h
  - Right-click src folder → New → Source file
    - Call the file PointOperations.c

Move your code

- Next we need to move our code
- Publicly available content goes into **.h** files
- Private content and code implementations go into **.c** files
- Move into **PointOperations.h**
  - The `typedef struct` code
- Move into **PointOperations.c**
  - The `makePoint` function

Adding the wiring

- **main.c** and **PointOperations.c** need to know about **PointOperations.h**
- Add #includes into both files, like this:
  - **#include "PointOperations.h"**

Function prototypes in the .h

- Additionally **main.c** needs to know about the **makePoint** function (currently only in private **.c** file)
- Add this function prototype to **PointOperations.h**
  - Point **makePoint(int xx, int yy);**

- The compiler automatically knows that the implementation of the function is within the .c file of this module
- Any .c file that #includes "PointOperations.h" can now call that function (it's publically available)

PointOperations.h

```
#ifndef POINTOPERATIONS_H_
#define POINTOPERATIONS_H_

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

Point makePoint(int xx, int yy);

#endif /* POINTOPERATIONS_H_ */
```

PointOperations.c

```
#include "PointOperations.h"

Point makePoint(int xx, int yy)
{
  Point result;
  result.x = xx;
  result.y = yy;
  return result;
}
```

main.c

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

int main(void) {
  Point myPoint = makePoint(3,5);
  printf("myPoint.x = %d  myPoint.y =
  %d\n",myPoint.x,myPoint.y);
  return EXIT_SUCCESS;
}
```

Try it out

- Save all 3 files, build and run
  - Ctrl Shift S, Ctrl B, Ctrl F11
- 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 **Session23GeometryOperations**
- Look at the code and try running the program
- Good trick, if you get a 'Binaries not found' error
  - Make a small change to main.c (like adding a space)
  - Save main.c (Ctrl S) to mark it as needing to be rebuilt
  - Build (Ctrl B) to build program
  - Run (Ctrl F11) to run code
    - Sometimes I need to do that cycle TWICE
      - Seems to make things happy assuming I have no code errors

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
  - 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 Line Segment Operations (after you write those functions)

PointOperations Module

- Functions in this module:

```
Point makePoint(int xx, int yy);
void printPoint(Point currentPoint);
double calculateDistance(Point pt1, Point pt2);
```

LineSegmentOperations Module

- Functions in this module:

```
LineSegment makeLineSegment(Point pt1, Point pt2);
void printLineSegment(LineSegment currentLine);
double calculateLength(LineSegment currentLine);
```

Calculate distance function

- TODOs #1 & #2
- Notice that **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)
  - http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html

Continued expansion

- Implementing the LineSegmentOperations module
  - LineSegmentOperations.h
  - LineSegmentOperations.c
- TODO #3
  - For a line segment, what should the fields be?
  - Do the quiz question.
  - Then create a new LineSegment variable type

Add more struct types

- TODOs #4 thru #7
- Add a makeLineSegment function
  - Receives two Points returns the LineSegment
  - From main call this function to make a LineSegment
- Add a printLineSegment function
  - Code provided but uses MY field names
  - From main call printLineSegment to print your line

Calculate the line segment length

- TODOs #8 thru #10
- Write a **calculateLength** function for a line segment.
  - Hint: Can you call the distance function we already wrote to **avoid** copy & paste?

Get started

- The rest of the time is your time to finish the 10 TODO's
- Ask questions as you need help
- If you finish early, checkout and start reading HW23 RectangleStructs
- Go ahead!

A C Program in Multiple Files

- Check out ***Session23RectangleStructs*** 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.
  - **Session23RectangleStructs.c** contains a main function to test the various functions of the rectangle module.
- Both of the **.c** files must include the **.h** file.

Add functions for homework

```
/* Makes a rectangle from the given coordinates.  A
 * rectangle is  made up of two points. */
Rectangle makeRect(int x1, int y1, int x2, int y2);

/* Returns the x-coordinate of the left-most edge given
 * of the rectangle. */
int getLeft(Rectangle r);

/* Returns the x-coordinate of the right-most edge of the
 * given rectangle. */
int getRight(Rectangle r);

/* Returns the y-coordinate of the top-most edge given
 * of the rectangle. */
int getTop(Rectangle r);
```

Add additional functions

```
/* Returns the y-coordinate of the bottom-most edge of the
 * given rectangle. */
int getBottom(Rectangle r);

/* Returns TRUE if the given rectangles touch and FALSE if
 * not they do touch. */
boolean areIntersecting(Rectangle q, Rectangle r);

/* Returns a new rectangle representing the overlapping
 * area of  the two given rectangles.  */
Rectangle intersect(Rectangle q, Rectangle r);
```