Questions about the first half of the semester?

Questions about HW3 or HW4
Recap from Lab 6

- Code blocks
- Global variable scoping
- Two dimensional arrays
  - arrays of strings
- Debugging

Recap from Lab 7

- Writing a README and comments
- Function prototypes (but I am still not sure everyone gets it)
- Preprocessors
  - #include
  - #define
- Bit Operators
- Debugging

More on preprocessors

- #ifdef
  - Allows for code to be compiled if symbol is not defined.
  - #ifdef DEBUG
  - printf("This is production code");
  - #endif
- #else
  - basically does the same thing
  - #ifdef DEBUG
  - printf("This is test code");
  - #else DEBUG
  - printf("This is production code");
  - #endif

- You can use these techniques to debug as well as write regular code
  - Helps in commenting
  - /* lots of code */
More on preprocessors

- You can use these techniques to debug as well as write regular code
  - Helps in commenting
    - "***** I want to comment this testing section
    - section_report();
    - " Handle the end of section stuff */
    - dump_table();
    - " end of commented out section */
- What is wrong with this code?
- You can fix it by writing
  #ifdef DEBUG
  section_report();
  /* handle the end of section stuff */
  dump_table();
  #endif

Structs

- Used to define your own types
  struct structure-name {
    field-type field-name;
    field-type field-name;
    ....
  }
  variable-name;

Structs II

- So an example would be
  struct bin {
    char name [30]; // name of the part
    int quantity; // how many in the bin
    int cost; // the cost of the single part
  } printer_cable_bin; // where we put the cables

- Here printer_cable_bin is a variable of type struct bin
- You can omit the variable name
Structs III

- The dot operator
  - In order to access one of the fields of the struct, for a particular variable, use the form `variable.field`
  - `eg: printer_cable_bin.cost = 1295;`
  - `eg: total_cost = printer_cable_bin.cost * printer_cable_bin.quantity`

Structs IV

- I said earlier that you don't have to define variables when defining the struct.
  - So can I do, later in the code — `int printer_cables_bin;` (i.e. just like I use int or char)
  - Answer: No

- How to do it correctly
  
  ```c
  struct bin printer_cables_bin; // This doesn't define any of the values inside of bin, therefore those remain undefined
  ```

- So you can either assign them one at a time or you can do the following
  ```c
  struct bin printer_cable_bin = {
  "Printer Cables",
  0,
  1295
  }; // However, this notation can only be used at the time of declaration
  ```
- Note the semicolons and the commas

Structs V

- (Shortcut) Initializing values — `struct bin {
  char name [30]; // name of the part
  int quantity;   // how many in the bin
  int cost;      // the cost of the single part
} printer_cable_bin = {
  "Printer Cables",
  0,
  1295
};`

- Note the commas and the semicolon
Structs VI

- Structs typically go outside all methods.
- You can have them inside methods but then those are local only to the method, this is NOT RECOMMENDED.

```c
#include<stdio.h>

int main(void) {
    struct a {
        int b;
        double c;
    };
    struct a suhit; /* = { 6 , 7.213432 };*/
    suhit.b = 5;
    suhit.c = 3.2;
    printf("%d\n", suhit.b);
    printf("%f\n", suhit.c);
    return 0;
}
```

Unions

- There are like structs, however they have only one memory space.

```c
union structure-name {
    field-type field-name;
    field-type field-name;
    ....
} variable-name;
```

Unions II

```c
struct bin {
    char name [30]; // name of the part
    int quantity; // how many in the bin
    double cost; // the cost of the single part
    } printer_cable_bin; // where we put the cables

union bin {
    char name [30]; // name of the part
    int quantity; // how many in the bin
    double cost; // the cost of the single part
    } printer_cable_bin; // where we put the cables
```

- Make space for largest variable
Unions III

- You can overwrite quantities, in union
  printer_cables_bin.name = "Printer Cables"
  printer_cables_bin.cost = 10;
  printf("The name of the bin is %s\n",
         printer_cables_bin.name);
  - What will the produce?
  - Answer: Unexpected result
  - You must keep track of which field you used

- So why use this?
  - Memory space saving

Typedefs

- Struct allows you to create a data type/structure
- Typedefs allow the programmer to define their own variable type

Typedefs II

- Usage
  - typedef type-declaration;
  - where type-declaration is the same as variable declaration, except that a type name is used instead of a variable name
  - eg: typedef int count; //creates a new type count that is the same as an integer
  - Now you can say – count a; //equal to int a;
### Typedefs III

- But you can get more complex
  - typedef int group[10];
  - You can now say group classroom, which will create a variable classroom of 10 integers

```c
main()
{
    typedef int group[10];
    group class;
    for (i=1; i<10; i++)
        class[i] = 0;
    return 0;
}
```

### Typedefs IV

- But you can get more complex
  - typedef struct bin bin
  - This creates a variable type bin of type `struct bin`, and you can now say `bin` printer_cables_bin, instead of `struct bin` printer_cables_bin

```c
struct bin {
    char name[30];
    int quantity;
    int cost;
};
typedef struct bin bin;
bin printer_cables_bin = {"Printer Cables", 10, 1290};
```

### Enums

- This is designed for variables that contain only a limited set of values
- Traditionally, if you wanted to set up the days of a week, you would:

```c
typedef int week day;
const int Sunday = 0;
const int Monday = 1;
const int Tuesday = 2;
const int Wednesday = 3;
const int Thursday = 4;
const int Friday = 5;
const int Saturday = 6;

week day today = Tuesday;
```
Enums II

- That was cumbersome
- You can say
  
  ```
  enum week_day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
  ```

  ```
  enum week_day today = Tuesday;
  ```

- Usage
  
  ```
  enum enum-name (tag-1, tag-2, ...) variable-name;
  ```

Enums III

- You can omit variable-name, like in struct and union
- C implements the enum type as compatible with integer, so it is legal to say
  
  ```
  today = 5; // though this may throw a warning
  // will make today Thursday
  ```

Enums IV – more examples

```
enum week_day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum day d1, d2; // makes d1 and d2 of type
// enum day

d1=Friday;
if (d1==d2)
...
```
 Enums V – more examples

- You can use it to do switches

```c
enum week_day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
day find_next_day(day d) {
    day next_day;
    switch(d) {
        case Sunday:
            next_day = Monday;
            break;
        case Monday:
            next_day = Tuesday;
            break;
        case Tuesday:
            next_day = Wednesday;
            break;
        case Wednesday:
            next_day = Thursday;
            break;
        case Thursday:
            next_day = Friday;
            break;
        case Friday:
            next_day = Saturday;
            break;
        case Saturday:
            next_day = Sunday;
            break;
    }
    return next_day;
}
```

 Arrays of Structs

```c
struct time {
    int hour;
    int minute;
    int second;
};
const int MAX_LAPS = 4;
struct time lap[MAX_LAPS];
lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

 Arrays of Structs II

- Another way of initializing

```c
struct time start_stop[2] = {
    {10, 0, 0},
    {12, 0, 0}
};
```
Structs with arrays

```c
struct mailing {
    char name[60];
    char address1[60];
    char address2[60];
    char city[40];
    char state[2];
    long int zip;
};

struct mailing list[MAX_ENTRIES];
list[count].name[0]=S;
```

Casting

- `(type) expression`
- You already know this
  ```c```
  ```
  int a;
  float b, total;
  total = (float)a + b;
  ```

Assignment

- Read Ch. 12 from the Practical C Programming book
- Start reading Ch. 13 for next class
- This class is going to get hard (pointers and memory allocation)
- HW4
  - Don’t wait too long