Structures & Enumerations

- So far, we've been unable to create our own data types.
- We've been able to use built-in types, and create arrays of built-in types
- Based on what we know now, how would you do the following:
  - Create an employee management system. Each employee has a first name, last name, employee ID, social security number, and salary.
  - Probably would create an array for each attribute, and share an index for each employee.
    * E.g. the 10th person is index 9 across every array
    * This is bad design!
- What if we could create an *employee* that had a first name, last name, employee ID, social security number, and salary?
  - Then, we would only need one array, where each element of the array is a complete employee
- Structs enable this grouping of basic types to form a more complex type.
- We can make our own types!

Structs

- A struct is a data structure that contains multiple pieces of data.
- We define structs using the `struct` keyword:

```
struct employee{
  char first_name[100];
  char last_name[100];
  int employee_id;
  int ssn;
  float salary;
};
```

- That's it!
- How do we use a struct?
- Instantiate a struct with:

```
struct employee mark;
```

- How we change values?
  - Access part of a struct using the `.` operator:

```
mark.ssn = 0123456789;
```

- An example usage:

```c
#include <stdio.h>
#include <string.h>

struct employee{
  char first_name[100];
  char last_name[100];
  int employee_id;
  int ssn;
  float salary;
};

void print_employee(struct employee e);

int main(){
  struct employee mark;
  strcpy(mark.first_name, "Mark");
  strcpy(mark.last_name, "Edmonds");
  mark.employee_id = 31358;
  mark.ssn = 1234567890;
  mark.salary = 1000000;
  print_employee(mark);
}

void print_employee(struct employee e){
  printf("%s, %s. ID: %d, SSN: %d, Salary: $%.2f\n", e.last_name, e.
      first_name, e.employee_id, e.ssn, e.salary);
}
```

- It is common practice to **typedef** your structure so you can create instances without the `struct` keyword (we have done this with every other type thus far):

```c
typedef struct {
  char first_name[100];
  char last_name[100];
  int employee_id;
  int ssn;
  float salary;
} Employee;
```

- Now we can create instances like this:

```
Employee mark; // equivalent to our old struct employee mark before
```

- Everything else stays the same (except we replaced `struct employee` with `Employee` in all declarations/usages)!

Struct initialization

- We can initialize each member of a struct using an initializer list (like what we did for an array.
- For example, we could replace `main` above with the following:

```
struct employee mark = { "Mark", "Edmonds", 31358, 1234567890,
    1000000}; // order here matters! corresponds to order of variables
    in struct.
print_employee(mark);
```

Pointers to structs

- We can have a pointer to a struct as well. For instance, we could write:

```
struct employee mark;
struct employee *mark_ptr = &mark;
```

- This is the same logic as before with pointers! Nothing special here, even if it looks odd.
- Accessing pointers works the same way as before, but we also have a shortcut:

```
(*mark_ptr).ssn = 1234567890; // the . operator has higher precedence
    than the * operator, so we need the parentheses
mark_ptr->ssn = 1234567890; // the same as above, but nicer notation.
    The -> operator dereferences and accesses the corresponding member
```

- That's all there is to structs. Just a useful way to group data to make code more readable, reliable, and maintainable.

Enumerations

- Mappings between labels and integers
- Enumerations are not composed of any data types only labels.
- Example enum to represent colors

```
enum color {
    red,
```

```
    orange,
    yellow,
    green,
    cyan,
    blue,
    purple,
};
```

- Under the hood, red will be assign the value `0`, orange `1`.
- How do you use them?

```
enum color value = red;
if(value == green){
  printf("We should never execute this statement\n");
}
```

- We can also use them in switch statements conveniently:

```
#include <stdio.h>
#include <string.h>

enum color {
  red,
  orange,
  yellow,
  green,
  cyan,
  blue,
  purple,
};

int main(){
  enum color value = red;
  switch(value){
    case red:
      printf("color is red\n");
      break;
    case orange:
      printf("color is orange\n");
      break;
    case yellow:
      printf("color is yellow\n");
      break;
```

```
26      case green:
27        printf("color is green\n");
28        break;
29      case cyan:
30        printf("color is cyan\n");
31        break;
32      case blue:
33        printf("color is blue\n");
34        break;
35      case purple:
36        printf("color is purple\n");
37        break;
38    }
39  }
```

- Example usage: suppose we wanted to compare against the day of the week. We could use string comparisons for everything, but this is clunky, hard to read and string comparison is computationally expensive.
  - Because enumerations are really just labeled integers, we can make code more readable by using them!

Header files

- Header files contain C declarations and macros.
- Header files are included into your source code using `#include` preprocessor directive.
  - We have been including system-level header files with `#include <stdio.h>`, etc.
- When we write our own header files, we will use quotes instead of <> to surround the filename
  - E.g. `#include "myheader.h"`

Why bother?

- When you say `#include`, the preprocessor fetches the corresponding header file and literally copies its contents on the same line as the `#include` (thereby replacing the `#include` statement with many lines of code)
- Header files end with the extension `.h`
- This means we don't have to manually copy the contents of a header when we want to include functionality written by someone else. This would be very error-prone and prevents updating the code base in one step
  - If everyone is using the same header, and that header is updated, programs that use that header will be updated when they are recompiled

What should you put in a header?

- Headers should only contain declarations, not implementations
- This means headers can contain:
  1. Function prototypes: `int sum(int a, int b);`
  2. Structure/enumeration declarations: `struct p {int x; int y;};`
  3. Macros/Defines: `#ifndef HEADER #define HEADER #endif`

What should you NOT put in a header?

- Headers should not contain any implementation of any sort, only declarations
- This means headers should not contain:
  1. Function definitions/implementations: `int sum(int a, int b){ return a + b; }`
- Why?
  - Suppose we have function definitions in a header, let's call it `myheader.h`, what will be linker see if `source1.c` and `source2.c` both `#include "myheader.h"`?
    * The same function definition twice! This means it won't know which one to actually execute when the function name is called
  - A general rule: multiple declarations is fine for the linker, but multiple definitions is not.

Example:

- Suppose we have `myheader.h` with the following

```
1 char *test();
```

- Suppose we have `main.c` with the following:

```
1 #include "header.h"
2
3 int main (void) {
4    puts (test ());
5 }
```

- The preprocessor will copy the contents of `myheader.h` and place them into `main.c`

```
1 int x;
2 char *test (void);
3
4 int main (void) {
5    puts (test ());
```

```
}
```

Header guards

- Header guards protect against including the same header multiple times. This means the contents will copied twice, which will result in a compiler error (for using the same symbol twice).
- We can easily guard against this with the following scheme:

```
#ifndef MY_HEADER_NAME   // if MY_HEADER_NAME is not defined
#define MY_HEADER_NAME   // define MY_HEADER_NAME

// header contents

#endif // end the if
```

Employee Example

`main.c`

```
#include <stdio.h>
#include <string.h>
#include "employee.h"

int main(){
  size_t num_employees = 3;
  size_t len;
  struct employee employees[num_employees];
  float sum_salary = 0.0;

  for (int i = 0; i < num_employees; i++){
    printf("Employee %d input\n", i+1);
    printf("First name: ");
    fgets(employees[i].first_name, 100, stdin);
    // replace newline from fgets with null character
    len = strlen(employees[i].first_name);
    employees[i].first_name[len-1] = '\0';

    printf("Last name: ");
    fgets(employees[i].last_name, 100, stdin);
    len = strlen(employees[i].last_name);
    employees[i].last_name[len-1] = '\0';
```

```
23
24      printf("Employee ID: ");
25      scanf("%d", &employees[i].employee_id);
26      printf("Social Security Number: ");
27      scanf("%d", &employees[i].ssn);
28      printf("Salary: ");
29      scanf("%f", &employees[i].salary);
30
31      // clear buffer from scanf (prepare for next employee)
32      char c;
33      while((c = getchar()) != '\n' && c != EOF) { }
34    }
35
36    for(int i = 0; i < num_employees; i++){
37      print_employee(employees[i]);
38      sum_salary += employees[i].salary;
39    }
40
41    printf("The company needs $%.2f to pay the employees\n", sum_salary);
42  }
```

employee.h

```
1  #ifndef employee_h
2  #define employee_h
3
4  struct employee{
5    char first_name[100];
6    char last_name[100];
7    int employee_id;
8    int ssn;
9    float salary;
10 };
11
12 void print_employee(struct employee e);
13
14 #endif
```

employee.c

```c
#include <stdio.h>
#include "employee.h"

void print_employee(struct employee e){
  printf("%s, %s. ID: %d, SSN: %d, Salary: $%.2f\n", e.last_name, e.
      first_name, e.employee_id, e.ssn, e.salary);
}
```

- `employee.c` is the *implementation file* for things related to the employee struct
- `employee.h` is the *header file* for all declarations related to the employee struct
- `main.c` is the main file, which is the bulk of the program

Exercises

1. What is wrong with the following C declarations?
    1. `struct point ( double x, y )`
    2. `struct point { double x, double y };`
    3. `struct point { double x; double y }`
    4. `struct point { double x; double y; };`
    5. `struct point { double x; double y; }`
2. What is the difference among the following three programs?

Program 1

```c
#include <stdio.h>
struct point { double x; double y; };
int main(void) {
    struct point test;
    test.x = .25; test.y = .75;
    printf("[%f %f]\n", test.x, test.y);
    return 0;
}
```

Program 2

```c
#include <stdio.h>
typedef struct { double x; double y; } Point;
int main(void) {
    Point test;
    test.x = .25; test.y = .75;
    printf("[%f %f]\n", test.x, test.y);
    return 0;
```

```
}
```

Program 3

```c
#include <stdio.h>
typedef struct { double x; double y; } Point;
int main(void) {
    Point test = {.25, .75};
    printf("[%f %f]\n", test.x, test.y);
    return 0;
}
```

3. Write a program that uses the employee structure above to get 5 employee's data from a user and print the results. The sum of the salaries should be computed as well, as a record for the company's total salary expenditure.

```c
#include <stdio.h>
#include <string.h>

typedef struct{
  char first_name[100];
  char last_name[100];
  int employee_id;
  int ssn;
  float salary;
} Employee;

void print_employee(Employee e);

int main(){
  const size_t n_employees = 5;
  Employee employee_list[n_employees];
  float sum_salary = 0.0;
  for(int i = 0; i < n_employees; i++){
    printf("Employee %d input\n", i+1);
    printf("First name: ");
    fgets(employee_list[i].first_name, 100, stdin);
    // fgets will place the newline, so we need to manually remove it
    size_t len = strlen(employee_list[i].first_name);
    if (employee_list[i].first_name[len-1] == '\n'){
      employee_list[i].first_name[len-1] = '\0';
    }
    printf("Last name: ");
```

```
28      fgets(employee_list[i].last_name, 100, stdin);
29      len = strlen(employee_list[i].last_name);
30      if (employee_list[i].last_name[len-1] == '\n'){
31        employee_list[i].last_name[len-1] = '\0';
32      }
33      printf("Employee ID: ");
34      scanf("%d", &employee_list[i].employee_id);
35      printf("Social Security Number: ");
36      scanf("%d", &employee_list[i].ssn);
37      printf("Salary: ");
38      scanf("%f", &employee_list[i].salary);
39      // clear buffer from scanf (prepare for next employee)
40      char c;
41      while((c = getchar()) != '\n' && c != EOF) { }
42    }
43    for(int i = 0; i < n_employees; i++){
44      sum_salary += employee_list[i].salary;
45      print_employee(employee_list[i]);
46    }
47    printf("The company will need $%.2f to pay these employees\n",
          sum_salary);
48  }
49
50  void print_employee(Employee e){
51    printf("%s, %s. ID: %d, SSN: %d, Salary: $%.2f\n", e.last_name, e.
          first_name, e.employee_id, e.ssn, e.salary);
52  }
```