CMSC 313
Introduction to Computer Systems
Lecture 6
Structures and Unions, cont. & Input/Output

Alan Sussman

als@cs.umd.edu

Administrivia

- Project 1 questions?
  - due tomorrow, 6PM
- Read Reek, Chapter 10: Structures and Unions, and Chapter 15: Input/Output

Chapter 10, Reek

STRUCTURES AND UNIONS

Structure storage

- How much space does a structure use in memory?

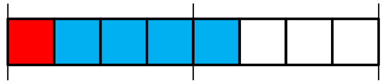
```
struct one {
  double b;
  int a;
} s1;

printf("%d\n", sizeof(s1));
```

- Assuming **int**s use 4 bytes and **double**s 8 bytes each, prints "**12**"

Structure storage, cont.

- But due to alignment issues, things aren't always that simple:

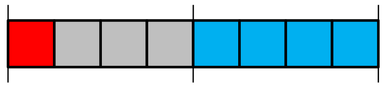
```
struct two {
  char a;
  int b;
} s2;
```



Wrong - blue **int** isn't properly aligned (and whatever comes after this won't be, either)

- **int**s must begin at 4-byte boundaries, so s2 must be 8 bytes, not 5.

Right - blue **int** is properly aligned, and gray bytes in memory are left unused



- To minimize unused space, order fields from longest to shortest

Assigning and comparing structures

- Each field is copied for an assignment

```
struct ex_struct a, b;
...
a = b;
```

- Is **a == b** true now? Two issues:
  - **a** and **b** are still separate objects in memory
  - can't just compare bits - what if there's unused space?
- Because it doesn't make sense to do these types of comparisons, the **==** won't compile

Structure initialization

- Much like array initialization
- The items listed in the initializer are assigned to the fields in order
- Zeroes used to fill uninitialized fields when an initializer is used
- Example:

```
typedef struct {
  int i;
  char ch;
  double d;
} Ex_struct;
Ex_struct a = {4, 's', 3.5};
Ex_struct b = {5, 'g'};
```


Nested structure example

```
/* a Section contains a number like 0101, and
 * how many students are enrolled */
typedef struct {
  int number;
  int num_students;
  int start_time;
} Section;

/* a Course contains a number like 313,
 * and two Sections */
typedef struct {
  int course_number;
  Section section1, section2;
} Course;

Section s = {101, 30, 1300};
Course c = {213, {}, {201, 30, 1200}};
...
c.section1 = s;                /* referring to a whole Section */
c.section2.num_students = 29;  /* referring to one field of a
                                  Section in the Course */
```

Structures and functions

- Structure arguments are passed by value
- We can return structures from functions

```
Section add_students(Section sec, int students_to_add) {
  Section new_section = sec;
  new_section.num_students += students_to_add;
  return new_section;
}
Section s = { 0101, 10, 1400 }, t;
...
t = add_students(s, 26);
```


Aside: parameters are variables, too!

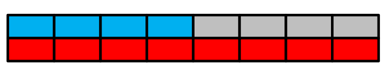
- Because arguments are passed and returned by value, you can use the parameters as variables:

```
Section add_students(Section sec, int students_to_add) {
  sec.num_students += students_to_add;
  return sec;
}
Section s = { 0101, 10, 1400 }, t;
...
t = add_students(s, 26);
```


Unions

- Look much like structures
- But all fields share the same memory space, so are only as large as largest field
- Only one field valid at a time

```
typedef union {
  int i;
  double d;
} Number;
...
Number a, b;
a.i = 2;
b.d = 3.14159;
printf(“%d\n”, b.i);
```



blue bytes are used when accessed as an int, red bytes used when accessed as a double

Making unions more useful

- using an enum and struct along with the union can help keep track of which field is in use

```
typedef struct {
  enum { INT, DOUBLE } type;
  union {
    int i;
    double d;
  } value;
} Better_number;

void print_number(Better_number num) {
  switch (num.type) {
    case INT:    printf(“%d”, num.value.i);
                 break;
    case DOUBLE: printf(“%f”, num.value.d);
                 break;
    default:     printf(“?????”);
                 break;
  }
}
```

Section 13.4, Reek

Command Line Arguments

Command line arguments

- When executing a command like "**`ls -l`**", or "**`emacs puzzles.c`**", the various parts of the command line are called arguments to the program
- We can access these by including parameters to the **`main()`** function
- These parameters are represented as strings (similar to the **`String[] args`** you'd use in Java's **`main()`** method).

Accessing command line arguments

- This program (named **`prog`**) will print out each argument, one per line
  - Note: the type "**`char *`**" is also used to represent strings in C - we'll learn why soon

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++)
    printf("Arg #%d: %s\n", i, argv[i]);
  return 0;
}
```

- What if we execute "**`./prog -l 53 -c -d`**"?

Accessing command line args, cont.

- If we execute "**`./prog -l 53 -c -d`**"?
  - Output is:

```
Arg #0: ./prog
Arg #1: -l
Arg #2: 53
Arg #3: -c
Arg #4: -d
```

Chapter 15, Reek

INPUT/OUTPUT

Error reporting

`void perror(const char *message);`

- prints a description of the most recent error to have occurred (in a system call and some library calls), along with the message you provide (format: "*message*: error desc.\n")
- For example, say we have an error opening a file; the call **`perror("Can't open filename.txt")`** could result in:

  `Can't open filename.txt: No such file or directory`
- System knows what error occurred by setting the global variable **`errno`** (defined in **`errno.h`**)

Terminating execution

`void exit(int status);`

- prototype in **`stdlib.h`**
- "immediately" ends execution when called
- status is viewable by the shell
  - **`exit(0);`** generally means OK
  - **`exit(1);`** or any nonzero generally means error encountered
  - can use "**`echo $?`**" in tcsh to see the exit status of the last program executed
- can use constants **`EXIT_SUCCESS`** and **`EXIT_FAILURE`** instead of 0/1

Standard I/O Library

- **`stdio.h`** contains prototypes and constants for I/O routines
- Most I/O is stream-based and buffered to make things more efficient
  - line buffered
  - fully buffered
  - unbuffered
- Can use the function **`fflush()`** to flush buffers immediately

Types of streams

- Text streams are composed of lines of text, each terminated by a newline
  - there are library functions to deal with text streams in three ways:
    - character-oriented I/O
    - line-oriented I/O
    - formatted I/O
- Binary streams are composed of just plain data

The type **`FILE *`**

- Variables of type **`FILE *`** are used to represent open streams
- Three predefined streams for every program:
  - **`stdin`**: standard input (redirect with **<**)
  - **`stdout`**: standard output (redirect with **>**)
  - **`stderr`**: standard error (redirect both stdout and stderr with **>&** -- in tcsh only!)

Stream I/O overview

1. Declare a **`FILE *`** variable for the file
2. Use **`fopen()`** to open the file
3. Read or write (or both!)
4. Use **`fclose()`** when done

Opening files

```
FILE *fopen(char *filename, char *mode);
```

- arguments are strings
- **`mode`** specifies access mode:
  - **`"r"`** - read; file must already exist
  - **`"w"`** - write; if file exists, it is overwritten
  - **`"a"`** - append; if file does not exist, it's created
  - there are other modes …
- returns **`NULL`** on failure; can use **`perror()`** to see why it failed

Opening files, example

```
#include <stdio.h>

int main() {
  FILE *fp = fopen("infile.txt", "r");
  if (fp == NULL) {
    perror("can't open infile.txt");
    exit(EXIT_FAILURE);
  }
  /* read the file and close when done */
  return EXIT_SUCCESS;
}
```


Closing files

`int fclose(FILE *fp);`

- closes a file, flushing output if necessary
- returns 0 on success
- Failure does occur - the closing operation can be interrupted by the OS, or other, worse things

Line-oriented I/O

```
char *fgets(char *buf, int size,
            FILE *stream);
```

- reads chars from **`stream`**, storing in **`buf`**, stopping when either
  - a newline is read; or
  - **`size`** - 1 characters are read
- null byte is always appended to string
- **`NULL`** returned on error or **`EOF`**
- on success (no error and non-**`EOF`**), returns **`buf`**