CS 240: Programming in C

Lecture 7: Arrays, Memory Layout of Data

PURDUE UNIVERSITY

Announcements

- Homework 2 due tonight!
- Work on homework 3

Homework 3

hw3.h

```
extern int g_mission_count;
extern char g_astronaut_and_mission[MAX_MISSIONS][3][MAX_NAME_LEN];
extern char g_equipment[MAX_MISSIONS][2][MAX_NAME_LEN];
extern float g_op_hours[MAX_MISSIONS][2];
extern char g_mission_dates[MAX_MISSIONS][2][MAX_DATE_LEN];
extern int g_experiments[MAX_MISSIONS][2];
```

- extern is a *declaration*
  - It tells the compiler what the variable looks like, but it does not allocate space for it!
  - You still must define it somewhere!

Arrays of structures

- We can create arrays of structures just as we can create arrays of anything else

```
struct person people[4];
```

- Initialization is similar to before:

```
struct person people[4] = {
  { “Kirk”,  “Captain”,                {10, 20, 25, 9} },
  { “Spock”, “First Officer”,          {-1,  2,  5, 8} },
  { “McCoy”, “Chief Medical Officer”, {94, 33,  2, 8} },
  { “Scott”, “Chief Engineer”,         { 7,  7,  2, 8} },
};
```

Arrays of structures example

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

struct person {
  char name[40];
  char title[25];
  int  codes[4];
};

struct person crew[200]; /* global! */

void print_person(struct person);
```

Arrays of structures example

```
int main() {
  int index = 0;

  strncpy(crew[0].name,  “Kirk”, 40);
  strncpy(crew[0].title, “Captain”, 25);
  crew[0].codes[0] = 10;
  crew[0].codes[1] = 20;
  crew[0].codes[2] = 40;

  strncpy(crew[1].name,  “Ensign”, 40);
  strncpy(crew[1].title, “Redshirt”, 25);
  crew[1].codes[0] = 1;
```

Arrays of structures example

```
  int index;
  for (index = 0; index < 200; index++) {
    if (crew[index].name[0] != '\0') {
      print_person(crew[index]);
    }
  }
  return 0;
}

/* Assume that print_person is defined
 * below.
 */
```

Arrays of structures example

```
$ vi ex2.c
$ gcc -Wall -Werror -std=c99 -g -o ex2 ex2.c
$ ./ex2
Name:  Kirk
Title: Captain
Codes: 10, 20, 40, 0

Name:  Ensign
Title: Redshirt
Codes: 1, 0, 0, 0

$
```

Notes about previous example

- When you define something as a global data structure, anything that is not initialized is automatically made zero
  - Sometimes this is good, sometimes not
- We only defined the first two elements of the big array
- You can check if the first character of a string is NUL by:

```
if (string[0] == '\0') ...
```

Array initialization

- You can partially initialize an array!

```
int my_numbers[200] = { 5, 5, 3, 4, 5 };
```

- Only the first five elements are explicitly initialized. The rest are set to zero
- This is true not only for global arrays but for arrays allocated inside functions as well

Array auto-sizing

- You can define and initialize an array without explicitly saying what its size is

```
int my_array[] = { 1, 7, 0, 1 };
```

- This array would be size 4
- There are no zero elements at the end of the array since we're letting the compiler figure out how large it is

Arrays of structures

- Same idea...

```
struct point {
  int x;
  int y;
};

int almost_pointless() {
  struct point dots[] = { {1, 2},
                          {3, 4} };
  return dots[1].x;
}
```

strncpy()

- What's wrong with this?

```
int main() {
  char another_str[16] = “123456789012345”;
  char my_str[] = “Hello, world!”;


  strncpy(another_str, my_str, strlen(my_str));
  printf(“%s\n”, another_str);

  return 0;
}
```

strncpy()

- What's wrong with this?

```
int main() {
  char another_str[16] = "123456789012345";
  char my_str[] = "Hello, world!";


  strncpy(another_str, my_str, strlen(my_str));
  printf("%s\n", another_str);

  return 0;
}
```


strncpy() will not NUL terminate the string!

strncpy()

- Does this fix it?

```
int main() {
  char another_str[16] = “123456789012345”;
  char my_str[] = “Hello, world!”;


  strncpy(another_str, my_str, strlen(my_str));
  another_str[strlen(my_str)] = ‘\0’;
  printf(“%s\n”, another_str);

  return 0;
}
```

strncpy() overflow

- No! What if my_str is longer than another_str?

```
int main() {
  char another_str[16] = “123456789012345”;
  char my_str[40] = “12345678901234567890” \
                    “1234567890123456789”;

  strncpy(another_str, my_str, strlen(my_str));
  another_str[strlen(my_str)] = ‘\0’;
  printf(“%s\n”, another_str);

  return 0;
}
```

strncpy()

- What about now?

```
int main() {
  char another_str[16] = “123456789012345”;
  char my_str[40] = “12345678901234567890” \
                    “12345678901234567890”;

  strncpy(another_str, my_str, strlen(another_str));
  another_str[strlen(another_str)] = ‘\0’;
  printf(“%s\n”, another_str);

  return 0;
}
```

strncpy()

- No! What if another_str is shorter than its buffer?

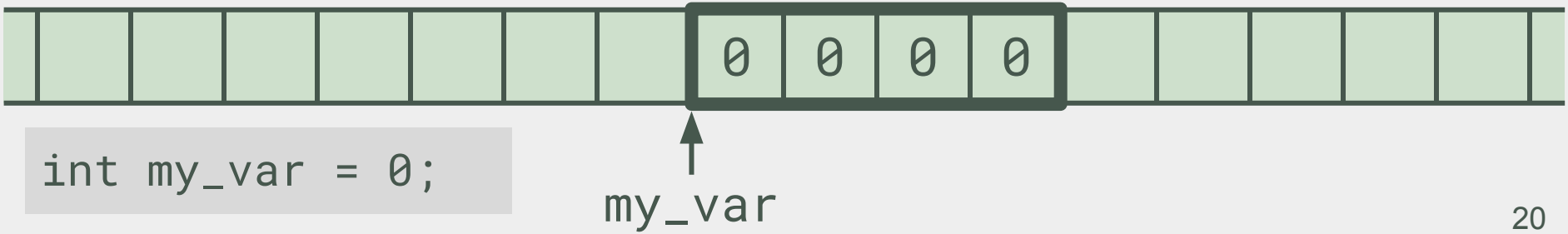
```
int main() {
  char another_str[16] = “1”;
  char my_str[40] = “12345678901234567890” \
                    “1234567890123456789”;

  strncpy(another_str, my_str, strlen(another_str));
  another_str[strlen(another_str)] = ‘\0’;
  printf(“%s\n”, another_str);

  return 0;
}
```

Data layout in memory

- Everything that contains a value uses memory
- Memory space looks like a long, continuous stream of bytes
- And everything that contains a value occupies one or more bytes of memory

Variables

- When we define a variable, the compiler creates a space for it in memory somewhere
- Whenever we use the name of the variable, it gets translated into that “somewhere”
- Some types of variables consume several bytes of memory
  - An int is usually 4 bytes long

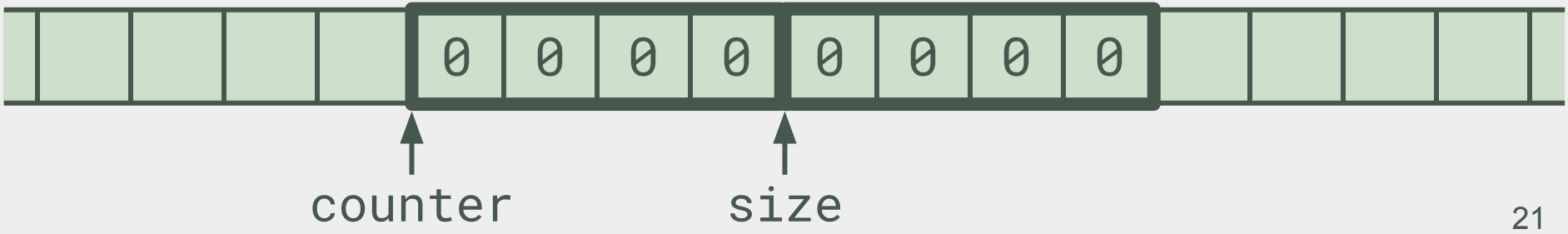



```
int my_var = 0;
```

Variables

- Variables that are defined near each other are usually near to each other in memory

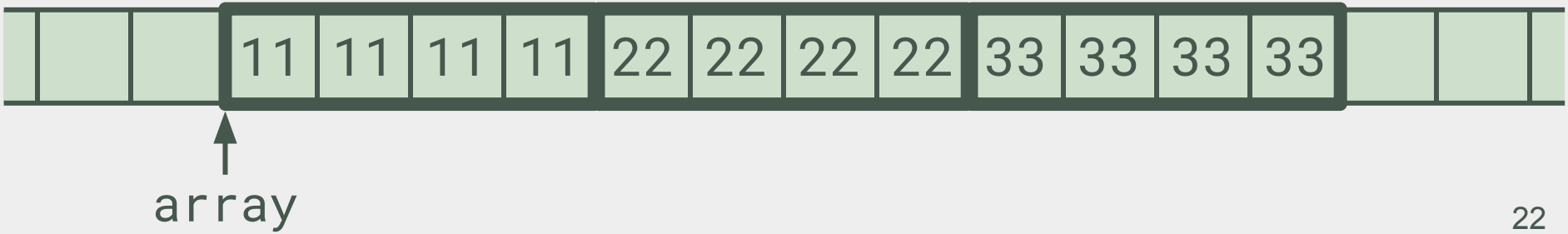
```
int counter = 0;
float size = 0.0;
```



Arrays

- Arrays of items are **guaranteed** to be packed together in memory

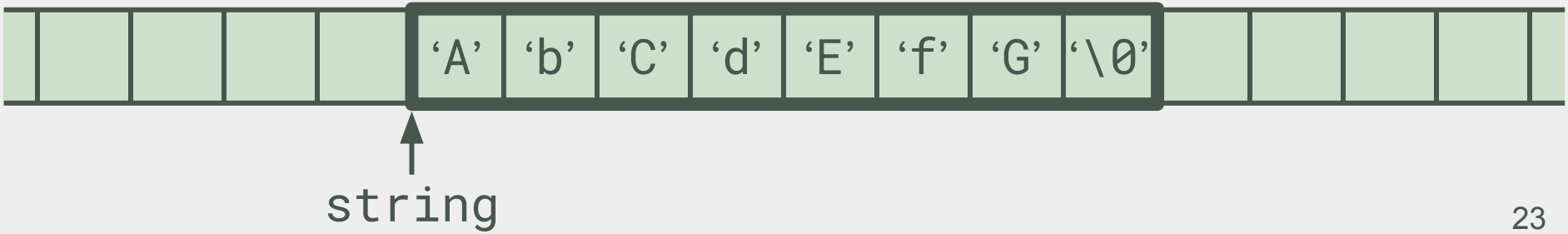
```
int array[3] = { 0x11111111,
                 0x22222222,
                 0x33333333 };
```



Strings

- A string in C is an **array** of characters
- All strings delimited by “ are said to be null-terminated
  - Terminated by a zero byte
- strcpy(), strcmp(), etc. will search for the null

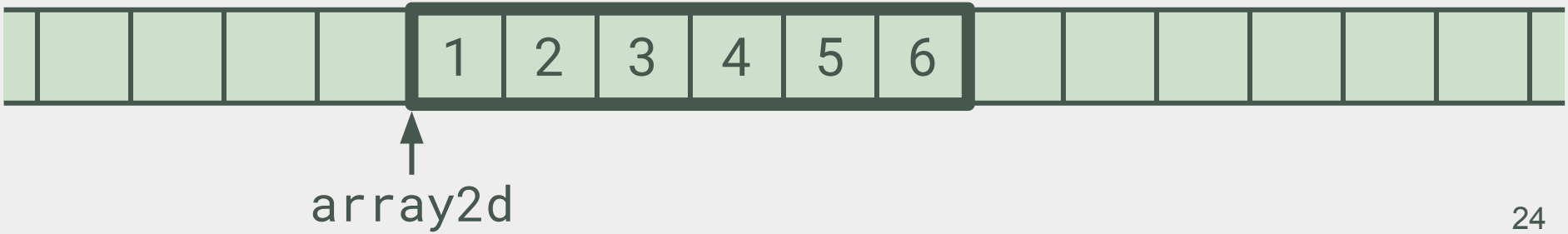
```
char string[8] = “AbCdEfG”;
```



Two dimensional arrays

- How does a 2-D array get stored in memory?

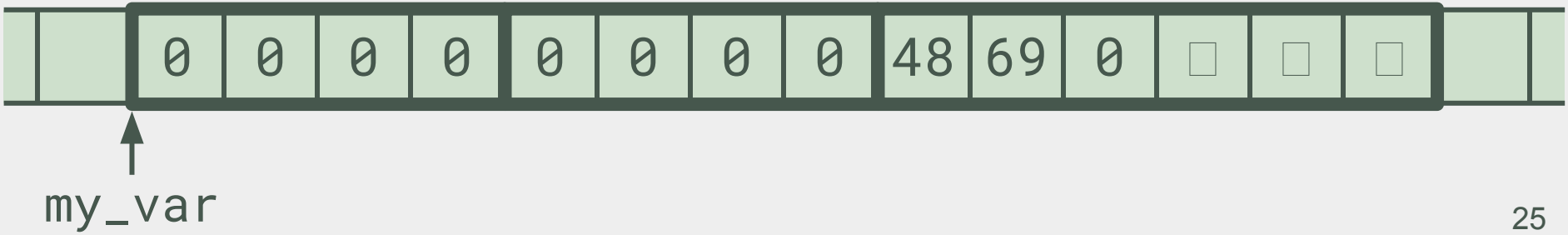
```
char array2d[2][3] = { {1, 2, 3},
                       {4, 5, 6} };
```



Structures

- Structure members are placed in memory just like arrays
- They are guaranteed to be packed next to each other

```
struct my_stuff {
  int i;
  float f;
  char c[6];
} my_var = { 0, 0, “Hi” };
```



Variable sizes and types

- How do you know the size and type of a variable?
- It may have a different allocated size on different machines with different compilers
  - A long would be 4 bytes on x86, but would be 8 bytes on x86_64
- We don’t want our software to misbehave when compiled on a different system.
- Fortunately, we don’t have to remember what the size is

sizeof()

- The sizeof() operator can tell us the size (number of bytes) of any:
  - Variable definition
  - Type declaration

```
int array[100];
printf("Size of char  = %d\n", sizeof(char));
printf("Size of array = %d\n", sizeof(array));
```

Correct strncpy()

- Use sizeof() to get the size of the buffer!

```
int main() {
  char another_str[16] = “1”;
  char my_str[40] = “12345678901234567890” \
                    “1234567890123456789”;

  strncpy(another_str, my_str, sizeof(another_str));
  another_str[sizeof(another_str) - 1] = ‘\0’;
  printf(“%s\n”, another_str);

  return 0;
}
```

What is the size of this struct?

```
struct strange {
  int x;  /* four bytes */
  int y;  /* four bytes */
  int z;  /* four bytes */
  char c; /* one byte */
};

int main() {
  printf("size = %d\n", sizeof(struct strange));
  return 0;
}
```

```
$ ./strange
size = 16
```

Right...

- The size of the previous structure is 16 bytes
- On most modern computers, an integer must reside on an even boundary if it is to be efficiently accessed

```
int my_int;
assert(&my_int % 4 == 0)

short my_short;
assert(&my_short % 2 == 0)
```

Padding

- Structures are often padded so that data elements occur at the correct offset
  - E.g., ints must be 4-byte aligned, longs must be 8-byte aligned, etc.
- Some architectures cannot handle unaligned accesses
- For others, they are very slow

If a structure is not padded

- If the structure was not padded, an array of these structures would look like this:

```
struct strange arr[3];
```

Incorrect!

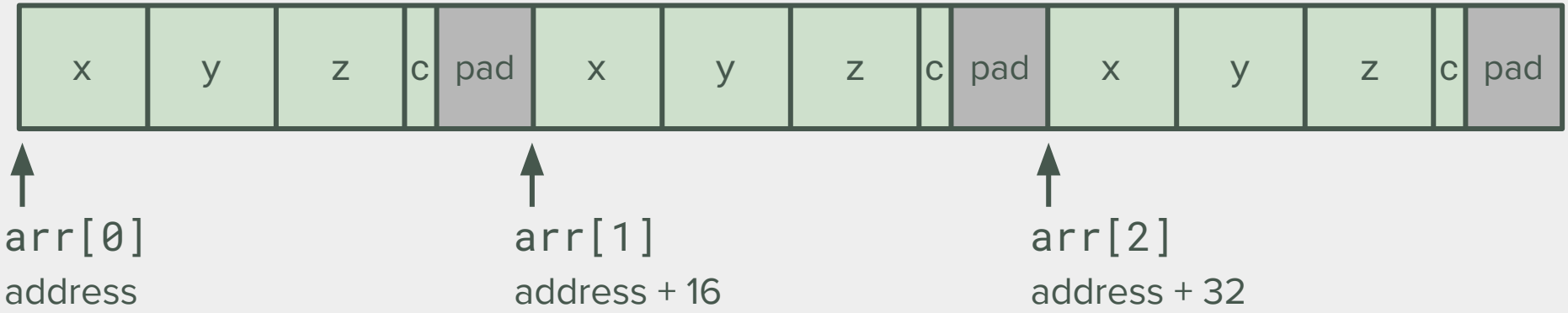
- If address is a proper location for an int, address + 13 isn't

When a structure is padded

- When an odd-sized array is created, it is padded to align all of its fields properly

```
struct strange arr[3];
```

Correct!



- Now all of its integers are on a proper boundary

When a structure is padded

- You can’t (shouldn’t) access the pad space
- Note that padding may be added at several places in the structure

How to create inefficient structs

- Here's a structure that uses space inefficiently

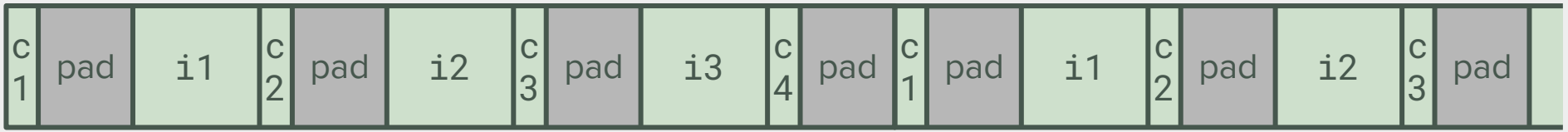
```
struct bad {
  char c1;
  int i1;
  char c2;
  int i2;
  char c3;
  int i3;
  char c4;
}; /* size = 28 */
```

- How big would you say it is?

How to create inefficient structs

- All values must be properly aligned...

- And must remain aligned in an array...



How to create efficient structs

- We can simply reorder the variables to reduce padding

```
struct bad {
  char c1;
  int i1;
  char c2;
  int i2;
  char c3;
  int i3;
  char c4;
}; /* size = 28 */
```

```
struct not_so_bad {
  int i1;
  int i2;
  int i3;
  char c1;
  char c2;
  char c3;
  char c4;
}; /* size = 16 */
```

Structure alignment rule of thumb

- When creating a structure, order the fields top to bottom by their relative size
  - double
  - long long
  - pointer
  - long
  - int/float
  - short
  - char
- Doing so will result in padding being added only to the end of the structure - if at all

Takehome Quiz #2

- Due 9/12 at 2:20 pm
- Use the template!
  - Solutions not using the template will not get full credit
- Handwritten only!
  - Or no credit

Takehome Quiz #2

Run the program on the next slide, and answer these questions:

1. Draw the memory map as in slides 19-25
   a. Use `setarch -R ./your_program` to run your program
   b. Hint: use “%p” with printf to print the address of a variable, e.g.:
   c. And run on data.cs.purdue.edu

```
printf(“%p\n”, &var);
```

2. Are there any gaps between space allocated for the variables? If so, why?
   a. Hint: padding only applies to structures
3. The order of variables in memory may be unexpected. Can you explain it?

Takehome Quiz #2

```
#include <stdio.h>

int main() {
  char buf[6] = “Hi!”;
  int my_int = 0xbeefbeef;
  struct a_struct {
    long l;
    char c;
  } my_struct = { 1701, ‘Z’ };
  short my_short = 0xf00d;

  printf(“%s %d %ld %c %hd\n”, buf, my_int, my_struct.l,
                               my_struct.c, my_short);
  return 0;
}
```

For next lecture

- Read K&R 2.3, 4.4, 6.8-6.9, A8.3-A8.4
  - ...and skim K&R 2
- Practice the examples!!

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.