U N I V E R S I T Y

**CS 240: Programming in C**

**Lecture 7: Arrays**
**Memory Layout of Data**

Prof. Jeff Turkstra




# Announcements

- Work on Homework 3!
  - Done by Sunday!
- Feasting with Faculty tomorrow 12pm
  - Look for the sign on the door (2 private dining rooms toward back of the area)
  - You can show up to any (all!) of them!



# Grades

- Homework grades are typically available the day after the assignment is due
  - No more announcements, your job to check
- Lecture quizzes same
- Takehome quizzes generally take one week to grade
  - Then released on Gradescope
    - Then in gradebook a week later
  - Regrade request deadline is based on Gradescope release



# Homework 3

- What's wrong with this? (Assume we check fscanf()'s return value)

```
char buf[1024];
fscanf(in_fp, "%[^\n]", buf);
if (strlen(buf) > MAX_NAME_LEN) {
  fclose(in_fp);
  in_fp = NULL;
  return BAD_RECORD;
}
```



# Homework 3

- How about this?

```
#define MAX_NAME_LEN (40)

char buf[MAX_NAME_LEN];
fscanf(in_fp, %40[^\n]", buf);
...
```



# Homework 3

- Don't forget the NUL terminator!

```
#define MAX_NAME_LEN (40)

char buf[MAX_NAME_LEN];
fscanf(in_fp, %39[^\n]", buf);
...
```

## Quizzes

- Must use the template
- Must be handwritten
- Otherwise 0
  - This has been discussed previously



## Debug output

- We have fairly strict file size limits on output
- If you leave a bunch of printf()s in your code you might hit it
- You've been warned
  - Future assignments that's a score of 0



## Feasting with Faculty

- Tomorrow! 12pm!
- Earhart Private Dining Room



## What about hw3.h?

```
extern char g_rental_history[MAX_RENTALS][3][MAX_BUF_LEN ];
extern char g_vehicle_info[MAX_RENTALS][3][MAX_BUF_LEN];
extern float g_rental_stats[MAX_RENTALS][4];

extern int g_rental_count;
```

- extern is also 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!



## Notes

- Do not use variable length arrays in this class
- Some of you are copying code and concepts from things outside of course material
  - fgetc(), sizeof(), malloc(), etc
  - You're probably cheating.
  - You're also making your life more difficult



## Reading

- In K&R:
  - Read Sections: 4.4, 6.8-6.9, A8.3-A8.4
  - ...and skim Chapter 2 (read 2.3)
- In Beej's:
  - Read Chapter 6, ignore 6.2
  - Read Chapter 14
  - Read sections 12.2-12.4

# Definitions vs. declarations

- Definition: allocates storage for a variable (or function)
- Declaration: announces the properties of a variable (or function)
- What's this?

```
struct hey {
  int   zap;
  float zing;
};
```

And this?

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



# Arrays of structures

- We can create arrays of structures just as we can create arrays of anything else. E.g.:

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

- Initialization is similar to before:

```
struct person people[4] = {
  { "Mai Elkady", "TA", {1, 2, 3, 4} },
  { "Nan Jiang", "TA", {2, 3, 4, 5} },
  { "Zach Bryant", "TA", {3,4,5,6} },
  { "Julie Stevenson", "TA",
    {4, 5, 6, 7} },
};
```



# Array of Structures Example (page 1)

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

struct person {
  char name[40];
  char title[15];
  int  codes[4];
};

struct person crowd[100]; /* global! */

void print_person(struct person);
```



# Array of Structures Example (page 2)

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

  strncpy(crowd[0].name, "Jeff", 40);
  strncpy(crowd[0].title, "Speaker", 15);
  crowd[0].codes[0] = 10;
  crowd[0].codes[1] = 20;
  crowd[0].codes[2] = 40;

  strncpy(crowd[1].name, "Student", 40);
  strncpy(crowd[1].title, "Listener", 15);
  crowd[1].codes[0] = 1;
```



# Array of Structures Example (page 3)

```
  for (index = 0; index < 100; index++) {
    if (crowd[index].name[0] != '\0') {
      print_person(crowd[index]);
    }
  }
  return 0;
}

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



# The result...

```
$ vi ex2.c
$ gcc -Wall -Werror -std=c99 -g -o ex2
ex2.c
$ ./ex2
Name:  Jeff
Title: Speaker
Codes: 10, 20, 0, 9

Name:  Student
Title: Listener
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') …`



## Purdue Trivia

- Purdue is home to Indiana's first and only nuclear reactor
  - Built in 1962
  - Built by Lockheed Corporation
  - Three stories beneath the Duncan Annex of EE
  - Criticality on August 30, 1962
  - Dedication September 27

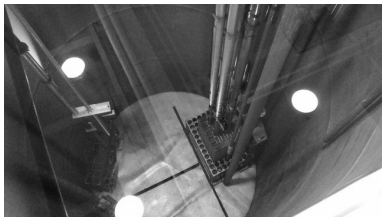












## Array initialization

- You can partially initialize an array! E.g.:

```
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



## Array auto-sizing

- You can define and initialize an array without explicitly saying what its size is. E.g.:

```
int my_array[] = { 1, 1, 2, 2, 3, 3, 7 };
```

- What would the size of this array be?
- 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_string);

  return 0;
}
```



## strncpy()

- Do not do 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_string);

  return 0;
}
```

strncpy() will not NUL terminate the string!



## strncpy() fixed?

- Don't do this either...

```
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_string);

  return 0;
}
```

# strncpy() fixed?

- Don't do this either…

```
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_string);

  return 0;
}
```

Only works because my_str happens to be smaller than another_str. What happens if my_str changes to something larger in the future?



# strncpy() overflow

- Oops…

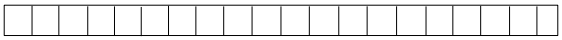
```
int main() {
  char another_str[16] = "123456789012345";
  char my_str[40] = "1234567890123456789" \
                    "123456789012345678";

  strncpy(another_str, my_str, strlen(my_str));
  another_str[strlen(my_str)] = '\0';
  printf("%s\n", another_string);

  return 0;
}
```

What's the right thing to do here?



# Data layout in memory

- Everything that contains a value uses memory
- Everything that contains a value uses memory
- Everything that contains a value uses memory
- Memory space looks like a long, continuous stream of bytes



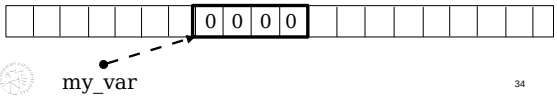
- And everything that contains a value occupies one or more bytes of memory



# 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. E.g., an 'int' is usually 4 bytes long.

```
int my_var = 0;
```

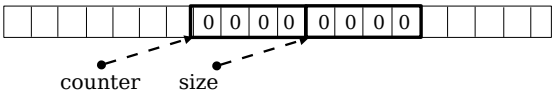





# More variables

- Variables that are defined near each other are usually near to each other in memory. e.g.:

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

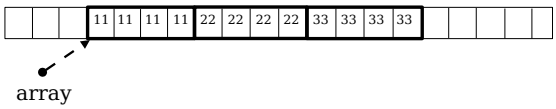





# Arrays

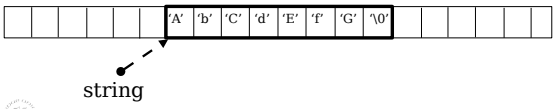
- Arrays of items are guaranteed to be packed together in memory. e.g.:

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



# Strings

- A string in C is an array of characters
- How are these characters stored?
- All strings delimited by (“) characters are said to be null-terminated (terminated by a zero byte)
- strcpy(), strcmp(), etc will search for the null. E.g.:
  `char string[8] = “AbCdEfG”;`

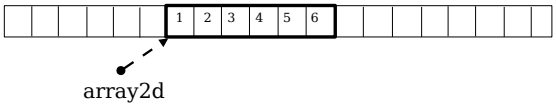





# Two dimensional arrays

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

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

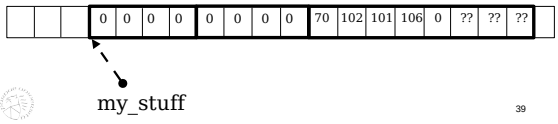





# Structures

- 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[8];
} my_var = { 0, 0, “Ffej” };
```






# How do you know the size of variables and types?

- A variable of a certain size may have a different allocated size on different machines with different compilers.
  - E.g. long X would be four bytes on x86 or Sparc but would be eight bytes long on Alpha, Sparc64, or 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()

- :-)

```
int main() {
  char another_str[16] = “123456789012345”;
  char my_str[40] = “1234567890123456789” \
                    “123456789012345678”;

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

  return 0;
}
```

# Takehome Quiz 4

```
#include <stdio.h>

int main() {
  char buf[11] = "Purdue";
  int my_int = 0x8badf00d;
  char my_char = 'X';
  short my_short = 0xbeef;

  printf("%s %d %c %hd\n", buf, my_int, my_char, my_short);

  return 0;
}
```

1. Draw the memory map as described previously
   - Remember to use setarch -R ./your_exe when running!
   - And run on data.cs.purdue.edu!
2. Are there any gaps between the space allocated for the variables?
   - If so, why might that be?



# For next lecture...

- Read!
- In K&R:
  - Read Sections: 4.4, 6.8-6.9, A8.3-A8.4
  - ...and skim Chapter 2 (read 2.3)
- In Beej's:
  - Read Chapter 6, ignore 6.2
  - Read Chapter 14
  - Read sections 12.2-12.4



# Boiler Up!