# *CS 240: Programming in C*

## Lecture 8: Binary File I/O, Unions, Enums, Bitwise Operators

PURDUE UNIVERSITY

# Announcements

- Homework 4 is out
  - Try out hw4_view

# Homework 2 histogram

```
291 scores total...
100+:  (0)
 100: ======================== (179)
  90: == (10)
  80: == (9)
  70: = (6)
  60: == (9)
  50: = (3)
  40: = (1)
  30: == (14)
  20: === (17)
  10: = (8)
   0: ===== (35)
```

# Announcements

- Midterm 1 is next week!
  - 9/24 at 8:00 pm
  - CL50 224 and MATH 175
  - Seating chart will be released soon
- Practice questions and practice midterm on website
- You will have to handwrite code

# Binary file I/O

- Given an open FILE pointer, we can use fread() and fwrite() to read or write “raw” memory items to or from a file

```
fwrite(void *ptr, int size, int num, FILE *fp);
fread(void *ptr, int size, int num, FILE *fp);
```

- This allows you to “dump” a data structure directly into a **binary** format file

# Example

```
#include <stdio.h>

struct xx {
  int x;
  int y;
};

int main() {
  struct xx try = { -1, -1 };
  FILE *fp = fopen(“input.file”, “rb”);
  int status = fread(&try, sizeof(struct xx), 1, fp);
  printf(“Read values (%d, %d) with return %d\n”,
         try.x, try.y, status);
  fclose(fp);
  return 0;
}
```

# *Return values*

- Both fread() and fwrite() return the number of **items** that were read or written
- On error, they return a short item count (or zero)

# *Uses of fread()*

- Recall the prototype:

```
fread(void *ptr, int size, int num, FILE *fp);
```

```
fread(&try, sizeof(struct xx), 1, fp);
```

- The “`void *`” means we can pass a pointer to “anything”
- What value should this call to fread() return?

- How many bytes are read by this operation?
- How would we read a whole file full of the structures?
- Is there any data format checking with this?

# fwrite() example

```
#include <stdio.h>

int main() {
  struct xx try = { 1, 2 };
  FILE *fp = fopen(“xx.out”, “wb”);
  if (fp == NULL) {
    return -1;
  }

  int ret = fwrite(&try, sizeof(struct xx), 1, fp);
  printf(“ret: %d\n”, ret);

  fclose(fp);
  fp = NULL;
  return 0;
}
```

# *fwrite()ing multiple structs*

```
#include <stdio.h>

int main() {
  struct xx xxs[20];
  for (int i = 0; i < 20; i++) {
    xxs[i].x = i + 1;
    xxs[i].y = -(i + 1);
  }

  FILE *fp = fopen(“xxs.out”, “wb”);
  int ret = fwrite(xxs, sizeof(struct xx), 20, fp);
  printf(“ret: %d\n\n”, ret);

  fclose(fp);
  return 0;
}
```

# fwrite()ing multiple structs

```
$ vi ex1.c
$ gcc -o ex1 ex1.c
$ ./ex1
ret: 20

$ cat xxs.out




$
```

# *fwrite()ing multiple structs*

```
$ hexdump -C xxs.out
00000000  01 00 00 00 ff ff ff ff  02 00 00 00 fe ff ff ff  |................|
00000010  03 00 00 00 fd ff ff ff  04 00 00 00 fc ff ff ff  |................|
00000020  05 00 00 00 fb ff ff ff  06 00 00 00 fa ff ff ff  |................|
00000030  07 00 00 00 f9 ff ff ff  08 00 00 00 f8 ff ff ff  |................|
00000040  09 00 00 00 f7 ff ff ff  0a 00 00 00 f6 ff ff ff  |................|
00000050  0b 00 00 00 f5 ff ff ff  0c 00 00 00 f4 ff ff ff  |................|
00000060  0d 00 00 00 f3 ff ff ff  0e 00 00 00 f2 ff ff ff  |................|
00000070  0f 00 00 00 f1 ff ff ff  10 00 00 00 f0 ff ff ff  |................|
00000080  11 00 00 00 ef ff ff ff  12 00 00 00 ee ff ff ff  |................|
00000090  13 00 00 00 ed ff ff ff  14 00 00 00 ec ff ff ff  |................|
000000a0
```

File offset

Contents

ASCII representation

PURDUE UNIVERSITY

# *fwrite()ing multiple structs*

- What's the difference between:

```
int ret = fwrite(xxs, sizeof(struct xx), 20, fp);
```

```
int ret = fwrite(xxs, sizeof(xxs), 1, fp);
```

- What will `ret` be in both cases? Assuming no errors
- Will the output files be different?

# *fread()ing multiple structs*

- We can do the same thing with fread()

```
int ret = fread(xxs, sizeof(struct xx), 20, fp);
```

```
int ret = fread(xxs, sizeof(xxs), 1, fp);
```

# *Summary of fread()/fwrite()*

- Moves a “memory image” to or from a file
- The file is **not portable**
- Different systems have different formats for integers, floats, etc.
  - Endianness
- No data type checking

# Endianness

- Recall we wrote the integer value 1 to a file...

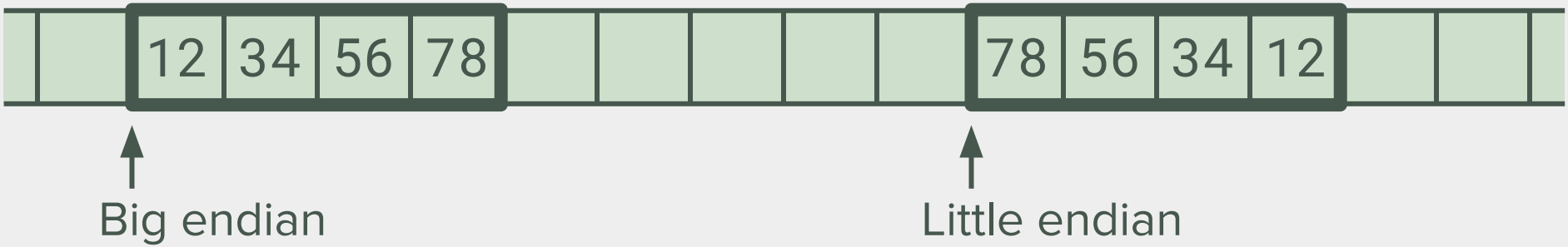
```
$ hexdump -C xxs.out
00000000 01 00 00 00 ff ff ff ff  02 00 00 00 fe ff ff ff  |................|
```

- Endianness is the order of **bytes** in a word or multi-byte value
  - It does not impact **bit** ordering for individual bytes!
- Two schemes:
  - Big-endian: most significant byte first (lowest address)
  - Little-endian: least significant byte first

# Example

- Consider the integer value 305419896
- In hexadecimal:
  - 0x12345678
- Each pair of hexadecimal values corresponds to 8 bits or 1 byte

# When stored in memory...




- Each box is 1 byte. We can look at it in binary too:
  - 0x12 = 0b00010010
  - 0x34 = 0b00110100
  - 0x56 = 0b01010110
  - 0x78 = 0b01111000
- 0x12345678 = 0b00010010 00110100 01010110 01111000
- 0x78563412 = 0b01111000 01010110 00110100 00010010

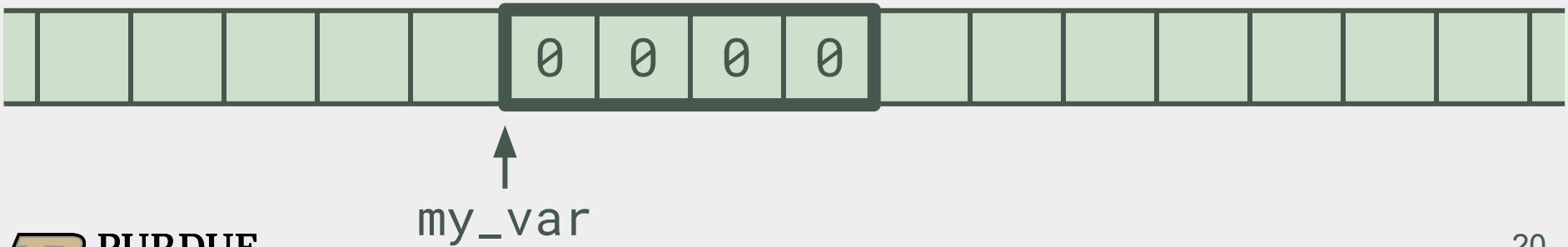
# *Binary file portability*

- Most machines use little endian
- Some file formats specify endianness
  - e.g., JPEG is always stored in Big endian
  - Little endian machines must convert the values
- Some file formats test endianness
  - E.g., read a known value from the file, check the first byte

# union

- Similar to structs, but internal elements overlap

```
union my_union {
  int i;
  float f;
  char c;
} my_var;
```



# Union example

```
union my_union my_var;
my_var.i = 86593;

printf("%c\n", my_var.c);
```

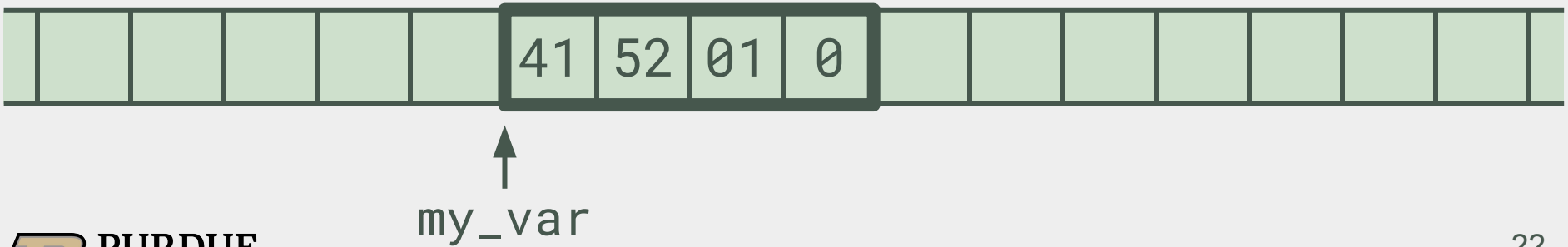
```
A
```

# Union example

```
union my_union my_var;
my_var.i = 86593;

printf("%c\n", my_var.c);
```

```
A
```



# Initialization

```
union my_union {
  int i;
  float f;
};
union my_union my_var = { 5 };
```

- Assumes you are initializing the first field!
- C99 has designated union initializers:

```
union my_union my_var = { .f = 5.0 };
```

# *Why unions?*

- When you really need to save space in your program and you know that some data will be one of two types
- Deep operating system hacking
  - Peripheral I/O manipulation
- Format conversion
- If you need it, you'll know
  - Don't use it in this class

# enum

- Attaches labels to values

```
enum color {
  RED,
  GREEN,
  BLUE
};
enum color my_hue = GREEN;
```

# enum example

```
#include <stdio.h>

enum color {RED, GREEN, BLUE};
int main() {
  enum color my_hue = GREEN;

  switch (my_hue) {
    case RED:
    case GREEN:
      printf(“Red or Green.\n”);
      break;
    case BLUE:
      printf(“Blue.\n”);
      break;
  }
  return 0;
}
```

# *enums can also have values*

- You can assign exact values to the enum declaration's members

```
enum british_transport {
  LAND=1,
  SEA=2,
  AIR=3,
  SUBMARINE=2,
  FLYING_SAUCER=400
};
```

# Use of that enum

```
#include <stdio.h>

int main() {
  enum british_transport craft = AIR;
  printf(“Value of craft is %d\n”, craft);
  return 0;
}
```

# *Bitwise operators*

- You regularly use logical operators:
  - ||, && in compound if statements
- What does this mean?

```
if (x) printf("x = %d\n", x);
```

- And this?

```
if (x && y) printf("x = %d\n", x);
```

- There are also bitwise operators: | &
- What does this mean?

```
if (x & y) printf("x = %d\n", x);
```

# *The difference between && & &*

- Logical operators check whether the quantities are zero or non-zero

```
if (x && y) { ... }
```

...really means:

```
if ((x != 0) && (y != 0)) { ... }
```

- The result of && is either 1 or 0
- Use logical operators to make a **yes/no** decision

# *The difference between && & &*

- Bitwise operators work on all of the bits

```
char x = 5;  /* binary 00000101 */
char y = 6;  /* binary 00000110 */
char z = 0;  /* binary 00000000 */

z = x & y;   /* result 00000100 */
```

- There are also OR (|), XOR (^) and NOT (~) operators
- Use bitwise operators when you want to work on the **bits** of a quantity

# Truth tables

AND

| X | Y | O |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR

| X | Y | O |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XOR

| X | Y | O |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

NOT

| X | O |
|---|---|
| 0 | 1 |
| 1 | 0 |

# We also have shift operators

- You can take a bunch of bits and shift them left or right

```
char x = 10;  /* binary 00001010 */
char y = 0;

y = x << 3;   /* result 01010000 */
```

- Note that every shift left is equivalent to a multiplication by two
- Above statement is equivalent to:

```
y = x * 2^3
```

# Example: cut a range of bits

- Suppose we want to write a function that accepts a 32-bit integer and pulls a range of bits from somewhere in the middle:

```
unsigned int bit_range(unsigned int num,
                       unsigned int bits,
                       unsigned int offset) {
  return ((num >> offset) & ((1 << bits) - 1));
}
```

- You'll have to stare at that for a while to understand it...

# Bit setting / clearing

- Use bitwise operators to set/clear bits in numbers…

```
int color = 44;  /* binary 00101100 */
int blue = 7;    /* binary 00000111 */

printf(“Color with all blue is %d\n”,
       color | blue);  /* 00101111 */
printf(“Color with no blue is %d\n”,
       color & ~blue); /* 00101000 */
```

# Bit checking

- How can we determine if a specified bit is set?
  - i.e., set to 1

```
char bits = 44;  /* binary 00101100 */
char mask = 8;   /* binary 00001000 */

if ((bits & mask) == mask) {
  printf(“The bit is set\n”);
}
else {
  printf(“The bit is cleared\n”);
}
```

# *For next lecture*

- Read K&R Chapter 5
  - ...and/or Beej Chapter 7
  - Probably repeatedly
- Understand the operators & and *

# Slides

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