PURDUE
UNIVERSITY®

CS 240: Programming in C

Lecture 8: Memory Layout of Data Padding Binary File I/O

Prof. Jeff Turkstra

Announcements

- For homeworks, plan ahead!
  - Poor planning on your part does not constitute an emergency on our part!
- Midterm exam is slowly creeping up on us – Monday, March 3
  - Remember to write your code out by hand first, then type it in and compile/run it
  - Sample exam and questions available ~1 week before the exam

Homework 2

- Issue with RNG seed in the test modules provided to students
- When compiled for grading, this issue went away
- Some student code had overflow issues that were not triggered until grading
- We're going to fix the grading environment to match the test module students used

Lecture Quiz 5

- I adjusted the location API timeout to the detriment of some of you
  - This has been reverted
- We will not count Lecture Quiz 5

Homework 4

- Fun little trip into imaginary number land
- Remember Euler’s identity?

$$e^{\pi i} + 1 = 0$$

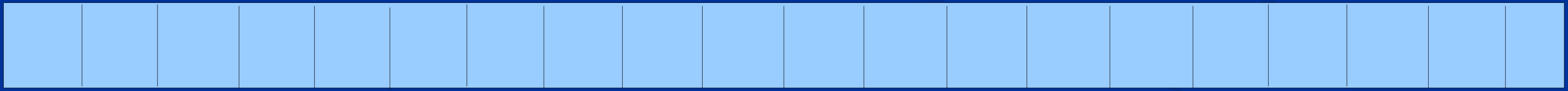
- Use hw4_view – it’s beautiful!

Reading

- Read Chapter 5
  - ...and/or Chapter 7 in Beej’s
  - Probably repeatedly

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

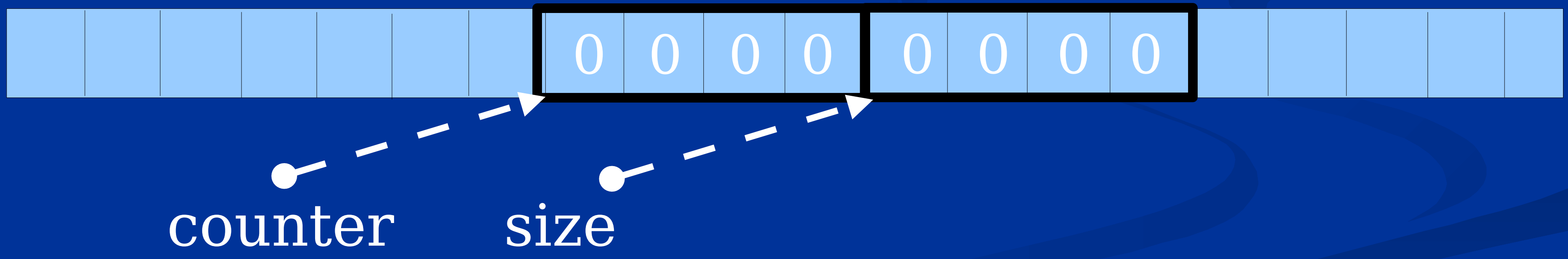


- And everything that contains a value occupies one or more bytes of memory

More variables

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

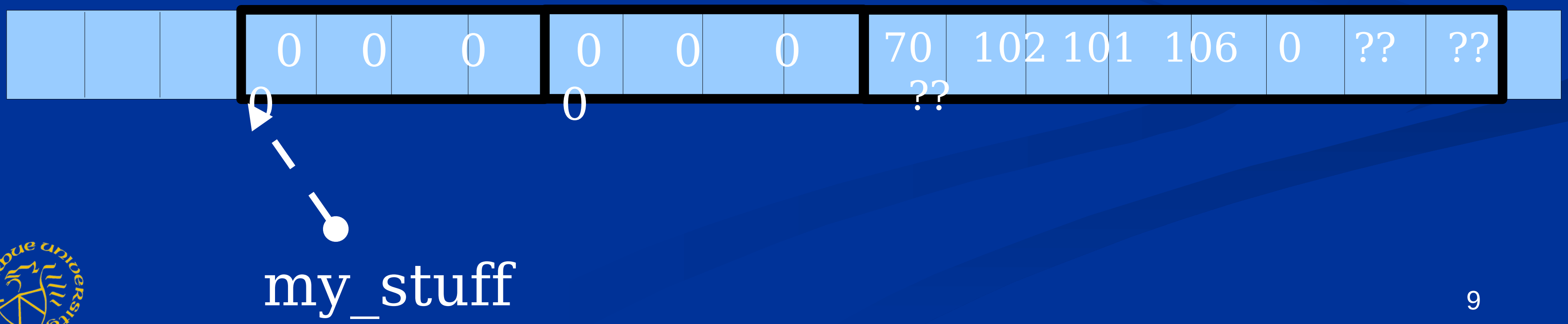
```
int counter = 0;
float size = 0.0;
```



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…

Correct strncpy()

- :-)

```
int main() {
  char another_str[16] = "123456789012345";
  char my_str[40] = "1234567890123456789" \
                    "12345678901234567 8";

  strncpy(another_str, my_str, sizeof(another_str));
  another_str[sizeof(another_str) - 1] = '\0';
  printf("%s\n", another_string);

  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;
}
```

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. E.g.:

```
int my_int;                    short my_short;
&my_int % 4 == 0               &my_short % 2 ==
0
```

What is the size of this struct?

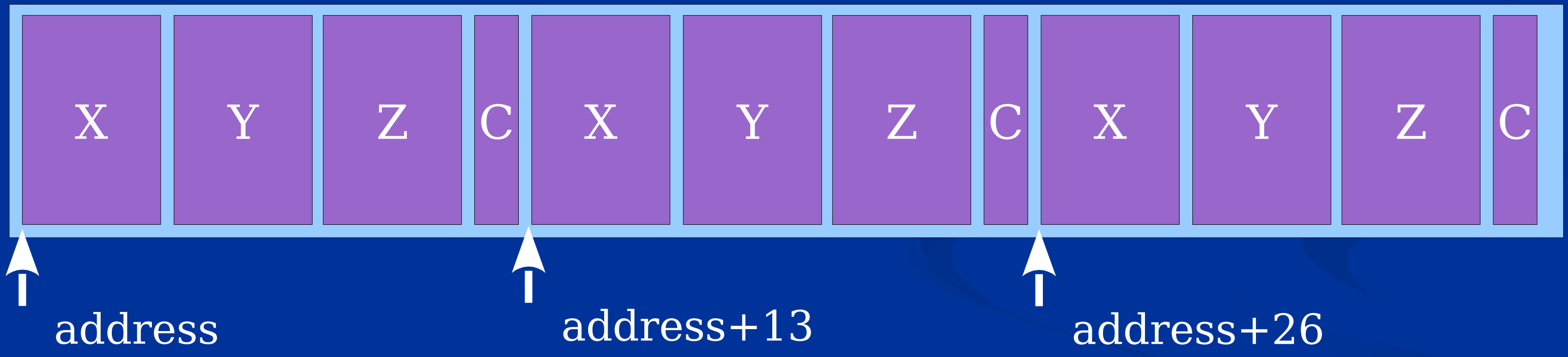
```
struct strange {
  int x;  /* four bytes */
  int y;  /* four bytes */
  int z;  /* four bytes */
  char c; /* one byte */
};

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

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 (intel), they are very slow

If a structure is not padded

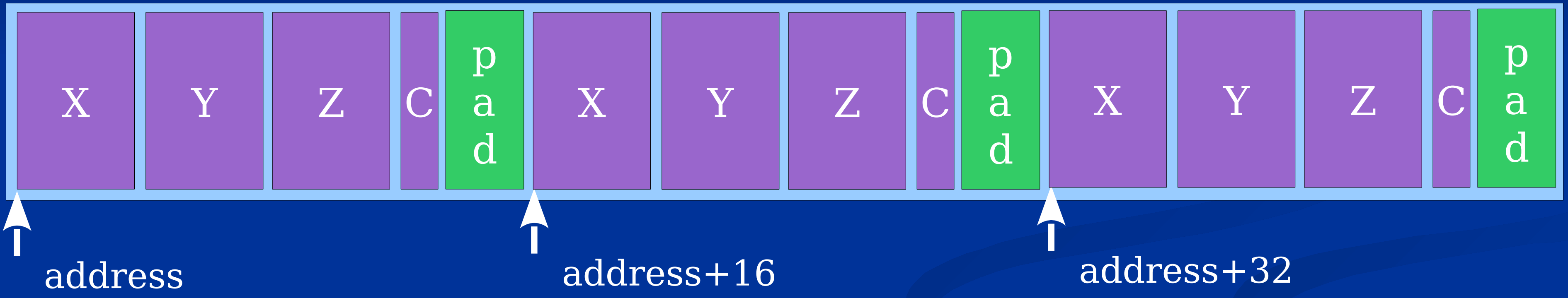
- If the structure was not padded, an array of these structures would look like this:



- If address is a proper location for an integer, then address + 13 is certainly not

When a structure is padded

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



- Now all of its integers are on a proper boundary
- 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:

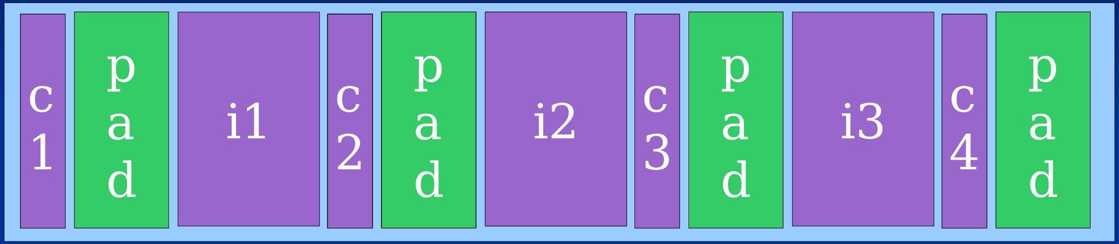
- Here's a structure that uses space inefficiently:

  ```
  struct bad {
    char c1;
    int i1;
    char c2;
    int i2;
    char c3;
    int i3;
    char c4;
  };
  ```

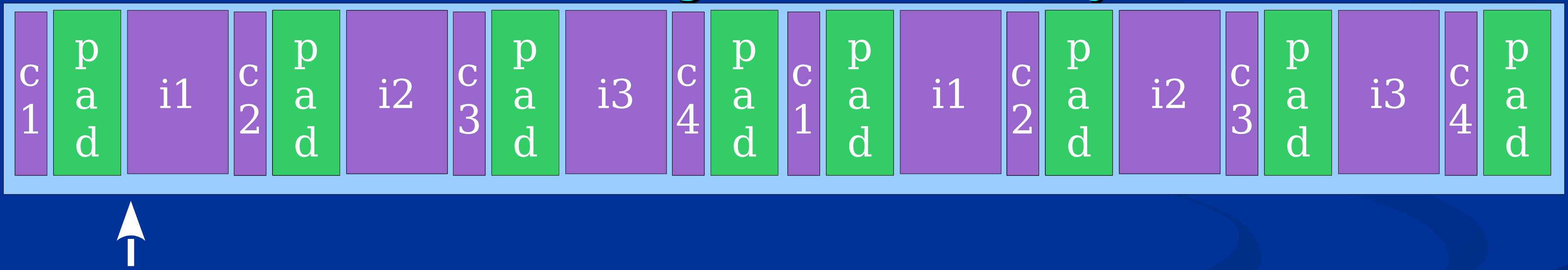
- How big would you say it is?

When a structure is padded

- All values must be properly aligned…



- And must remain aligned in an array…



no padding?
i1 won't be
properly
aligned

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;
};
```

- How big would you say it is?

How to create inefficient structs:

- Here’s a structure that uses space inefficiently:

```
struct bad {                struct not_so_bad {
  char c1;                    int i1;
  int i1;                     int i2;
  char c2;                    int i3;
  int i2;                     char c1;
  char c3;                    char c2;
  int i3;                     char c3;
  char c4;                    char c4;
};  /* size = 28 */         }; /* size = 16 */
```

- How big would you say it is? We can do better.

Structure alignment: Rule of thumb

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

Purdue Trivia

In 1935, during a Purdue football game at Northwestern University the band donned lights on their uniforms while performing at halftime. With the stadium lights turned off for the performance, the band drew such awe from radio broadcaster Ted Husing, he referred to them as a "truly All-American marching band," hence the current title of the band.

fread(), fwrite()

- 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 = 0;
  int status = 0;
  fp = fopen(“input.file”, “rb”);
  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 these structures?
- Is there any data format checking with this?

fread()ing multiple structures

```
#include <stdio.h>

int main() {
  FILE *fp = 0;
  struct xx try = { 0, 0 };

  fp = fopen(“input.file”, “rb”);
  if (!fp) {
    return -1;
  }
  while (fread(&try, sizeof(struct xx), 1, fp) == 1) {
    printf(“Read (%d, %d)\n”, try.x, try.y);
  }
  fclose(fp);
  fp = NULL;
  return 0;
}
```

fwrite() example

```
#include <stdio.h>

struct xx {
  int x;
  int y;
};

int main() {
  struct xx try = { 0, 1 };
  FILE *fp = 0;
  fp = fopen(“output.file”, “wb”);
  fwrite(&try, sizeof(struct xx), 1, fp);
  fclose(fp);
  fp = NULL;
  return 0;
}
```

fwrite() curiosities

- Recall the prototype for fwrite():

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

fwrite(&try, sizeof(struct xx), 2, fp);
```

- We're getting close to talking about pointers…
  - "void *" means we can pass a pointer to "anything"
- How many bytes are written by this operation?
- What value should this call to fwrite() return?

Summary: 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
- Little endian (intel) vs big endian (motorola, sparc)
- No data type checking

For next lecture

- Read Chapter 5
  - …and/or Chapter 7 in Beej’s
  - Probably repeatedly
- Have you started Homework 4?

Boiler Up!