CS1 – Exam 1 Review

Michael McAlpin

UCF

Things since we started …

- Pointers
  - segfaults
  - malloc
  - free
- Structures
  - create & initialize
- File I/O
  - open
  - read
  - write
  - close
- Big O
- SUMMATIONS (see L6)

Pointers - Obtaining an address

- Every variable declared in C has a memory address.
- Access it using the unary operator, & , and print it using the %p conversion character
- Create an integer named ***a*** with a value of 42 & create an integer pointer named ***p***, set it to point to ***a***

```
int a = 42;
// Create a variable capable of storing the address
int *p;
// Assign the address of 'a' to 'p'.
p = &a; //We say that 'p' now points to 'a'.
printf("(value) a: %d\n", a);
printf("(addr) &a: %p\n", &a);
```

Pointers - All together now…

```
int x = 52;// Create a place to store an integer.

int *p; // variable capable of storing the address
of 'x'

p = &x; // Assign the address of 'x' to 'p'.
        //('p' now points to 'x')

// Print the results to verify.
printf("(value) x: %d\n", x);
printf("(addr) &x: %p\n", &x);
printf("(value) p: %p\n", p);
printf("(addr) &p: %p\n", &p);
```

Segfault^2

- Dereferencing a NULL pointer

```
int *p = NULL;
*p = 42; //Crashes here…
```

- FYI
  - Typically, create a pointer and initialize it to NULL.
  - The default way to initialize pointers in C.
  - Indicates pointer is not currently holding a valid memory address.
  - AKA Defensive Programming.
- Example of strong(ish) defense

```
if (p == NULL)
    printf("Initially, p is NULL.\n");
```

Segfault^3

- Array out of bounds

```
int i, array[10];
for (i = 0; i < 1000000000; i++)
    array[i] = 0;
```

- BLAM!

NB

May not occur until array[i] is much more than 10. Why?

malloc Usage[1]

- Malloc
  - `type * varName = malloc(varSize);`
  - Notes:
    - Returns a memory address of the buffer of the requested size
    - On error returns a NULL
    - *DOES NOT INITIAL MEMORY CONTENTS.*
- Free
  - `free(varName);`

malloc Usage^2

- Let's get DEFENSIVE!!!

```
int * array = malloc(10 * sizeof(int));
if (NULL == array) {
  fprintf(stderr, "malloc failed\n");
  return(-1);
}
```

- Why?

malloc Usage[4]

- Creation and destruction is a cycle that must be managed…
  - For example:
    - **`int *p = malloc(500 * sizeof(int));`**
    - must have a corresponding **`free(p);`**
- Why?

Other ideas for buffers

- An array of ten integers?
  - `int array[10];`
    - Static or dynamic? Why?
  - `int * arrayInts = malloc(10 * sizeof(int));`
    - Static or dynamic? Why?
- Accessing the contents of a buffer
  - Logically equivalent?

```
* arrayInts = 42;
arrayInts[0] = 42;
```

Pointers[0]

- Given:

```
const char* p = "abc";
```

- the compiler produces:

```
Memory Address (hex)    Variable name    Contents
1000                                     'a' == 97 (ASCII)
1001                                     'b' == 98
1002                                     'c' == 99
1003                                     0
...
2000-2003               p                1000 hex
```

Pointers[1]

- Can be used to directly manipulate data in a variable
- The good news
  - *call by value* versus *call by reference*
  - Directly process data in an array (OR buffer)
  - Can be useful – defensively **`const`**
- The bad news
  - can attempt illegal modification (aka ?)

Structs – as in STUCTURE

- A defined collections of one or more variables
  - May be different types
  - grouped together under a single name
- Advantage – streamlines organization
- Provides convenient
  - access
  - handling

Structs – mechanics[1]

- Defined as follows:

```
typedef struct student
{
        char *fName;
        char *lName;
        int pid;
} student;
```

- Things to note:
  - typedef (alias for another data type)
  - structure tag name (*student*)
  - member definitions
  - structure type name (*student*)
  - ***convention is to use the same identifier for the tag name and the type name***

Structs – mechanics^2

- Declaration

  student s;

- Note that NOTHING in **s** is initialized – as if it was any other data type
- Initialization – example

```
s.fName = malloc(sizeof(char) * (22 + 1));
s.lName = malloc(sizeof(char) * (32 + 1));
s.pid = 421764;
```

Structs – mechanics[3]

- Declaration

  student s;

- Note that NOTHING in **s** is initialized – as if it was any other data type
- Initialization – example

```
s.fName = malloc(sizeof(char) * (22 + 1));
s.lName = malloc(sizeof(char) * (32 + 1));
s.pid = 421764;
```

NB The dot operator (.) provides access to the specific member of the struct.

Structs – Pointers[1]

- Declaration:

  student *s;

- Accessing the members either by -
  - Dereference the pointer, then access the members as normal, with the dot operator (.):

    (*s).lName
  - Use the arrow operator (->) which takes care of the dereferencing :

    s->lName

Structs – Pointers^2

- DMA for a struct

```
student *s;
student *s = malloc(sizeof(student));
```

- Accessing the members either by -
  - Dereference the pointer, then access the members as normal, with the dot operator (.):

```
(*s).lName
```

  - Use the arrow operator (->) which takes care of the dereferencing :

```
s->lName //PREFERRED STYLE
```

How to initialize the fName & lName members?

Structs – Pointers[3]

- Initializing the data in the struct pointed to by **s**

```
s->fName = malloc(sizeof(char) * (22 + 1));
s->lName = malloc(sizeof(char) * (32 + 1));
s->pid = 421764;
strcpy(s->fName, "Zaphod");
strcpy(s->lName, "Beeblebrox");
s->pid = 421764;
```

File IO - OPEN

```
char err_msg[1024], filename[20];
FILE *ifp;
strcpy(filename, "input.txt");
if ((ifp = fopen(filename, "r")) == NULL)
{
   sprintf(err_msg, "Attempted to open: %s\n”,
           filename);
   debug(err_msg, 1);
   panic("Fail on open input file->main()!\n");
}
```

- On success returns FILEPOINTER (ifp)
- Error handling
  - NULL is returned and *errno*is set to indicate the error

File IO - Read

```
char buffer[100];
FILE *ifp = fopen("input.txt", "r");
if (ifp == NULL)
   panic("ERR:could not open file input.txt\n");
//Read loop
while (fscanf(ifp, "%s", buffer) != EOF)
   {
      printf("(read string): %s\n", buffer);
    }
```

- What is the filename?
  - This is a “hard coded” filename. (bad dog!)
  - More about this later…

File IO - Write

```
char buffer[100];
FILE *ofp = fopen(”output.txt", "rwb");
if (ofp == NULL)
    panic("ERROR: could not open output fil\n");
//Read loop
while (fscanf(ifp, "%s", buffer) != EOF)
   {
      fprintf(ofp, “%s\n", buffer);
    }
```

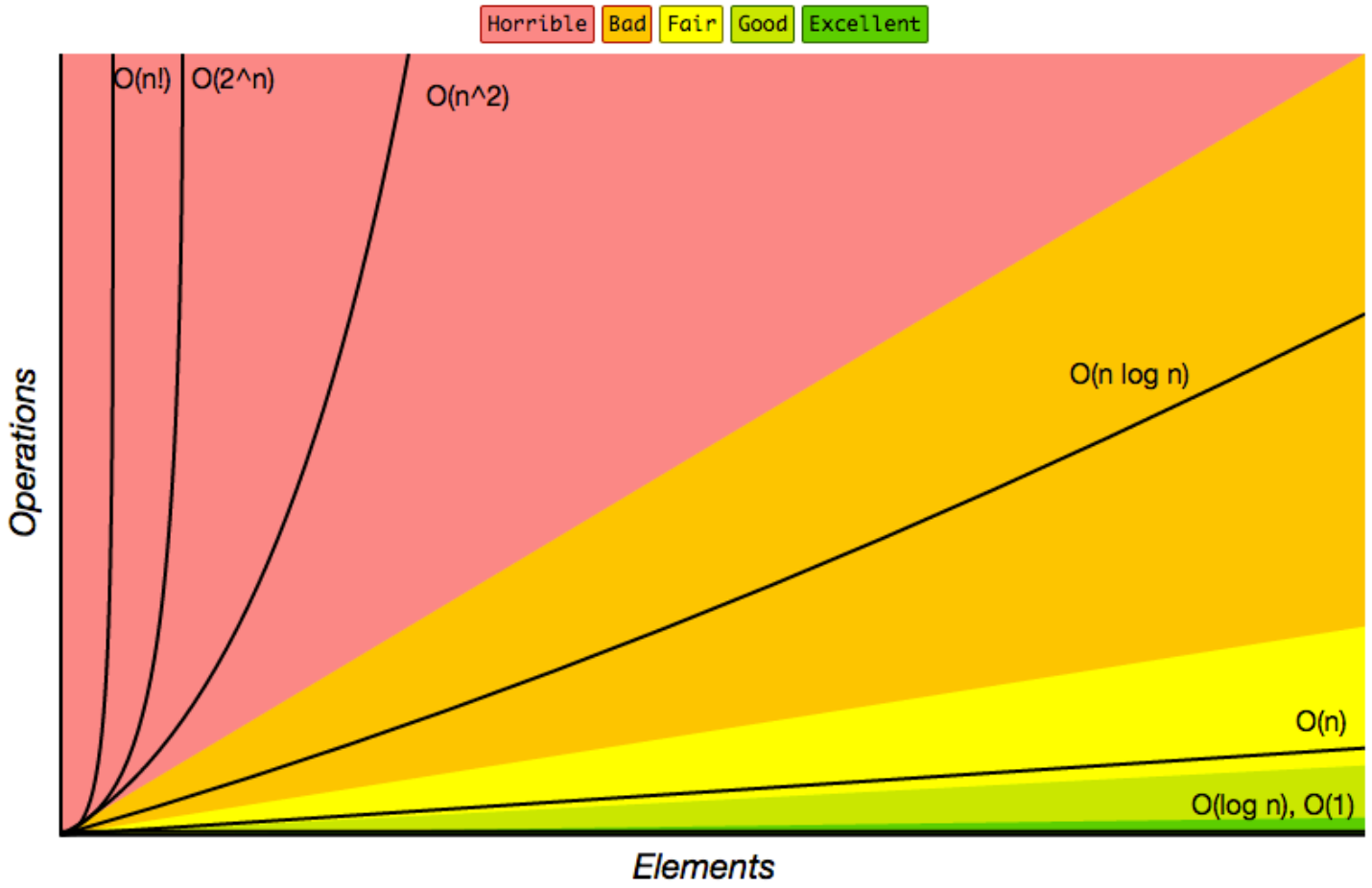
- Reads the input file (see earlier slides for details)
  - This is a “hard coded” filename. (bad dog!)
  - Note the permissions, R W & B

File IO - CLOSE

```
fclose(ifp);
fclose(ofp);
```

- Flushes the stream pointed to by stream
  - (writing any buffered output data using fflush(3))
  - closes the underlying file descriptor.
- Successful completion 0 is returned.
- On error
  - EOF is returned
  - errnois set to indicate the error.
- REGARDLESS - any further access (including another call to fclose()) to the stream results in *undefined behavior*.

Big “O” – the big picture



Big "O" – the words

- Big-O notation is a relative representation of the complexity of an algorithm
- We only care about the most significant portion of complexity
  - Given $45n^3 + 2x^2 + 19$
    - What is the most significant part of the equation?
- Is not an absolute measure of performance
  - Not the best case
  - Not the expected case
  - Yup – the worst case
- find ***O*** by dropping constants & looking for highest-order term

Big "O" – the cases

- O (1) : Constant time.
- O (lg n) : Logarithmic time.
- O (n) : Linear time
- O (n lg n): Linearithmic time.
- $O(n^2)$: Quadratic time.
- O(n!) : Factorial time.

O(1) – Constant time

- O (1) = O (10) = O (2^{100}) - why?
  - Even though the constants are huge, they are still constant.
  - if you have an algorithm that takes 2^{100} discreet steps, regardless of the size of the input
    - the algorithm is still O (1) - runs in constant time;
    - Not dependent upon the size of the input .

O(1) – Constant time code

- The code:

```
int fooBoo(int n){
return n+1;}
```

O (lg n) : Logarithmic time

- This is faster than linear time;
  - O (log 10 n) = O (ln n) = O (lg n)
  - CS most concerned with lg n , which is the base-2 logarithm
    - Why is this the case?
  - The fastest time bound for search.
- Used for:
  - Binary searches, balanced tree searches, binomial heaps

O (n) : Linear time

- Need to examine every single bit of your input. At least once.
- iterates over data one or more times
- The code:

```
int foo0(int *array, int n){
 int i, sum = 0;
for (i = 0; i < n; i++)
   sum += array[i];
return sum;}
```

O (n lg n)

- Fastest time bound we can currently achieve for sorting a list of elements.
- Used for
  - Fast Fourier transforms
  - Heapsort
  - Quicksort
  - Merge Sort

  NB O(n lg n) = O(lg n!)

$O(n^2)$: Quadratic time the code

- The code:

```
int foo2(int n){
  int i, j, x = 0;

  for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
                  // How many times
      x++;        // executed?
  return x;}
```

Recap

- Pointers
  - segfaults
  - malloc
  - free
- Structures
  - create & initialize
- File I/O
  - open
  - read
  - write
  - close
- Big O
- SUMMATIONS (see L6-Summations.pdf in webcourses)