C bugs, arrays, the shell, data in memory

CS 211: Computer Architecture

Fall 2020

Common bug 1

scanf's arguments must be pointers to a valid location

```
int val;
scanf("%d", val);
```

Common bug 2

Reading uninitialized memory

```
int* matvec(int** A, int* x) {
    int* y = malloc(n * sizeof(int));
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            y[i] += A[i][j] * x[j];
    return y;
}
```

Common bug 3

Allocating the (possibly) wrong sized object

```
int** p;
p = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
    p[i] = malloc(m * sizeof(int));
```

Common bug 4

Overwriting memory

```
int** p;
p = malloc(n * sizeof(int*));
for (i = 0; i <= n; i++)
    p[i] = malloc(m * sizeof(int));
```

Common bug 5

Misunderstanding pointer arithmetic

```
int* search(int* p, int val) {
    while (*p && *p != val)
        p += sizeof(int);
    return p;
}
```

Common bug 6

Referencing nonexistent variables

```
int* foo () {
    int val;
    return &val;
}
```

Common bug 7

Freeing blocks multiple times

```
x = malloc(n * sizeof(int));
// ... manipulate x ...
free(x);
y = malloc(m * sizeof(int));
// ... manipulate y ...
free(x);
```

Common bug 8

Use after free

```
x = malloc(n * sizeof(int));
// ... manipulate x ...
free(x);
// ...
y = malloc(m * sizeof(int));
for (i = 0; i < m; i++)
    y[i] = x[i]++;
```

Common bug 9

Failing to free blocks

```
int *x = malloc(n * sizeof(int));
// ...
```

Common bug 10

Freeing only part of a data structure

```
typedef struct List {
    int val;
    struct List* next;
} List;

void foo() {
    List *head = malloc(sizeof(List));
    head->val = 0;
    head->next = NULL;
    // ...
    free(head);
}
```

Arrays vs. pointers

Pointer	Array
holds address	holds data
access is indirect	access is direct
“dynamic” data	“static” data

Arrays vs. pointers

```
char* s1 = "hello";
char s2[] = "hello";
```

Arrays vs. pointers

```c
int x = 42;
int* p = &x;
int a[100];

printf("%p\n", p);
printf("%p\n", &p);
printf("%p\n", a);
printf("%p\n", &a);
```

Pipes and redirection

We can redirect the output from a program to a file:

```
./myProgram > out.txt
```

We can redirect stderr too:

```
find /etc -name "*color*" 2> /dev/null
```

Pipes and redirection

We can take the output from one program and give it as input to another

```
./myProgram | ./myOtherProgram
```

Pipes and redirection

Given a CSV file of student names, emails, courses, and #credits:

```
Bob Smith,bob@rutgers.edu,CS111,78
Carol Ford,carol@rutgers.edu,CS211,43
Alice Jones,alice@rutgers.edu,CS211,92
...
```

- Print names/emails of CS211 students in descending order of #credits
- Find the average number of credits of all students

Pipes and redirection

```
Bob Smith,bob@rutgers.edu,CS111,78
Carol Ford,carol@rutgers.edu,CS211,43
Alice Jones,alice@rutgers.edu,CS211,92
```

Print CS211 email list:

```
cat students.csv | \
  grep ',CS211,' | \
  perl -pe 's/(.*),(\d*)$/$2 $1/' | \
  sort -rn | \
  perl -pe 's/^\d* ([^,]*),([^,]*),.*$/"$1" <$2>/'
```

Pipes and redirection

```
Bob Smith,bob@rutgers.edu,CS111,78
Carol Ford,carol@rutgers.edu,CS211,43
Alice Jones,alice@rutgers.edu,CS211,92
```

Find average number of credits of all students:

```
cat students.csv | \
  cut -d, -f4 | \
  awk '{ sum += $1; n++ } END { print sum/n }'
```

C mask operators

- `x & m` - do a bitwise AND operation
- `x | m` - do a bitwise OR operation
- `x ^ m` - do a bitwise XOR operation
- `~x` - do a bitwise NOT operation

Examples:

- `1 & 1 = 1`
- `17 | 68 = 85`
- `17 | 20 = 21`
- `85 ^ 83 = 6`

C shift operators

- `x << n` - shift x left by n bits
- `x >> n` - shift x right by n bits

Note:

- A left shift = multiplying by 2
- A right shift = dividing by 2 (and discarding remainders)

Examples:

- `1 << 1 == 2`
- `25 << 2 == 100`
- `25 >> 1 == 12`

Shifts and masks

- Given the integer 1234, grab the 3rd byte

Data sizes

C type	Min size	Typical size
char	1	1
short int	2	2
int	2	4
long int	4	8
pointer		8
float		4
double		8

Endianness

How is a 4-byte int stored in memory?

```
int x = 1;
```

- Most significant byte (MSB) first → big endian
- Least significant byte (LSB) first → little endian

How can we tell which our system uses?