Memory-Related Perils and Pitfalls in C

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

Dereferencing Bad Pointers

- The classic `scanf` bug

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

- Will cause `scanf` to interpret contents of `val` as an address!
  - Best case: program terminates immediately due to segmentation fault
  - Worst case: contents of `val` correspond to some valid read/write area of virtual memory, causing `scanf` to overwrite that memory, with disastrous and baffling consequences much later in program execution

Dereferencing Bad Pointers

- The classic `scanf` bug

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

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```c
/* return y = Ax */
int *matvec(int **A, int *x) {
    int *y = (int *)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;
}
```
Overwriting Memory

- Allocating the (possibly) wrong sized object

```c
int **p;
p = (int **)malloc( N * sizeof(int) );
for (i=0; i<N; i++) {
    p[i] = (int *)malloc( M * sizeof(int) );
}
```

Overwriting Memory

- Off-by-one error

```c
int **p;
p = (int **)malloc( N * sizeof(int *) );
for (i=0; i<=N; i++) {
    p[i] = (int *)malloc( M * sizeof(int) );
}
```

Overwriting Memory

- Not checking the max string size

```c
char s[8];
int i;
gets(s); /* reads “123456789” from stdin */
```

- Basis for classic buffer overflow attacks
  - Your lab assignment #3

Overwriting Memory

- Misunderstanding pointer arithmetic

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

- Referencing a pointer instead of the object it points to
  ```c
  int *getPacket(int **packets, int *size) {
    int *packet;
    packet = packets[0];
    packets[0] = packets[*size - 1];
    *size--; // what is happening here?
    reorderPackets(packets, *size);
    return(packet);
  }
  ```

- ‘-’ and ‘*’ operators have same precedence and associate from right-to-left, so -- happens first!

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns
  ```c
  int *foo () {
    int val;
    return &val;
  }
  ```

Freeing Blocks Multiple Times

- Nasty!
  ```c
  x = (int *)malloc( N * sizeof(int) );
  <manipulate x>
  free(x);
  ...
  
y = (int *)malloc( M * sizeof(int) );
  <manipulate y>
  free(x);
  ```

- What does the free list look like?
  ```c
  x = (int *)malloc( N * sizeof(int) );
  <manipulate x>
  free(x);
  free(x);
  ```
Referencing Freed Blocks

- Evil!

```c
x = (int *)malloc( N * sizeof(int) );
<manipulate x>
free(x);
```

```c
y = (int *)malloc( M * sizeof(int) );
for (i=0; i<M; i++)
    y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

- Slow, silent, long-term killer!

```c
foo() {
    int *x = (int *)malloc(N*sizeof(int));
    ...
    return;
}
```

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```c
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head =
        (struct list *)malloc( sizeof(struct list) );
    head->val = 0;
    head->next = NULL;
    <create and manipulate the rest of the list>
    ...
    free(head);
    return;
}
```

Dealing With Memory Bugs

- Conventional debugger (gdb)
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs

- Debugging malloc (UToronto CSRI malloc)
  - Wrapper around conventional malloc
  - Detects memory bugs at malloc and free boundaries
    - Memory overwrites that corrupt heap structures
    - Some instances of freeing blocks multiple times
    - Memory leaks
  - Cannot detect all memory bugs
    - Overwrites into the middle of allocated blocks
    - Freeing block twice that has been reallocated in the interim
    - Referencing freed blocks
Dealing With Memory Bugs (cont.)

- Some malloc implementations contain checking code
  - Linux glibc malloc: `setenv MALLOC_CHECK_2`
  - FreeBSD: `setenv MALLOC_OPTIONS AJR`
- Binary translator: `valgrind` (Linux), Purify
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Can detect all errors as debugging `malloc`
  - Can also check each individual reference at runtime
    - Bad pointers
    - Overwriting
    - Referencing outside of allocated block

What about Java or ML or Python or ...?

- In memory-safe languages, most of these bugs are impossible.
  - Cannot perform arbitrary pointer manipulation
  - Cannot get around the type system
  - Array bounds checking, null pointer checking
  - Automatic memory management
- But one of the bugs we saw earlier is possible. Which one?

Memory Leaks with GC

- not because of forgotten `free()` -- we have GC!
- unneeded “leftover” roots keep objects reachable
- Sometimes nullifying a variable is not needed for correctness but is for performance.
- Bigger issue with reference counting GC.

```
Root nodes

Heap nodes

reachable
Not-reachable (garbage)
```