Memory-Related Perils and Pitfalls in C

		Slide	Program stop possible?	Fixes:
A)	Dereferencing a non-pointer			
B)	Freed block – access again			
C)	Freed block – free again			
D)	Memory leak – failing to free memory			
E)	No bounds checking			
F)	Reading uninitialized memory			
G)	Referencing nonexistent variable			
H)	Wrong allocation size			

Find That Bug! (Slide 2)

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

Error Type: ☐ Prog stop Possible? ☐ Fix:

Find That Bug! (Slide 3)

```
int* foo() {
    int val = 0;

    return &val;
}
```

Error Type: ☐ Prog stop Possible? ☐ Fix:

Find That Bug! (Slide 4)

```
int **p;

p = (int **)malloc( N * sizeof(int) );

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

- `N` and `M` defined elsewhere (`#define`)

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug! (Slide 5)

```
/* 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;
}
```

- `A` is `N`x`N` matrix, `x` is `N`-sized vector (so product is vector of size `N`)
- `N` defined elsewhere (`#define`)

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug! (Slide 6)

- The classic `scanf` bug
  - `int scanf(const char *format)`

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

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug! (Slide 7)

```
x = (int*)malloc( N * sizeof(int) );
    // manipulate x
free(x);

    ...

y = (int*)malloc( M * sizeof(int) );
    // manipulate y
free(x);
```

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug! (Slide 8)

```
x = (int*)malloc( N * sizeof(int) );
   // manipulate x
free(x);

   ...

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

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Find That Bug! (Slide 9)

```
typedef struct L {
    int val;
    struct L *next;
} list;

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

Error Type: ☐ **Prog stop Possible?** ☐ **Fix:**

Dealing With Memory Bugs

Non-testable Material

- 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.)

Non-testable Material

- 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