Memory Related Pitfalls

Common Memory Related Issues

Credit to Dr. Robucci's CMPE311 course for all slide content

Memory-Related Perils and Pitfalls

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

Using * and ++ together ptrplusplus.c

- Avoid using * and ++ in the same expression.
- What's the difference among
  - *ptr++
    - *ptr++ dereferences ptr, then increments ptr
  - (*ptr)++
    - (*ptr)++ performs a post-increment on what ptr points to
  - ++*ptr
    - ++*ptr increments ptr, then dereferences ptr
  - ++(*ptr)
    - ++(*ptr) performs a pre-increment on what ptr points to

Dereferencing Bad Pointers

- The classic scanf bug is to pass variable itself instead of an address
  - Typically reported as an error by the compiler.

```
int val;

...

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

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero is wrong, see calloc if needed.

```
/* return y = A times x */
int *matvec(int A[N][N], int x[N]) {
   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;
}
```

Overwriting Memory

- Allocating the (possibly) wrong sized object

```
int i, **p;

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

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

- Here, second line should have been sizeof(int *).

Overwriting Memory

- Not checking the max string size

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

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

- Basis for classic buffer overflow attacks
  - 1988 Internet worm
  - Modern attacks on Web servers
  - AOL/Microsoft IM war

Overwriting Memory

- Misunderstanding pointer arithmetic

```
int *search(int *p, int val) {

   while (*p != NULL && *p != val)
      p += sizeof(int);

   return p;
}
```

- Remember,
  p+=N;
  already adds N times the sizeof (int 8) to p

Referencing Nonexistent Variables

- Another error I commonly seen is returning a pointer to a local variable (the variable's lifetime ceases at the end of the function and pointer is not safe to use.)
- Forgetting that local variables disappear when a function returns

```
int * sum(int a, int b){
  int c = a + b;
  return &c;
}
```

Freeing Blocks Multiple Times

- Nasty!

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

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

- Note: Considering setting pointers to NULL after deallocation as a bookkeeping measure. Later, you can check if a pointer is NULL before using it. Deallocating a NULL pointer has no effect.

Referencing Freed Blocks

- Evil!

```
x = malloc(N * sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M * sizeof(int));
for (i = 0; i < M; i++)
   y[ i ] = x[ i ]++;
```

Note: Considering setting pointers to NULL after deallocation as a bookkeeping measure. Later, you can check if a pointer is NULL before using it. Deallocating a NULL pointer has no effect.

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```
int foo(int x,int y, int z) {
    int result;
    int *p = malloc(n * sizeof(int));
    …
    //forget to free p

    return result;
}
```

- Here, a function allocated memory and tracked it with a pointer which doesn't exist after the function. The memory is no longer tracked by the program but is left allocated. Over time the system may run out of memory.

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
typedef struct list {
   char * nameString;
   struct list *next;
} LIST_t;
foo() {
   struct list *head = malloc(sizeof(LIST_t));
   head->val = malloc((NAMESIZE+1)*sizeof(char));
   head->next = NULL;
   <create and manipulate the rest of the list>
free(head); //only use of free
   return;
}
```

- There are multiple problems here. Freeing head only deallocated a small block of memory consisting of two pointers. The nameString block was not deallocated nor was the rest of the linked list structure.

Dealing With Memory Bugs

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

- Some malloc implementations contain checking code
  - Linux glibc malloc: setenv MALLOC_CHECK_ 2

Dealing With Memory Bugs (cont.)

- Binary translator: valgrind (Linux)
  - 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
- Garbage collection (Boehm-Weiser Conservative GC)
  - Let the system free blocks instead of the programmer.