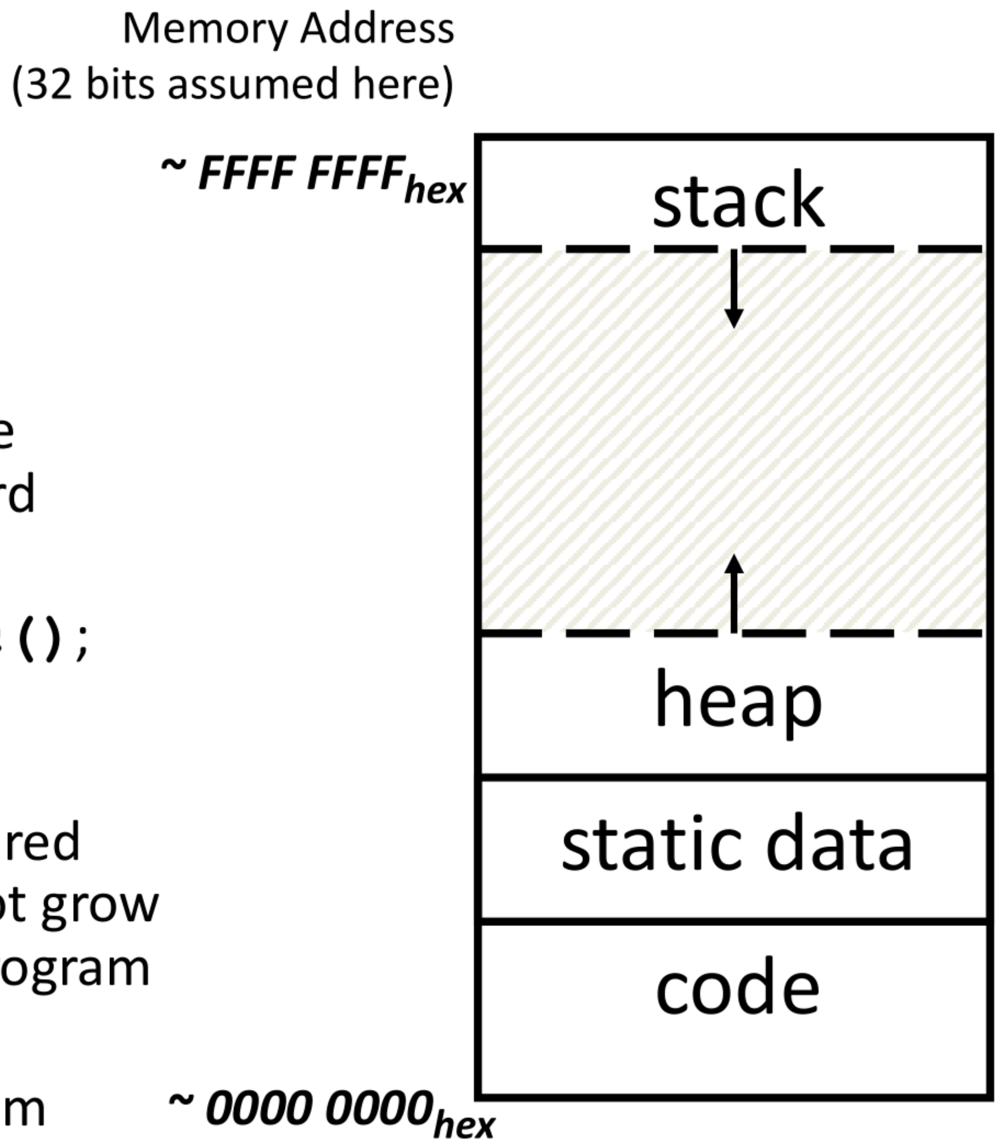
CA Discussion 3

Ziyi Yu　俞子逸

C Memory Management

- Program's *address space* contains 4 regions:
  - stack: local variables inside functions, grows downward
  - heap: space requested for dynamic data via **`malloc()`**; resizes dynamically, grows upward
  - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code: loaded when program starts, does not change



```c
#define MAX_NAME_LEN 50
int num_people = 0;
void add_people(char **list){
    char name2[] = "Van";
    list[num_people] = calloc(MAX_NAME_LEN, sizeof(char));
    strcpy(list[num_people], name2);
    num_people += 1;
}
int main(){
    const int list_size = 100;
    char **name_list = malloc(sizeof(char *) * list_size);
    char *name1 = "Billy";
    add_people(name_list);
    add_people(name_list);
    return 0;
```

4 (a) Fill in <, >, = or can't decide for these four questions based on what the given C expressions evaluate to. You cannot assume malloc return heap address sequentially in C standard.

name_list ______________ &list_size

&name_list ______________ &num_people

name_list[1] ______________ name_list

&name1 ______________ &list

Solution: 1. <
2. >
3. can't decide
4. >

3 (b) Fill in static, stack, heap or code for these three questions according to their address type in memory.

name1 ______________________

*name_list ______________________

&(name2[1]) ______________________

Solution: 1. static
2. heap
3. stack

Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time

sizeof() **v.s.** strlen()

```c
#include <stdio.h>
#include <string.h>

int main(){
    /* I am using a 64bit system */
    char* ptrStr = "abcde";
    char listStr1[] = "abcde";
    char listStr2[10] = "abcde";

    printf("%d\n",sizeof(ptrStr));          /* sizeof: 8, a ptr takes 8 bytes, 64bit! */
    printf("%d\n",strlen(ptrStr));          /* strlen: 5 */

    printf("%d\n",sizeof(ptrStr+2));        /* sizeof: 8, a ptr takes 8 bytes, 64bit! */
    printf("%d\n",sizeof(*(ptrStr+2)));     /* sizeof: 1 */
    printf("%d\n",strlen(ptrStr+2));        /* strlen: 3 */


    printf("%d\n",sizeof(listStr1));        /* sizeof: 6 */
    printf("%d\n",strlen(listStr1));        /* strlen: 5 */

    printf("%d\n",sizeof(listStr2));        /* sizeof: 10 */
    printf("%d\n",strlen(listStr2));        /* strlen: 5 */

    return 0;
}
```

```c
#include <libc.h>

/* Takes a string and makes it awesome! */
int make_ca(char * str, size_t length){

    char awesome[] = "CA is so awesome!";

    /* if str is too small we need to get more memory! */
    if(length < strlen(awesome) ){
        str = malloc(sizeof(char) * strlen(awesome));
    }

    strcpy(str, awesome);
}

int main(int argc, char *argv[]){

    char ca[] = "CA is OK.";
    char * CA = malloc(6);
    memcpy(CA, ca, strlen(ca));

    make_ca(ca, strlen(ca));
    make_ca(CA, strlen(CA));
    /* We want to print an awesome string! */
    printf(" %s %s ",ca, CA);

}
```

Bugs

- Line 9: comparison with strlen instead of sizeof (for 0-terminator)
- Line 10: strlen instead of sizeof (or +1) for malloc =>
  - Line 13: write past end of array (if malloc was used)
- Line 4: Ownership of pointer str not clear =>
  - Line 10: Potential memory leak
- Line 4: New pointer is not returned/ no pointer to pointer is used
- Line 20: memcpy over length of CA
- Line 20: 0-terminator is not copied!
- Line 22 &23: better: call with array size
- Line 14 & 27: return missing!

Agenda

- Pointers
- Pointers & Arrays
- C Memory Management
- C Bugs

Common Memory Problems

- Using uninitialized values
- Using memory that you don’t own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Using Memory You Don't Own

- What is wrong with this code?
- Using pointers beyond the range that had been malloc'd
  - –May look obvious, but what if mem refs had been result of pointer arithmetic that erroneously took them out of the allocated range?

```
int *ipr, *ipw;
void ReadMem() {
      int i, j;
      ipr = (*int) malloc(4 * sizeof(int));
      i = *(ipr - 1000); j = *(ipr + 1000);
      free(ipr);
   }
void WriteMem() {
      ipw = (*int) malloc(5 * sizeof(int));
      *(ipw - 1000) = 0; *(ipw + 1000) = 0;
      free(ipw);
   }
```

Faulty Heap Management

- What is wrong with this code?

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  …
  free(pi);
}

void main() {
  pi = malloc(4*sizeof(int));
  foo();
  …
}
```

Faulty Heap Management

- Memory leak: *more mallocs than frees*

```
int *pi;
void foo() {
  pi = malloc(8*sizeof(int));
  /* Allocate memory for pi */
  /* Oops, leaked the old memory pointed to by pi */
  …
  free(pi); /* foo() is done with pi, so free it */
}

void main() {
  pi = malloc(4*sizeof(int));
  foo(); /* Memory leak: foo leaks it */
  …
}
```

Faulty Heap Management

- What is wrong with this code?

```
int *plk = NULL;
void genPLK() {
        plk = malloc(2 * sizeof(int));
  … … …
        plk++;
}
```

Faulty Heap Management

- Potential memory leak – handle has been changed, do you still have copy of it that can correctly be used in a later free?

```
int *plk = NULL;
void genPLK() {
        plk = malloc(2 * sizeof(int));
  … … …
        plk++;
}
```

Faulty Heap Management

- What is wrong with this code?

```
void FreeMemX() {
        int fnh = 0;
        free(&fnh);
}

void FreeMemY() {
        int *fum = malloc(4 * sizeof(int));
        free(fum+1);
        free(fum);
        free(fum);
}
```

Faulty Heap Management

- Can’t free non-heap memory; Can’t free memory that hasn’t been allocated

```
void FreeMemX() {
        int fnh = 0;
        free(&fnh);
}

void FreeMemY() {
        int *fum = malloc(4 * sizeof(int));
        free(fum+1);
        free(fum);
        free(fum);
}
```

Using Memory You Haven’t Allocated

- What is wrong with this code?

```
void StringManipulate() {
        const char *name = “Safety Critical";
        char *str = malloc(10);
        strncpy(str, name, 10);
        str[10] = '\0';
        printf("%s\n", str);
}
```

Using Memory You Haven't Allocated

- Reference beyond array bounds

```
void StringManipulate() {
       const char *name = “Safety Critical";
       char *str = malloc(10);
       strncpy(str, name, 10);
       str[10] = '\0';
  /* Write Beyond Array Bounds */
       printf("%s\n", str);
  /* Read Beyond Array Bounds */
}
```

Using Memory You Don’t Own

- What’s wrong with this code?

```
char *append(const char* s1, const char *s2) {
  const int MAXSIZE = 128;
  char result[128];
  int i=0, j=0;
  for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
  }
  for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
  }
  result[++i] = '\0';
  return result;
}
```

Using Memory You Don't Own

- Beyond stack read/write

```
char *append(const char* s1, const char *s2) {
   const int MAXSIZE = 128;
   char result[128];
   int i=0, j=0;
   for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
          result[i] = s1[j];
   }
   for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
          result[i] = s2[j];
   }
   result[++i] = '\0';
   return result;
}
```

`result` is a local array name – stack memory allocated

Function returns pointer to stack memory – won't be valid after function returns

Using Memory You Don't Own

- What is wrong with this code?

```
typedef struct node {
        struct node* next;
        int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
            head = head->next;
    }
    return head->val;
}
```

Using Memory You Don't Own

- Following a NULL pointer to mem addr 0!

```
typedef struct node {
        struct node* next;
        int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
            head = head->next;
    }
    return head->val;
}
```

Managing the Heap

- `realloc(p,size)`:
  - Resize a previously allocated block at `p` to a new `size`
  - If `p` is `NULL`, then `realloc` behaves like `malloc`
  - If `size` is `0`, then `realloc` behaves like `free`, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

  E.g.: allocate an array of 10 elements, expand to 20 elements later

```
int *ip;
ip = (int *) malloc(10*sizeof(int));
/* always check for ip == NULL */
… … …
ip = (int *) realloc(ip,20*sizeof(int));
/* always check for ip == NULL */
/* contents of first 10 elements retained */
… … …
realloc(ip,0); /* identical to free(ip) */
```

Using Memory You Don't Own

- What is wrong with this code?

```
int* init_array(int *ptr, int new_size) {
  ptr = realloc(ptr, new_size*sizeof(int));
  memset(ptr, 0, new_size*sizeof(int));
  return ptr;
}

int* fill_fibonacci(int *fib, int size) {
  int i;
  init_array(fib, size);
  /* fib[0] = 0; */ fib[1] = 1;
  for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
  return fib;
}
```

Using Memory You Don't Own

- Improper matched usage of mem handles

```
int* init_array(int *ptr, int new_size) {
   ptr = realloc(ptr, new_size*sizeof(int));
   memset(ptr, 0, new_size*sizeof(int));
   return ptr;
}
```

Remember: `realloc` may move entire block

```
int* fill_fibonacci(int *fib, int size) {
   int i;
   /* oops, forgot: fib = */ init_array(fib, size);
   /* fib[0] = 0; */ fib[1] = 1;
   for (i=2; i<size; i++)
            fib[i] = fib[i-1] + fib[i-2];
   return fib;
}
```

What if array is moved to new location?

And In Conclusion, …

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it
- Pointer is a C version (abstraction) of a data address
  - `*` "follows" a pointer to its value
  - `&` gets the address of a value
  - Arrays and strings are implemented as variations on pointers
- C is an efficient language, but leaves safety to the programmer
  - Variables not automatically initialized
  - Use pointers with care: they are a common source of bugs in programs

And In Conclusion, …

- C has three main memory segments in which to allocate data:
    - Static Data: Variables outside functions
    - Stack: Variables local to function
    - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code