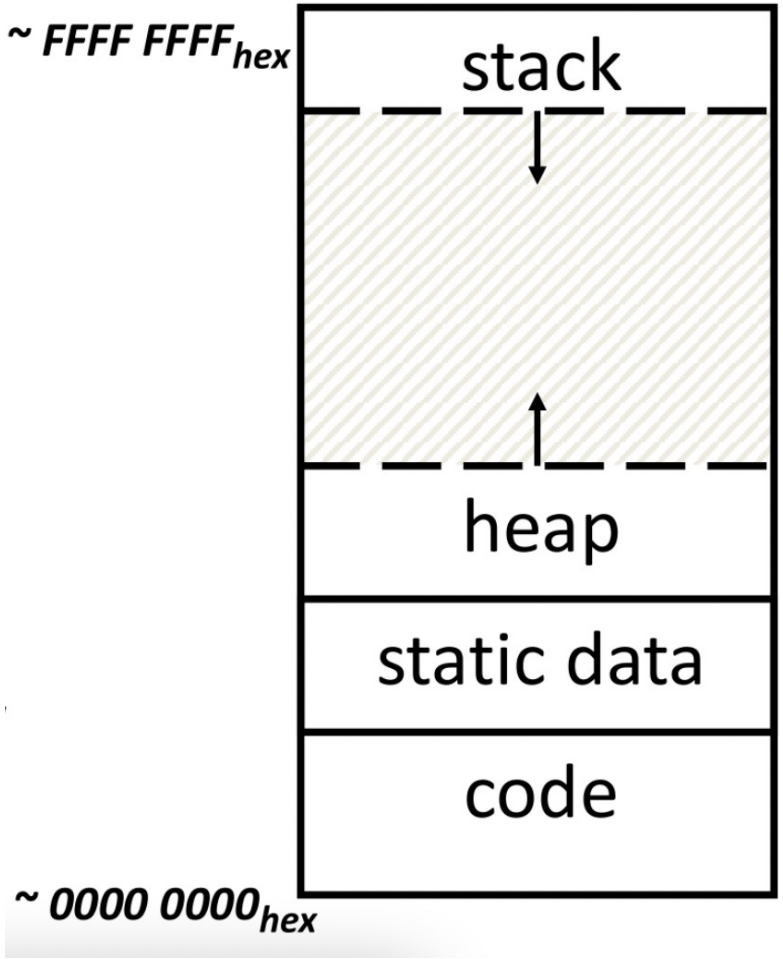
CS110-Discussion 3

Tianyuan Wu, ShanghaiTech University

Memory management in C

- Program's address space contains 4 regions
  - Stack: local variables inside functions, grow downwards.
  - Heap: Space for dynamic data, requested via "malloc", grows upwards.
  - Static data: Variables defined outside functions, does not grow or shrink. But can be modified.
  - Code: Loaded when program starts. Can not be modified.



Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  static const int year = 2019;
4
5  int main (void) {
6      char name[] = "Jose";
7      char *game = "The Elder Scrolls";
8      int *ver = malloc(sizeof(int));
9      *ver = 5;
10     /* Break! */
11     printf("Until %d, %s's favourite game is
           %s %d.\n", year, name, game, *ver);
12     return 0;
13 }
```

Expressions:

&year	_____
name	_____
game	_____
ver	_____
&ver	_____

Solution

- &year: static
- name: stack
- game: static
- ver: heap
- &ver: 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() **vs.** 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;
}
```

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);
   }
```

Using Memory You Don’t Own

- NULL pointer issues

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

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

Memory Leaks

- 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 */
  …
}
```

Potential Memory Leaks

- 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++;
}
```

Misuse of `free()`

- 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);
}
```

Finding Bugs

```
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;
}
```

Bugs

```
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

Finding Bugs!

```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!

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!
- `calloc(p,size)`:
  - Allocation with initialization: malloc() + memset()

Finding Bugs!

```
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;
}
```

Bugs

- Improper matched usage of memory 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;
}

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;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?

Can we do better?

- Many language implements more advanced MM
- C++: Smart pointers, RAII, …

Can we do better?

Pointer categories

`unique_ptr` (C++11)	smart pointer with unique object ownership semantics (class template)
`shared_ptr` (C++11)	smart pointer with shared object ownership semantics (class template)
`weak_ptr` (C++11)	weak reference to an object managed by `std::shared_ptr` (class template)
`auto_ptr` (deprecated in C++11) (removed in C++17)	smart pointer with strict object ownership semantics (class template)

Helper classes

`owner_less` (C++11)	provides mixed-type owner-based ordering of shared and weak pointers (class template)
`enable_shared_from_this` (C++11)	allows an object to create a `shared_ptr` referring to itself (class template)
`bad_weak_ptr` (C++11)	exception thrown when accessing a `weak_ptr` which refers to already destroyed object (class)
`default_delete` (C++11)	default deleter for `unique_ptr` (class template)

Smart pointer adaptors

`out_ptr_t` (C++23)	interoperates with foreign pointer setters and resets a smart pointer on destruction (class template)
`out_ptr` (C++23)	creates an `out_ptr_t` with an associated smart pointer and resetting arguments (function template)
`inout_ptr_t` (C++23)	interoperates with foreign pointer setters, obtains the initial pointer value from a smart pointer, and resets it on destruction (class template)
`inout_ptr` (C++23)	creates an `inout_ptr_t` with an associated smart pointer and resetting arguments (function template)

Can we do better?

- Rust: Ownerships – References and Borrowing

Filename: src/main.rs

```rust
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Can we do better?

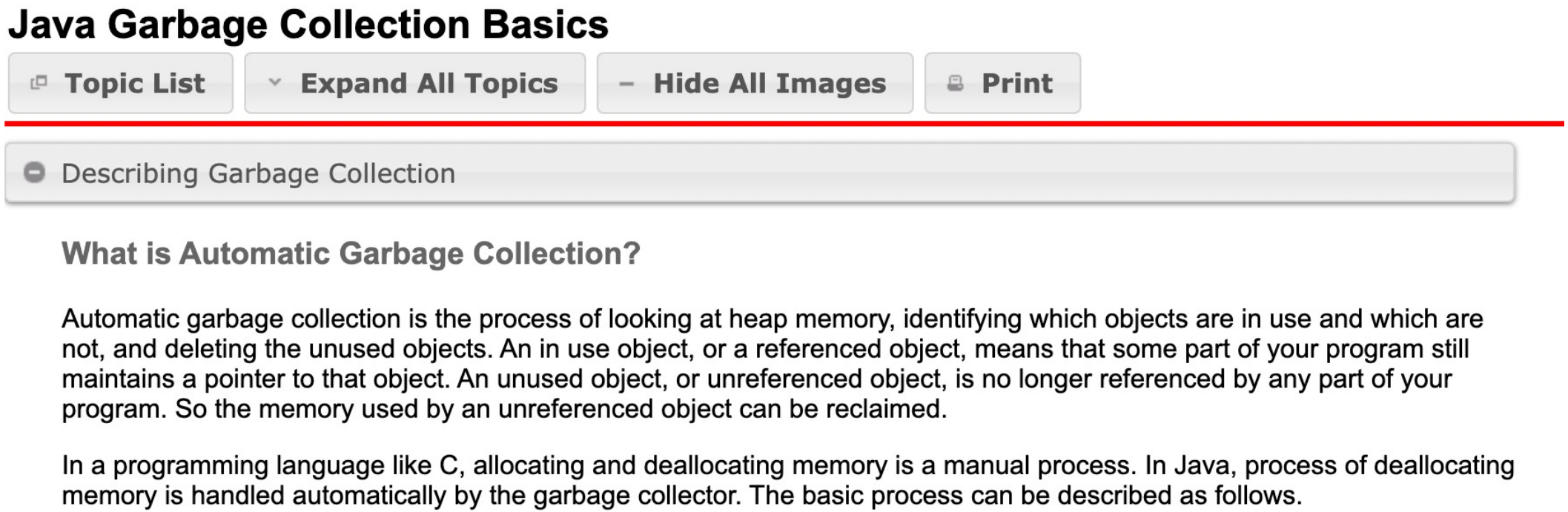
- Rust: Ownerships – References and Borrowing

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- help: consider changing this to be a mutable reference
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be

For more information about this error, try `rustc --explain E0596`.
error: could not compile `ownership` due to previous error
```

Can we do better?

- Java, Python,…: Garbage collection



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

Conclusion Cont’d

- 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

Thanks!