Memory Allocation

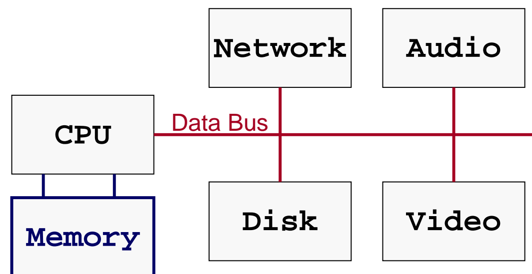
CS 217

Memory Allocation

- Good programmers make efficient use of memory
- Understanding memory allocation is important
  - Create data structures of arbitrary size
  - Avoid "memory leaks"
  - Run-time performance

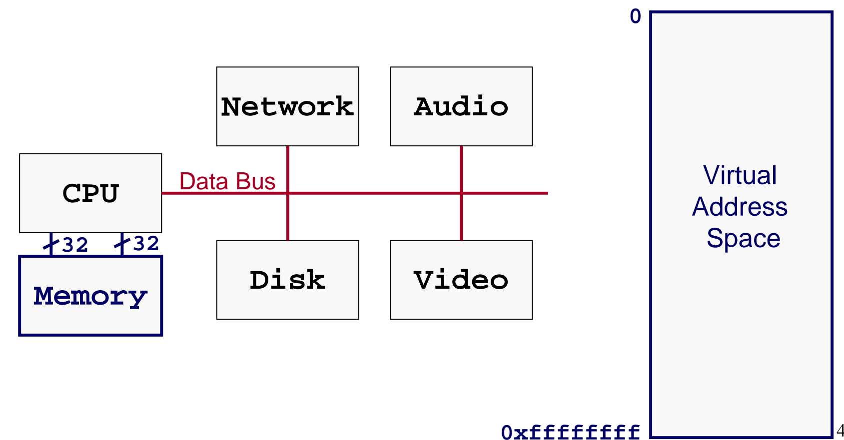
Memory

- What is memory?
  - Storage for variables, data, code, etc.



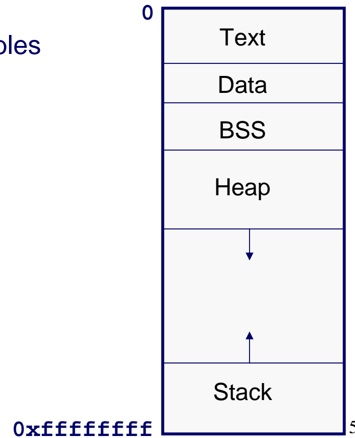
Memory

- What is memory?
  - Storage for variables, data, code, etc.
  - Unix provides virtual memory



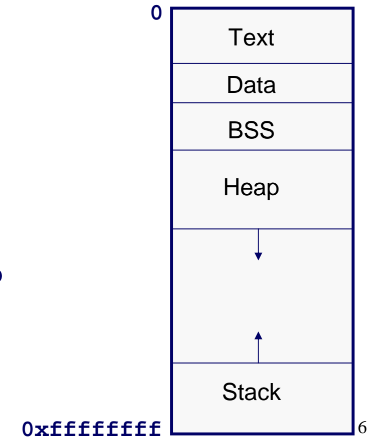
Memory Layout

- How is memory organized?
  - Text = code, constant data
  - Data = initialized global and static variables
  - BSS = (Block Started by Symbol) uninitialized (zero) global & static variables
  - Stack = local variables
  - Heap = dynamic memory



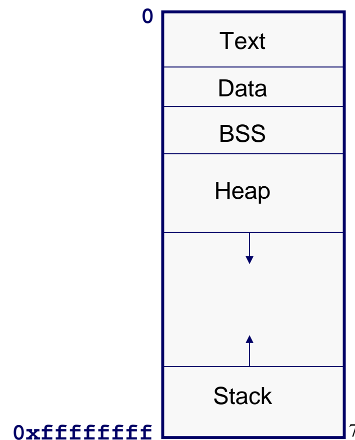
Memory Layout

```
char string = "hello"
int iSize;

char *f(void)
{
    char *p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```

Memory Allocation

- How is memory allocated?
  - Global and static variables = program startup
  - Local variables = function call
  - Dynamic memory = malloc()



Memory Allocation

```
int iSize;                   ← allocated in BSS, set to zero at startup

char *f(void)
{
    char *p;                 ← allocated on stack at start of function f
    iSize = 8;
    p = malloc(iSize);       ← 8 bytes allocated in heap by malloc
    return p;
}
```

Memory Deallocation

- How is memory deallocated?
  - Global and static variables = program finish
  - Local variables = function return
  - Dynamic memory = free()
- All memory is deallocated at program termination
  - It is good style to free allocated memory anyway

Memory Deallocation

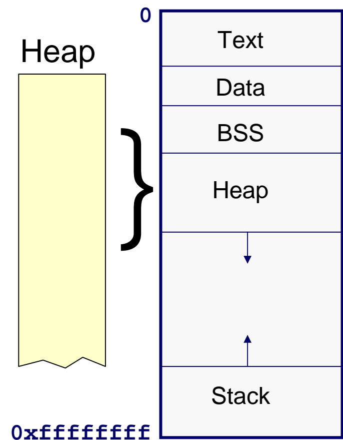
```
int iSize;                  ←— available until program termination

char *f(void)
{
    char *p;                ←— deallocated by return from function f
    iSize = 8;
    p = malloc(iSize);      ←— deallocate by calling free(p)
    return p;
}
```


Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

`size_t` is a typedef for an appropriate-sized unsigned int, e.g., `typedef unsigned size_t`

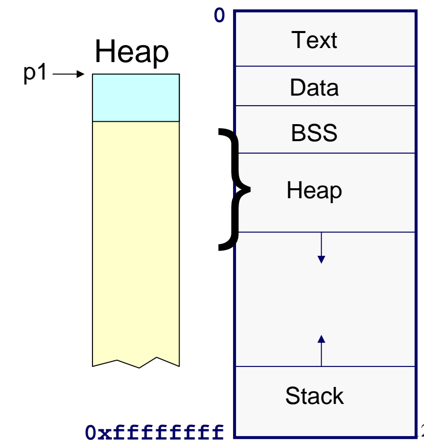
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

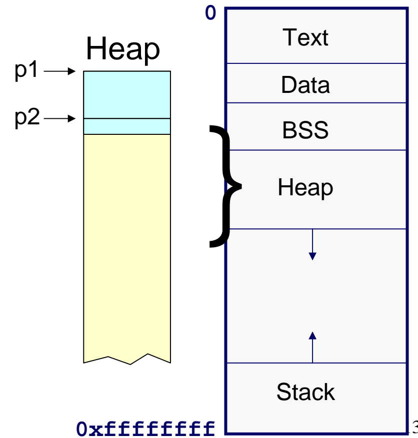
```
➡ char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

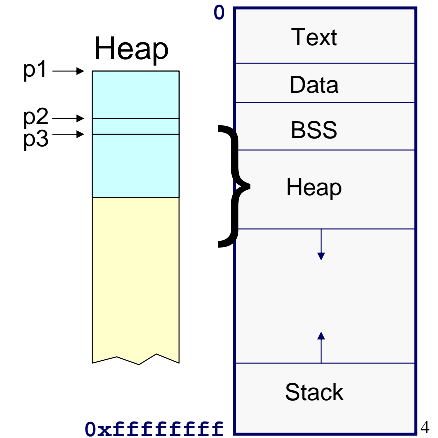
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

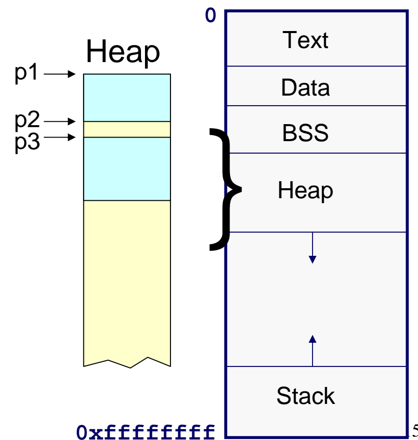
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

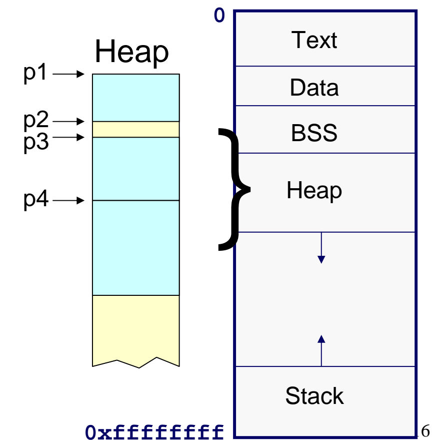
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

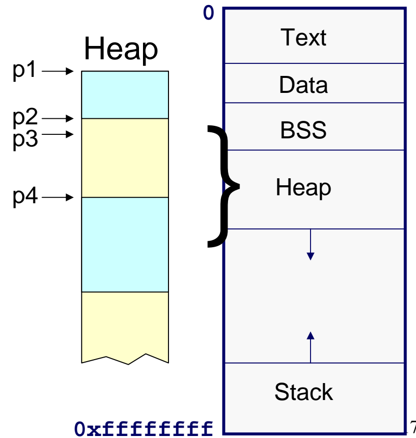
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

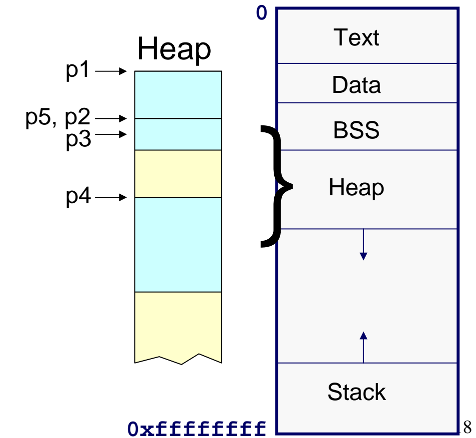
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

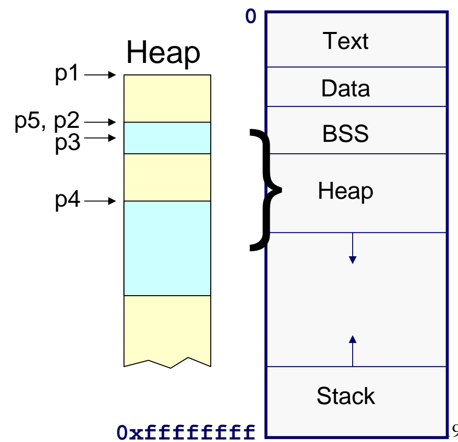
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

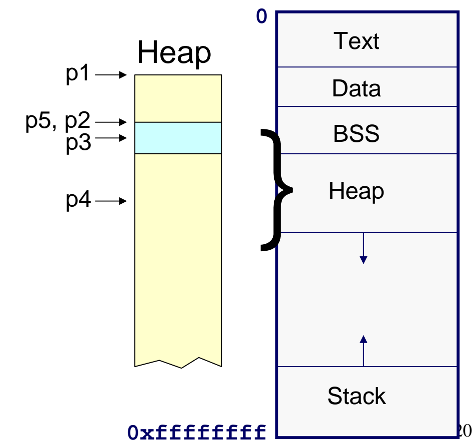
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

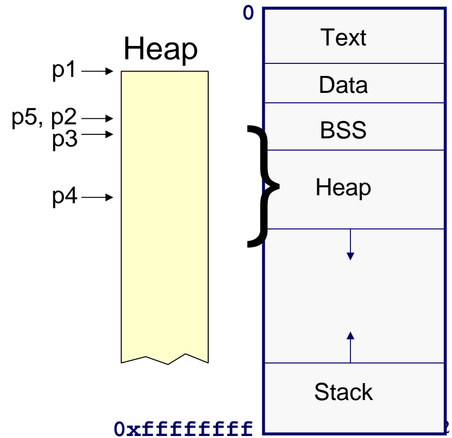
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



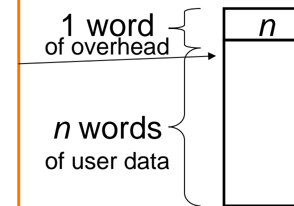
0
Text
Data
BSS
Heap
Stack
0xffffffff

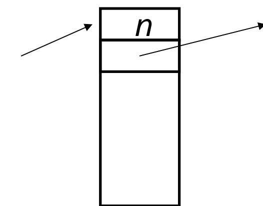
Memory allocator ADT

- Malloc & free are the operations of an ADT
  - How do they work inside?
- First answer: it's an ADT, you're not supposed to ask!
- Second answer:

malloc(s)
n = ⌈s / sizeof(int)⌉

free(p)
put p into linked list of free objects



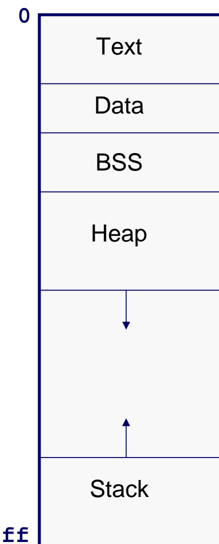


Dangling pointers

- Dangling pointers point to data that's not there anymore
- Avoid dangling pointers!
- Example:

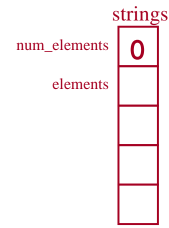
Example Code I

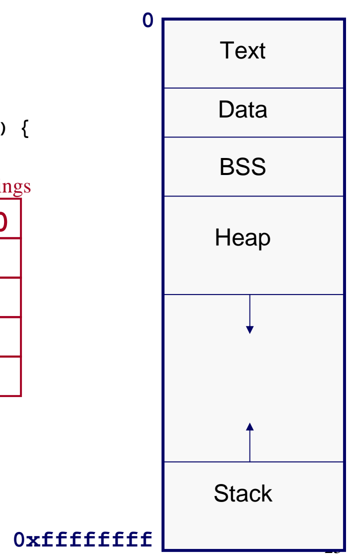
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Example Code I

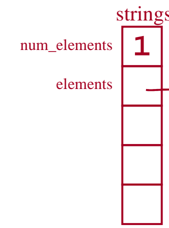
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```

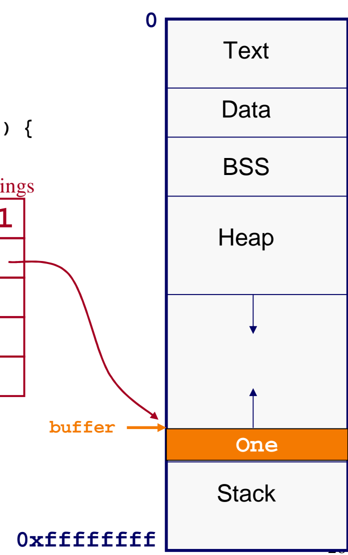




Example Code I

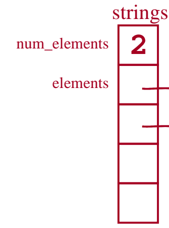
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```

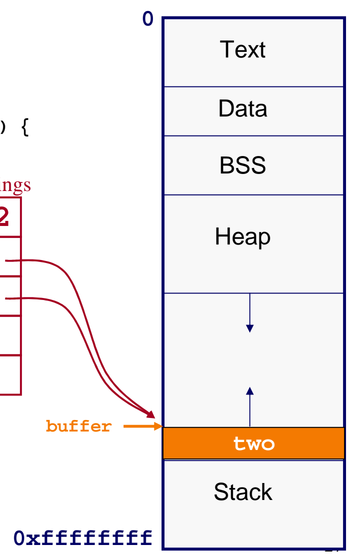




Example Code I

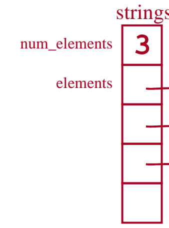
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```

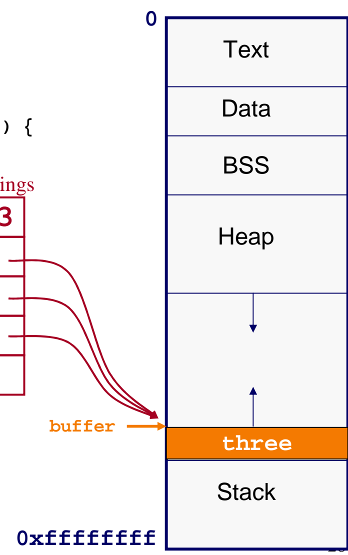




Example Code I

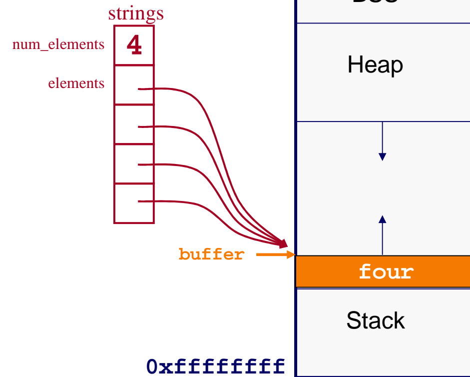
```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```





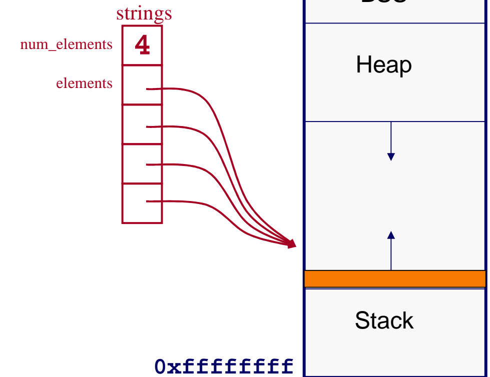
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



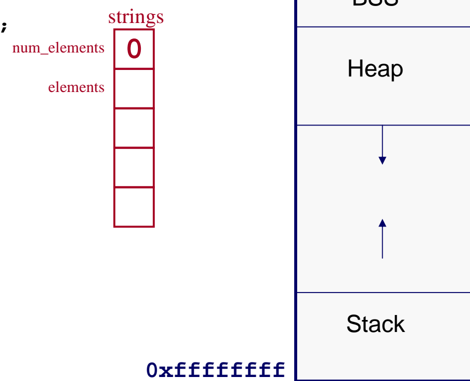
Example Code I

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    Array_insert(strings, buffer);
  }
}

...

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



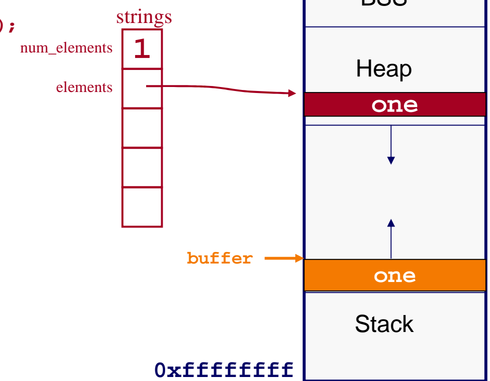
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



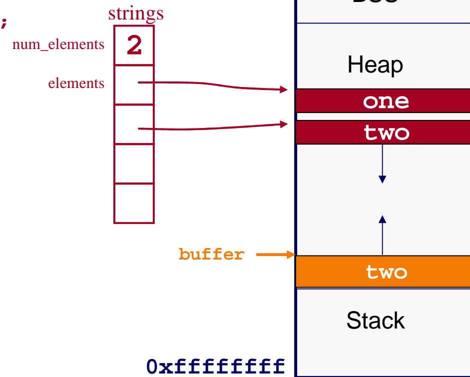
Example Code II

```
...

void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



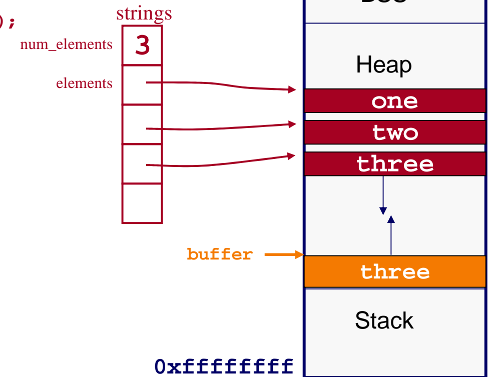
Example Code II

```
...
void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



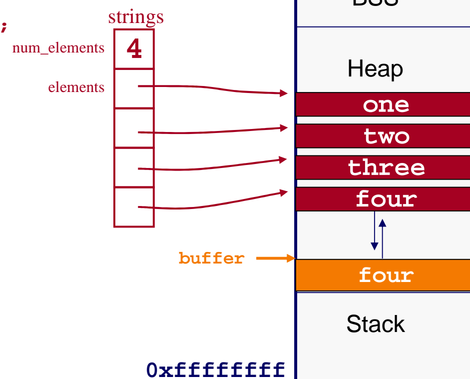
Example Code II

```
...
void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



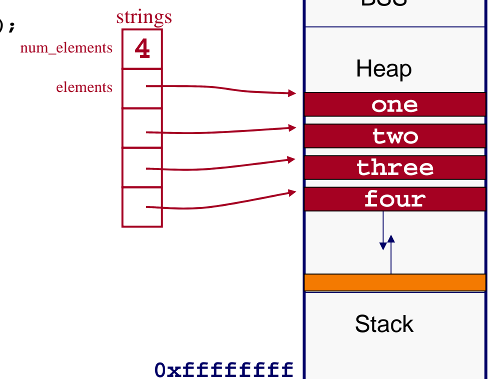
Example Code II

```
...
void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer+1));
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Example Code II

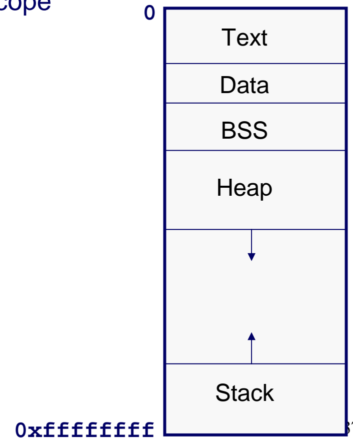
```
...
void ReadStrings(Array_T strings, FILE *fp)
{
  char buffer[MAX_STRING_LENGTH];
  while (fgets(buffer, MAX_STRING_LENGTH, fp)) {
    char *string = malloc(strlen(buffer)+1);
    strcpy(string, buffer);
    Array_insert(strings, string);
  }
}

int main()
{
  Array_T strings = Array_new();

  ReadStrings(strings, stdin);
  SortStrings(strings, strcmp);
  WriteStrings(strings, stdout);

  Array_free(strings);

  return 0;
}
```



Static Local Variables

- `static` keyword in declaration of local variable means:
  - Available (if within scope) throughout entire program execution
  - Variable is allocated from Data or BSS, not stack
  - Acts like global variable with limited scope

```
int iSize;

char *f(void)
{
    static int first = 1;
    if (first) {
      iSize = GetSize();
      first = 0;
    }
    ...
}
```



Memory Initialization

- Local variables have undefined values

```
int count;
```

- Memory allocated by malloc has undefined values

```
char *p = malloc(8);
```

- If you need a variable to start with a particular value, use an explicit initializer

```
int count = 0;
p[0] = '\0';
```

- Global and static variables are initialized to 0 by default

```
static int count = 0;
```

is the same as

```
static int count;
```

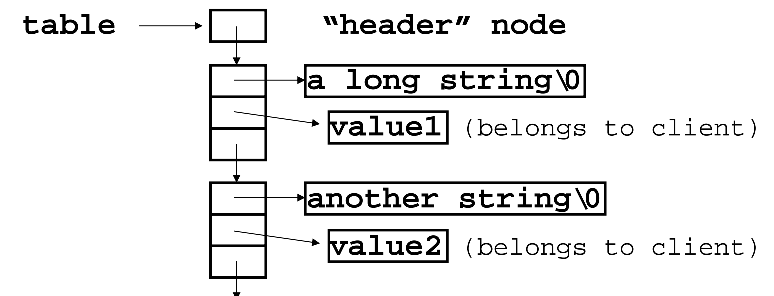
It is bad style to depend on this

Summary

- Three types of memory
  - Global and static variables = BSS
  - Local variables = stack
  - Dynamic memory = heap
- Three types of allocation/deallocation strategies
  - Global and static variables (BSS) = program startup/termination
  - Local variables (stack) = function entry/return
  - Dynamic memory (heap) = malloc()/free()
- Take the time to understand the differences!

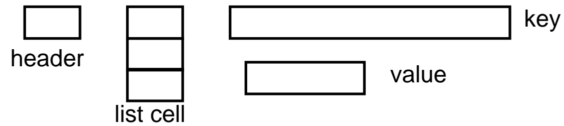
ADT Implementation

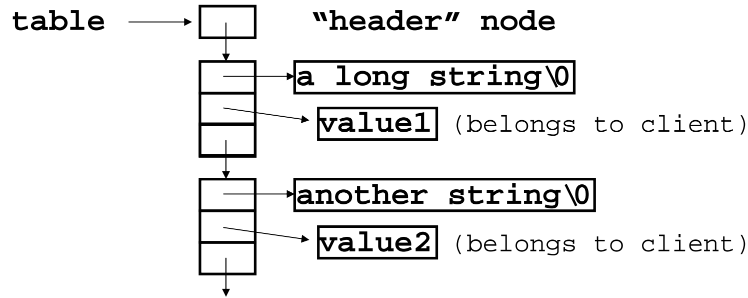
- Recall the simple implementation of the symtable ADT:



Memory Management Issues

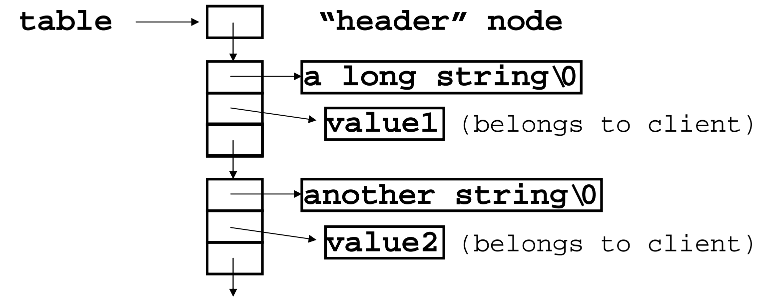
- Does ADT or client "own" the data?
  - Who mallocs/frees each kind of node?





Who Free What

- Who frees the list header and the list cells?



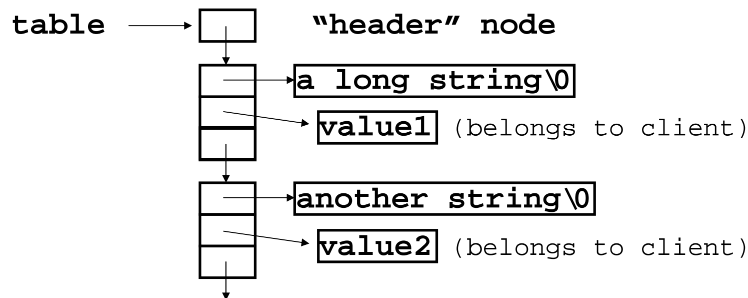
Who Free What

- What happens if,

```
{SymTable_T table; . . . free(table);}
```

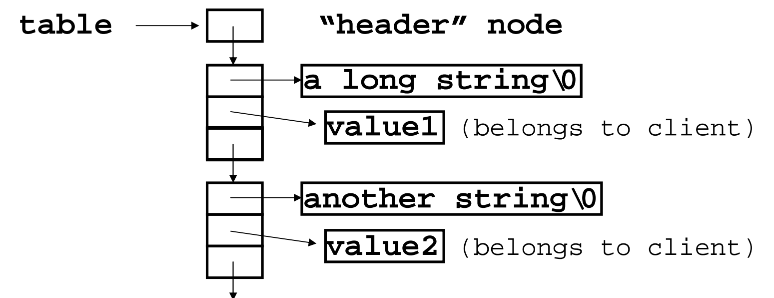
then the list cells don't get freed!

- So, ADT must "own" headers and list cells



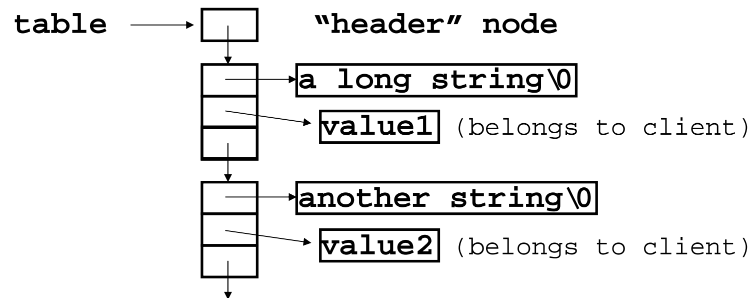
Who Free What, cont'd

- Who frees the value pointers?



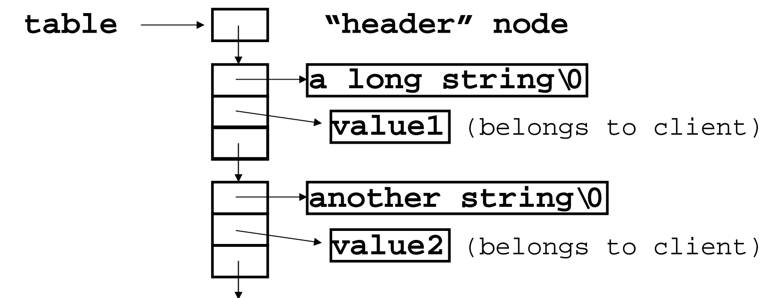
Who Free What, cont'd

- Who frees the value pointers?
- What's wrong with the ADT freeing the value pointers?



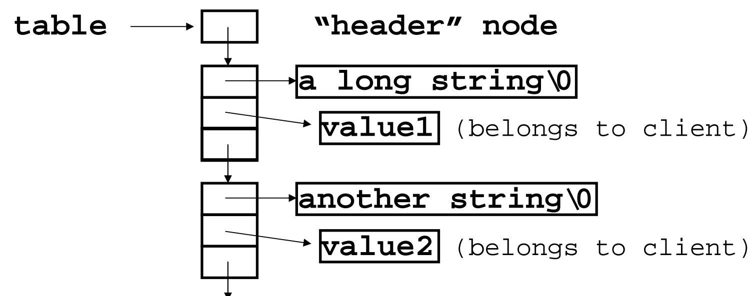
Who Free What, cont'd

- ADT just sees `void *value;`
- Value pointer might be root of big data structure, all the pieces need to be freed.
- Thus, client must "own" the value nodes.



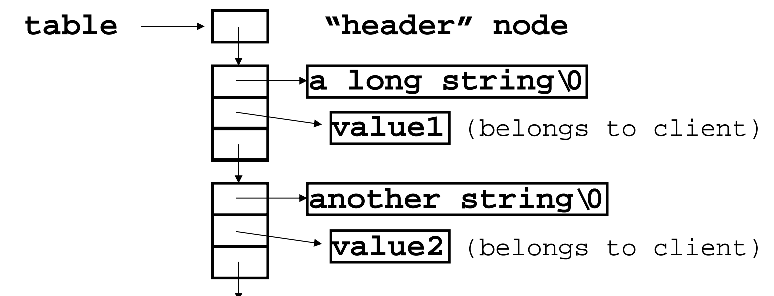
Who Owns The Key?

- Who frees keys?



Who Owns The Key?

- Both client and ADT "know" about `char *key;`
- Therefore, we are faced with a design choice
- Choice 1: client owns the key.
  - Consequence: must call SymTable_put only with a string that will last a long time. *(But our client didn't do that!)*

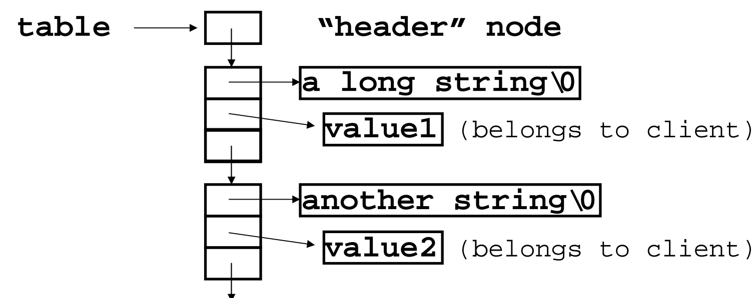


Previous Example Overwrites "line"

```
int main(int argc, char *argv[]) {
 char line[MAXLINE];
 SymTable_T table = SymTable_new();
 struct stats *v;
 while (fgets(line, MAXLINE, stdin)) {
   v = SymTable_get(table, line);
   if (!v) {
        v = makeStats(0);
        SymTable_put(table, line, v);
   }
 SymTable_map(table, maybeprint, NULL);
 return EXIT_SUCCESS;
}
```


Choice 2: ADT owns the key

- Consequence: `SymTable_put` must copy its `key` argument into a newly malloc'ed string object.



Put Away Your Toys…

- When client is done with a symbol table, it should give the memory back.
- But client can't call `free` directly (as we already demonstrated)
- So there must be an interface function for client to say "I'm done with this"
- It should free the header, list cells, strings
  `SymTable_free(SymTable_T table);`
- Should it free the values?
  - Can't do it by calling `free` directly (as we already demonstrated)
  - Another design choice!

Options to Free Values

- Option 1: Client frees all the values before calling `SymTable_free(table)`
  - Can do this using SymTable_map(table, free_it, NULL);
  - Minor bother: temporarily leaves dangling pointers in the table
  - Minor bother: it's clumsy
- Option 2: `SymTable_free` calls client function

```
void SymTable_free(SymTable_T table,
    void (*f)(char *key, void *value, void *extra),
    void *extra);
```

/* Free entire table. During this process, if f is not NULL, apply f to each binding in table. It is a checked runtime error for table to be NULL. */

- We will choose Option 1.