Memory Allocation

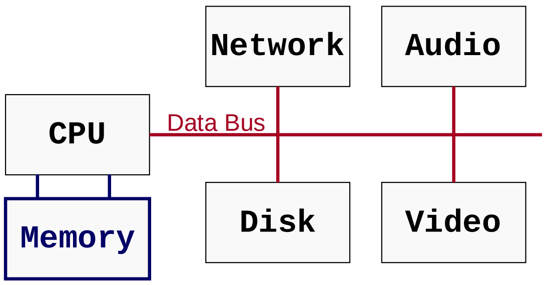
CS 217

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

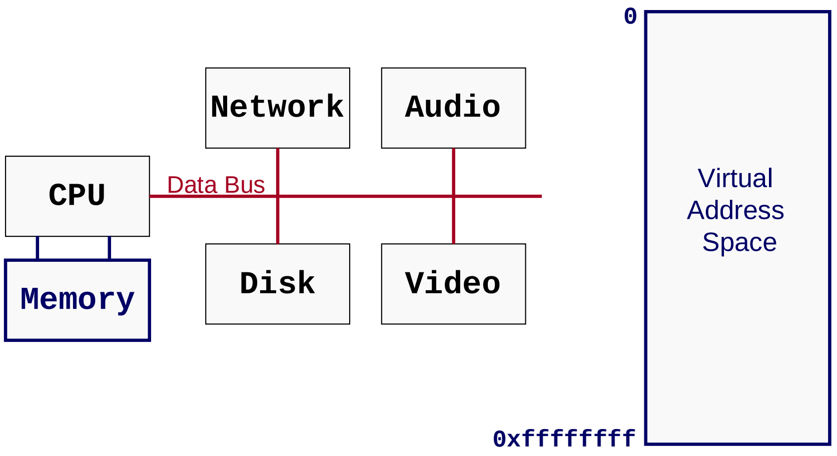
Memory

- What is memory?
  - Flip-flops storing bits for variables, data, code, etc.



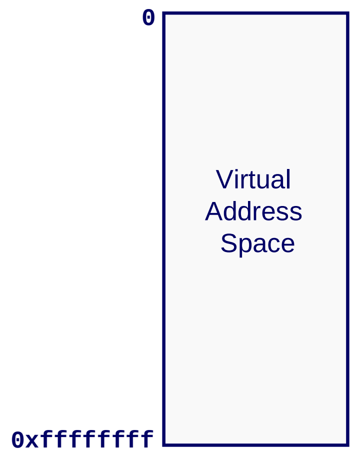
Memory

- What is memory?
  - Flip-flops storing bits for variables, data, code, etc.
  - Unix provides virtual memory



Memory

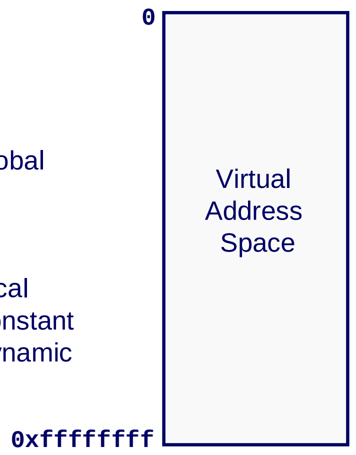
- What is stored in memory?



Memory

- What is stored in memory?
  - Code
  - Constants
  - Global and static variables
  - Local variables
  - Dynamic memory (malloc)

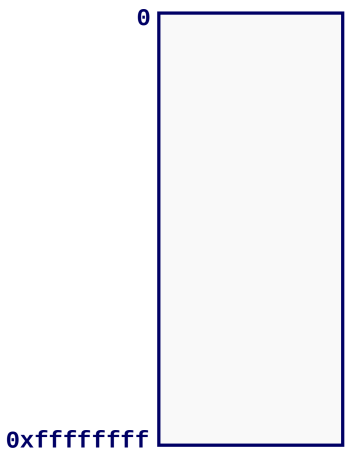
```
int iSize;                   ← global

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



Memory Layout

- How is memory organized?
  - Code
  - Constants
  - Global and static variables
  - Local variables
  - Dynamic memory (malloc)

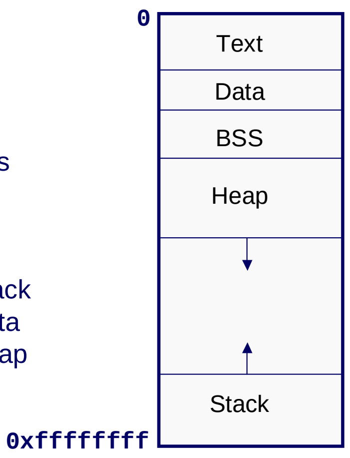
```
int iSize;

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



Memory Layout

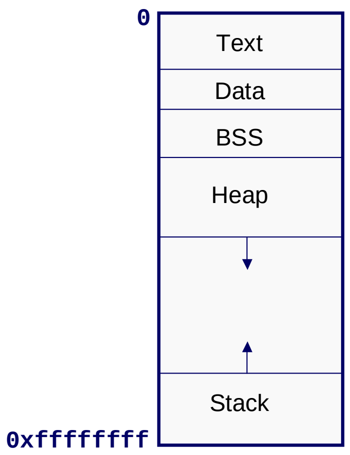
- How is memory organized?
  - Text = code
  - Data = constants
  - BSS = global and static variables
  - Stack = local variables
  - Heap = dynamic memory

```
int iSize;                ← bss

char *f(void)
{
    char *p;              ← stack
    iSize = 8;            ← data
    p = malloc(iSize);    ← heap
    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

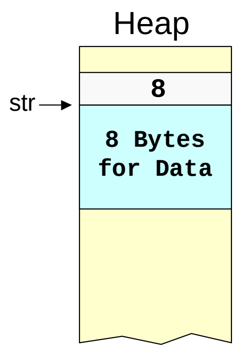
Memory Deallocation

```
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);
```

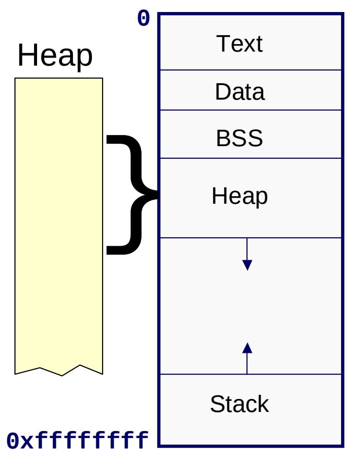
```
char *str = malloc(8);
...
free(str);
```



Dynamic Memory

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

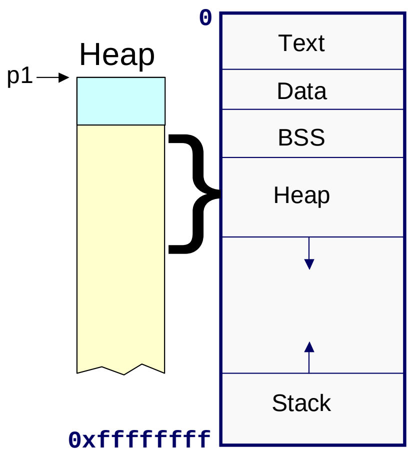
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

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

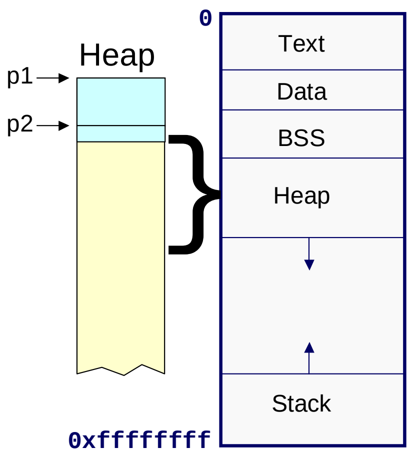
```
➡ char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

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

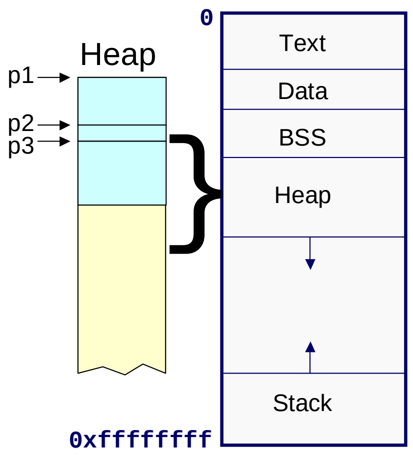
```
  char *p1 = malloc(3);
➡ char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

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

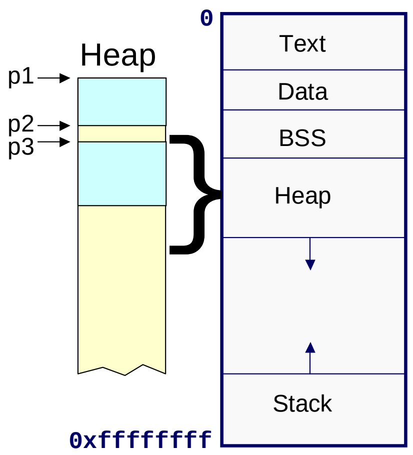
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

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

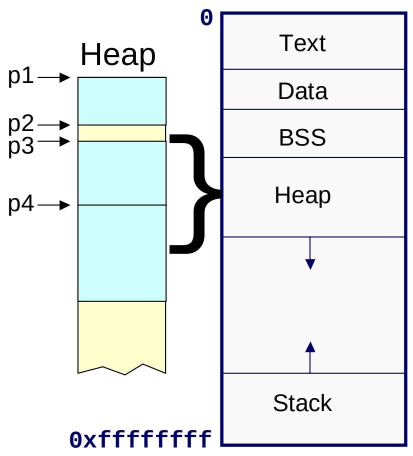
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Dynamic Memory

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

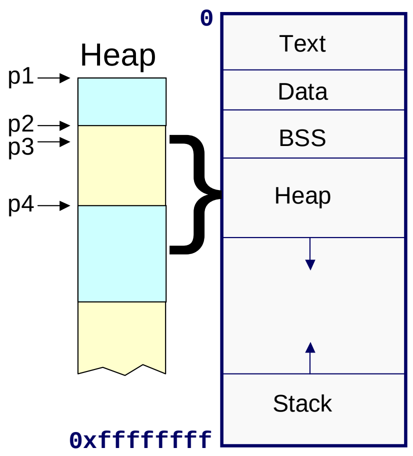
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
➡ char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

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

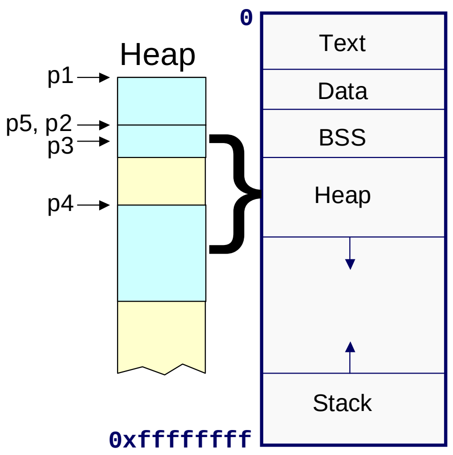
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
➡ free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Dynamic Memory

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

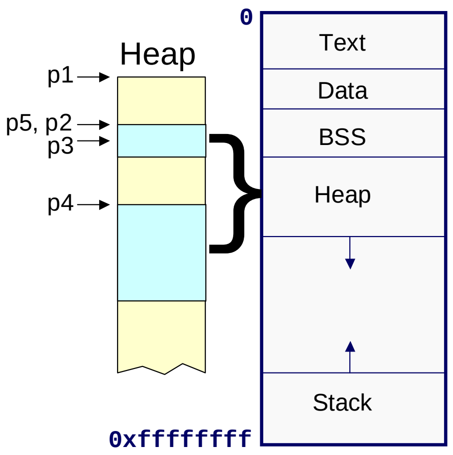
```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
   char *p3 = malloc(4);
   free(p2);
   char *p4 = malloc(6);
   free(p3);
⇨  char *p5 = malloc(2);
   free(p1);
   free(p4);
   free(p5);
```



Dynamic Memory

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

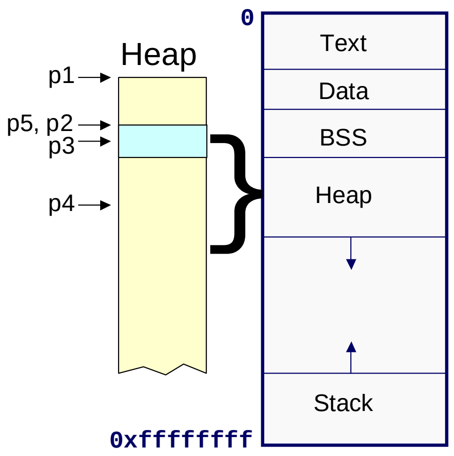
```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
   char *p3 = malloc(4);
   free(p2);
   char *p4 = malloc(6);
   free(p3);
   char *p5 = malloc(2);
⇨  free(p1);
   free(p4);
   free(p5);
```



Dynamic Memory

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

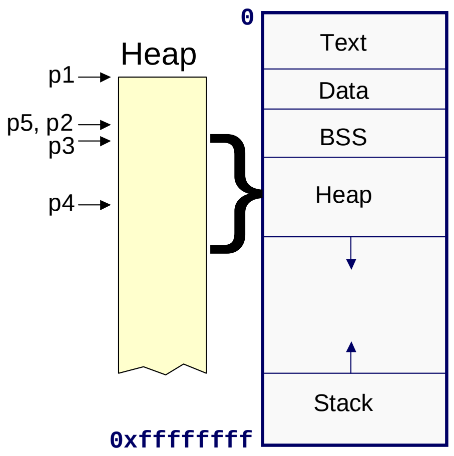
```
    char *p1 = malloc(3);
    char *p2 = malloc(1);
    char *p3 = malloc(4);
    free(p2);
    char *p4 = malloc(6);
    free(p3);
    char *p5 = malloc(2);
    free(p1);
⇨   free(p4);
    free(p5);
```



Dynamic Memory

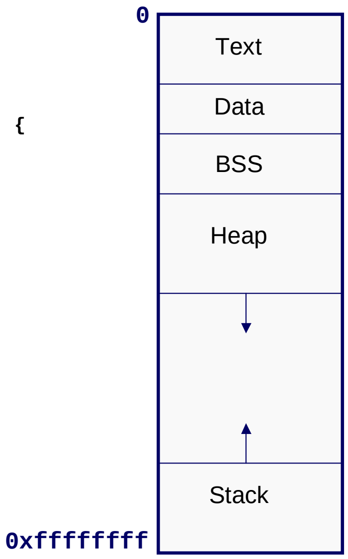
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
    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);
```



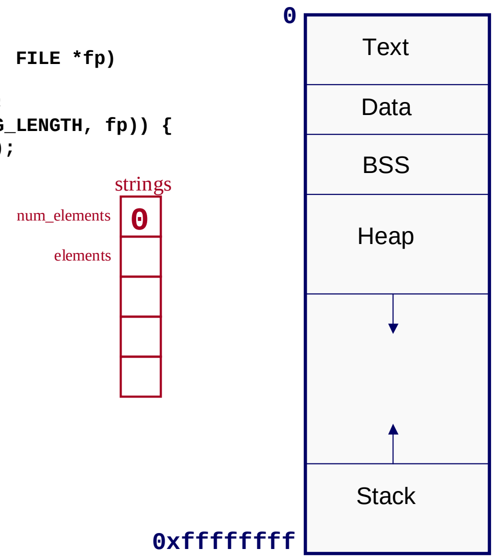
Example Code I

```
...

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;
}
```



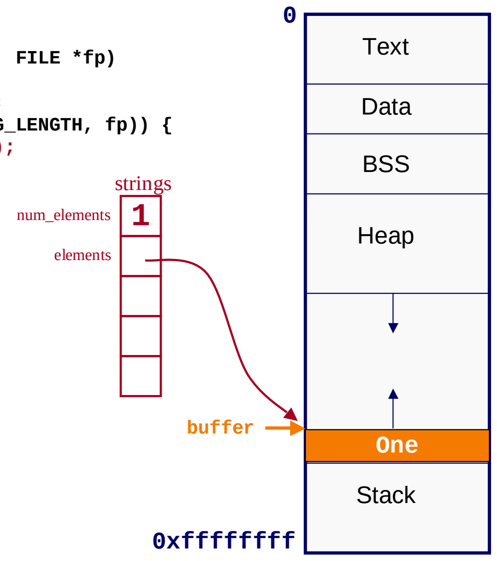
Example Code I

```
...

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;
}
```



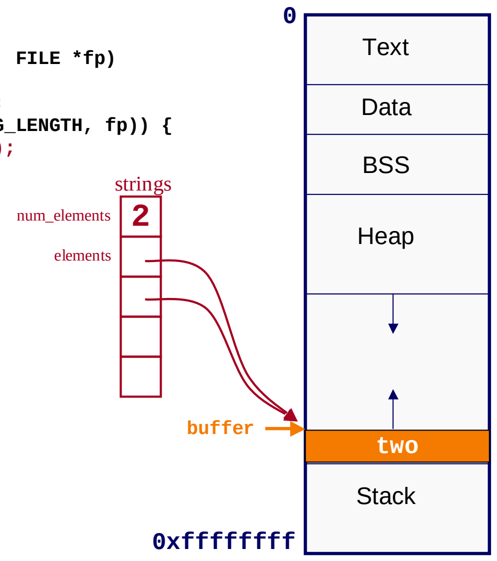
Example Code I

```
...

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;
}
```



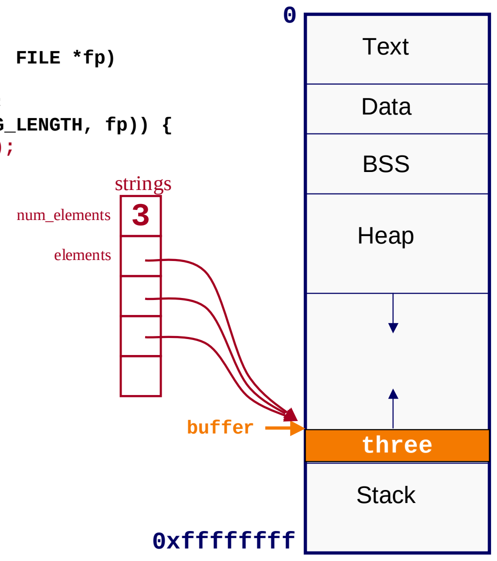
Example Code I

```
...

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;
}
```



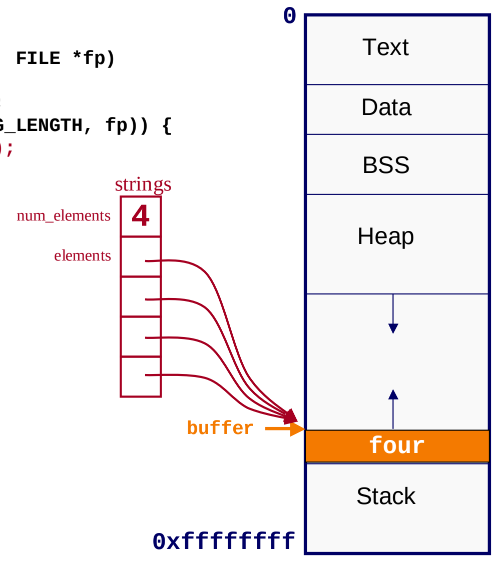
Example Code I

```
...

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;
}
```



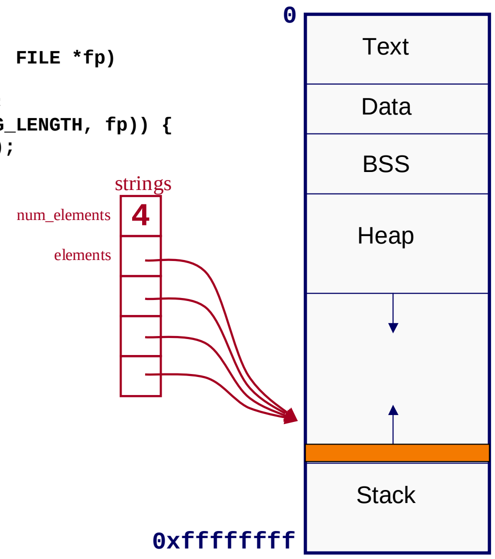
Example Code I

```
...

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;
}
```



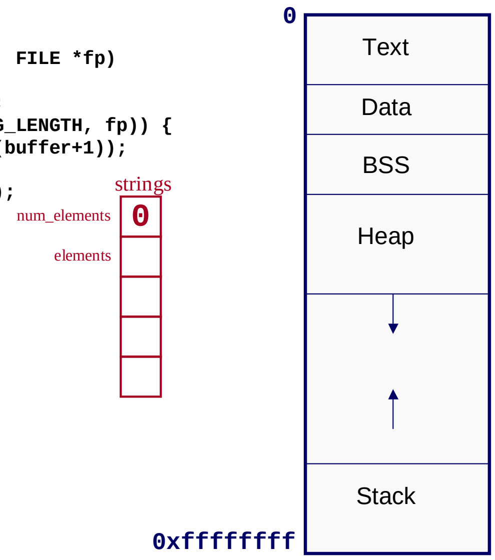
Example Code I

```
...

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;
}
```



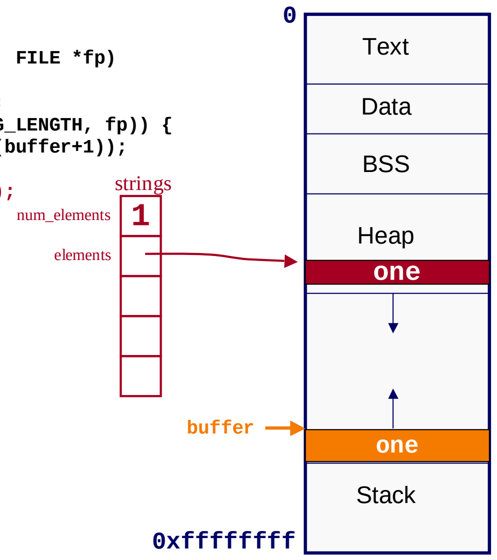
Example Code II

```
...

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;
}
```



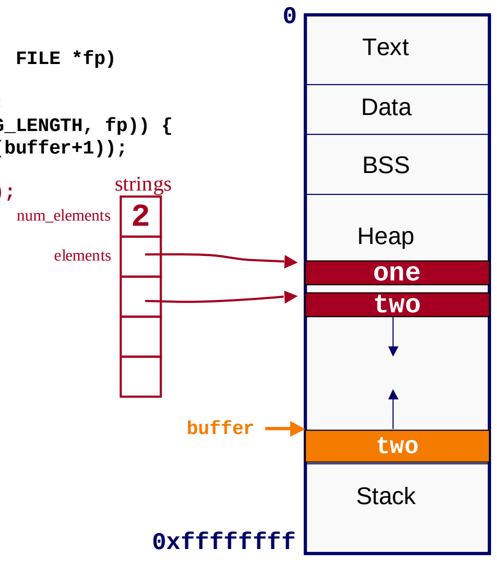
Example Code II

```
...

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;
}
```



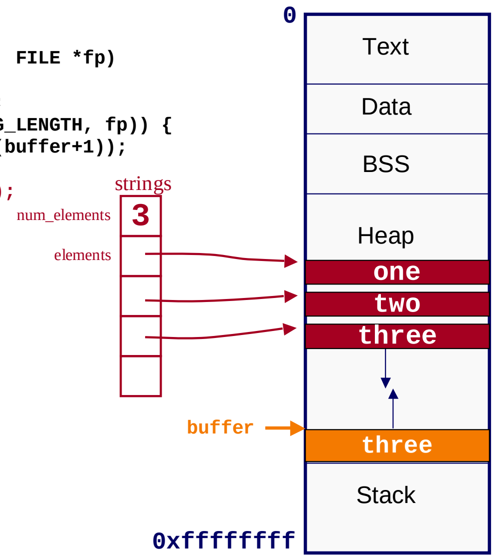
Example Code II

```
...

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;
}
```



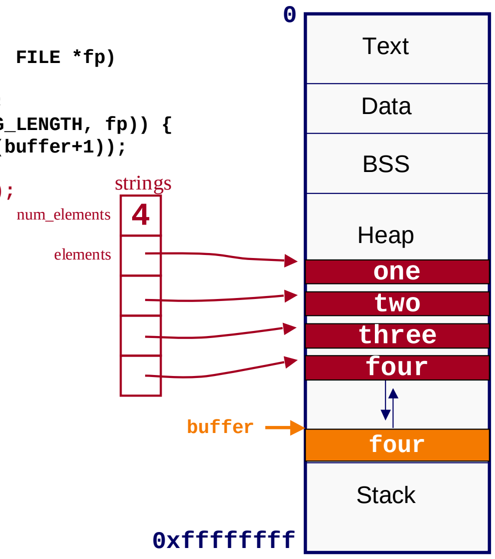
Example Code II

```
...

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;
}
```



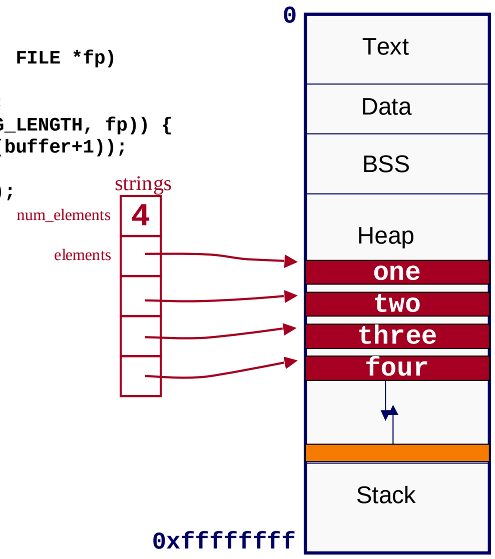
Example Code II

```
...

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;
}
```



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!

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

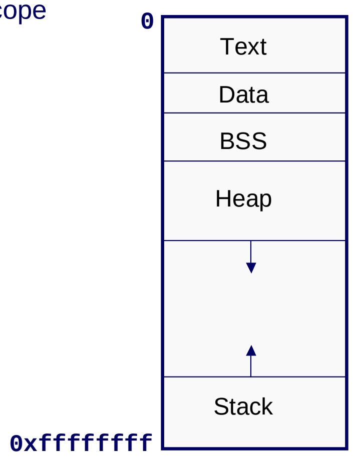
```
static int count = 0;
```
is the same as
```
static int count;
```

It is bad style to depend on this

Static Local Variables

- `static` keyword in declaration of local variable means:
  - Available (if within scope) throughout entire program execution
  - Variable is allocated from 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;
    }
    ...
}
```


0
Text
Data
BSS
Heap
Stack
0xffffffff