Today's Goals

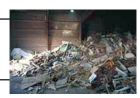
- Memory management
  - Dynamic memory allocation
  - The heap
  - Memory layout
- **`malloc`** and **`free`**
- Other heap functions

Section 1

Dynamic Memory Allocation

- The most important usage of pointers.
- C's data structures are normally fixed in size, i.e. static.
  - Static data structures must have their sizes decided at time of compilation
  - Arrays are good examples
  - Allocated on stacks
- Through pointers, C supports the ability to allocate storage during program execution.

The Heap



- The pool of memory from which dynamic memory is allocated is separate, and is known as the heap.
- There are library routines to allocate and free memory from the heap.
- Heap memory is only accessible through pointers.
- Mixing statically and dynamically allocated memory is not allowed.

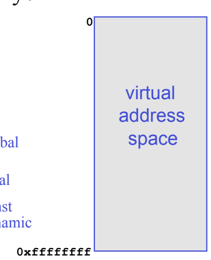
Memory

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

```
int SIZE;            <- global

char* f(void) {
  char *c;           <- local

  SIZE = 10;         <- const
  c = malloc(SIZE);  <- dynamic
  return c;
}
```



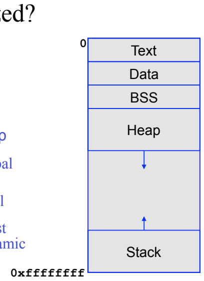
Memory Layout

- How is memory organized?
  - Code – Text
  - Constants – Data
  - Global and static variables – BSS
  - Local variables – Stack
  - Dynamic memory (malloc) – Heap

```
int SIZE;            <- global

char* f(void) {
  char *c;           <- local

  SIZE = 10;         <- const
  c = malloc(SIZE);  <- dynamic
  return c;
}
```



Function Call Mechanism

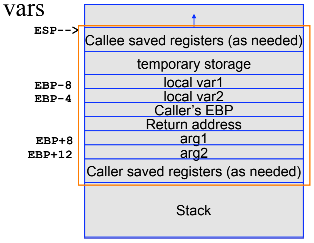
- Activation record (of a function call), also known as a stack frame
- A block of memory that contains:
  - Parameters passed to the function
  - Local variables declared in the function
  - Return address – pointer to the instruction to be executed after the function call

Call Stack

- A call stack is a region of memory that manages activation records
- The call stack is initialized with the activation record of **`main`**
- Activation record of a function is
  - Pushed onto the stack at the function call
  - Popped off the stack on return from the call
    - The reason why local variables are only present during the function call

A Typical Stack Frame

- **`int foo(int arg1, int arg2);`**
- Two local vars



Stack Frame Details

- Stack grows upwards
- **`ESP`** and **`EBP`** are registers, used to point to the top of the stack and the base
- Saved registers – on return:
  - Callee must store return value to **`EAX`** before returning
  - Other registers must be restored if modified during function call

Section 2

`malloc()` and **`free()`**

- Library routines for managing the heap
- Dynamically allocate and free arbitrary-sized chunks of memory in any order
  - **`void *malloc (size_t size);`**
    - Allocates a block of **`size`** bytes from the heap
    - Returns a pointer to the block allocated (casting to correct type required)
    - **`size_t`** is an unsigned integer type used for very large integers.
  - **`void free (void *ptr);`**
- **`#include<stdlib.h>`**

Example: Allocating an **`int`** Array

```
int *a;
a = (int *) malloc(sizeof(int)*6);
a[5] = 3;
free(a);
```

- Never attempt to free memory that has not been previously allocated via **`malloc`**!
- Memory allocated through **`malloc`** is not cleared or initialized in anyway.

Example: String Allocations

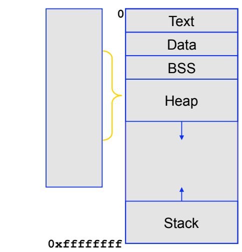
```
char* newStr(char *str) {
  char *s;
  s = (char *) malloc(strlen(str) + 1);
  return strcpy(s, str);
}
```

```
char* newStr2(char *str, char *str2){
  char *s;
  s = (char *) malloc(strlen(s) + strlen(s2) + 1);
  strcpy(s, str); return strcat(s, str2);
}
```

- By default **`void*`** will be casted to **`char*`**, so in fact no casting is necessary here.

Dynamic Memory Layout

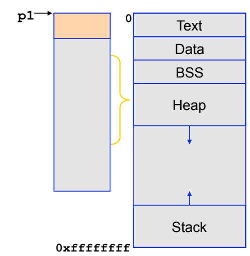
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 13 Lec11

Dynamic Memory Layout

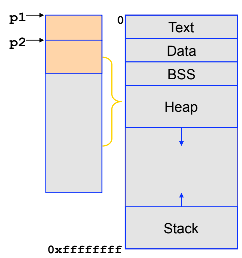
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 14 Lec11

Dynamic Memory Layout

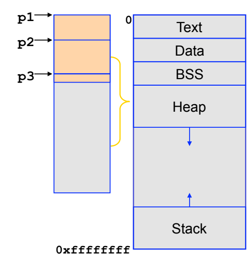
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 15 Lec11

Dynamic Memory Layout

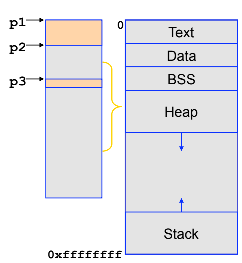
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 16 Lec11

Dynamic Memory Layout

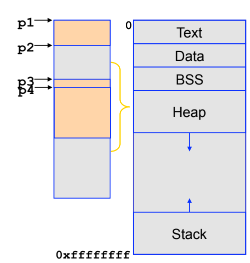
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 17 Lec11

Dynamic Memory Layout

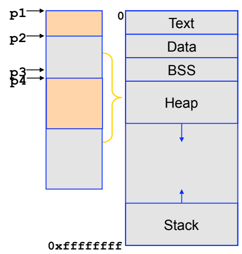
```
char *p1 = malloc(3);
char *p2 = malloc(4);
char *p3 = malloc(1);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



CS246 18 Lec11

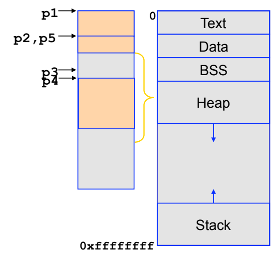
Dynamic Memory Layout

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



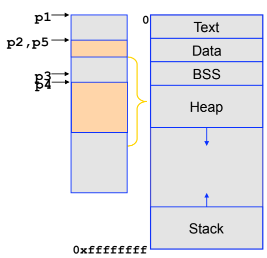
Dynamic Memory Layout

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



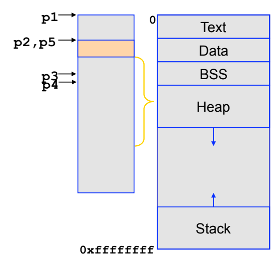
Dynamic Memory Layout

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



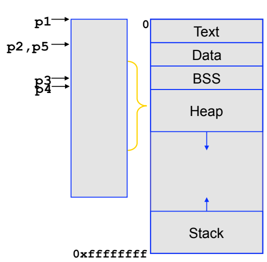
Dynamic Memory Layout

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



Dynamic Memory Layout

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



Example: Allocating 2d Array

```
void error() {
  printf("Out of memory!\n");
  exit(1);
}

int main() {
  int **a, r, c, i, j;

  scanf("%d", &r);
  if ((a =(int **)malloc(sizeof(int *)*r)) != NULL){
    scanf("%d", &c);
    for (i=0; i<r; i++) {
      if ((a[i]=(int *)malloc(sizeof(int)*c)) != NULL)
   {
        for (j=0; j<c; j++)
          a[i][j] = i*c + j;
      }
      else error();
    }
  }
  else error();
  return 0;
}
```

21-malloc.c

The Love-hate Relationship with **`malloc`**

- Most experience C-programmers have such a delimma.
  - **`malloc`** is fast, efficient and flexible
  - The dreaded memory leak – neglecting to free memory
  - Reaching beyond **`malloc`**ed bounds
  - Heap fragmentation – this is not really a programming error, and is therefore even harder to fix

Section 3

Other Heap Functions

- **`void *calloc(size_t n, size_t size);`**
  - Allocates space for an array with **`n`** elements, each of which is **`size`** bytes long.
  - **`calloc`** also initializes the array by setting all bits to **`0`**.
- **`void *realloc(void *ptr, size_t size);`**
  - **`realloc`** resizes memory (pointed to by **`ptr`**, must be result of previous call to **`malloc`** or **`calloc`**) to the new size specified by **`size`**.
  - Returns a **`NULL`** if expansion attempt fails.
  - If called with **`NULL`** as 1st argument **`ptr`**, behaves like **`malloc`**.
  - If called with **`0`** as 2nd argument, behaves like **`free`**.

Memory Types and Allocations

- Three types of memory
  - Global and static variables – BSS – Program start up/termination
  - Local variables – Stack – Function entry/return point
  - Dynamic memory – Heap – **`malloc`**/**`free`** or program termination

Summary

- Learn how to handle memory management in C
- **`malloc`** and related functions are essential to C programming
- Learn the good habit of freeing memory whenever possible