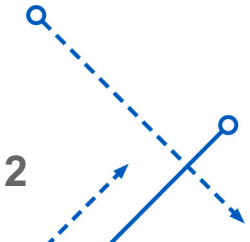
C – Dynamic Memory Allocation

Karthik Dantu
Ethan Blanton
Computer Science and Engineering
University at Buffalo
kdantu@buffalo.edu

Portions of this lecture are borrowed from the U-W CSE 333 course slides

Administrivia

- Piazza has a search bar – use it!
- Corollary – name your posts descriptively so others can find them!
- GitHub – commit regularly
- Git – learn features such as tagging
- Don't push `.o` and executable files or other build products



Memory Allocation

So far, we have seen two kinds of memory allocation:

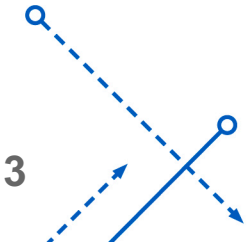
```
int counter = 0;    // global var

int main(int argc, char** argv) {
  counter++;
  printf("count = %d\n",counter);
  return EXIT_SUCCESS;
}
```

```
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main(int argc, char** argv) {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return EXIT_SUCCESS;
}
```

- `counter` is ***statically***-allocated
  - Allocated when program is loaded
  - Deallocated when process gets reaped
- `a, x, y` are ***automatically***-allocated
  - Allocated when function is called
  - Deallocated when function returns



Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient:
  - We need memory that persists across multiple function calls but not the whole lifetime of the program
  - We need more memory than can fit on the stack
  - We need memory whose size is not known in advance to the caller

```
// this is pseudo-C code
char* ReadFile(char* filename) {
  int size = GetFileSize(filename);
  char* buffer = AllocateMem(size);

  ReadFileIntoBuffer(filename, buffer);
  return buffer;
}
```

Dynamic Memory Allocation

- What we want is ***dynamically***-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual memory management*)
    - A garbage collector collects it (*automatic memory management*)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches

Aside: NULL

- NULL is a memory location that is guaranteed to be invalid
  - In C on Linux, NULL is 0x0 and an attempt to dereference NULL *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {
  int* p = NULL;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

malloc()

- General usage: `var = (type*) malloc(size in bytes)`

- **malloc** allocates a block of memory of the requested size

  Returns a pointer to the first byte of that memory

  And returns NULL if the memory allocation failed!

  You should assume that the memory initially contains garbage

  You'll typically use sizeof to calculate the size you need

```
// allocate a 10-float array
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

calloc()

- General usage:

```
var = (type*) calloc(num, bytes per element)
```

- Like **malloc**, but also zeros out the block of memory

  Helpful when zero-initialization wanted (but don't use it to mask bugs – fix those)

  Slightly slower; but useful for non-performance-critical code

  malloc and **calloc** are found in `stdlib.h`

```
// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

free()

- Usage: `free(pointer);`

- Deallocates the memory pointed-to by the pointer

  Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **malloc** or **calloc**)

  Freed memory becomes eligible for future allocation

  Pointer is unaffected by call to free

    Defensive programming: can set pointer to NULL after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

Practice

- Which lines have errors?

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

  return EXIT_SUCCESS;
}
```

Required Reading

- K&R 7.8.5