Data Structures in C

Prof. Georg Feil

Dynamic Memory Management

Summer 2018

Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- C for Programmers (supplementary textbook)
  - Chapter 12, section 12.3
  - Chapter 14, section 14.9

Dynamic Memory Management in C

- One of the powerful features of C is that we can allocate a region of memory (almost any size) then use pointers to work with that memory
- Data can be accessed in any way we like. We can treat a double precision floating point number as a sequence of 8 bytes, or 64 bits
- We're in complete control over the lifetime of our structures (*objects*)
  - Memory can be allocated, deallocated, reallocated as needed

Dynamic Memory Management in C

- C programs can build complex data structures in memory, where one part of the data structure may link to another part using pointers

- Dynamic memory allocation is based on a C library function called malloc
  - It uses an area of memory called the heap

- The way malloc allocates memory is very similar to how the Java 'new' keyword allocates memory

➢ Data structures can grow or shrink as needed

malloc

```
void* malloc(size_t bytes)
```

- To access malloc, #include <stdlib.h>
- The parameter is how many **bytes** to allocate
  - We can often use sizeof to calculate how many bytes are needed
  - Recall: size_t is equivalent to an unsigned long int
- The malloc function returns a void pointer
  - C lets us assign a void pointer to variables of any pointer type
  - If memory allocation failed, the pointer will be NULL
- The memory contents are not initialized!
  - We must fill/assign the memory a value before using it

malloc example: Array of int

- Here's how to allocate memory for an int array size 100

```
int* arr = malloc(100 * sizeof(int));
```

- For this array we must use a pointer data type
  - The array will be located in heap memory, unlike arrays we've seen before which go on the stack or in the static data area
- This is a good way to work with large arrays
  - We use sizeof to guarantee that the size is exactly right
- We can work with this array using [ ], for example

```
arr[0] = 5;                  // Set first element to 5
printf("%d\n", arr[99]);  // Print last element
```

malloc and NULL

- The value NULL (all upper case) represents an invalid pointer
  - The pointer has a value (it's not undefined) but can't be used
  - If you try to use it (dereference), your program will crash
- Memory allocation functions like malloc will return NULL on errors, for example if you try to request more memory than is available
- For production quality code it may be good to check for errors, e.g.

```
char* mem = malloc(1000000000);  // Allocate 1 billion bytes
if (mem == NULL) {
    printf("Out of memory!");
    exit(1);
}
```

malloc example: Student structure

- Imagine we have a Student structure which contains a name, ID, GPA etc.

- Here’s how to allocate memory for one student

```
Student* st = malloc(sizeof(Student));
```

- We can work with this variable using ->, for example

```
printf("%s\n", st->name); // Print the student’s name
st->id = 12345678;          // Set the student’s ID
```

Exercise 1: malloc

- Create a program that allocates an array of 1000 integers using malloc
- Fill the array with the numbers 0, 1, 2, 3, 4, ...
- After filling is complete, print the contents of the array

Deallocating memory

- Memory allocated using malloc is reserved for the program until it stops running
- If you don’t need the memory anymore you must deallocate it using free
  - There is no automatic deallocation or garbage collection in C
- If you don’t free memory, a program that uses malloc will keep using more and more memory until it eventually runs out of memory!
  - This is called a memory leak

free

```
void free(void* ptr)
```

- Pass a pointer that was previously obtained from malloc (or a related memory allocation function)
- If you pass any other memory address bad things will happen (unpredictable!) This includes
  - An address not from malloc
  - An address that has already been freed
- There should be exactly one ‘free’ for every ‘malloc’
- Program must stop using the pointer!

Exercise 2: free

- In your program from Exercise 1, add code to free memory before the program quits

Other ways to allocate memory – calloc

```
void* calloc(size_t num, size_t size)
```

- The calloc function provides a convenient way to allocate arrays. There are two parameters
  - num – The number of array elements
  - size – The size of each element in bytes (normally use sizeof here)
- Unlike malloc, the calloc function clears (zeroes) memory
  - After allocating an array of numbers you can be sure they'll all be zero
- Like malloc, calloc will return NULL if allocation failed

Exercise 3: calloc

- Change your program from Exercise 2 to use calloc instead of malloc
- Instead of duplicating the size (1000) in several places in the program, make the program use a constant
- Also change the array data type to array of double

Dynamic Arrays

Our first dynamic data structure

Dynamic Arrays

- A regular array has a fixed size that can’t be changed
- A dynamic array is different: We can change its size
- To implement a dynamic array in C,
  1. Allocate the array using calloc or malloc
  2. When needed, resize the array using realloc

realloc

```
void* realloc(void* ptr, size_t size)
```

- The realloc function changes the size of a memory block previously obtained from malloc/calloc
  - ptr – The previously obtained memory address
  - size – The new size in **bytes**
- When making a memory block smaller, memory is “removed” from the end (program must stop using it!)
- When making a block larger, *either*
  - Memory is added at the end, *or*
  - A new block is allocated, and data is automatically copied over

Using realloc to enlarge a dynamic array

- To make an array "grow" dynamically you can use the realloc function whenever you need to make the array larger
- Remember that the newly allocated part of the array won't be initialized,
  - If needed, fill the new part of the array with zeros using a loop
- Try not to call realloc too often, remember it's not efficient (why?)
  - How can we be sure not to call realloc very often as a dynamic array grows?

Exercise 4: realloc

- Start with your program from Exercise 3
  - Recall that this program fills an array size 1000 with numbers 0, 1, 2, 3, …, 999
- Add code to resize the array to double its original size using realloc (you might want to “change” your size constant!)
- Fill the new array elements with values 999, 998, 997, …, 0
- Print the resulting array
- Also print the address of the array (pointer) before and after resizing
  - What does this tell you about how realloc resized the array?
  - Try making the array smaller/bigger, does the behavior change?
- Free the memory allocated for the array before quitting