Dynamic Memory – Overview

- Dynamically allocated memory is stored in the Heap-section of memory.
- It is solely controlled by the programmer (unlike, for example, stack frames or read-only data).
- As a result, it is the programmer's responsibility to *allocate* and *free* this type of memory.

Dynamic Memory – Example

- For example:

```
char *str = malloc(sizeof(char) * 3);
str[0] = 'H';
str[1] = 'I';
str[2] = '\0';
free(str);
```

- You cannot make any assumptions about the content of `str` after `malloc`.

Dynamic Memory – Common Errors

```c
int main(void) {
  char *str = malloc(sizeof(char) * 3);
  int *arr = malloc(sizeof(int) * 5)
  str[9001] = '?'; // heap-buffer-overflow
  free(arr);
  *(arr + 2) = 42; // heap-use-after-free
  free(arr);       // double-free
  return 0;
} // memory-leak (here: str has not been freed)
```

- A good rule of thumb: for each `malloc`, there should be a `free` somewhere in your code.

Dynamic Memory – Flavours of Allocation

```c
// malloc(size) allocates the requested memory of a
//   given size and returns a pointer to it.
// effect: allocates heap memory, must be freed
// requires: size >= 0
// time: O(1)
void *malloc(size_t size);

// recommended use:
int arr_len = 10;
int *arr = malloc(sizeof(int) * arr_len);
assert(arr);
```

Dynamic Memory – Flavours of Allocation

```
// realloc(data, size) allocates the requested memory of
//   a given size, copies the content of data over,
//   frees data, and returns a pointer to the newly
//   allocated memory.
// effect: allocates heap memory, must be freed
// requires: data is not NULL, size >= 0
// time: O(n)

void *realloc(void *data, size_t size);

// recommended use (but missing some error handling):
// continued from previous slide
arr_len -= 5;
arr = realloc(arr, sizeof(int) * arr_len);
assert(arr); // assume that arr has been mutated!
```

Basic Exercise – Reading Strings Dynamically

Implement the following function:

```
// read_paragraph() reads an arbitratry number of words
//   from the command line (via scanf("%s",...)), stores
//   them in a single string of increasing size, and
//   returns a pointer to this string. The maximum
//   length of a single word is 20 characters; words are
//   separated with the underscore character ('_').
```

Hints:

- Your string-growth strategy does not have to be efficient!
- Remember that `C`-strings must be `NULL`-terminated!
- Try not to use any stack-arrays (e.g., `char arr[21]`)!.

Medium Exercise – Splitting Strings

Implement the following function:

```
// substr(paragraph, substring) splits a string into two
//   parts. It stores the first 10 characters of
//   paragraph in substring and returns a char-ptr to a
//   new string that contains the remaining characters
//   of paragraph, i.e., the original paragraph with the
//   first 10 characters removed. The original paragraph
//   is free'd.
```

Hints:

- You have to shrink paragraph; therefore, you must return a pointer to the new (shrunk) paragraph!
- Consider the case where your paragraph length is below 10! How would you handle it; what would you return?

- Don’t forget the built-in string functions!
- Remember what you have learnt about using pointer-parameters to return values to the caller-function!

HARD Exercise – Arrays of String

Implement the following function:

```
// para_to_lines(paragraph, lines) splits a paragraph
//   into an array of strings of length 10. It
//   iteratively splits the paragraph into substrings
//   and stores each resulting substring in lines. It
//   returns an int indicating the number of substrings
//   in the string-array lines.
```

Hints:

- Don't be intimidated by `char ***`! Consider what each indirection means; ultimately, `char ***` is just a pointer to a string-array!
- Use `substr`! You can implement `para_to_lines` in about 10 lines of code!

- To be honest, this question is less about heap-memory and more about pointers.

Linked Data Structures – Overview

- In linked data structures, each element has a link to one (or more) other elements.
- Common linked data structures include
  - Linked Lists: each element has a link to the next element.
  - Doubly-linked lists: each element has a link to the next and the previous element.
  - Trees: each element has a link to n children.†
  - Graphs: each element has a link to n neighbours.†

† Simplified; please recall CS 135 for full definitions.

Linked Data Structures – `c`-specific

- In `C`, this link is implemented through pointers.
- Maintaining these pointers is one of the pitfalls when implementing linked data structures!
- Remember that you oftentimes have to distinguish between manipulating the first element (e.g., `tree->root` or `llist->front`), manipulating the last element (e.g., leaves or `llist->back`, and manipulating an element in between.
- Use diagrams and drawings **excessively** when planning on how to manipulate linked data structures!

Basic Exercise – Doubly-Linked What?!

Implement the following function:

```
// what(dllist) swaps the first and the last node in
//   dllist.
//
```

Hints:

- There is an easy way to do this: swapping the values stored in both nodes, and a fun way: mutating all the pointers. Try doing it the fun way!
- Use diagrams and drawings **excessively** when planning on how to manipulate linked data structures!

EXTRA HARD Exercise – Doubly-Linked Merge Sort

Implement the following function:

```
// dllist_sort(dllist) sorts dllist using merge sort.
```

Hints:

- Remember that merge sort has two aspects: sorting and merging. Split up the problem accordingly!
- Try not to create additional `dllnode` structures or free existing ones! Try using pointer-mutation only!
- Use diagrams and drawings **excessively** when planning on how to manipulate linked data structures!

- Good luck!