MEMORY MANAGEMENT

MEMORY MANAGEMENT IN C

- Memory allocation/deallocation is done differently in C as compared to what you may be used with other languages.
- There is no automatic memory management (garbage collection) and thus th programmer is responsible for releasing *dynamically* allocated memory whe no longer needed.
- Poor memory management can lead to memory leaks → system performance suffers as virtual memory is progressively paged to hard drive. The OS may crash.

MEMORY ALLOCATION

- Two mechanisms are used to allocate memory in C
  1. Declaring local variables. These are stored in a **stack** and are automatically freed when they become out of scope (e.g. exiting a function).

```
int parseInput(FILE *finput) {
   int N;
   int array[100];
   ...
}
```

     - Nice, right? Well there's a catch. The compiler needs to know in advance how much memory to allocate (inefficient) and the stack size is limited (not all your variables may fit).

MEMORY ALLOCATION

- Two mechanisms are used to allocate memory in C

  2. Requesting memory explicitly and storing variables in the **heap**.

     - You can allocate as much memory as the system allows, but you need to take care of releasing it.

```
...
   int N;
   int *array;
   ...
   array = (int *) malloc(N * sizeof(int));
```

     - The compiler cannot know what you intend to do and thus will not release it automatically. You have to do it manually when done:

```
free(array);
```

SEGMENTATION FAULT & CO

A few things can go wrong if not careful with variables allocated on the heap.

- Memory leaks → you forget to call `free()` when variables no longer needed.
- You try to free, but allocation failed or memory already deallocated.

```
char *fileName = malloc(255*sizeof(char));
if (fileName != NULL) {
  ...
  free(fileName);
}
```

- You write beyond the allocated length (corruption)

```
char *temp = malloc(64*sizeof(char));
memcpy(temp, data, dataLen);   // dataLen > 64 gives gives error
```

SEGMENTATION FAULT & CO

- You use an out of bounds array index

```
int *array = malloc(128*sizeof(int));
int N = 200;

for (i = 0; i < N; i++) {  // once i > 127, an error will occur
   ...
}
```

- You use an address, but memory has not been allocated

```
struct *listElement;
x = listElement->value;
```

SEGMENTATION FAULT & CO

- You return a pointer to a variable from the stack

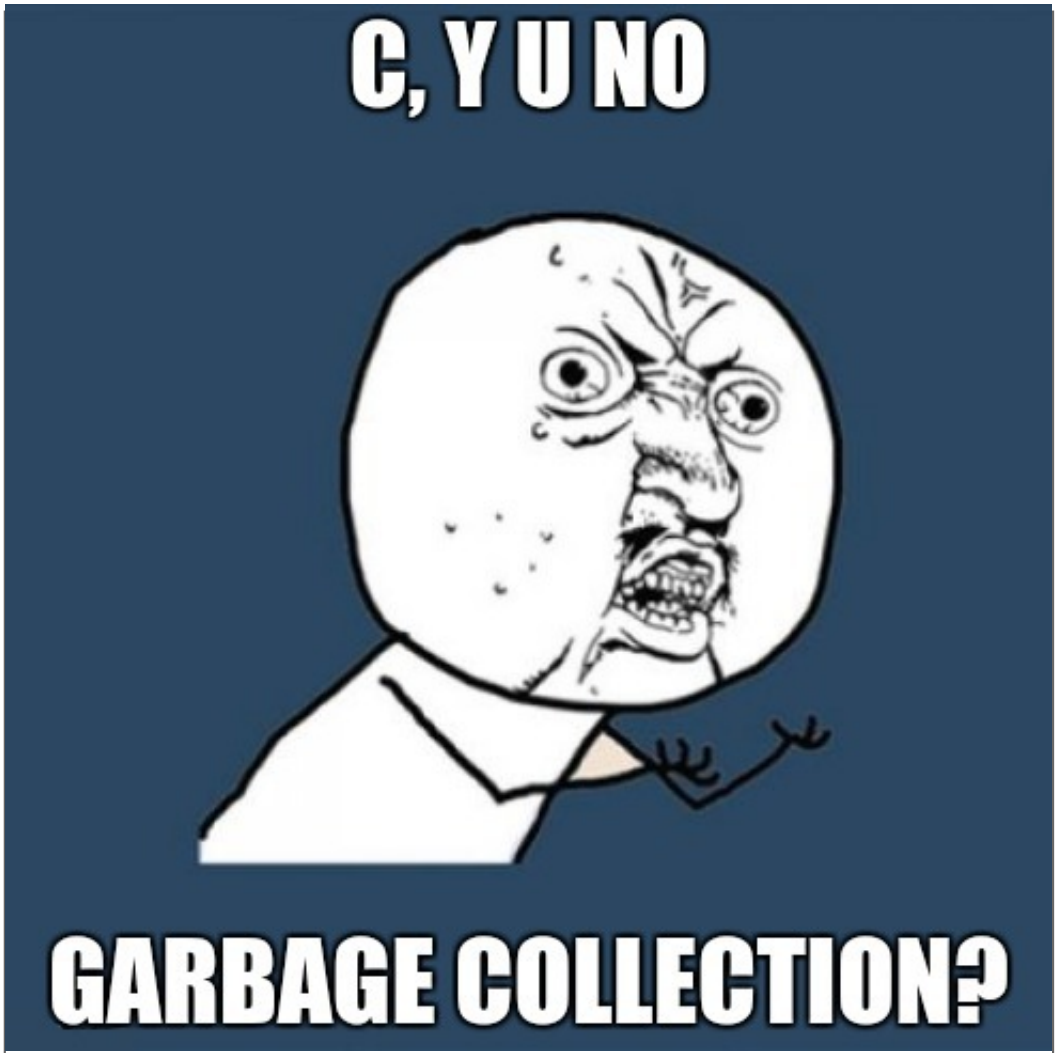
```
int *getCount() {
   int n;  // Local stack variable

   ...     // count the number of values that
           // divide by x in an array

   return &n;
}

int main(void){
  int *n;
  ...
  n = getCount();  // Stack given up by getCount(),
                   // &n no longer safe

  ... // n may be corrupt when needed later
}
```


C, Y U NO
GARBAGE COLLECTION?

WHY NO GARBAGE COLLECTION IN C?

- Garbage collection involves constructing a complex data structure for keepin track of allocations and references counting.
- This mechanism increases the complexity of the language and affects the performance (overhead).
- C is meant for designing very fast code, e.g. for operating systems, device drivers, etc.
- High performance is traded for convenience.

ARRAY & STRING HANDLING

ALLOCATING ARRAYS & MATRICES

We discussed how you can allocate memory dynamically for an array of N elements.

```
int N;
int *array;

array = (int *) malloc(N * sizeof(int));
```

But, how do you allocate memory for a matrix?

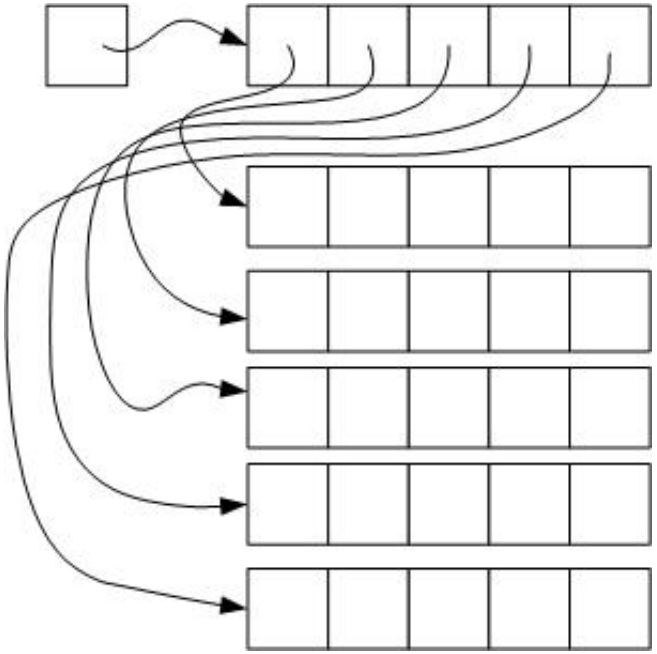
Common mistake

```
int N;
int **matrix;

matrix = (int **) malloc(N * N * sizeof(int));
```

MATRIX ALLOCATION

Remember, you are trying to allocate a pointer to an array of pointers to integer



MATRIX ALLOCATION

Approach 1

```
int i,N;
int **matrix;

matrix = (int **) malloc(N * sizeof(int*));  // rows
for(i = 0; i < N; i++)
   matrix[i] = (int *) malloc(N * sizeof(int));  // columns

// access the (i,j) element by
matrix[i][j] = ...
```

Approach 2 (define the matrix as an array)

```
int i,N;
int *matrix;

matrix = (int *) malloc(N * N * sizeof(int));

// and access the (i,j) element by
matrix[i*N+j] = ...
```

MATRIX DEALLOCATION

When using the first approach, first deallocate the memory allocated for each row

```
for(i = 0; i < N; i++)
   free(matrix[i]);
free(matrix);
```

When using the second approach, simply

```
free(matrix);
```

WHAT ABOUT ARRAYS OF STRUCTURES?

- Imagine the following

```
typedef struct {
  int groupSize;
  float* marks;
} GROUP;

int nGroups = 5;
GROUP *g;
```

- The same principle applies

```
g = (GROUP *) malloc(nGroups * sizeof(GROUP));
for (i = 0; i < nGroups; i++) {
   fscanf(stdin, "%d", &g[i].groupSize);
   g[i].marks = (float *) malloc(g[i].groupSize * sizeof(float));
   ...
}
```

STRING HANDLING

- Strings are simply arrays of characters terminated by the ASCII null characte '\0'.

```
char *str;
char string[100];

str = (char*) malloc(100*sizeof(char));
```

- C provides a set of functions in the standard library, that are useful for manipulating strings.
- Typical operations: copying, tokenizing, comparing, searching, etc.
- Most of these are given in the `<string.h>` header file, but a few exist in `<stdlib.h>` as well.

FUNCTIONS YOU MAY USE

- Copying

```
char* strcpy(char *dst, const char *src);
```

Copies **src** to **dst** including the terminating '\0' character. Returns **dst**.

```
char* strncpy(char *dst, const char *src, int len);
```

Copies at most **len** characters from **src** to **dst**. Appends '\0' to the copied characters if the length of **src** is less than **len**. Returns dst.

NB: careful with sizes to avoid memory corruption.

FUNCTIONS YOU MAY USE

- Comparing

```
int strcmp(const char *str1, const char *str2);
```

Returns:

- <0 if the first character that does not match has a lower value in **str1** than in **str2**;
- 0 if the contents of both strings are equal;
- >0 if the first character that does not match has a greater value in **str1** than in **str2**.

FUNCTIONS YOU MAY USE

- Searching

```
char* strstr(const char *str1, const char *str2);
```

  Returns: a pointer to the first occurrence of **str2** in **str1**, or NULL if not found.
- Examining

```
size_t strlen(const char *str);
```

  Returns: the length of the null-terminated string **str**, i.e. the offset of the terminating '\0' character.

FUNCTIONS YOU MAY USE

- Tokenising – a string into different tokens according to some delimiter(s):

```
char *strtok(char *str, const char *delim)
```

  - **str** broken into smaller strings.
  - **delim** may contain different characters to be used as delimiters.
  - Returns a pointer to the last token found or NULL if none found.
  - Can be called multiple times to find all tokens.

FUNCTIONS YOU MAY USE

Example

```
const char str[100] = "The quick brown fox jumps over the lazy dog";
const char delim[2] = " ";
char *token;

token = strtok(str, delim);   // gets first token

while(token != NULL) {   // retrieve all tokens; stop when no more found
   printf("%s\n", token);
   token = strtok(NULL, delim);
}
```

CONVERTING STRINGS TO NUMBERS

- Converting to floating-point numbers

```
double strtod(const char *str, char **ptr);
float strtof(const char *str, char **ptr);
```

Convert the initial portion of the string **str** to double or float. Return the floating-point value and store in **ptr** the offset of the non-numerical part (if any).

- Example:

```
char str[11] = "9.50 marks";
char *ptr;
float fVal;

fVal = strtof(str, &ptr);
printf("Number:%.2f\t String:%s\n", fVal, ptr);
// Number:9.50     String: marks
```

CONVERTING STRINGS TO NUMBERS

- Converting to (long) integer numbers

```
long int strtol(const char *str, char **ptr, int base);
```

Converts the initial portion of the string **str** to long int according to the given **base** value. Returns the long value and stores in **ptr** the offset of the non-numerical part.

- **base** must be between 2 and 36.
- if **base** is 0, the expected form is a decimal/octal/hexadecimal constant.

CONVERTING STRINGS TO NUMBERS

Example:

```
char str[11] = "60 seconds";
char *ptr;
long int liVal;

liVal = strtol(str, &ptr, 10);
printf("Number:%ld\t String:%s\n", liVal, ptr);
// Number:60     String: seconds
```

CONVERTING STRINGS TO NUMBERS

- Question: what will be the output of the following?

```
char *str;
double fVal;
long int liVal;

liVal = strtol("20.00mm", &str, 10);
printf("Number:%ld\t String:%s\n", liVal, str);

fVal = strtod("1e+2 litres", &str);
printf("Number:%.1lf\t String:%s\n",fVal,str);

liVal = strtol("FFGH", &str, 16);
printf("Number:%ld\t String:%s\n", liVar,str);
```

CONVERTING STRINGS TO NUMBERS

- Answer:

```
Number:20         String:.00mm
Number:100.0      String: litres
Number:255        String:GH
```

Note: A good resource for understanding other string manipulation functions is available **here**.

CODE OPTIMISATION

CODE OPTIMISATION

- Refactoring is done in between development of new functionality
  - Recall this makes it easier to test that this process has not changed the behaviour of your code.
- This is also a good time to do **some** optimisation
  - You should be in a good position to test that your optimisations have not negatively impacted correctness.

WHEN TO OPTIMISE?

- When you discover that your code is not running fast enough, it's probably wise to optimise it.
- Often this will come towards the end of the project.
- It should certainly come after you have something deployable.
- Preferably after you have developed and tested some major portion of functionality.

A PLAUSIBLE STRATEGY

- Perform no optimisation until the end of the project once **all** functionality is complete and tested.
- This is a reasonable approach; however:
- During development, you may find that your test suite takes a long time to run.
- Even one simple run to test the functionality you are currently developing may take minutes/hours.
- This can slow down development significantly, so it may be appropriate to do some optimisation at that point.

HOW TO OPTIMISE

- The very first thing you need **before** you could possibly optimise code is a *benchmark*.
- This can be as simple as timing how long it takes to run your test suite.
- $O(n^2)$ solutions will beat O(n log n) solutions on sufficiently small inputs, so your benchmarks must not be too small.

HOW TO OPTIMISE

Once you have a suitable benchmark then you can:

1. Save a copy of your current code;
2. Run your benchmark and record the run time;
3. Perform what you think is an optimisation on your source code;
4. Re-run your benchmark & compare the run times;
5. If you successfully improved the performance of your code keep the new version, otherwise revert changes;
6. Do one optimisation at a time.

HOW TO OPTIMISE

- However, bear in mind that you are writing a **stochastic** simulator
  - This means each run is different and hence may take a different time to run,
  - Even if the code has not changed or has changed in a way that does not affect the run time significantly.
  - Simply using the same input several times should be enough to reduce or nullify the effect of this.

PROFILING

- *Profiling* is **not the same** as *benchmarking*.
- *Benchmarking*:
  - determines how quickly your program runs;
  - is to performance what testing is to correctness.
- *Profiling*:
  - is used after benchmarking has determined that your program is running too slowly;
  - is used to determine which parts of your program are causing it to run slowly;
  - is to performance what debugging is to correctness.

BENCHMARKING & PROFILING

- Without benchmarking you risk making changes to your program that will lead to poorer performance.
- Without profiling you risk wasting effort optimising a part of code which is either already fast or rarely executed.

Documenting: Source code comments are a good place to explain *why* the code is the way it is.