# CS100 Lecture 8

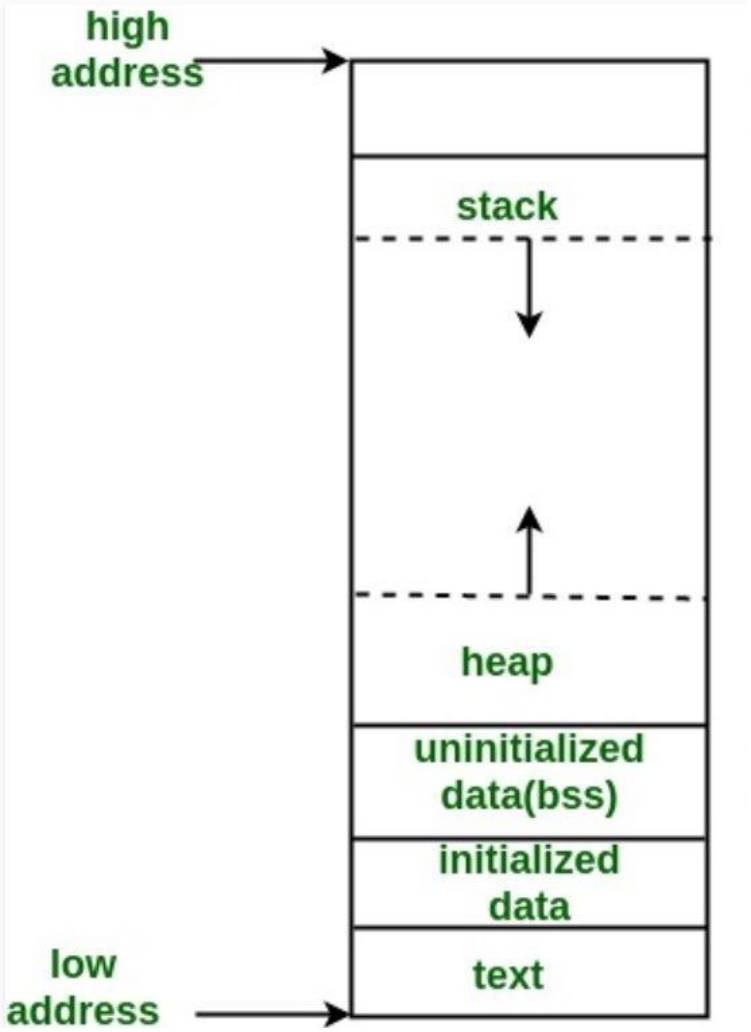
Dynamic Memory and Strings Revisited

## Contents

- Recap
- Command line arguments
- Example: Read a string of unknown length

# Recap

# Stack memory vs heap (dynamic) memory




- Stack memory is generally smaller than heap memory.
- Stack memory is often used for storing small and local objects.
- Heap memory is often used for storing large objects, and objects with long lifetime.
- Operations on stack memory is faster than on heap memory.
- Stack memory is allocated and deallocated **automatically**, while heap memory needs **manual** management.

## Use `malloc`

- Allocate memory for an `int`?
- Allocate memory for $n$ `int`s?
- Allocate memory for a "2-d" array with $n$ rows and $m$ columns?

## Use `malloc`

- Allocate memory for an `int`?

```
int *p = malloc(sizeof(int));
*p = 42;
printf("%d\n", *p);
```

- Allocate memory for $n$ `int`s?

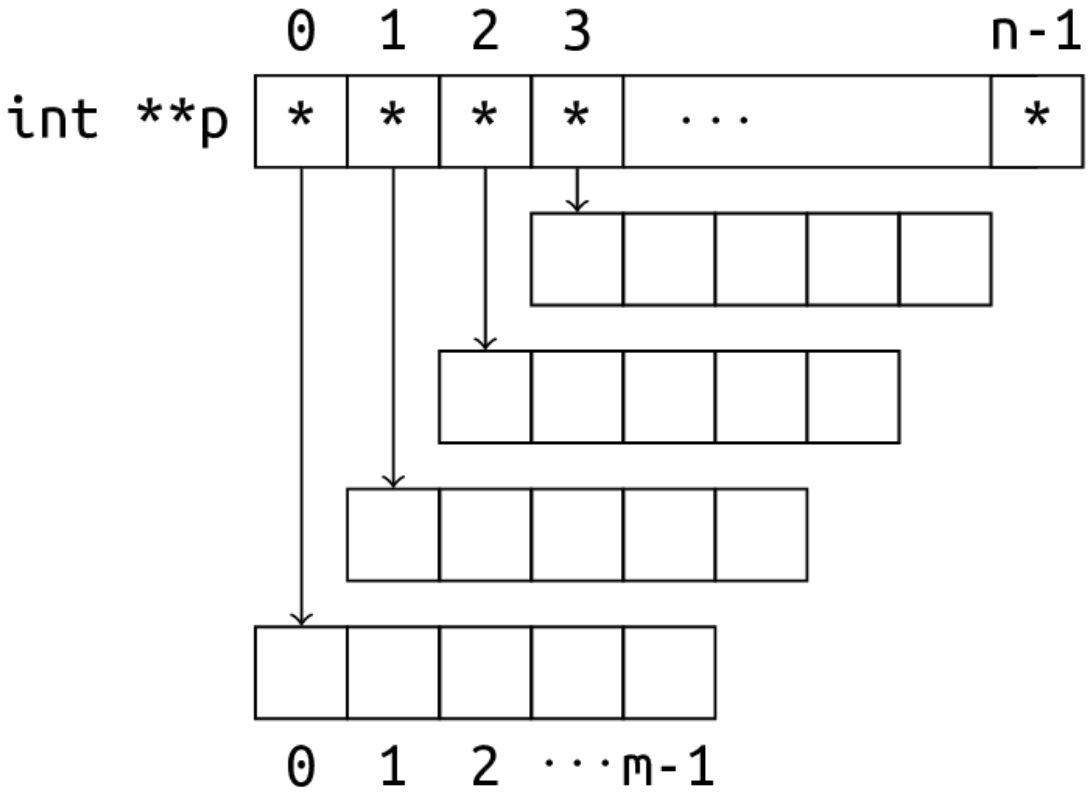
```
int *p = malloc(sizeof(int) * n);
for (int i = 0; i < n; ++i)
  scanf("%d", p + i); // What does `p + i` mean?
```

# Use `malloc`

- Allocate memory for a "2-d" array with $n$ rows and $m$ columns?

```
int **p = malloc(sizeof(int *) * n);
for (int i = 0; i < n; ++i)
  p[i] = malloc(sizeof(int) * m);

for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    scanf("%d", &p[i][j]);
```




`p` is a pointer to pointer to `int`,

- pointing to a sequence of pointers,
- each pointing to a sequence of `int`s.

## Use `malloc`

- Allocate memory for a "2-d" array with $n$ rows and $m$ columns?

  Another way: Allocate a "1-d" array of $nm$ elements:

```
int *p = malloc(sizeof(int) * n * m);
for (int i = 0; i < n; ++i)
  for (int j = 0; j < m; ++j)
    scanf("%d", &a[i * m + j]); // a[i * m + j] is the (i, j)-th entry
```

## Use `free`

- What pointer should be passed to `free`?
- What does `free(NULL)` do?
- What if we forget to `free`?
- After a call to `free(ptr)`, what does `ptr` become?
- What will happen if we `free` an address twice?

## Use `free`

- What pointer should be passed to `free`?
  - The pointer must be either **null** or **equal to a value returned earlier by an allocation function** (such as `malloc` and `calloc`).
- What does `free(NULL)` do?
  - Nothing.
- What if we forget to `free`?
  - Memory leak.

## Use `free`

- After a call to `free(ptr)`, what does `ptr` become?
  - `ptr` becomes a **dangling pointer**, which cannot be dereferenced.
- What will happen if we `free` an address twice?
  - Undefined behavior (and is often severe runtime error).

## Use `malloc` and `free`

Which of the following pieces of code deallocate(s) the memory correctly?

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

- `free(p);`
- `for (int i = 0; i < 100; ++i) free(p + i);`
- `free(p + 50); free(p);`
- `for (int i = 0; i < 10; ++i) free(p + i * 10);`

## Use `malloc` and `free`

Which of the following pieces of code deallocate(s) the memory correctly?

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

- `free(p);` **Yes**
- `for (int i = 0; i < 100; ++i) free(p + i);` **No**
- `free(p + 50); free(p);` **No**
- `for (int i = 0; i < 10; ++i) free(p + i * 10);` **No**

You cannot deallocate only a part of the memory block!

## Strings

- What is a string in C?
- How can we obtain the length of a string?
- How do we read / write a string?
- How does a function accept and handle a string?

## Strings

- What is a string in C?
  - A sequence of characters stored contiguously, with `'\0'` at the end.
- How can we obtain the length of a string?
  - `strlen(s)`
- How do we read / write a string?
  - `scanf` / `printf` with `"%s"`
  - `fgets`, `puts`

## Strings

- How does a function accept and handle a string?
  - The function accepts a `char *`, indicating the start of the string.
  - The end of the string is found by searching for the first appearance of `'\0'`.
  - What is the result of `printf(NULL)`?

## Strings

- How does a function accept and handle a string?
  - The function accepts a `char *`, indicating the start of the string.
  - The end of the string is found by searching for the first appearance of `'\0'`.
  - What is the result of `printf(NULL)`?
    - Undefined behavior! `printf` expects a string for the first argument, which should contain at least a character `'\0'`.

* Differentiate between **the null character `'\0'`**, **the empty string `""`** and **the null pointer `NULL`**.

# Command line arguments

## Command line arguments

The following command executes `gcc.exe`, and tells it the file to be compiled and the name of the output:

```
gcc hello.c -o hello
```

How are the arguments `hello.c`, `-o` and `hello` passed to `gcc.exe`?

- It is definitely different from "input".

## A new signature of `main`

```
int main(int argc, char **argv) { /* body */ }
```

Run this program with some arguments: `.\program one two three`

```
int main(int argc, char **argv) {
  for (int i = 0; i < argc; ++i)
    puts(argv[i]);
}
```

Output:

```
.\program
one
two
three
```

## A new signature of `main`

```c
int main(int argc, char **argv) { /* body */ }
```

where

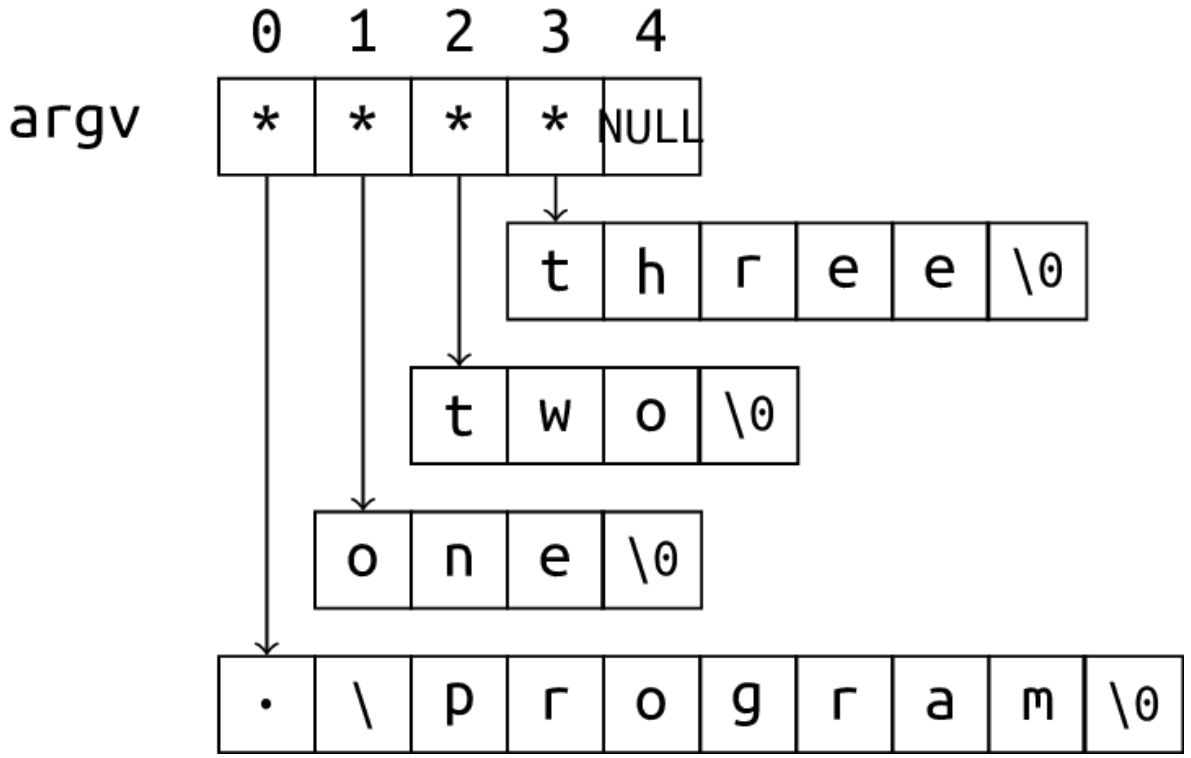
- `argc` is a non-negative value representing the number of arguments passed to the program from the environment in which the program is run.
- `argv` is a pointer to the first element of an array of `argc + 1` pointers, of which
  - the last one is null, and
  - the previous ones (if any) point to strings that represent the arguments.

If `argv[0]` is not null (or equivalently, if `argc > 0`), it points to a string representing the program name.

# Command line arguments

```
int main(int argc, char **argv) { /* body */ }
```

`argv` points to **an array of pointers** that point to the strings representing the arguments:



# Example: Read a string of unknown length

## Read a string

`fgets(str, count, stdin)` reads a string, but at most `count - 1` characters.

`scanf("%s", str)` reads a string, but not caring about whether the input content is too long to fit into the memory that `str` points to.

For example, the following code is likely to crash if the input is `responsibility`:

```
char word[6];
scanf("%s", word);
```

`scanf` does nothing to prevent the disaster.

- It does not even know how long the array `word` is!

## Read a string of unknown length

Suppose we want to read a sequence of non-whitespace characters, the length of which is unknown.

- Use `malloc` / `free` to allocate and deallocate memory dynamically.
- When the current buffer is not large enough, we allocate a larger one and copies the stored elements to it!

# Ignore leading whitespaces:

```
char *read_string(void) {
  char c = getchar();
  while (isspace(c))
    c = getchar();









}
```

# Set a buffer with initial capacity.

```
char *read_string(void) {
  char c = getchar();
  while (isspace(c))
    c = getchar();

  char *buffer = malloc(INITIAL_SIZE);
  int capacity = INITIAL_SIZE;
  int cur_pos = 0; // The index at which we store the input character.









}
```

# Write a loop to store and read characters.

```
char *read_string(void) {
  // ignore leading whitespaces

  char *buffer = malloc(INITIAL_SIZE);
  int capacity = INITIAL_SIZE;
  int cur_pos = 0; // The index at which we store the input character.

  while (!isspace(c)) {
    if (cur_pos == capacity - 1) { // `-1` is for '\0'.



    }
    buffer[cur_pos++] = c;
    c = getchar();
  }

}
```

# When the buffer is full, allocate a new one twice as large.

```
char *read_string(void) {
  // ...
  while (!isspace(c)) {
    if (cur_pos == capacity - 1) { // `-1` is for '\0'.
      char *new_buffer = malloc(capacity * 2);
      memcpy(new_buffer, buffer, cur_pos); // copy everything we have stored
                                           // to the new buffer
      capacity *= 2;
      buffer = new_buffer;
    }
    buffer[cur_pos++] = c;
    c = getchar();
  }

}
```

* Are we done?

# Do not forget to `free`!

```
char *read_string(void) {
  // ...
  while (!isspace(c)) {
    if (cur_pos == capacity - 1) { // `-1` is for '\0'.
      char *new_buffer = malloc(capacity * 2);
      memcpy(new_buffer, buffer, cur_pos); // copy everything we have stored
                                           // to the new buffer
      free(buffer);  // !!!!!!!!!!
      capacity *= 2;
      buffer = new_buffer;
    }
    buffer[cur_pos++] = c;
    c = getchar();
  }

}
```

# Don't consume more than what we need from the input.

```
char *read_string(void) {
  // ...
  while (!isspace(c)) {
    if (cur_pos == capacity - 1) { // `-1` is for '\0'.
      // ...
    }
    buffer[cur_pos++] = c;
    c = getchar();
  }

  // Now, `c` is a whitespace. This is not part of the contents we need.
  ungetc(c, stdin); // Put that whitespace back to the input.

  return buffer;
}
```

* Are we done?

# Don't forget the null character!

```
char *read_string(void) {
  // ...
  while (!isspace(c)) {
    if (cur_pos == capacity - 1) { // `-1` is for '\0'.
      // ...
    }
    buffer[cur_pos++] = c;
    c = getchar();
  }

  // Now, `c` is a whitespace. This is not part of the contents we need.
  ungetc(c, stdin); // Put that whitespace back to the input.

  buffer[cur_pos] = '\0'; // Remember this!!!

  return buffer;
}
```

## Use

```c
int main(void) {
  char *content = read_string();
  puts(content);
  free(content);
}
```

Remember to `free` it after use!