Memory Management in C
(Dynamic Strings)

Personal Software Engineering

Memory Organization

- The call stack grows from the top of memory down.
- Code is at the bottom of memory.
- Global data follows the code.
- What's left – the "heap" - is available for allocation.
Allocating Memory From The Heap

```c
void *malloc( unsigned nbytes )
```
- Allocates `nbytes` of memory in the heap.
- Guaranteed not to overlap other allocated memory.
- Returns pointer to the first byte (or `NULL` if the heap is full).
- Similar to constructor in Java – allocates space.
- Allocated space is uninitialized (random garbage).

```c
void free( void *ptr )
```
- Frees the memory assigned to `ptr`.
- The space `must` have been allocated by `malloc`.
- **No garbage collection in C (or C++).**
- Can slowly consume memory if not careful.
Examples: Make a Copy of a String

```c
#include <stdlib.h>
#include <string.h>

/*
 * Return a copy of an existing NUL-terminated string.
 */
char *make_copy(char *orig) {
    char *copy;
    copy = malloc(strlen(orig) + 1);
    strcpy(copy, orig);
    return copy;
}
```

Examples: Make a Copy of a String

```c
#include <stdlib.h>
#include <string.h>

/*
 * Return a copy of an existing NUL-terminated string.
 */
char *make_copy(char *orig) {
    char *copy;
    copy = malloc(strlen(orig) + 1);
    strcpy(copy, orig);
    return copy;
}
```

Uninitialized pointer - until we assign something to it we have NO idea where it points.
Examples: Make a Copy of a String

```c
#include <stdlib.h>
#include <string.h>

/*
 * Return a copy of an existing NUL-terminated string.
 */
char *make_copy(char *orig) {
    char *copy;
    copy = malloc(strlen(orig) + 1);
    strcpy(copy, orig);
    return copy;
}
```

Allocate space and assign address of first byte to pointer `<copy>`

Examples: Make a Copy of a String

```c
#include <stdlib.h>
#include <string.h>

/*
 * Return a copy of an existing NUL-terminated string.
 */
char *make_copy(char *orig) {
    char *copy;
    copy = malloc(strlen(orig) + 1);
    strcpy(copy, orig);
    return copy;
}
```

Enough space to hold the characters in `<orig>` plus the terminating `NUL`
Examples: Make a Copy of a String

```c
#include <stdlib.h>
#include <string.h>

/*
 * Return a copy of an existing NUL-terminated string.
 */
char *make_copy(char *orig) {
    char *copy;
    copy = malloc(strlen(orig) + 1);
    strcpy(copy, orig);
    return copy;
}
```

Once `copy` points to some space we can copy `orig` to that space.

Return the pointer to the allocated space with the desired string copy.
The caller now "owns" this space.
Examples: Catenate 2 Strings

/*
 * Return a pointer to concatenated strings.
 */
char *catenate(char *s1, char *s2) {
    char *cat;
    int space_needed = strlen(s1) + strlen(s2) + 1;
    cat = malloc(space_needed);
    strcpy(cat, s1);
    strcpy(cat + strlen(s1), s2);
    return cat;
}
Examples: Catenate 2 Strings

/*
 * Return a pointer to concatenated strings.
 */

char *catenate(char *s1, char *s2) {
    char *cat;
    int space_needed = strlen(s1) + strlen(s2) + 1;
    cat = malloc(space_needed);
    strcpy(cat, s1);
    strcpy(cat + strlen(s1), s2);
    return cat;
}
Examples: Catenate 2 Strings

/*
 * Return a pointer to concatenated strings.
 */
char *catenate(char *s1, char *s2) {
  char *cat;
  int space_needed = strlen(s1) + strlen(s2) + 1;
  cat = malloc(space_needed);
  strcpy(cat, s1);
  strcpy(cat + strlen(s1), s2);
  return cat;
}

Add string <s2> to the end of the copied <s1>.

Return the address of the final concatenated strings.
Caller now "owns" this space.
Example: Client Side

char *p1 = make_copy("Hello, ");
char *p2 = make_copy("world!");

char *p3 = catenate(p1, p2);
char *p4 = catenate("Hello, ", "world!");
Example: Client Side

char *p1 = make_copy("Hello, ");
char *p2 = make_copy("world!");

char *p3 = catenate(p1, p2);
char *p4 = catenate("Hello, ", "world!");

Concatenate the two constant strings.
Example: Client Side

char *p1 = make_copy("Hello, ");
char *p2 = make_copy("world!");

char *p3 = catenate(p1, p2);
char *p4 = catenate("Hello, ", "world!");

So what is the difference between the 2 calls to `catenate`?

The constant strings have preallocated static storage. The dynamic strings (p1 and p2) are in dynamically allocated space.
Example: Client Side

```c
char *p1 = make_copy("Hello, ");
char *p2 = make_copy("world!");

char *p3 = catenate(p1, p2);
char *p4 = catenate("Hello, ", "world!");
```

So what is the difference between the 2 calls to `catenate`?
The constant strings have preallocated static storage.
The dynamic strings (`p1` and `p2`) are in dynamically allocated space.

Dynamically allocated space must eventually be freed or memory will slowly fill up with unused garbage.

Example: Client Side

```c
char *p1 = make_copy("Hello, ");
char *p2 = make_copy("world!");

char *p3 = catenate(p1, p2);
char *p4 = catenate("Hello, ", "world!");
```

So what is the difference between the 2 calls to `catenate`?
The constant strings have preallocated static storage.
The dynamic strings (`p1` and `p2`) are in dynamically allocated space.

Dynamically allocated space should eventually be freed or memory will slowly fill up with unused garbage.

Example: suppose we only want the concatenated result in `p3`. Then:
```c
free(p1);
free(p2);
```
Problems: Orphan Storage

char *p1;
p1 = catenate("Merchant ", "of ");
p1 = catenate(p1, "Venice");

Result of first call on catenate:

```
p1 = Merchant of
```
Problems: Orphan Storage

```c
char *p1;
p1 = catenate("Merchant ", "of ");
p1 = catenate(p1, "Venice");
```

Result of first call on `catenate`:

```
p1  Merchant of
```

Result of second call on `catenate`:

```
p1  Merchant of
p1  Merchant of Venice
```

Permanently lost memory!
Problems: Dangling Reference

```c
char *p1;
char *p2;
p1 = catenate("Merchant ", "of ");
...
free(p1);
    . . . p1 not changed . . .
p2 = make_copy(p1);
```

Problems: Dangling Reference

```c
char *p1;
char *p2;
p1 = catenate("Merchant ", "of "); ← Allocate space assigned to p1
    . . .
free(p1);
    . . . p1 not changed . . .
p2 = make_copy(p1);
```
Problems: Dangling Reference

```c
char *p1;
char *p2;
p1 = catenate("Merchant ", "of ");

... free(p1);    // Free up space assigned to p1
    ... p1 not changed ...
p2 = make_copy(p1);
```

Problems: Dangling Reference

```c
char *p1;
char *p2;
p1 = catenate("Merchant ", "of ");

... free(p1);    // Reference to deallocated space!
    ... p1 not changed ...
p2 = make_copy(p1);
```
Moral of Our Story

THINK!
Moral of Our Story

THINK!

- Are you interested in the **pointer** or in what it **points to**?
- Random hacking won't work! You'll just tie yourself into knots.
Moral of Our Story

THINK!

- Are you interested in the *pointer* or in what it *points to*?
- Random hacking won't work! You'll just tie yourself into knots.
- MJL: After 45+ years in the field, I *still* have to reason carefully when using pointers - and I *still* make mistakes!

- If you are confused, lost, or bewildered: ask for help - *all* professionals need help at times.
Moral of Our Story

THINK!

- Are you interested in the `pointer` or in what it `points to`?
- Random hacking won't work! You'll just tie yourself into knots.
- MJL: After 45+ years in the field, I `still` have to reason carefully when using pointers - and I `still` make mistakes!
- If you are confused, lost, or bewildered: ask for help - `all` professionals need help at times.
- **BUT:** Be ready to explain `why` you did what you did.