CS107, Lecture 11 Extras
Stack and Heap

Reading: K&R 5.6-5.9 or Essential C section 6 on the heap
Ed Discussion: https://edstem.org/us/courses/46162/discussion/3644944
Each function **call** has its own *stack frame* for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

```c
int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
The Stack with Recursion

Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function **call** has its own *stack frame* for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own *stack frame* for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
The Stack with Recursion

Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
The Stack with Recursion

Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

Returns 1
Each function **call** has its own *stack frame* for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

Returns 6
The Stack with Recursion

Each function call has its own stack frame for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```

The memory diagram illustrates the stack frames for `main` and `factorial`. The memory initially starts at 0x0. The `main` function is on the stack, with `argc` and `argv` as arguments. The `factorial` function is called, and its stack frame is created below `main`. When `factorial` returns, the stack is popped. The call to `factorial(4)` returns 24.
Each function **call** has its own *stack frame* for its own copy of variables.

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
```
Each function call has its own stack frame for its own copy of variables.

int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

int main(int argc, char *argv[]) {
    printf("%d", factorial(4));
    return 0;
}
Extra Practice
strdup means string duplicate

How can we implement `strdup` using functions we’ve already seen?

```c
char *mystrdup(const char *str) {
    char *heapstr = _____(A)_____;
    _____(B)_____;
    _____(C)_____;
    return heapstr;
}
```

[Note] Use library functions:

- `<stdlib.h>`: malloc
- `<assert.h>`: assert
- `<string.h>`: strcpy, strlen
strdup means string duplicate

How can we implement `strdup` using functions we’ve already seen?

```c
1 char *mystrdup(const char *str) {
2     char *heapstr = malloc(strlen(str) + 1);
3     assert(heapstr != NULL);
4     strcpy(heapstr, str);
5     return heapstr;
6 }
```

char arrays differ from other arrays in that valid strings must be null-terminated (i.e., have an extra ending char).
(Note: library `strdup` doesn’t have an `assert`—it leaves the `assert` to the callee)
Where/how should we free memory below so that all memory is freed properly?

```c
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
  int *num = malloc(sizeof(int));
  *num = i;
  printf("%s %d
", ptr, *num);
}
printf("%s
", str);
```

**Recommendation:** Don’t worry about putting in frees until **after** you’re finished with functionality.
Memory leaks will rarely crash your CS107 programs.

**Answer in chat:**
“After line N: free(...);”
What if we didn’t free?

```
valgrind --leak-check=full --show-leak-kinds=all ...
```
Goodbye, Free Memory

Where/how should we free memory below so that all memory is freed properly?

```c
char *str = strdup("Hello");
assert(str != NULL);
char *ptr = str + 1;
for (int i = 0; i < 5; i++) {
    int *num = malloc(sizeof(int));
    *num = i;
    printf("%s %d\n", ptr, *num);
    free(num);
}
printf("%s\n", str);
free(str);
```

**Recommendation:** Don’t worry about putting in frees until *after* you’re finished with functionality.

Memory leaks will rarely crash your CS107 programs.
Write a function that takes in a heap-allocated `str1`, enlarges it, and concatenates `str2` onto it.

```c
char *strcat_extend(char *heap_str, const char *concat_str) {
    (_________________(1)__________________); // Allocate memory
    heap_str = realloc(___2A___,___2B___);
    (_________________(3)__________________); // Concatenate
    strcat(__(4A)___, ___4B___);
    return heapstr;
}
```

Example usage:
```c
cchar *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```
Write a function that takes in a heap-allocated `str1`, enlarges it, and concatenates `str2` onto it.

```c
char *strcat_extend(char *heap_str, const char *concat_str) {
    int new_length = strlen(heap_str) + strlen(concat_str) + 1;
    heap_str = realloc(heap_str, new_length);
    assert(heap_str != NULL);
    strcat(heap_str, concat_str);
    return heapstr;
}
```

Example usage:

```c
char *str = strdup("Hello ");
str = strcat_extend(str, "world!");
printf("%s\n", str);
free(str);
```