



CS107, Lecture 10 Extras

Stack and Heap

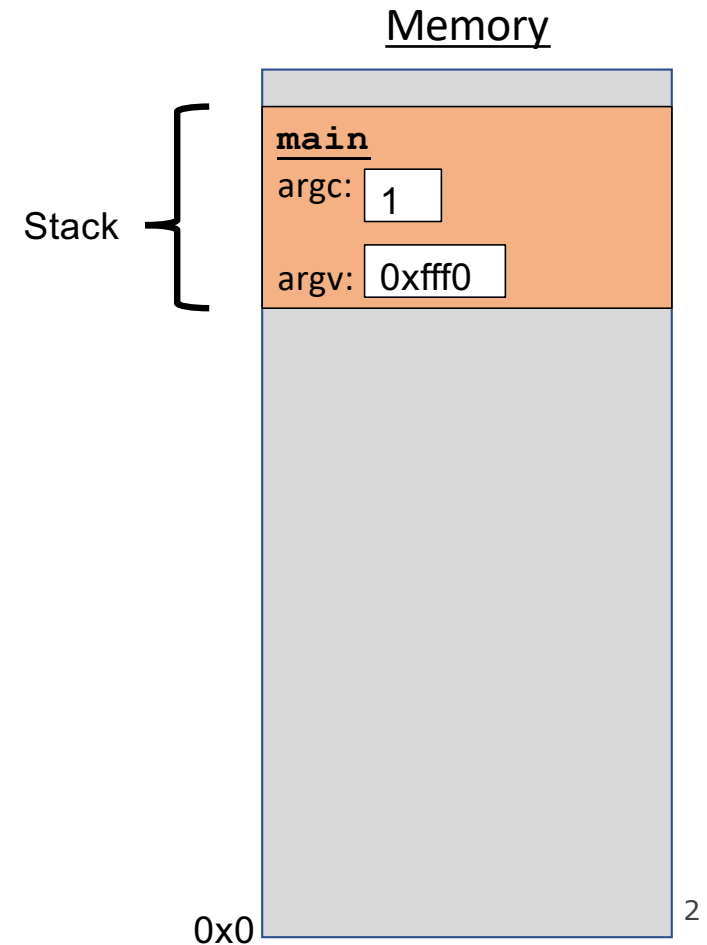
Reading: K&R 5.6-5.9 or Essential C section 6 on the heap

Ed Discussion: <https://edstem.org/us/courses/65949/discussion/5477907>

The Stack with Recursion

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

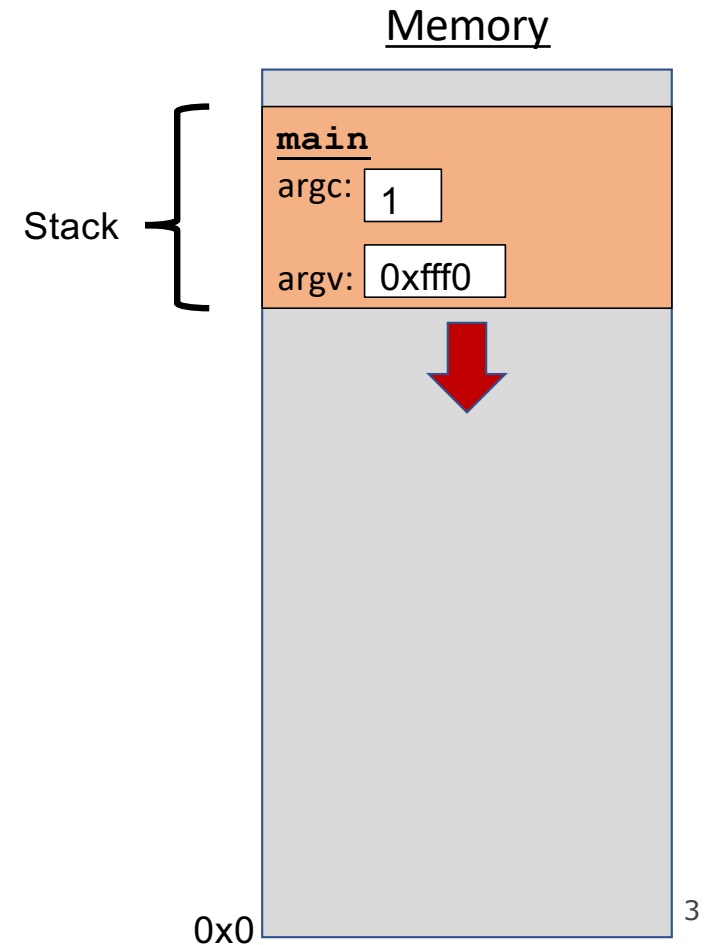
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

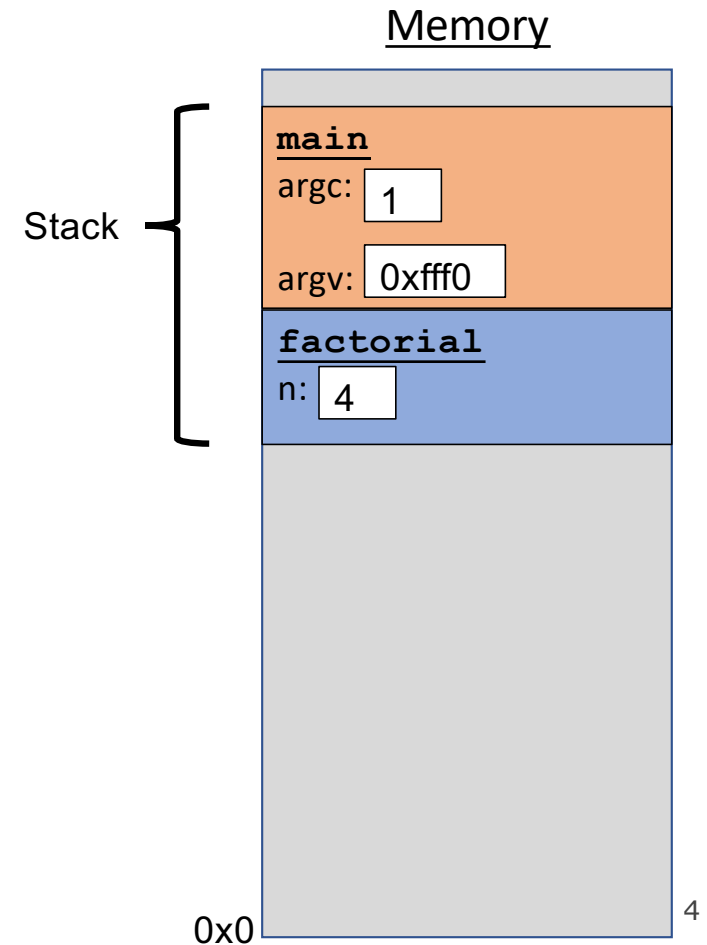
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

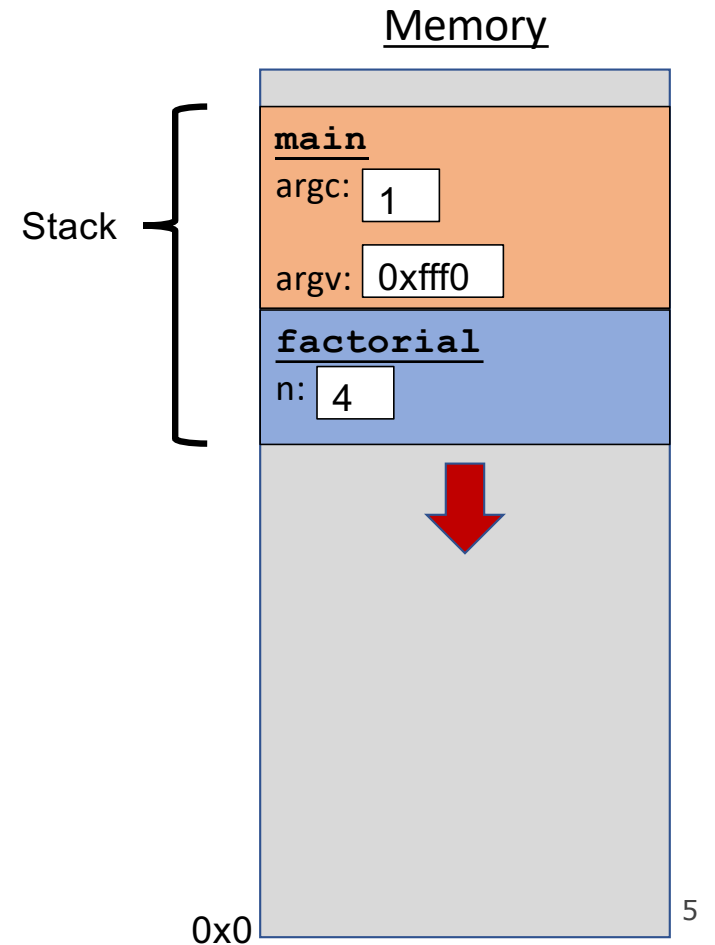
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

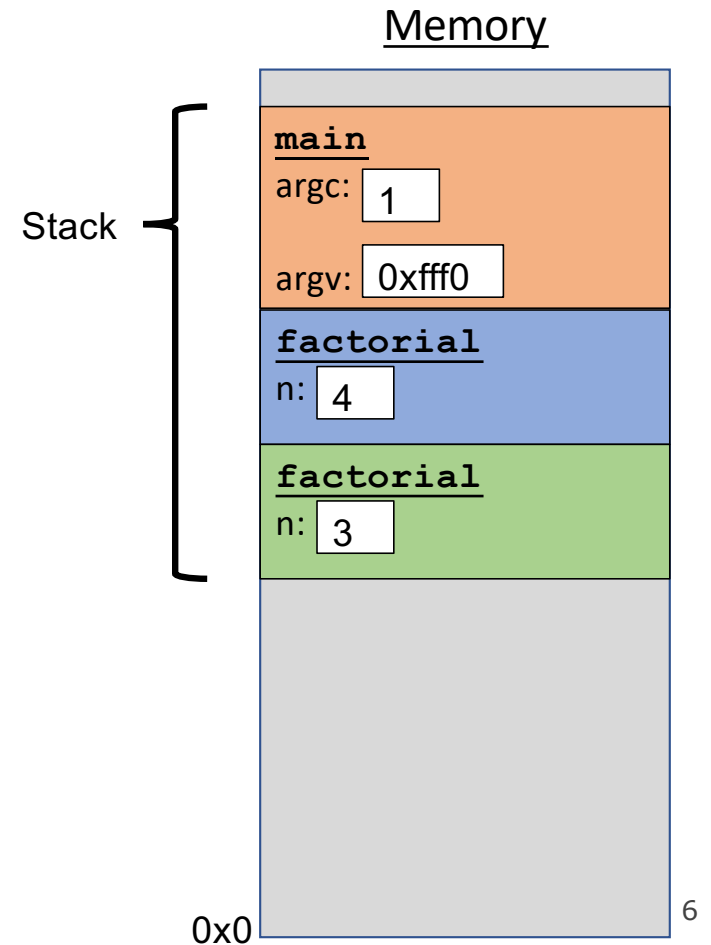
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

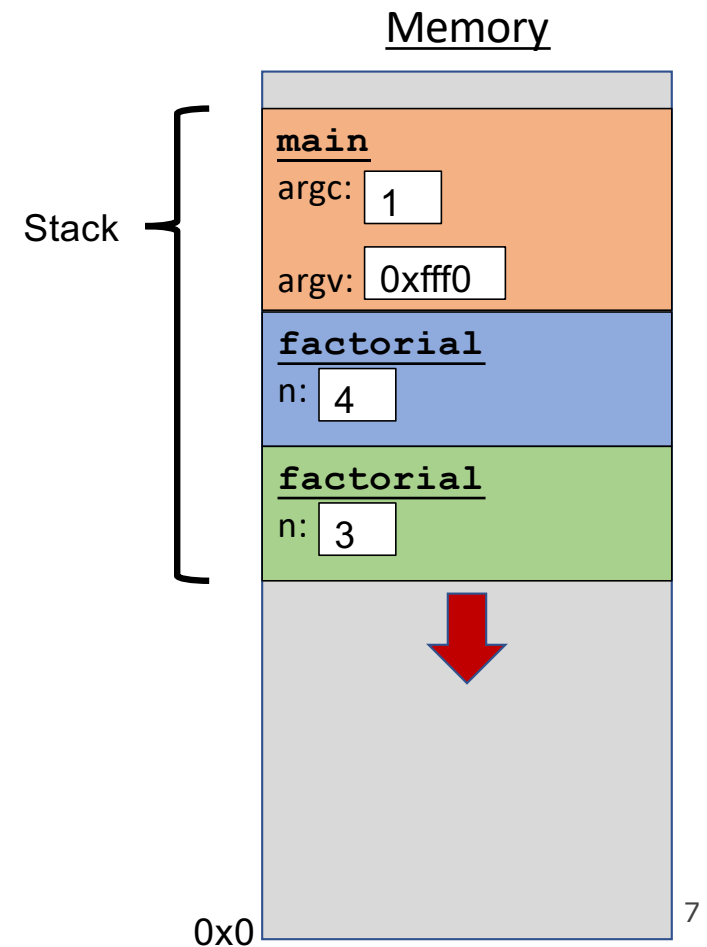
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

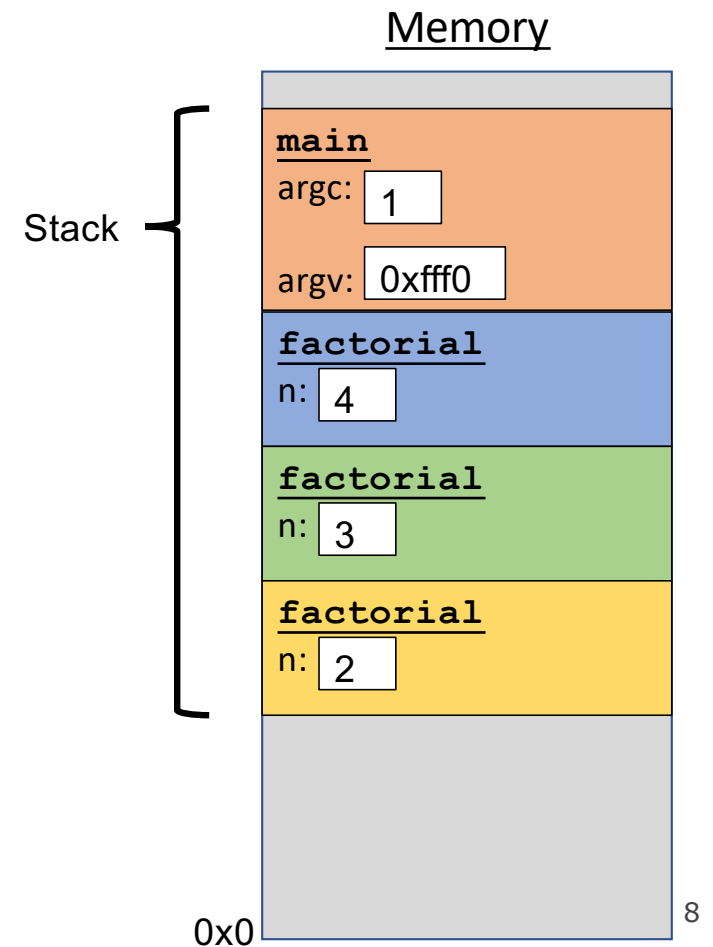
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

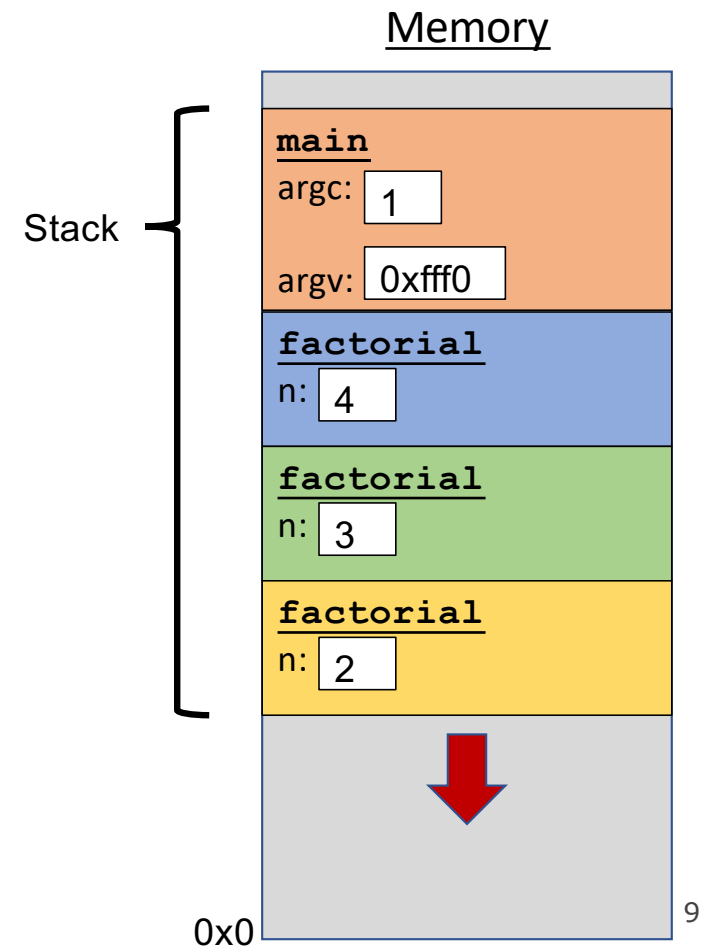
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

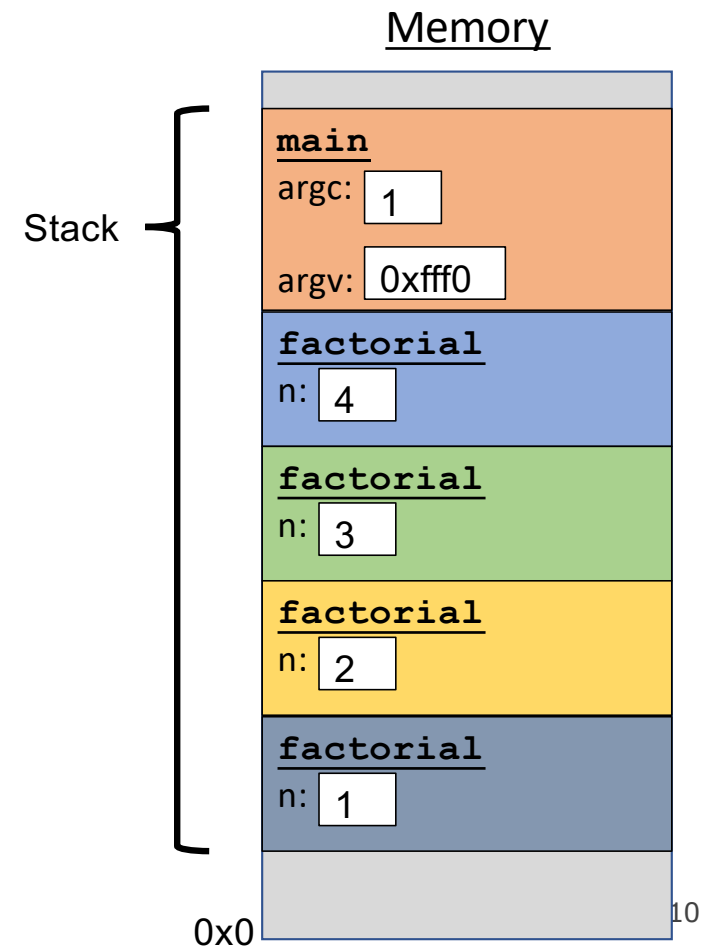
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

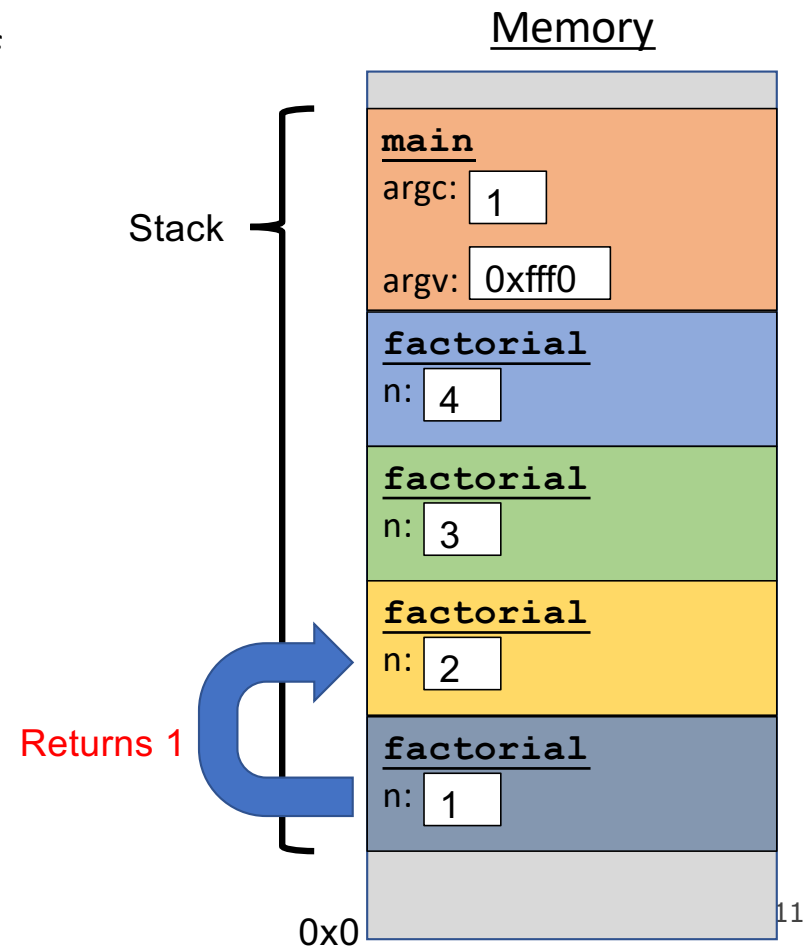
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

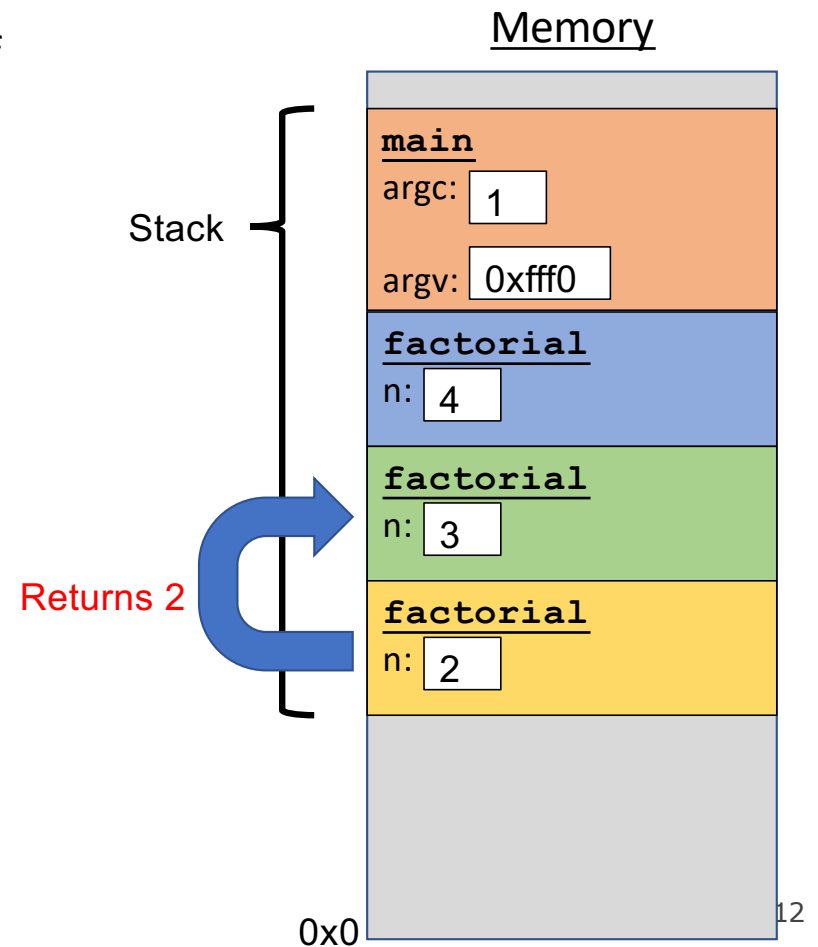
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

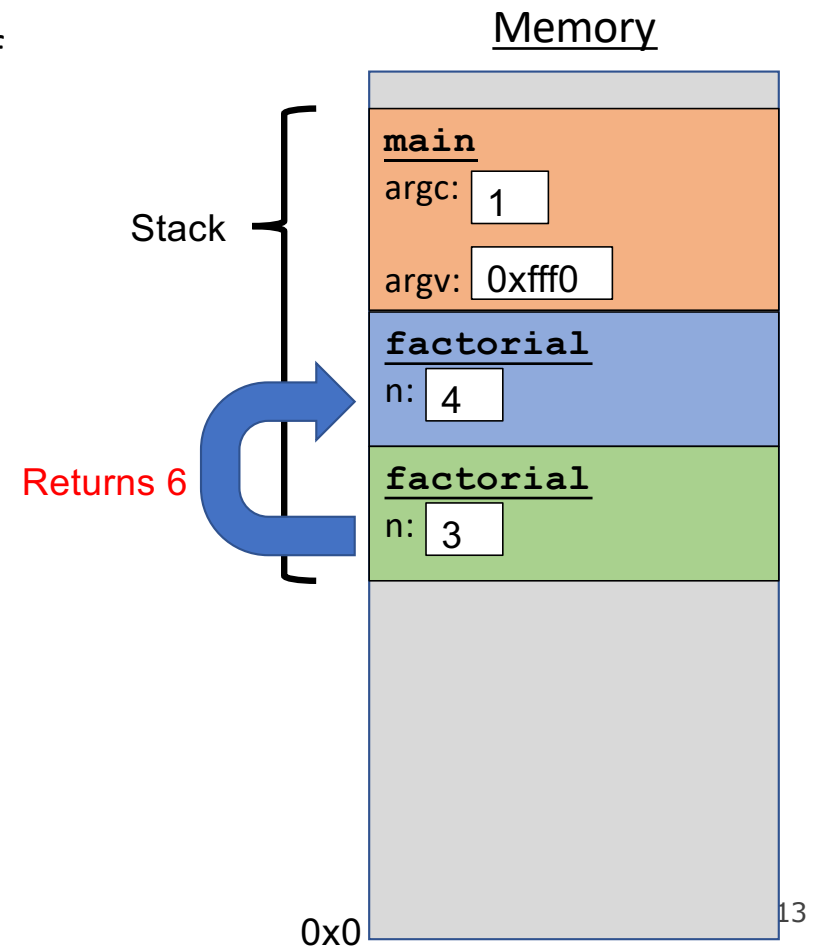
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

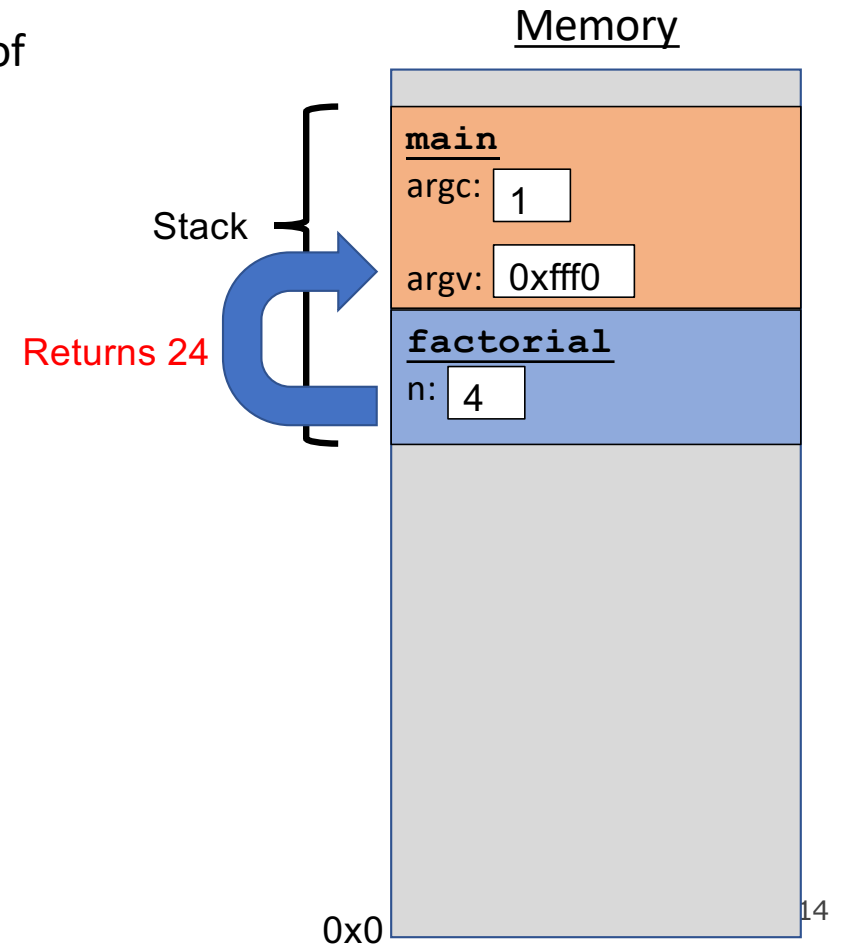
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

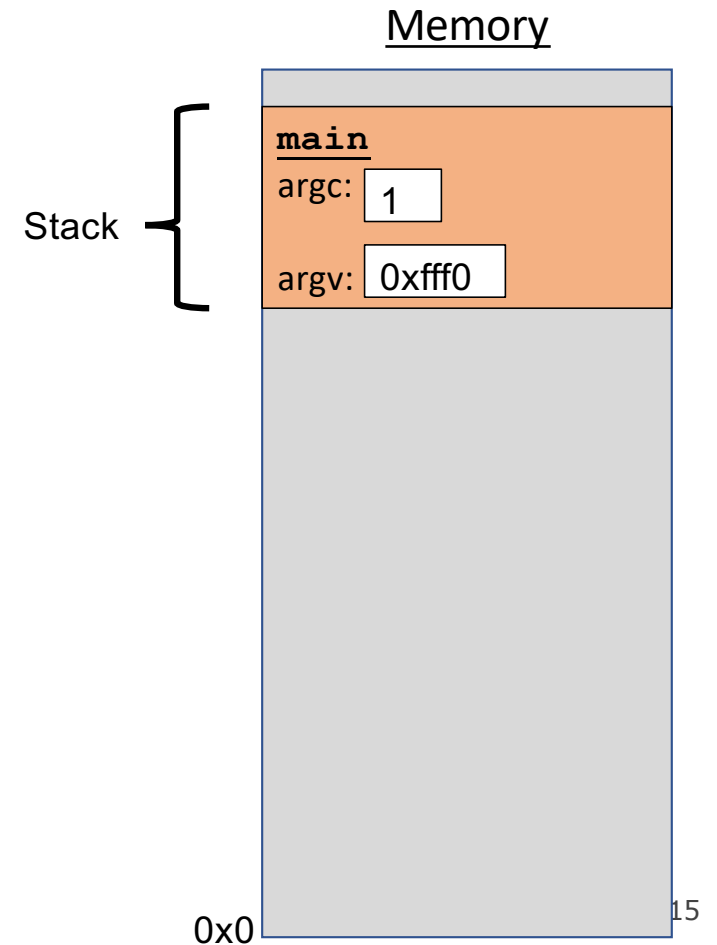
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```



The Stack with Recursion

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

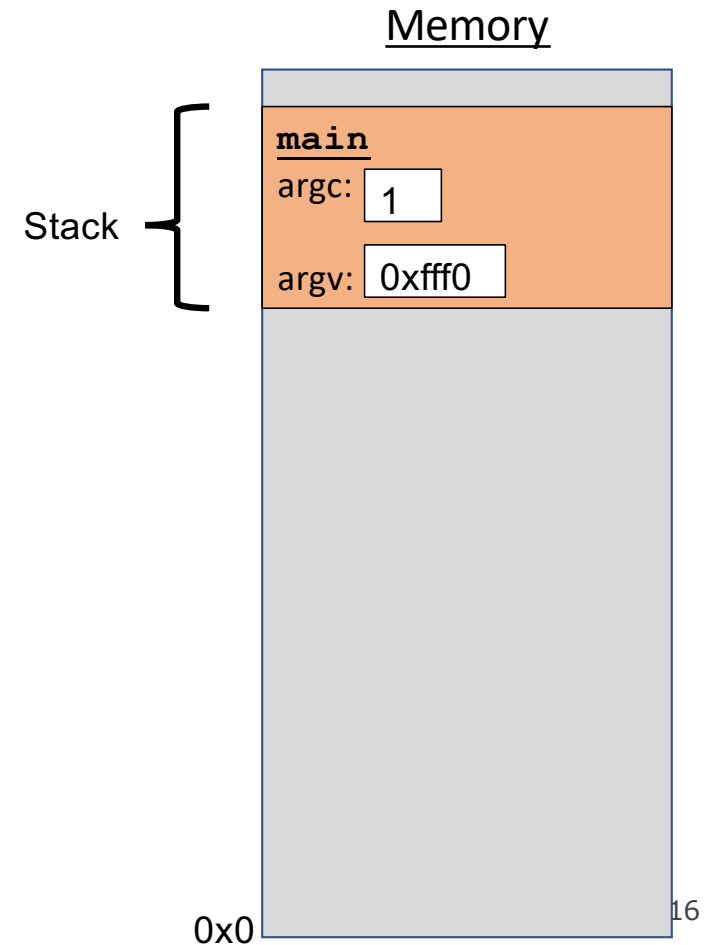
```
int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
int main(int argc, char *argv[]) {  
    printf("%d", factorial(4));  
    return 0;  
}
```

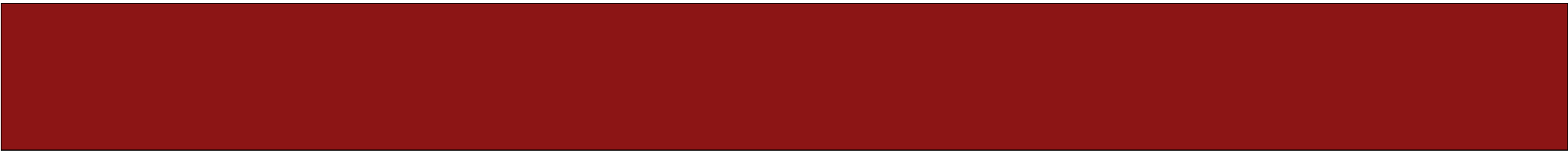


The Stack with Recursion

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?

```
1 char *mystrdup(const char *str) {  
2     char *heapstr = _____(A)_____;  
3     _____(B)_____;  
4     _____(C)_____;  
5     return heapstr;  
6 }
```

[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?

```
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)

Goodbye, Free Memory

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

```
1 char *str = strdup("Hello");
2 assert(str != NULL);
3 char *ptr = str + 1;
4 for (int i = 0; i < 5; i++) {
5     int *num = malloc(sizeof(int));
6     *num = i;
7     printf("%s %d\n", ptr, *num);
8 }
9 printf("%s\n", 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?

```
1 char *str = strdup("Hello");
2 assert(str != NULL);
3 char *ptr = str + 1;
4 for (int i = 0; i < 5; i++) {
5     int *num = malloc(sizeof(int));
6     *num = i;
7     printf("%s %d\n", ptr, *num);
8     free(num);
9 }
10 printf("%s\n", str);
11 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.

```
valgrind --leak-check=full --show-leak-kinds=all ...
```

strcat_extend

Write a function that takes in a heap-allocated **str1**, enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {  
2     (_____(1)_____);  
3     heap_str = realloc(____(2A)____, ____ (2B)____);  
4     (_____(3)_____);  
5     strcat(____(4A)____, ____ (4B)____);  
6     return heap_str;  
7 }
```

Example usage:

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



strcat_extend

Write a function that takes in a heap-allocated **str1**, enlarges it, and concatenates **str2** onto it.

```
1 char *strcat_extend(char *heap_str, const char *concat_str) {  
2     int new_length = strlen(heap_str) + strlen(concat_str) + 1;  
3     heap_str = realloc(heap_str, new_length);  
4     assert(heap_str != NULL);  
5     strcat(heap_str, concat_str);  
6     return heap_str;  
7 }
```

Example usage:

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