

Memory Model

CS 2022: Introduction to C

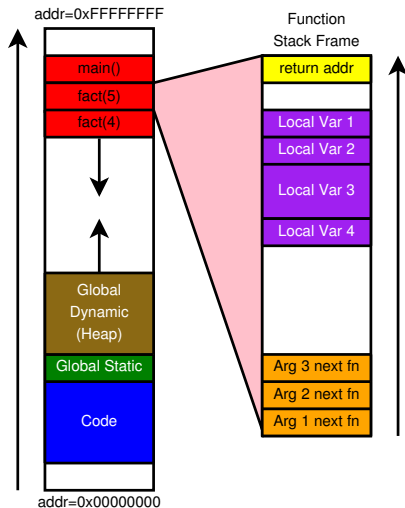
Instructor: Hussam Abu-Libdeh

(based on slides by Saikat Guha)

Fall 2011, Lecture 4

Memory

- ▶ Program code
- ▶ Function variables
 - ▶ Arguments
 - ▶ Local variables
 - ▶ Return location
- ▶ Global Variables
 - ▶ Statically Allocated
 - ▶ Dynamically Allocated



The Stack

Stores

- ▶ Function local variables
- ▶ Temporary variables
- ▶ Arguments for next function call
- ▶ Where to return when function ends

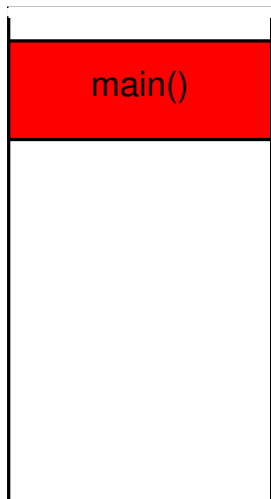
The Stack

Managed by compiler

- ▶ One stack frame each time function called
- ▶ Created when function called
- ▶ Stacked on top (under) one another
- ▶ Destroyed at function exit

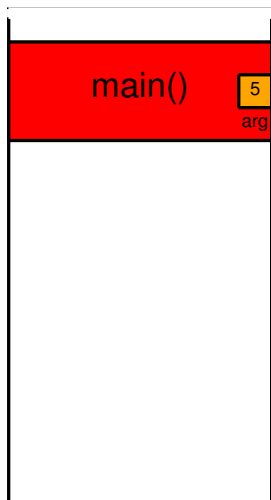
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



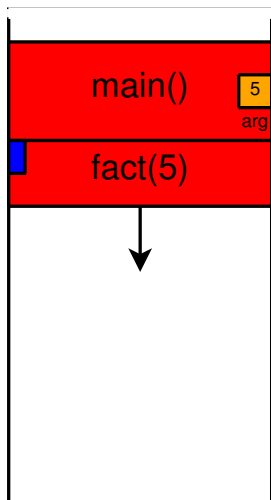
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



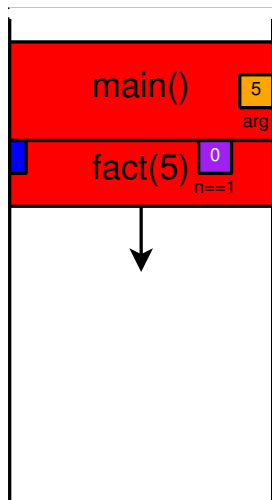
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



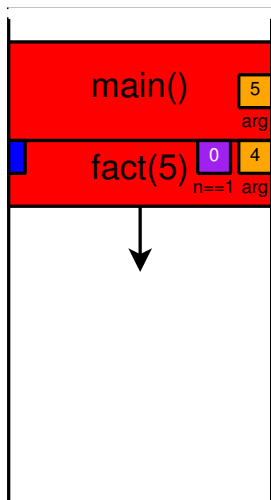
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



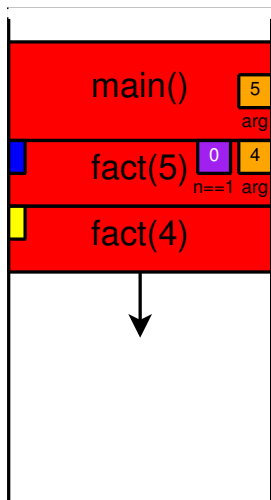
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



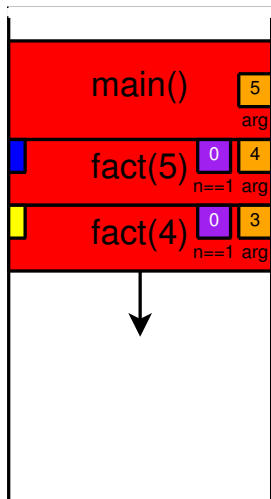
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



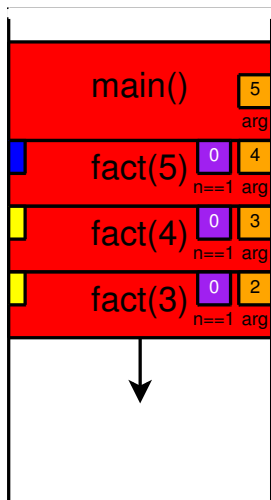
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



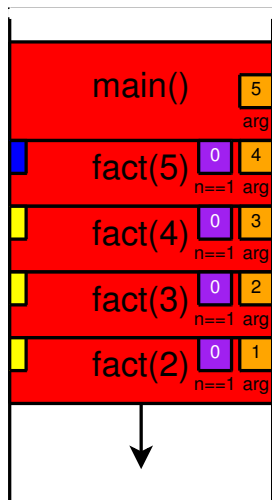
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



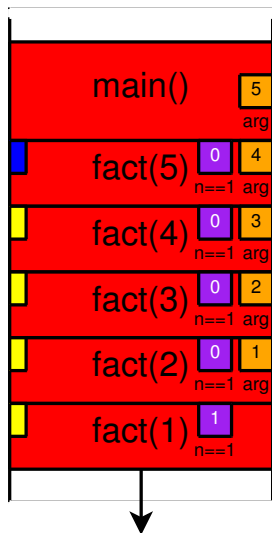
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



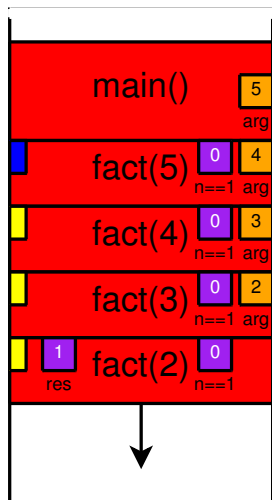
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



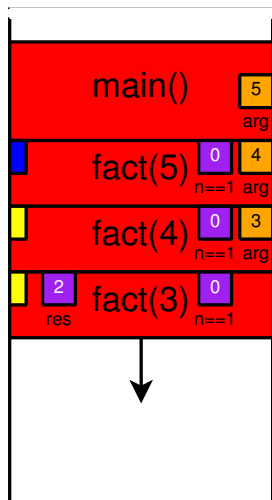
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



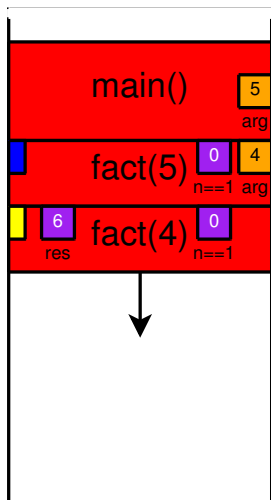
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



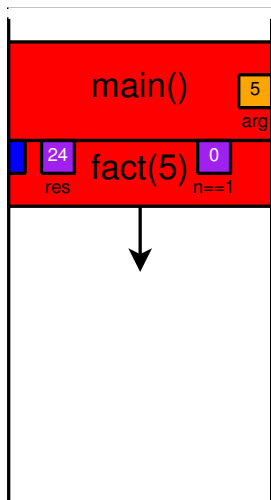
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



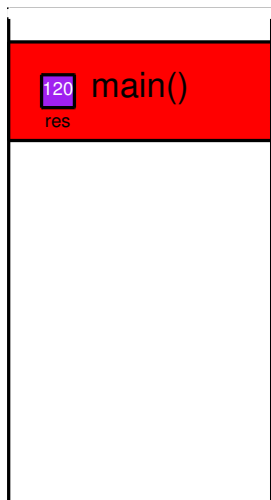
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



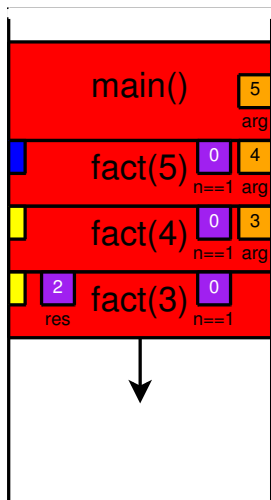
The Stack

```
int fact(int n) {  
    int res;  
    if (n == 1)  
        return 1;  
    res = fact(n-1);  
    return n * res;  
}  
  
int main() {  
    int res = fact(5);  
    return 0;  
}
```



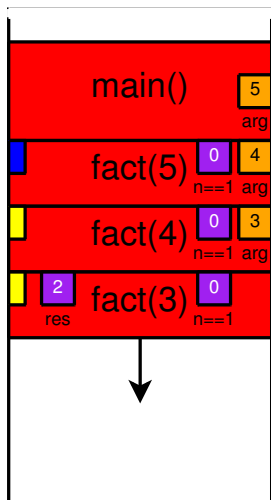
Stack games

- ▶ Locate the stack
- ▶ Find the direction of stack growth
- ▶ Finding size of stack frame



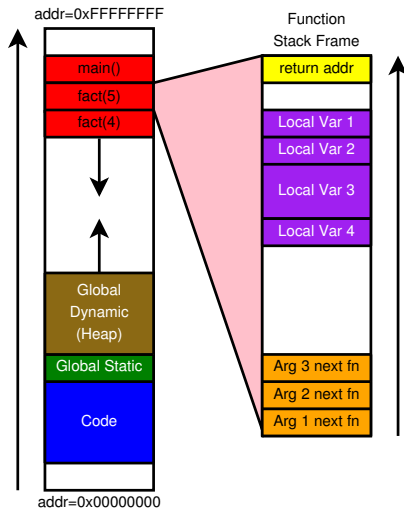
What can go wrong

- ▶ Run out of stack space
- ▶ Unintentionally change values on the stack
 - ▶ In some other function's frame
 - ▶ Even return address from function
- ▶ Access memory even after frame is deallocated



Memory Recap

- ▶ Program code
- ▶ Function variables
 - ▶ Arguments
 - ▶ Local variables
 - ▶ Return location
- ▶ Global Variables
 - ▶ Statically Allocated
 - ▶ Dynamically Allocated



Heap

Heap

Needed for long-term storage that needs to persist across multiple function calls.

Managed by programmer

- ▶ Created by `ptr = malloc(size)`
- ▶ Destroyed by `free(ptr)`

MUST check the return value from `malloc`

MUST explicitly free memory when no longer in use

Relevant Library

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

Includes definitions for `malloc()`, `free()`, and many other helpful functions.

- ▶ `void * malloc(size_t size);`
The `malloc()` function allocates *size* bytes of memory and returns a pointer to the allocated memory.
- ▶ `void free(void *ptr);`
The `free()` function deallocates the memory allocation pointed to by *ptr*.

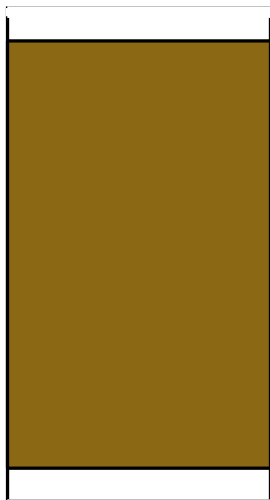
The Heap

```
int main() {
    int *p, *q, *r;

    p = (int *)malloc(sizeof(int));
    q = (int *)malloc(
        sizeof(int) * 10);
    r = (int *)malloc(sizeof(int));

    if (p == NULL || !q || !r) {
        ... do cleanup ...
        return 1;
    }

    free(p);
    ... do other stuff ...
}
```



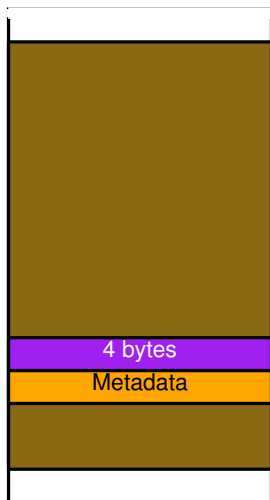
The Heap

```
int main() {
    int *p, *q, *r;

    p = (int *)malloc(sizeof(int));
    q = (int *)malloc(
        sizeof(int) * 10);
    r = (int *)malloc(sizeof(int));

    if (p == NULL || !q || !r) {
        ... do cleanup ...
        return 1;
    }

    free(p);
    ... do other stuff ...
}
```



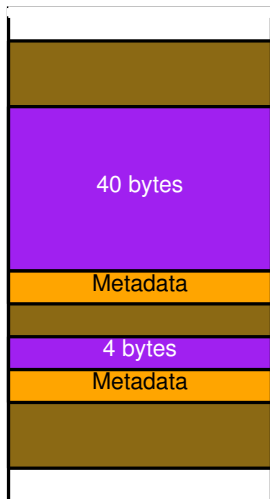
The Heap

```
int main() {
    int *p, *q, *r;

    p = (int *)malloc(sizeof(int));
    q = (int *)malloc(
        sizeof(int) * 10);
    r = (int *)malloc(sizeof(int));

    if (p == NULL || !q || !r) {
        ... do cleanup ...
        return 1;
    }

    free(p);
    ... do other stuff ...
}
```



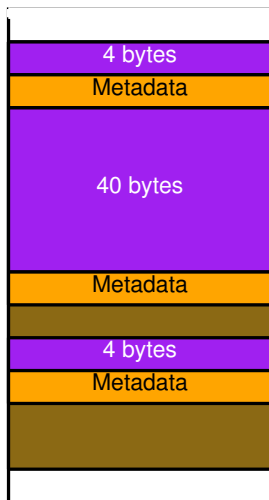
The Heap

```
int main() {
    int *p, *q, *r;

    p = (int *)malloc(sizeof(int));
    q = (int *)malloc(
        sizeof(int) * 10);
    r = (int *)malloc(sizeof(int));

    if (p == NULL || !q || !r) {
        ... do cleanup ...
        return 1;
    }

    free(p);
    ... do other stuff ...
}
```



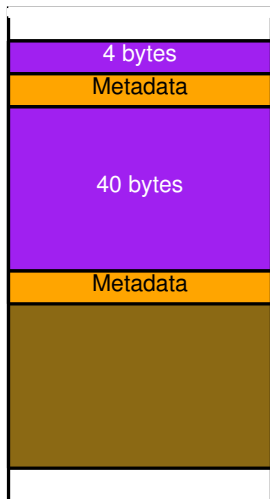
The Heap

```
int main() {
    int *p, *q, *r;

    p = (int *)malloc(sizeof(int));
    q = (int *)malloc(
        sizeof(int) * 10);
    r = (int *)malloc(sizeof(int));

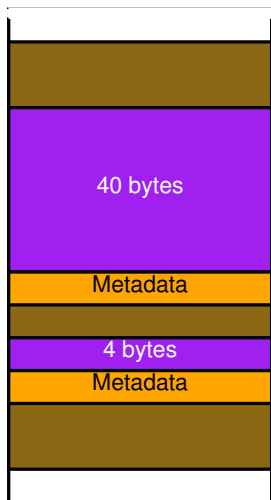
    if (p == NULL || !q || !r) {
        ... do cleanup ...
        return 1;
    }

    free(p);
    ... do other stuff ...
}
```



Heap games

- ▶ Locate the heap
- ▶ How freespace is managed
- ▶ Find how memory is allocated
 - ▶ How is fragmentation avoided



What can go wrong

- ▶ Run out of heap space `malloc` returns 0
- ▶ Unintentionally change other heap data
 - ▶ Or clobber heap metadata
- ▶ Access memory after free'd
- ▶ free memory twice
- ▶ Create a memory leak

