Welcome to the CS3410 C Primer

Please sit in the front rows so that you can see terminal output

If you can't read this, then you are too far away

C Primer

CS3410
Paul Upchurch & Jason Yosinski

Material

Introduction to writing C programs on a UNIX system.

Same material as CS2022, but condensed into three 2-hour sessions.

Knowledge of a modern high-level language is helpful (C++, Java). Otherwise, Google is your friend.

Schedule

January 28 Monday	Hello World, pointers, memory model, UNIX
February 7 Thursday	Arrays, structured data, debugging, I/O (file and network)
February 11 Monday	Preprocessor, serialization, threads, advanced topics (goto, exceptions, assembly), C for Java programmers

More info

See the course web page for CS2022.

Slides, homeworks and example code by Hussam Abu-Libdeh.

www.cs.cornell.edu/courses/CS2022/2011fa/

UNIX Access

All students have UNIX accounts in the CSUGLab.

1. Create your password at http://www.csuglab.cornell.edu/userinfo/

2. ssh to csugXX.csuglab.cornell.edu

This info will be on the first homework.

Introduction to C

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

(based on slides by Saikat Guha)

Fall 2011, Lecture 1

History of C

- Writing code in an assembler gets real old real fast
  - Really low level (no loops, functions, if-then-else)
  - Not portable (different for each architecture)
- BCPL (by Martin Richards): Grandparent of C
  - Close to the machine
  - Procedures, Expressions, Statements, Pointers, ...
- B (by Ken Thompson): Parent of C
  - Simplified BCPL
  - Some types (int, char)

History of C

- C (by Kernighan and Ritchie)
  - Much faster than B
  - Arrays, Structures, more types
- Standardization
- Portability enhanced
- Parent of Objective C, Concurrent C, C*, C++

When to use C

- Working close to hardware
  - Operating System
  - Device Drivers
- Need to violate type-safety
  - Pack and unpack bytes
  - Inline assembly
- Cannot tolerate overheads
  - No garbage collector
  - No array bounds check
  - No memory initialization
  - No exceptions

When not to use C

Use JAVA or C# for ...

- Quick prototyping
  - Python or Ruby are even better here
- Compile-once Run-Everywhere
- Reliability is critical, and performance is secondary
  - C can be very reliable, but requires tremendous programmer discipline
  - For many programs, JAVA can match C performance, but not always

Hello World

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

(based on slides by Saikat Guha)

Fall 2011, Lecture 2

Environment

- OS: GNU/Linux
- Editor: vim
- Compiler: gcc
- Debugger: gdb

Structure of a C Program

Overall Program

```
<some pre-processor directives>

<global declarations>
<global variables>

<functions>
```

Structure of a C Program

Functions

```
<function header>
<local declarations>

<statements>
```

hello.c: Hello World

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Compiling and Running

- `$ gcc -o hello hello.c`
- `$ ./hello`
  `Hello World`

vars.c: Variables

```
#include <stdio.h>

int main() {
    int a, b, c;

    a = 10;
    b = 20;
    c = a * b;

    printf("a=%d b=%d c=%d\n", a, b, c);
    return 0;
}
```

```
a=10 b=20 c=200
```

func.c: Functions

```
#include <stdio.h>

int add(int a, int b) {
    printf("a=%d b=%d\n", a, b);
    return a+b;
}

int main() {
    printf("ret=%d\n", add(10, 20));
    return 0;
}
```

```
a=10 b=20
ret=30
```

cond.c: Conditionals

```
#include <stdio.h>

int main() {
    int i = 10;
    if (10 == i) {
        printf("equal to ten\n");
    } else {
        printf("not equal to ten\n");
    }
    return 0;
}
```

```
equal to ten
```

loop.c: Loops

```
#include <stdio.h>

int main() {
    int i;
    for (i = 0; i < 10; i++) {
        printf("%d ", i);
    }
    printf("done.\n");
    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9 done.
```

rec.c: Recursion

```
#include <stdio.h>

void rec(int a) {
    printf("in %d\n", a);
    if (a > 0) rec(a-1);
    printf("out %d\n", a);
}

int main() {
    rec(2);
    return 0;
}
```

```
in 2
in 1
in 0
out 0
out 1
out 2
```

cmdarg.c: Command Line Args

```
#include <stdio.h>

int main(int argc, char **argv) {
    int n, m;

    n = atoi(argv[1]);
    m = atoi(argv[2]);

    printf("Argument 1: %d\nArgument 2: %d\n", n, m);

    return 0;
}
```

```
Argument 1:  10
Argument 2:  20
```

Pointers

CS 2022: Introduction to C

Instructor: Hussam Abu-Libdeh

(based on slides by Saikat Guha)

Fall 2011, Lecture 3

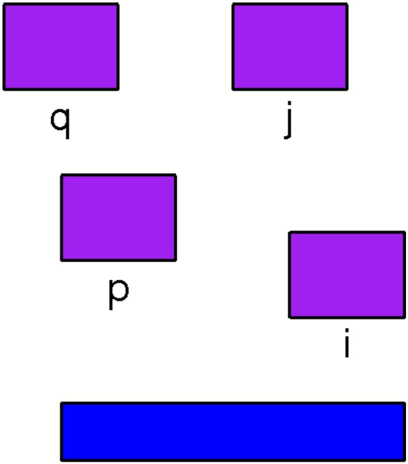
Pointer

Pointer

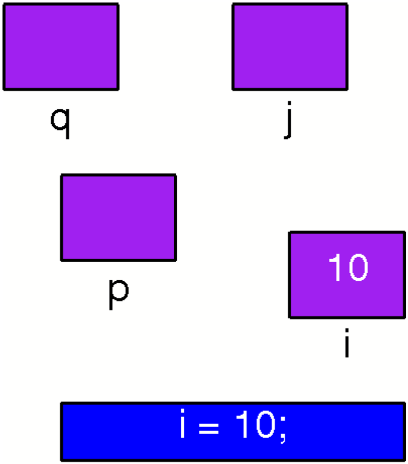
A pointer is just another variable that points to another variable. A pointer contains the memory address of the variable it points to.

```
int i;                  // Integer
int *p;                 // Pointer to integer
int **m;                // Pointer to int pointer

p = &i;                 // p now points to i
printf("%p", p);        // address of i (in p)

m = &p;                 // m now points to p
printf("%p", m);        // address of p (in m)
```

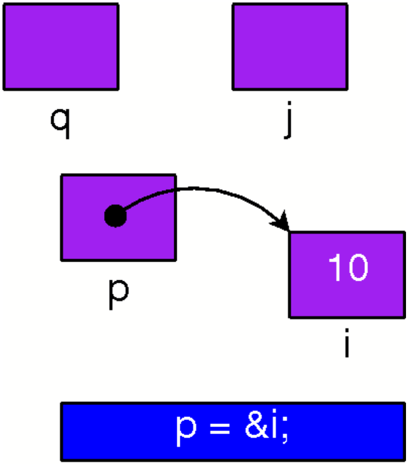
Pointers



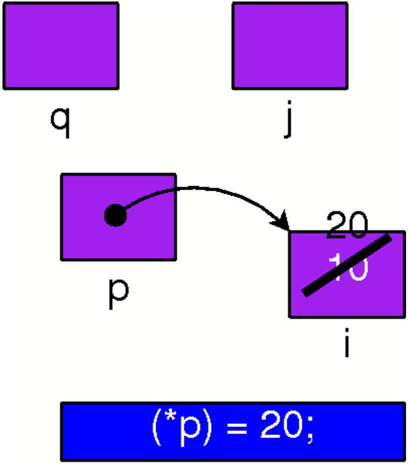
Pointers



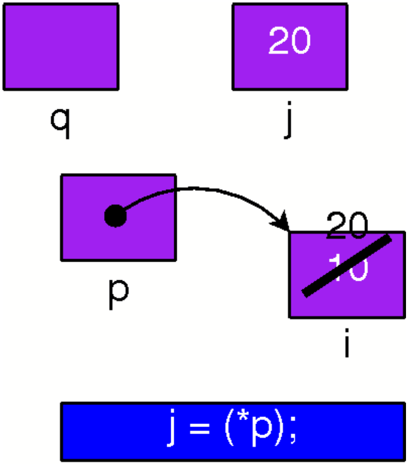
Pointers



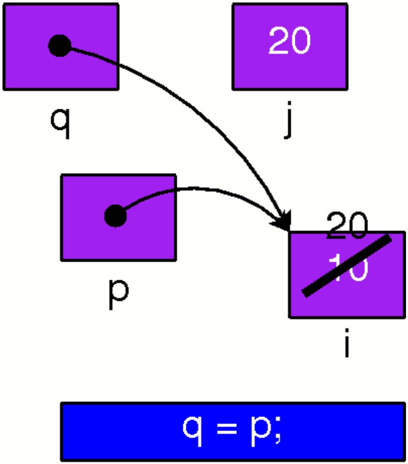
Pointers



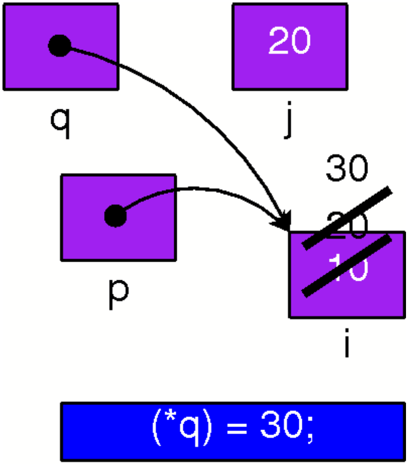
Pointers



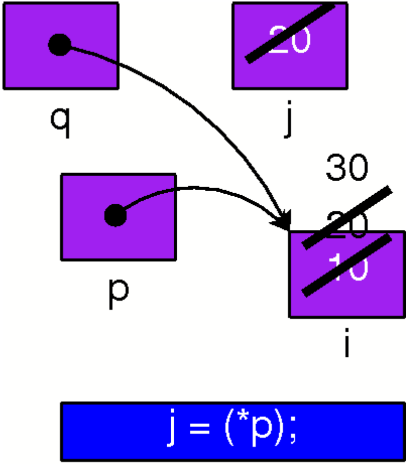
Pointers



Pointers



Pointers



swap1.c: Swap

```
#include <stdio.h>
int main() {
    int a, b;
    int *p, *q;

    a = 10; b = 20;
    p = &a; q = &b;
    printf("Before: %d, %d, %d, %d",
        a, b, *p, *q);

    p = &b;
    q = &a;

    printf("After: %d, %d, %d, %d",
        a, b, *p, *q);
    return 0;
}
```

```
Before:  10, 20, 10, 20
After:  10, 20, 20, 10
```

swap2.c: Swap

```
#include <stdio.h>
int main() {
    int a, b;
    int *p, *q;

    a = 10; b = 20;
    p = &a; q = &b;
    printf("Before: %d, %d, %d, %d",
        a, b, *p, *q);

    a  = 20;
    b  = 10;

    printf("After: %d, %d, %d, %d",
        a, b, *p, *q);
    return 0;
}
```

```
Before:  10, 20, 10, 20
After:  20, 10, 20, 10
```

swap3.c: Swap

```
#include <stdio.h>
int main() {
    int a, b;
    int *p, *q;

    a = 10; b = 20;
    p = &a; q = &b;
    printf("Before: %d, %d, %d, %d",
        a, b, *p, *q);

    a  = 20; b  = 10;
    p  = &b; q  = &a;

    printf("After: %d, %d, %d, %d",
        a, b, *p, *q);
    return 0;
}
```

```
Before:  10, 20, 10, 20
After:  20, 10, 10, 20
```

Pointers to Pointers!

```
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    int *p = &a, *q = &b;
    int **m = &p, **n = &q;

    printf("X: %d %d %d %d %d %d\n",
        **m, **n, *p, *q, a, b);

    *m = *n; m = n;
    *m = &a; n = &p;
    **n = 30;

    printf("Y: %d %d %d %d %d %d\n",
        **m, **n, *p, *q, a, b);
    return 0;
}
```

X: __ __ __ __ __ __

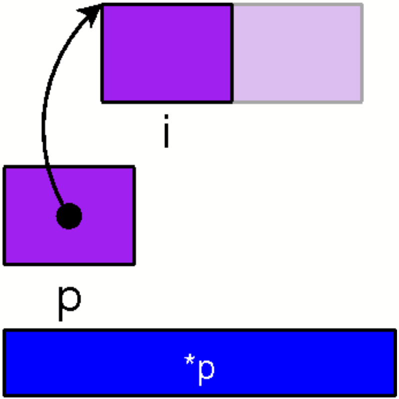
Y: __ __ __ __ __ __

Pointers to Pointers!

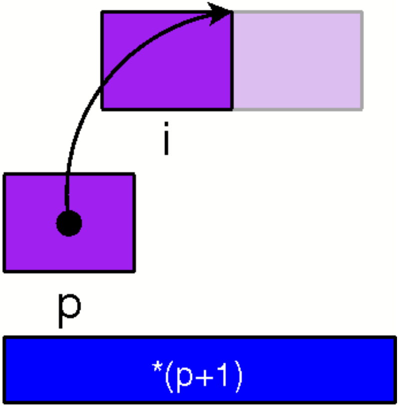
```
#include <stdio.h>
int main() {
    int a = 10, b = 20;
    int *p = &a, *q = &b;
    int **m = &p, **n = &q;

    printf("X: %d %d %d %d %d %d\n",
        **m, **n, *p, *q, a, b);

    *m = *n; m = n;
    *m = &a; n = &p;
    **n = 30;

    printf("Y: %d %d %d %d %d %d\n",
        **m, **n, *p, *q, a, b);
    return 0;
}
```

```
X: 10 20 10 20 10 20
Y: 10 30 30 10 10 30
```

Pointer Arithmetic



Pointer Arithmetic



Memory in C

Variables

- Independent variables are a figment of your imagination.
- When in C, think of memory cells. Each memory cell has an integer address.
- You can access any memory cell at any time from any function.
- Variable names are simply shortcuts for your convenience.

Nameless Variables

```
#include <stdlib.h>

int main() {
    int *p = (int *)malloc(sizeof(int));

    *p = 42;
    return 0;
}
```

A poor man's array

```
int * newarray(int siz) {
    return (int *)malloc(siz * sizeof(int));
}

void set(int *arr, int idx, int val) {
   *(arr+idx)  = val;
}

int get(int *arr, int idx) {
    return *(arr + idx);
}
```

Multiple Return Values

```
void getab(int *a, int *b) {
    *a = 10;
    *b = 20;
}

int main() {
    int a, b;

    getab(&a, &b);
}
```

Pointers Recap

- `int *ptr;`
- Pointers are variables that store memory addresses of other variables
- Type of variable pointed to depends on type of pointer:
  - `int *ptr` points to an integer variable
  - `char *ptr` points to a character variable
  - Can cast between pointer types:
    `my_int_ptr = (int *) my_other_ptr;`
  - `void *ptr` has an unspecified type; must be cast to a type before used

Pointers Recap

- Two main operations:
  - ∗ *dereference*: gets the value at the memory location stored in a pointer
  - & *address of*: gets the address of a variable
  - `int *my_ptr = &my_var;`
- Pointer arithmetic: directly manipulate a pointer's content to access other memory locations
  - **Use with caution!**: can crash your program due to bad memory accesses
  - However, it is useful in accessing and manipulating data structures
- Pointers to pointers
  - `int **my_2d_array;`

Memory Model

CS 2022: Introduction to C

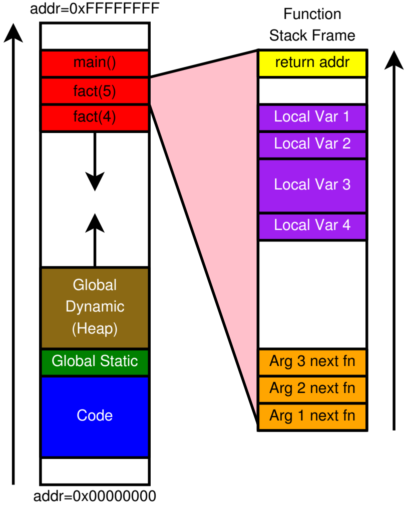
Instructor: Hussam Abu-Libdeh

(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

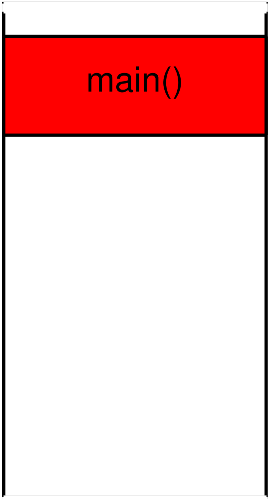
The Stack

Managed by compiler

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

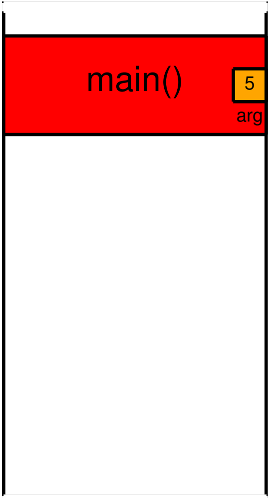
The Stack

```
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;
}
```



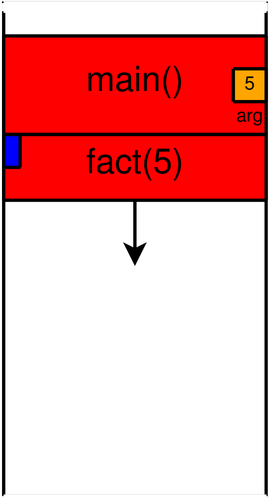
The Stack

```
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;
}
```



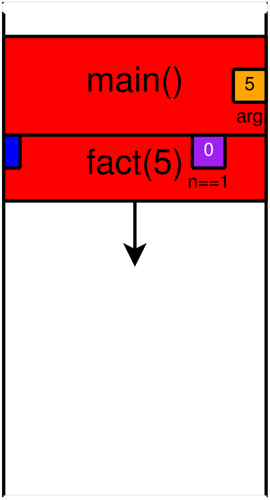
The Stack

```
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;
}
```



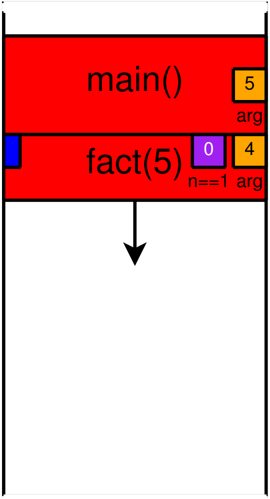
The Stack

```
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;
}
```



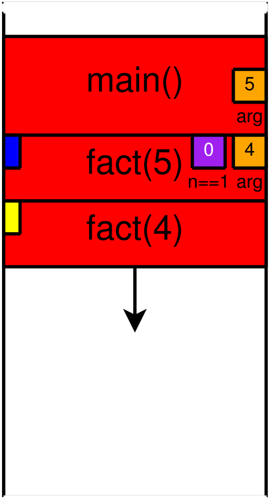
The Stack

```
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;
}
```



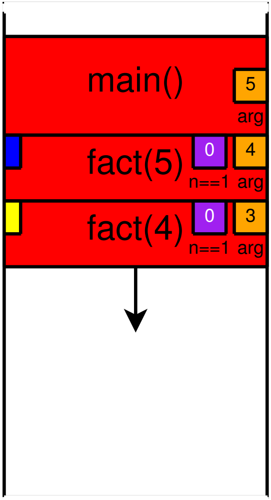
The Stack

```
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;
}
```



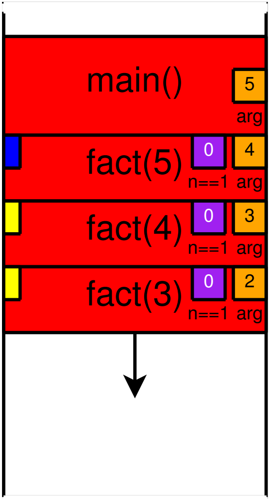
The Stack

```
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;
}
```



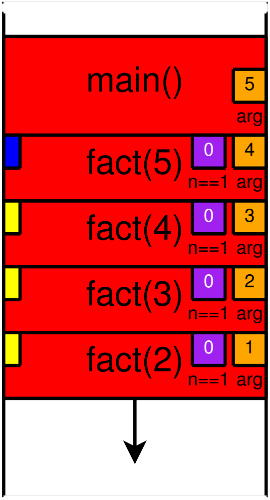
The Stack

```
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;
}
```



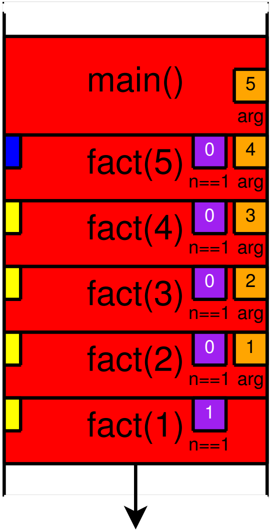
The Stack

```
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;
}
```



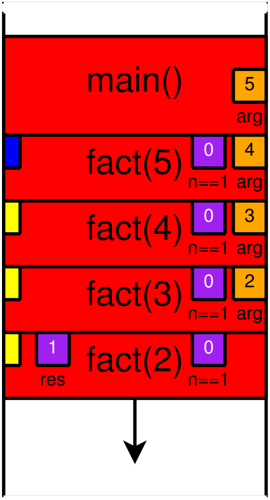
The Stack

```
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;
}
```



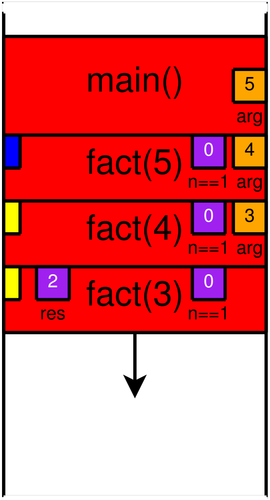
The Stack

```
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;
}
```



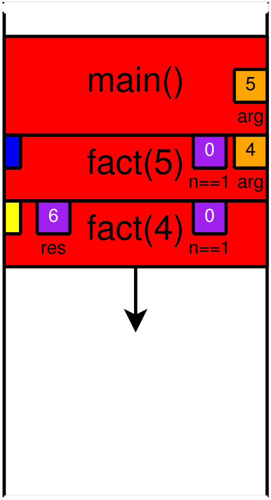
The Stack

```
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;
}
```



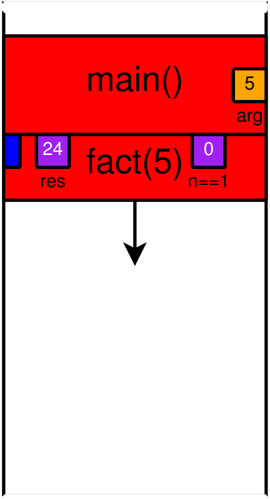
The Stack

```
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;
}
```



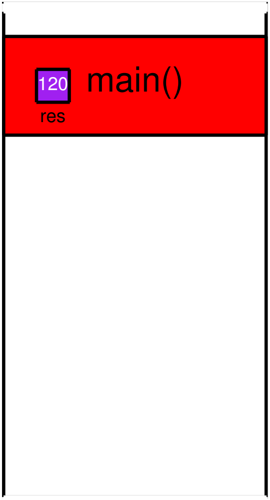
The Stack

```
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;
}
```



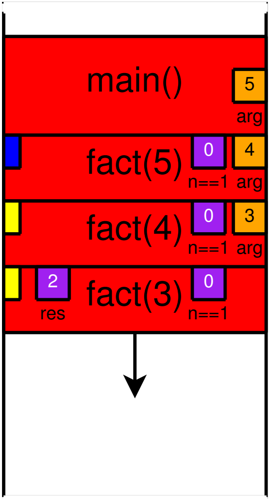
The Stack

```
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;
}
```



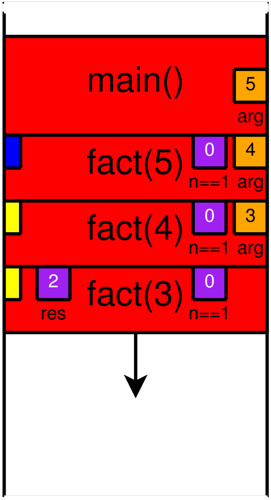
Stack games

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



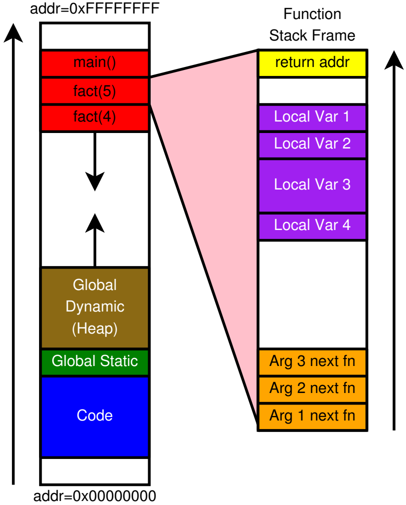
What can go wrong

- 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

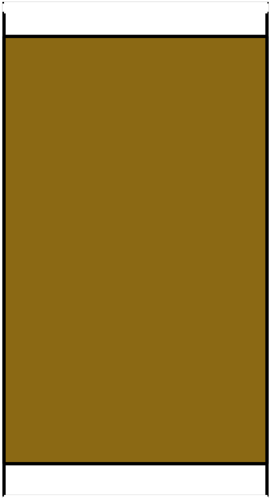
`#include <stdlib.h>`

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

- `void * malloc(size_t size);`
  The `malloc()` function alocates *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*.

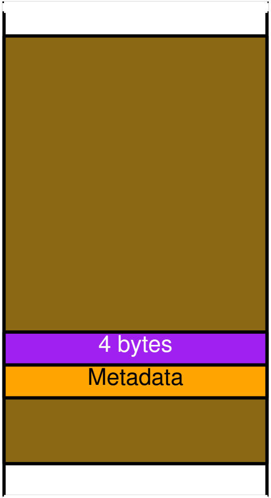
The Heap

```
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 ...
```



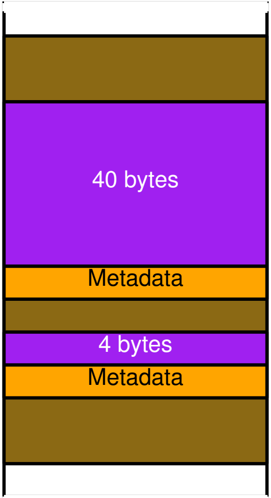
The Heap

```
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 ...
```



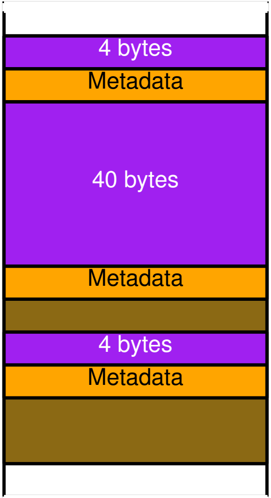
The Heap

```
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 ...
```



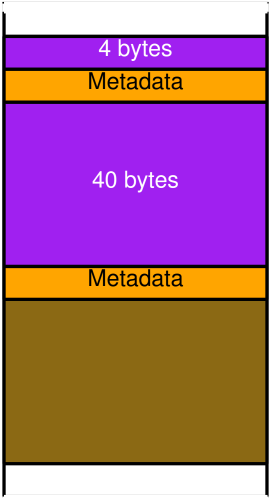
The Heap

```
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 ...
```



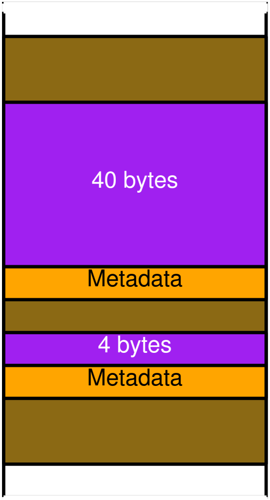
The Heap

```
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