CS 220: Introduction to Parallel Computing

Dynamic Memory

Lecture 10

Today's Schedule

- Project 1 Info
- Dynamic Memory Allocation

Today's Schedule

- **Project 1 Info**
- Dynamic Memory Allocation

Project 1

- P1 is now available on the course webpage
- You will get to work with:
  - Files, strings, tokenization
  - structs, dynamic memory allocation
  - Pointers! ☺
- Due 2/23
- …And: tentative midterm date: 2/28

Code Style (1/3)

- Be aware of your code formatting!
- Be consistent:

```
if (something) {
        ...
}
```

Or:

```
if (something)
{
        ...
}
```

Code Style (2/3)

- Don’t mix spaces and tabs
  - A tab character might be represented by 8 spaces on your machine and 4 on mine
  - Choose one and go with it
    - The examples I’ve given use spaces
- Use consistent spacing:

```
    if (something) {
    x = y;
                    z = q;
}
```

Code Style (3/3)

Commenting

- You don't have to comment everything. For instance:
  - `int i = 6; /* Create i and set it to 6 */`
  - Example of a **bad** comment
- Include comments above each non-obvious function you create.
  - What it does, what its inputs/outputs are
- Comment tricky/confusing parts of your code to make them more understandable
- Don't submit your project with big blocks of unused/commented out code

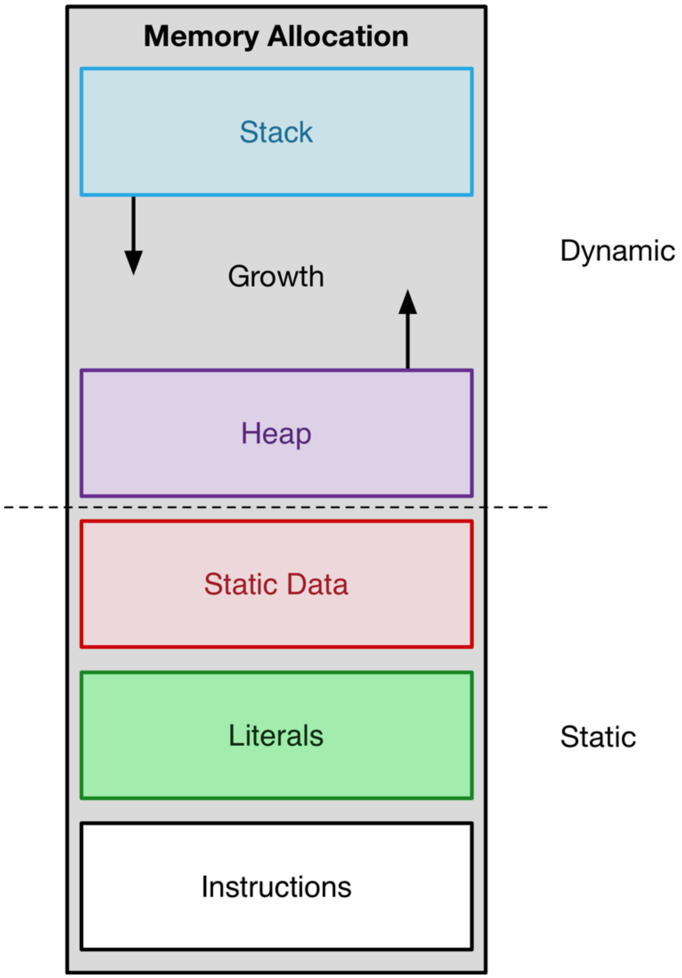
Today's Schedule

- Project 1 Info
- **Dynamic Memory Allocation**

Memory Allocation

- A running instance of a program is called a **process**
- Processes are allocated system memory to store instructions, literals, and more
- At run time, there are two places memory is allocated:
  - Stack
  - Heap

Memory Layout



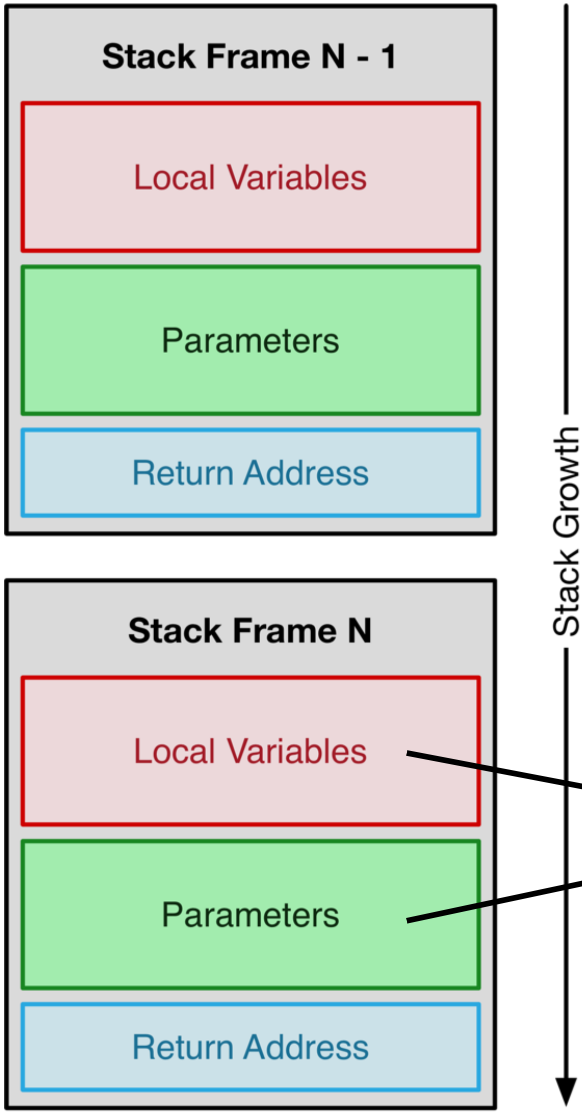
- **Stack**: generally responsible for temporary data
  - Scratch space
  - Made up of **stack frames**
- **Heap**: long-lived data

Stack

- Thus far, we've allocated everything to the stack
  - `int a = 5;`
- A good fit if we already know what data we're working with ahead of time
  - If we know a user wants to enter a number, we set aside some memory for them to do it
- If we don't know what data will be coming in ahead of time, then we need to place it on the **heap**

Stack Frame

- Each function call has a *stack frame*
  - You may also see these called **activation records**
- The stack frame contains the local variables, return address, and parameters
  - In other words, the "execution environment" for each function call
- Stack frames get pushed onto the stack with each function call
  - Unchecked recursive functions can lead to stack overflow




```c
int main(int argc, char *argv[]) {
    hello(1);
    return 0;
}

int hello(int i) {
    int j = i + 1;
    printf("Hello world!\n");
    return j;
}
```

Stack Overflow

We can cause a *stack overflow* by making the stack grow too large. Consider a recursive function:

```
int foo()
{
    return foo();
}
```

Heap

- The heap is where we **dynamically** allocate memory
- This is achieved using the malloc() function
- Allocating memory dynamically lets us cope with changing inputs
  - Perhaps a user wants to load a file: we can't just allocate a huge variable ahead of time and hope it fits
- How would we store a file in memory anyway? There's not exactly a "file" primitive type…

Allocating Memory: malloc

- `#include <stdlib.h>`
- `void * malloc(size_t size);`

- Remember the size_t type from our **sizeof** operator?
- This sets aside a block of memory for us to use
  - We just need to give it the size
- Reminder: there is no guarantee the memory set aside is zeroed out

Freeing Memory: free()

- `#include <stdlib.h>`
- `void free(void * ptr_p);`

- Every malloc() must also have a free()
  - Without freeing the memory, you introduce **memory leaks**
  - Imagine doing this inside an infinite loop

Use after free()

```
/* What happens here? */
int *i = malloc(sizeof(int));
*i = 3;
printf("%d\n", *i);
free(i);
printf("%d\n", *i);
```

Dynamic Memory Functions

- calloc() – clears the memory and allocates it
  - `void * calloc(size_t num, size_t size);`

- realloc() – reallocates (resizes) dynamically-allocated memory
  - `void * realloc(void *ptr, size_t new_size);`

Demo

- Dynamically allocating structs
- Use after free
- calloc() vs malloc()