Stack & heap

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org

Stack & heap

We think of scope and lifetime in the context of *the stack* (or the *call stack*), which grows upwards as:

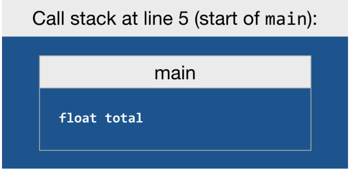
- New local variables are declared
- Functions call other functions

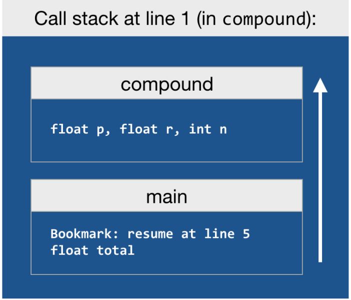
The bottom of the call stack is always `main` and its local variables

Stack

compound.c

```
0: float compound(float p, float r, int n) {
1:     return p * pow(1 + r/n, n);
2: }
3:
4: int main() {
5:     float total = compound(100.0, 0.10, 10);
6:     printf("%0.2f\n", total);
7:     return 0;
8: }
```





Upon function call, caller saves a “bookmark” for where to return to when callee finishes. Then room is made on the stack for the callee and its variables.

Stack

When functions return or when scopes are exited, stack shrinks

Stack overflow is when the stack grows so large it exhausts available memory

- E.g. because of a recursive function that never returns

Stack

Explains why a function can't return a locally-declared array:

scale.c

```
0: double* scale(double arr[5], double factor) {
1:    double scaled_array[5];
2:    for(int i = 0; i < 5; i++) {
3:        scaled_array[i] = arr[i] * factor;
4:    }
5:    return scaled_array;
6: }
7:
8: int main() {
9:    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
10:   double* scaled_array = scale(array, 2.0);
11:   printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
12:   return 0;
13: }
```

1 Call stack at line 10, before call:

main

```
double *scaled_array
double array[5]
```

2 Call stack at line 5:

scale

```
double scaled_array[5]
double factor
double arr[5]
```

main

```
Bookmark: resume at line 10
double *scaled_array
double array[5]
```

3 Call stack at line 10, after call:

main

```
double *scaled_array
        ^^^^^^^^^^^^
  points to scaled_array[5] in scale
  function, but it's dead & reclaimed
double array[5]
```

Stack arrays

When we declare an array, its size must be a “compile-time constant”

```
int array[400]; // we can do this

#define ARRAY_SIZE 400
int array[ARRAY_SIZE]; // this is also fine
```

`#define X Y` just means that everytime `X` appears in the program, it should be replaced with `Y`. It’s a “macro” rather than a variable because the substitution happens in the “preprocessing” step, prior to compilation.

Stack arrays

```c
int n = get_length_of_array();
int array[n]; // we shouldn't do this
```

C99 lets you do this, but earlier versions of C don't

It's considered bad style because it's easy to accidentally overflow the stack

- This is the only time you'll see it in these slides

Dynamic memory allocation

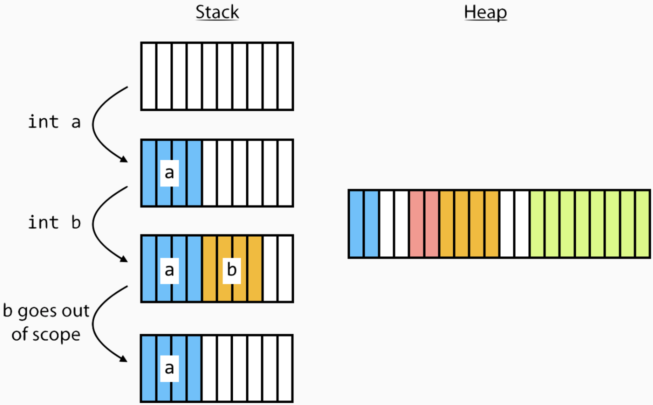
We're about to discuss dynamic memory allocation, where many of these issues are addressed:

- Flexible lifetime; we decide when allocated memory is allocated and deallocated (reclaimed)
- Allocated memory is *not* on the stack, can't cause stack overflow
- Allocation size need not be known at compile time
    - Can be a function of variables in the program

Stack vs. heap

So far, our variables and functions have used *the stack* to store data

We will soon be using a different area called *the heap*



Stack vs. heap

Stack: We declare variables; lifetime same as scope

- C takes care of allocating/deallocating memory as variables enter/exit scope

Heap: Lifetime is under our control

- We explicitly allocate and deallocate
  - E.g. with `malloc` and `free`, discussed later
- Operating System is places variables in memory in a non-overlapping way