## Scope, Lifetime, and the Stack

`const` pointers:

- For any type `T`, a `const T*` is a read-only pointer to a `T`
- `const int*T`
    - Can look but not touch
    - Can read the data that it points to but you can’t write it
    - But you change where the pointer points to
        - `const char* s = “hello”;`
        - `s = “goodbye”;        //this is ok`
        - `s[3] = ‘x’;          //not ok, gives compile-time error`
            - Error: assignment of read-only location ‘*s’
    - Allowed to turn a `T*` to a const `T*` at any time
        - Can pass a non `const*` to any `const*` function, but not the other way

Pointer casting:

- Casts convert from one type to another
- `(type) value`
    - `int a = 20; b =25;`
    - `double x = a / (double) b;      // x = 0.8, does floating point division`
- You can cast pointers too
    - `float f = 3.567;`
    - `int * p = (int*)&f;   // p points to f, int pointer pointing to a float variable`
    - `printf(“%08x\n”, *p);// interprets f as an int`
        - %x prints as hex
    - ints and floats are represented differently in binary
    - The computer doesn’t care about what bits mean
        - This kind of cast does not change anything in memory
            - f is still there and holds 3.567
        - It only changes how we view that memory

Scope and lifetime:

- Scope: where a name can be seen (anything you make that has a name)
- C has 3 levels of scope:
    - 1. Globals
        - Can be seen from ANY function in ANY file (like public static)
    - 2. Static global
        - Can be seen by any function in one file (like private static)
    - 3. Locals
        - Can only be seen by one function
- Try not to use global variables
    - Almost any problem you think you need a global variable for can be done by using a local and passing a reference

  - There are legit uses for them, but try to avoid
- Lifetime:
  - Every variable takes up space in memory
    - That memory must be allocated: reserved for the variable
    - When no longer needed, deallocated: released for other use
  - Lifetime: the time between allocation and deallocation
  - Global variables last from program start to program end
  - Local variables only last as long as the enclosing function
    - Only last as long as the enclosing brace block
    - Ends with its scope
  - Ownership: who is responsible for deallocating a piece of memory?
    - How do we determine when it’s okay to deallocate memory?
    - Locals owned by functions
    - Globals owned by programs
    - Local variables are allocated on the stack but when we use malloc (new in Java) it goes on the heap
      - Non local lifetime

Stack

- Caller: The one doing the calling
- Callee: the one being called
- Stack is an area of memory provided to your program by the OS
  - When your program starts, it's already there
- The stack holds information about function calls
- It’s not a strict stack
  - You can pop, push, peak
  - More like a resizable array
  - Grows and shrinks like a stack
- Each program (actually each thread) gets one stack
  - Only one function is running at a time

Activation records (AR):

- When a function is called, a bunch of data pushed to stack
  - This is the call’s activation record (or stack frame)
  - Contains local variables (including arguments) and the return address
- Low level layout
  - Each variable (including array variables) gets enough bytes in the activation record to hold its value
- Where the variable is located is up to the compiler
- The compiler aligns local variables using padding like with structs
- Call = push
- Return = pop
- Stack grows when we call a function and shrinks when it exits
- Return address is like a bookmark in caller so we know where to resume
- Recursive functions work this way
  - They work by using the call stack as an implicit stack data structure

- The stack is used constantly
    - So it needs to be really fast
- So we implement the stack as a pointer
    - The stack pointer (sp)
        - Push activation record: `sp-=(size of activation record)`
        - Pop activation record: `sp+=(size of activation record)`
    - But the activation record's memory is still there
    - So if we call another function, we're reusing the same memory for its activation record
    - This is why you have to initialize local variables
        - This is what leads to weird behavior
        - Why you don't return stack arrays