Lecture 5

Dr. Jack Lange

Variable Scope

- Recall dangerous.c

```
int * getInt()  /* this funct returns ptr to int */
{
    int num;
    printf("enter an int: ");
    scanf("%i", &num);  /* assume user entered some valid int */
    return &num;
}
```

- Returned a stale address from a function
  - It was out of scope
- But it was still legal in C
  - We just could not rely on memory being OK
  - Will work a lot of the time, but no guarantees

Local Variables

- C stores local variables on the stack
  - Compiler reserves memory when a function is called
  - Local variables exist inside the scope of the function
  - Scope is demarcated by { }'s (Does not have to be a function)
- How does reservation happen?
  - Compiler knows all the local vars declared in a function
  - Allocates space to store all of them on the stack
  - Local variables == stack variables

Stacks

- What do we know about stacks?
  - LIFO
  - The grow in one direction
  - You can push data onto them
  - You can pop data off of them
  - Push and pop always operate at the same end
- In general:
  - Stacks grow downwards
  - Memory addresses (indexes) decrease

Allocating stack memory

- Allocating stack space means increasing the size of the stack
  - Add more memory to the end of the stack
  - **DOES NOT INITIALIZE THE CONTENTS**
- The Stack Pointer (**`esp`**)
  - A special **register** that contains a memory address
    - The keyword is "Pointer"
  - Marks the current end of the stack
    - Address of the last byte of data on the stack
  - Memory is allocated by adding/subtracting from **`esp`**
- **Example: dangerous2.c**

Local variables

- Only local function variables are stored on the stack
  - Global data is stored in a special memory region
    - `char * name = "Jack Lange"`
- Scope and Stack allocations are not 1:1
- Static variables
  - Locally scoped, globally allocated

```
int do_something() {
    static int first_run = 0;

    if (first_run == 0) {
        // initialize something
        first_run = 1;
    }
}
```

Dynamic Memory

- Sometimes the stack is not good enough
  - You want to share data across functions
  - You want data to be persistent across function calls
  - The data size is too large for the stack
  - You don't know how much memory you will need at compile time
    - E.g. reading a file into memory (how big is the file?)
- A new way to get memory
  - Like new/delete from Java
  - Allocates memory from the **HEAP**
- `void * malloc(size_t size);`
  - **Allocates 'size' bytes of memory and returns the address**
- `void free(void * ptr);`
  - **Frees the memory allocated at the address 'ptr'**

The Heap

- A region of memory that is managed by the programmer
  - Programmer requests ***X*** bytes of memory
- Heap memory is not scoped
  - Memory is reserved until its released/free()'d
- Heap memory has no "name"
  - `int x;` --> compiler names 4 bytes of memory
    - When ***x*** goes out of scope its memory is automatically released (How?)
  - The compiler does not name a heap allocation
    - Not automatically released when it goes out of scope

Accessing heap memory

- Compiler does not associate a variable with heap allocations
  - *Q: How do we access it?* **A: Pointers**
- Review: what are pointers?
  - A variable that stores an address
  - Pointer variables are ***scoped***
    - Compiler manages the memory for the pointer ***variable***
- Pointers are used as ***handles*** for heap memory

Pointer pitfalls

- Memory Leaks
  - Pointers are scoped, heap is not
  - Allocated memory can be lost
    - What if all of the pointers with a heap address go out of scope?
    - This is a **MEMORY LEAK**
  - C does not garbage collect
- Stale Pointers

```
int * x = malloc(4);
free(x);
*x = 4;
```

Advanced pointer usage

- Returning heap allocated memory

```
int foo(char ** ret_str)
{
    char * str   = "Hello";
    int err_code = do_something();

    if (err_code == -1) {
        return -1;
    }

    *ret_str = str;
    return 0;
}
```

Example: double_ref.c