CSE 374 Lecture 10

(Week 4: C continued)

- Homework 2 due tonight
- Review previous lectures via website
- Office Hours This Week
  - Monday: Cynthia 11-12
  - Wednesday: Simon 4-5
  - Thursday: Dixon 3-4
  - Friday: Andrew 12-1
  - *More available upon request*
- Guest Lecture by Andrew on Friday: Debugging

Administrative Notes

(Using EdStem Well)

Please search previous messages in case your question has been answered.

Please do not post actual answers publicly: you can mark your post private (OR, you cse374-staff@cs.washington.edu)

Please note that it can take some time to address issues even after the issue has been acknowledged.

Quiz 3: General notes on quizzes

- Goal is to reinforce learning
  - Practical problems
    - Feel free to test answers before submitting
  - Retakes
    - Sorting out appropriate feedback settings
  - Generous grading
    - May require regrading - may take additional days.
    - Be patient!!

Quiz 3: Q2

My answer: `sed -i 's/sdel/clean/g' *`
Variations: `"  ./*  -s`

Notes:

- You must apply `sed` to something. (You need, `*` or `./*`)
- We have not talked about `xargs`; it is not necessary here
- And, while `` `ls` `` works, it will not always work
- Generally, simpler is better
- The option '--' doesn't do anything - it is an option tag with an empty string.

Quiz 3: Q6

What is strongly typed?

Truthfully, there is some debate. However, I should not have used the term ‘strongly typed’ here. C **is** a typed language, and datatypes must be declared in advance. This is similar to Java, and dissimilar to other modern languages such as Python. It also is one way in which C differs from bash script (where very variable is a string, although conversion to numbers is done).

https://hackernoon.com/actually-understand-statically-dynamically-strongly-weakly-typed-languages-axbpi3za2

Quiz 3: Q6

C and Java do differ a lot, though:

C is NOT object oriented - the program is structured around an algorithm/process (it is ‘procedural’), not around the data structures.

C is compiled and has access to memory - in Java the VM handles most of that for the programmers.

Quiz 3: Q

Compiling code

```
gcc myNewCode.c -o runme
```

Works, but it is not preferred. Please put the input file last. (We will see why this convention makes sense when we get to Makefiles.)

Also prefer `-Wall` and `-std=c11` options, but weren't required for the quiz.

Include file clarity

1. You create a .h file to share code with another caller
   a. Declare any variables and functions you want another caller to user
   b. Functions you want to use only in the same file are declare in the .c file
2. If you have `a.c`, which uses printf, you would include `<stdio.h>`
3. If you have `b.c`, which uses printf, and includes `"a.h"` you do not need to include `<stdio.h>`, however
4. Generally, include any header files needed for directly-called functions (promotes encapsulation), so `b.c` would include `<stdio.h>`

Function prototyping clarity

Functions called before declared are ‘implictly declared’, but functions ***should*** be prototyped.

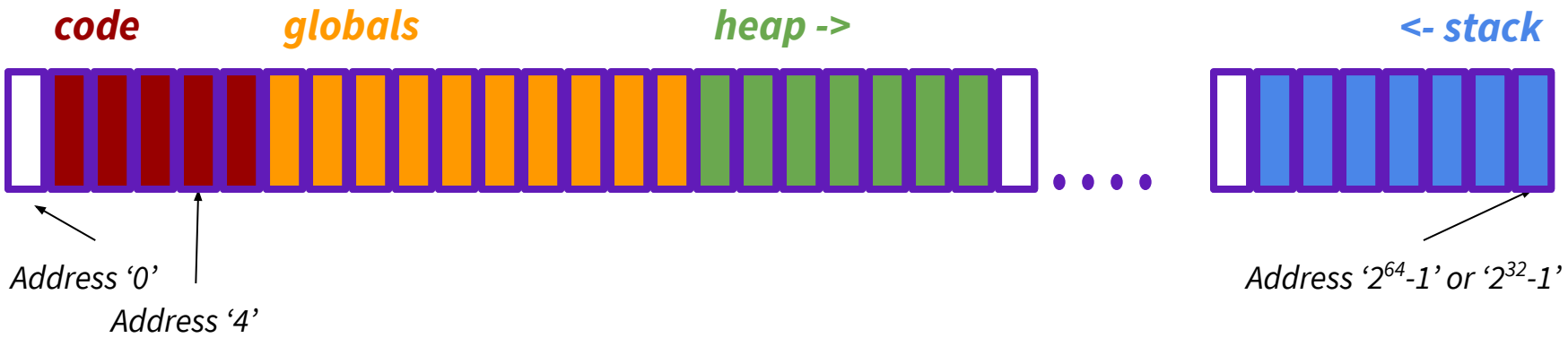
Function name and return type of int assumed, but not argument list - parameter checking is turned off.

Forbidden by c11 standard (try compiling with -std=c11)

It works if they are defined in the same file, but is not reliable and shouldn’t be done.

Storage

- Variables need a place to live in memory
- Get ‘allocated’ a physical space in memory (with an address)
- Size of memory allocation depends on datatype
- Get ‘deallocated’ to release the space in memory

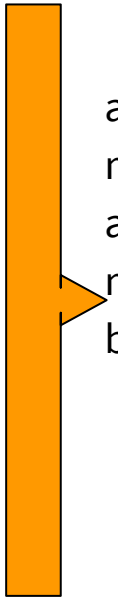


Scope

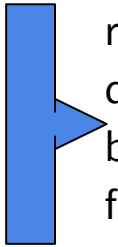
Variables may be accessed by the caller only at certain times - this is scope

Scope and storage are related, but not the same thing

- Global variables
  - Scope: entire program
  - Not desirable (violate encapsulation) But can be OK for truly global data like conversion tables, physical constants, etc.
- Static global variables
  - Scope: containing file
  - Static functions cannot be called from other files
- Static local variables
  - Scope: that function, rarely used
- Local variables (automatic)
  - Scope: that block – With recursion, multiple copies of same variable (one per stack frame/function activation)

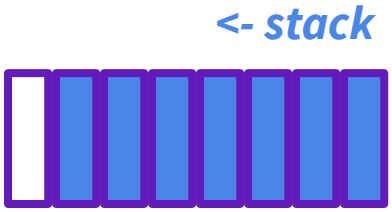


allocated before main, deallocated after main - memory in ‘global’ block



allocated “when reached” deallocated “after block” - memory in frame on stack

The stack



Stack stores active functions & local variables

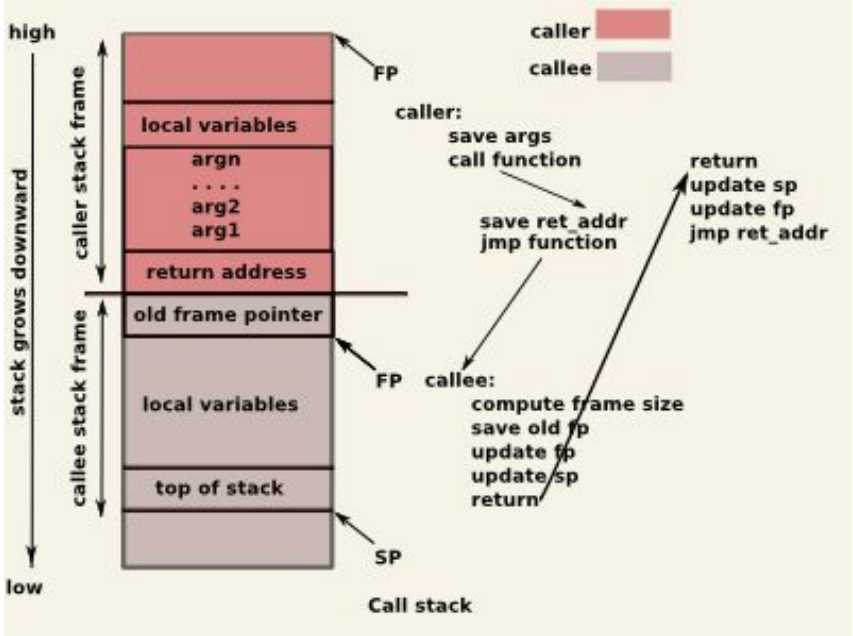
Frames deleted when function returns

Local variables do not persist

Local variables must have defined size

Can not make run-time adjustments

(Arrays must have length)



Initialization

Memory allocation and initialization are not the same thing

Unlike Java, you MUST provide a value to initialize a bit of memory

It is possible to access un-initialized bits
unlike Java with sets defaults and checks for initialization
best case scenario: you crash

L-values v. R-values

Left Side

Evaluated to locations (addresses) =

Right Side

Evaluated to values (the contents at the address)

Values may be numbers (or characters) OR addresses

```
9 = x;      // Nonsense, because 9 isn't a LOCATION
int x = 1; // Stores the VALUE 1 at a LOCATION which has the LABEL x.
x = 2;      // Stores the VALUE 2 at the LOCATION x.
int* xPtr = &x;  // Stores VALUE of address of x at a LOCATION labelled xPtr.
*xPtr = 3;  // Stores VALUE 3 at a LOCATION defined by address stored in xPtr.
int** xx = &(&x);// Nonsense, the r-value needs to resolve to a value.
                 // &x does indeed represent a value (the address x), but
                 // &(&x) refers to the address of the address of x -
                 // which is just a number and not stored anywhere
```

Arguments

Demo

Storage allocation and variable scope is like local variables (i.e. space is part of the function frame added to the stack, and the variable may be used in the function).

All arguments passed by value. (i.e. a copy of the value is made and assigned to the variable.)

Pointers to pointers

Levels of pointers make sense:

I.e.: `argv, *argv, **argv`

Or: `argv, argv[0], argv[0][0]`

But

`&(&p)` doesn't make sense

```
void f(int x) {
    int*p = &x;
    int**q = &p;
    // x, p, *p, q, *q, **q
}
```

Integer, pointer to integer, pointer to pointer to integer

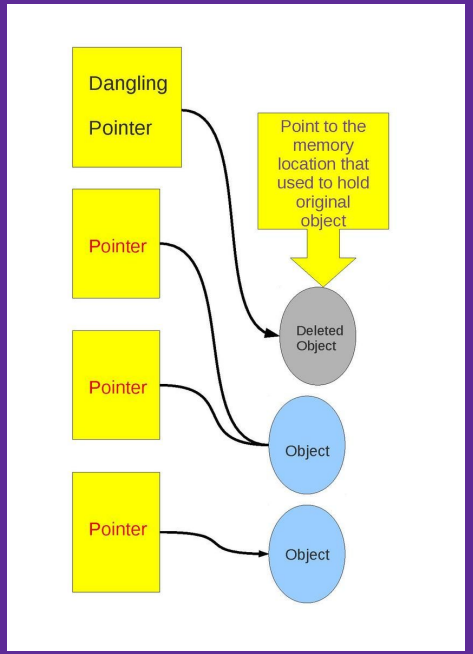
`&p` is the address of '`p`',

`&(&p)` would be the address of the address of `p`, but that value isn't stored separately anywhere and doesn't have an address

Try using `printf ("The address of x is %p\n", &x);`

Dangling pointers

Pointers referring to memory that has been released (*Demo*)



Garbage collecting languages (like Java) only delete memory that is unreachable to avoid this problem.

Pointer arithmetic

- If p has type T* or T[] and *p has type T
- If p points to one item of type T, p+1 points to a place in memory for the next item of type T
  - So, p[0] is one item of type T, p+i = p[i]
- T[] always has type T*, even if it is declared as T[]
  - Implicit array promotion

    Result: Arrays are always passed by reference, not by value. (The information passed is the address of where the values are stored.)

Arrays again

"A reference to an object of type array-of-T which appears in an expression decays (with three exceptions) into a pointer to its first element; the type of the resultant pointer is pointer-to-T."

http://c-faq.com/aryptr/aryptrequiv.html

Right: x is the array, which decays to a pointer to an int and &x returns a pointer to the entire array.

```
void f1(int* p) {  // takes a pointer
  *p = 5;
}
int* f2() {
    int x[3];     // x on stack, is pointer
    x[0] = 5;
    (&x)[0] = 5; // address of x, points to
                 // same place but different T
    *x = 5;      // put value at location x
    *(x+0) = 5;  // Also put value at x
    f1(x);
    f1(&x);     // wrong - watch types!
    x = &x[2]; // No!  X isn't really a pointer
    int *p = &x[2];
    return x; // correct type, but is a
              // dangling pointer
}
```

https://chortle.ccsu.edu/CPuzzles/CPuzzlesMain.html