cse30 discussion 3

Ibrahim Awwal

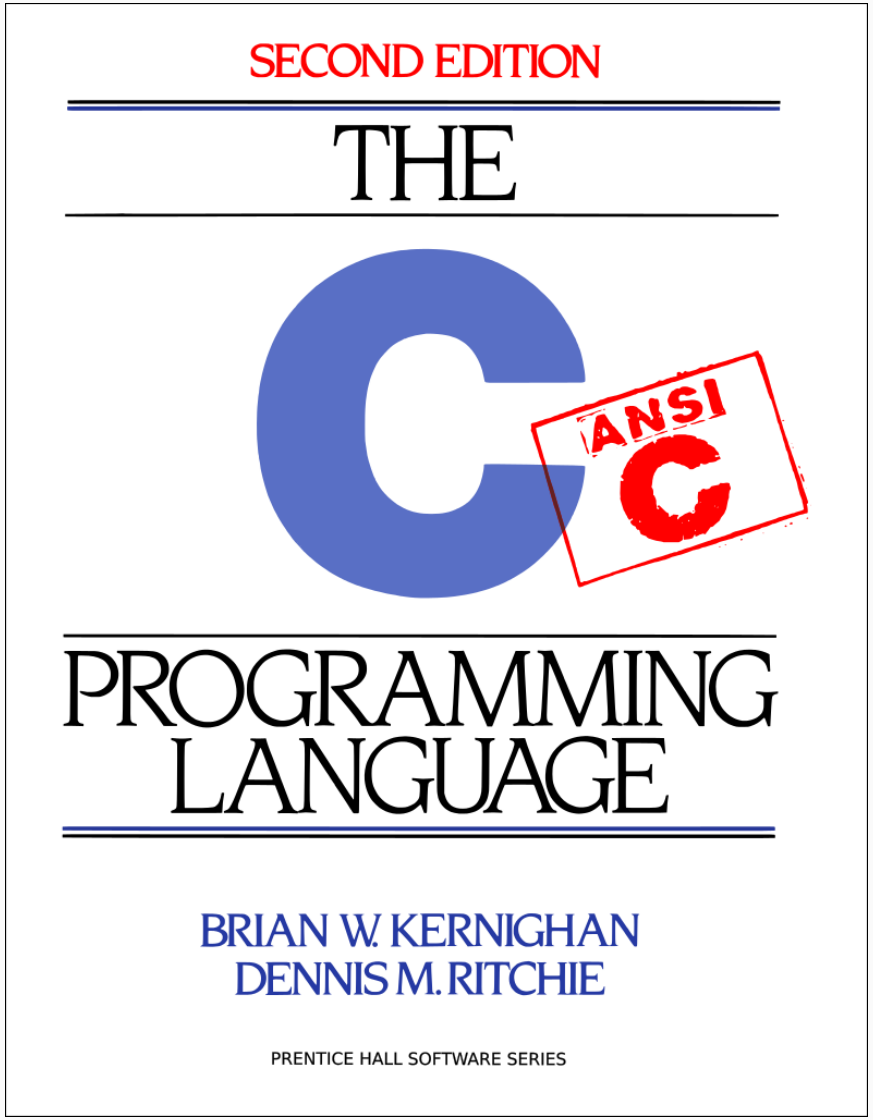
July 6, 2015

any questions?

- Raspberry Pi Setup
- Number representations
- C language
- Tools
- PA1

C

the c programming language (k&r)



pointers

- point to a location in memory
- declaration: `int *intPtr;`
- getting an address: `intPtr = &x;`
- "dereferencing" a pointer gets the value pointed to: `*intPtr`

pointer arithmetic

- In general, we cannot perform arbitrary assignments to a pointer and expect to read valid memory (often results in segfaults)
- Exception: we can add or subtract from a pointer to navigate an array
- Incrementing a pointer increments by `sizeof(type)` being pointed to, not by 1 memory address

strings

- `int strlen(char *string)`
- `int strcpy(char *dst, char *src)`
- `int strcmp(char *str1, char *str2)`

strlen

strlen

```
int strlen(char *string){
    int n;
    for (n = 0; *s != '\0', s++){
        n++;
    }
    return n;
}
```

strcpy

strcpy

```
char *strcpy(char *dst, char *src){
    int i = 0;
    while ((dst[i] = src[i]) != '\0'){
        i++;
    }
    return dst;
}
```

strcpy

```
char *strcpy(char *dst, char *src){
    int i = 0;
    while ((dst[i] = src[i]) != '\0'){
        i++;
    }
    return dst;
}
```

- src and dst cannot be overlapping, why?

strcpy with pointer arithmetic

```
char *strcpy(char *dst, char *src){
    while (*dst++ = *src++);
    return dst;
}
```

- How does this work?

strcmp

```
int strcmp(char *s, char *t){
    for ( ; *s == *t; s++, t++){
        if (*s == '\0'){
            return 0;
        }
    }
    return *s - *t;
}
```

the problem with strlen, strcpy, strcmp

- What's wrong with these functions?

the problem with strlen, strcpy, strcmp

- What's wrong with these functions?
- C does not store length alongside arrays

the problem with strlen, strcpy, strcmp

- What's wrong with these functions?
- C does not store length alongside arrays
- Naive versions can potentially access memory improperly if given non-terminated strings or insufficient space

buffer overflow

```
char           A[8] = "";
unsigned short B    = 1979;
strcpy(A, "excessive");
```

buffer overflow

```
char           A[8] = "";
unsigned short B    = 1979;
strcpy(A, "excessive");
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

buffer overflow

```
char           A[8] = "";
unsigned short B    = 1979;
strcpy(A, "excessive");
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

solution: passing the lengths of string to functions

- The C standard library includes string functions which take the maximum length of the string as arguments (eg. `strnlen`, `strncpy`, etc.)
- These versions are less prone to buffer overruns
- more info: `http://www.cplusplus.com/reference/cstring/`
- standard library functions also have man pages (eg. `man strcpy`, `man malloc`, etc.)

memory allocation

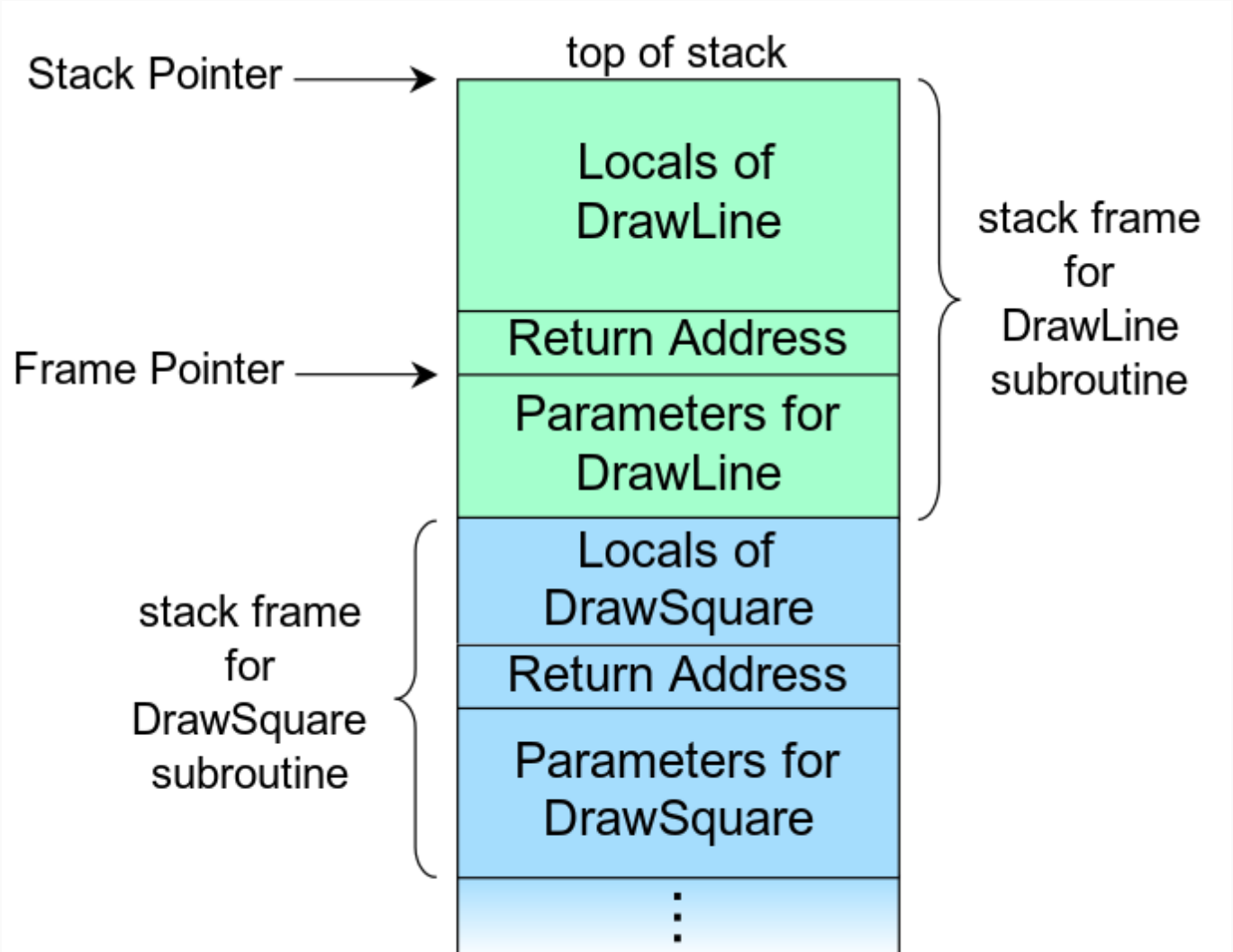
- stack
- heap
- static

example

```
1  #include <stdlib.h>
2
3  char *staticString = "The quick brown fox jumped.";
4
5  int main(int argc, char **argv){
6      int x = 4;
7      int *arr = (int *) malloc(10*sizeof(int));
8      return 0;
9  }
```

the stack

Assume `DrawSquare()` calls `DrawLine()`



lifetime of a variable

- stack: lives until the end of the function call in which it is defined

lifetime of a variable

- stack: lives until the end of the function call in which it is defined
- heap: lives until it is freed

lifetime of a variable

- stack: lives until the end of the function call in which it is defined
- heap: lives until it is freed
- static: until program ends

malloc

- to allocate memory on the heap, use `malloc`
- function signature: ~~ {.c} void* malloc (size_t size); ~~
- we *must* free memory allocated by malloc
- failure to do so is a memory leak
- freeing a pointer while others still hold references to it is also a potential error

malloc details (k&r 8.7)

- keeps a list of free blocks of memory
- each block contains a size, pointer to next block, and the free space
- when a request is made, the list is scanned until a block big enough is found
- when a block is found, it is returned and removed from the free list
- if no sufficiently large block exists, ask the OS for more

free details (k&r 8.7)

- scans the free list looking for the freed block's address
- adds an entry to the list if between two blocks
- merges free blocks if adjacent

more tools

valgrind

- Valgrind is a set of debugging and profiling tools
- The most common use for Valgrind is checking for memory errors

example c program

```c
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

errors in previous program

- problem 1: heap block overrun

errors in previous program

- problem 1: heap block overrun
- problem 2: memory leak – x not freed

let's run valgrind on this program