CSSE2310/7231 — 2.2

More Pointers

Pointers and structs

```c
struct Data {
    int length;
    char* str;
};

    struct Data d1;
    struct Data* d2=malloc(sizeof(struct Data));
        // modify the length field
    d1.length = 4;
    *d2.length = 4;       // ?? - no
    (*d2).length = 4;     // legal
    d2->length = 4;       // easier to read
```

->

```
struct Node {
    int value;
    struct Node* next;
};
```

->

```
struct Node {
    int value;
    struct Node* next;
};

// Want to change the value 3 hops along from n
    Node* n = ...
    ...
    (*(*(*(*n).next).next).next).value = 4;
// vs
    n->next->next->next->value = 4
```

Pointer related functions

- `memset()`
  - Set a chunk of bytes to a chosen value
- `memcpy()`
  - Copy bytes at one pointer to another

Where the variables are

Main places:

- Global variables + literals etc
- Function local variables (includes parameter variables)
  - Stack
  - Space allocated when the function is called, released when exiting/returning from the function
- Dynamically allocated storage (`malloc()`, `free()`)
  - Heap
  - Only cleaned up when explicitly told to[1]
  - Can store much bigger things than the stack can.

[1] While the program is running

Pointers to existing vars

```
int v1 = 7;
printf("v1=%d is at address %p", v1, (void*)&v1);
int* v2 = &v1;
*v2=14;
printf("v1=%d is at address %p", v1, (void*)&v1);
```

Lesson: pointers can point anywhere in memory, including to the stack.

void*?

void is used to indicate the lack of something:

- void fn.... — function doesn't return a value
- int fn(void) — function doesn't take any parameters

void* is a pointer without a type.

- Do not derefence a void*.
- How C deals with functions which take varying types.

The %p placeholder wants a void pointer (hence the cast).

Parameter passing

All C function parameters are passed by value.
You just need to know what is being passed.

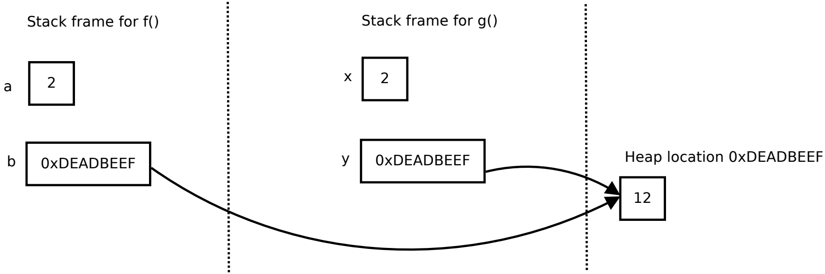
```
void g(int x, int* y);

// inside f()
int a;
int* b = malloc(sizeof(int));
g(a, b);
```

The values of `a, b` will be copied into `x, y` respectively.

This applies to the contents of structs.

Parameter passing



```
void g(int x, int* y) {
    x = 5;
    *y = 10
}
```

Swap

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

See `swap1.c`

“Pass by reference”

C is only pass by value, but we can simulate pass by reference by passing pointers by value.

See `swap2.c`

Strings

Strings

- *Stored in* arrays of char
  - Shortened (incorrectly) to “are arrays of char”[2].
- Well-formed strings end in a terminator byte '\0'

Not all arrays of char hold proper strings.

```
char buffer[7];
strcpy(buffer, "Hello");
```

buffer[0]=='H'
buffer[4]=='o'
buffer[5]=='\0'
buffer[6]==????

[2] That French pipe thing again

Strings

Strings don't (explicitly) store their length.

`strlen(s)` — finds length by counting chars until a terminator is found. eg:

```
int len(char s[]) {
    int i=0;
    for (;s[i] != '\0'; ++i) {
    }
    return i;
}
```

Strings

```
char buffer[6];
strcpy(buffer, "Hello");
char* greet=buffer;
printf("%s %s\n", greet, buffer);
greet=&(buffer[2]);
printf("%s %s\n", greet, buffer);
```

See `greet.c`

The type of strings is most commonly given as `char*`.

String operations

Example: Join argv entries together.

See `join.c`

String operations

Trying to guess how big your buffer needs to be is not great.

See `join2.c`.
Try to write this function without using `strlen()`, `strcpy()` or `strcat()`

Pitfall declaring multiple pointers

`int x, y, z;`
Declares `x, y` and `z` to be `int`s.

`int* x, y, z` ?

The `*` only affects the variable directly to its right.
So we'd end up with `x` being `int*` and `y` and `z` being `int`s.
Approaches:

- `int* x, *y, *z`
- `typedefs`

typedef

typedef *actualtype newname*;
eg:

```
typedef char* cptr;

// later

cptr x, y, z;
```

All three vars are char*.

structs

Can also use typedef on structs.

```
typedef struct {
    int id;
    double gpa;
    char* name;
} Student;

// later

Student s1;
```

structs

```
typedef struct Node {
    int value;
    struct Node* next;
} Node;

// Later

Node n;
```

Note: Can't use the `typedef`'d name inside the struct.