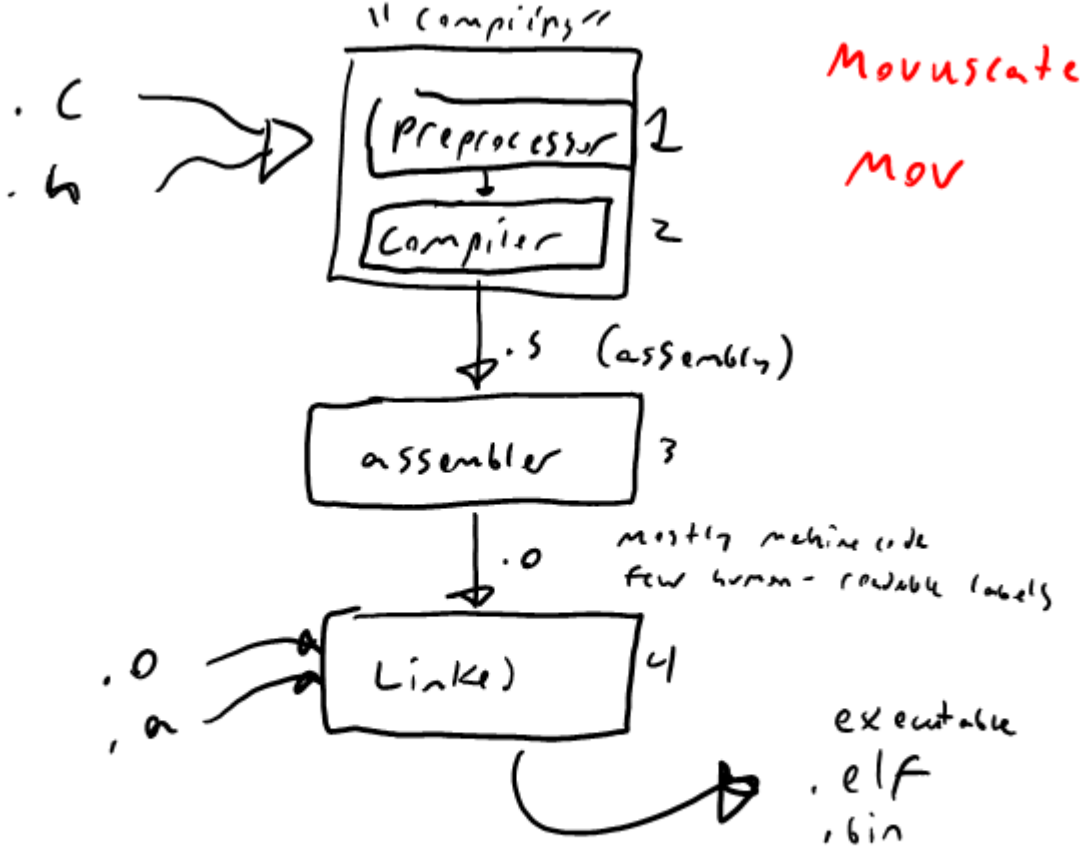
CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

Roadmap

- Announcements
- Battleboats Stuff
- Lab10 stuff?
- Any other stuff?
- C Dark Arts:
  - Weird enum tricks
  - Function pointers
  - Duff's Device
  - offsetof()
- Software Design Principles (?)



Mousecat

Mov

struct NMEA {

uint8_t

uint16_t

char [6]

char *

}

size (byte)

1

2

6

4

41

30

21

13

52

Scope

Lifetime | Namespace

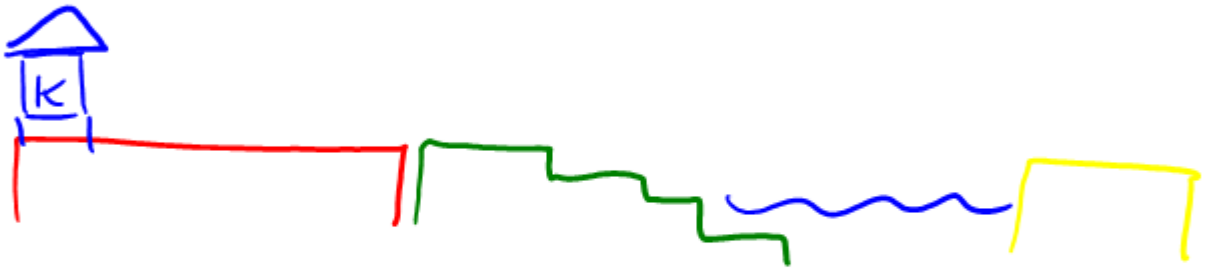
```
int i = 5
    2

Void foo() {
    static int i = 20;
                   2
    while(i > 10) i--;
          == 2
}

main {
    int i = 6   3
    if (j == 1) {
        foo()
    }
    printf("%d", i);
                 == 3
}
```

Lifetime of data

Static

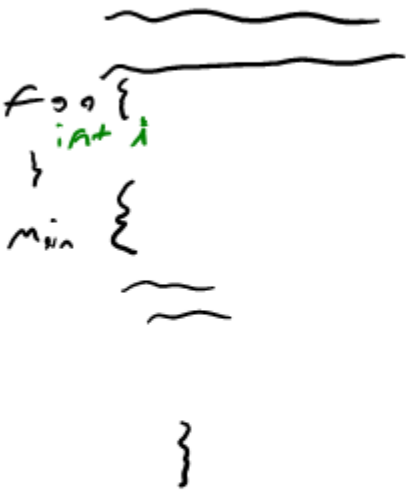
Stack

Heap



static K

Keyword static


foo {
int i
}
main {
}


possible
global
extern
int i
int i;
{
}
cannot
use
global
static i;
{
}

Agent.c:

```
#includes . . .
//if you want to run in simulator
#define TEST MODE   // comment out for release
#if def TEST mode
#define OLED Update(...)
   :
#endif

functions
```

Announcements

- If you have grade issues talk to me now
  - Grades will be finalized on September 4th

C Dark Arts

Stringification

- How to print out the state you're in?

```
//Main loop:
while (TRUE) {

    //if there is a top-level event, the Agent module should respond to it:
    if (battleboatEvent.type != BB_EVENT_NO_EVENT) {

        TraceEvent();

        Message message_to_send = AgentRun(battleboatEvent);

        TraceState();

        //send a message, if there is one to send:
        if (message_to_send.type != MESSAGE_NONE) {
            Transmission_StartSendingMessage(&message_to_send);
        }

        //consume the event:
        battleboatEvent.type = BB_EVENT_NO_EVENT;
```

Stringification

- Stringification macro operator:

```
#define printvar(x) printf( #x "= %d\n", x)

    random_num = rand();
    printvar(random_num);
```

output:

random_num = 17

Function Pointers

```
          include  "p16f84.inc"
          extern   SUM   ; The 2-byte number Hi:Lo

; Local declarations
          udata_ovr
I         res 2          ; Magic number hi:lo
COUNT     res 1          ; Loop count

TEXT      code

SQR_ROOT clrf   COUNT    ; Task 1: Zero loop count

          clrf   I       ; Task 2: Set magic number I to one
          clrf   I+1
          incf   I+1,f

SQR_LOOP movf   I+1,w    ; Task 3(a): Number - I
          subwf  SUM+1,f ; Subtract lo byte I from lo byte Num
          movf   I,w     ; Get high byte magic number
          btfss  STATUS,C ; Skip if No Borrow out
           addlw 1       ; Return borrow
          subwf  SUM,f   ; Subtract high bytes

          btfss  STATUS,C ; IF No Borrow THEN continue
           goto  SQR_END  ; ELSE the process is complete

          incf   COUNT,f  ; Task 3(c): ELSE inc loop count

          movf   I+1,w    ; Task 3(d): Add 2 to the magic number
          addlw  2
          btfsc  STATUS,C ; IF no carry THEN done
           incf  I,f      ; ELSE add carry to upper byte I
          movwf  I+1
          goto   SQR_LOOP

SQR_END   movf   COUNT,w  ; Task 4: Return loop count as the root
          return

          global SQR_ROOT
          end
```

Address pointer

Max Lichtenstein

Function Pointers

```
void applyFunctionToEachElement(
        float (*fun)(float),
        float array[3][3])
{
    int i, j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            array[i][j] = fun(array[i][j]);
        }
    }
}
```

```
typedf struct {
    Function *  FieldInt;
    Function *  DecideGuess
    Fun *       RegisterAttack
} AI;
```

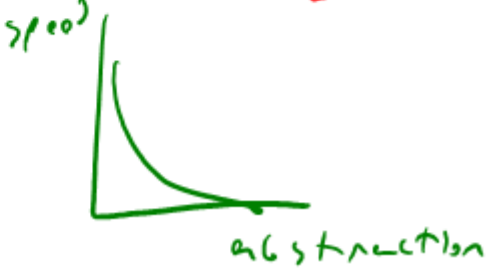
object-oriented C

- methods
- class

```
int RunMatch(AI1, AI2)
```

```
RunMatch(AI ai1, AI ai2){
    // ai1, ai2 get copied
    // copying 24 bytes
    ai1.Init();
```


```
RunMatch(AI* ai1, AI* ai2)
    // pass in pointer
    // copy 8 bytes
    ai1->Init();
```



Duff's Device

```
void duffs_device(register char * to,
        const register char * from, int count)
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do {
            *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
        } while (--n > 0);
    }
}
```

Duff's Device (original version)

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

offset_of()


```
#define OFFSET_OF(st, m) ((size_t)&(((st *)0)->m))

    printf("%d the address distance between leftChild "
           "and the base address of its Node is :\n%d\n",
           OFFSET_OF(Node, leftchild));
```

Max Lichtenstein

Software Engineering

Software Engineering

Principles

- Use consistent styling
- Summary:
  - Utilize whitespace
  - Good variable/function names
  - Comments that describe non-obvious code behavior
    - "How?" and "why?" are good questions to answer in comments

Software Engineering

Principles

- Modularity is important
- Why?
  - Supports code reuse
  - Simplifies changes
  - Allows for testing
- How?
  - Keep functions small
  - Minimize side effects
  - Information hiding/encapsulation

Software Engineering

Principles

- Information hiding/encapsulation
- Summary:
  - Hide unimportant details from the user
  - Protects the user from breaking things
  - Separates backend from frontend

Software Engineering

Mantras

- Keep it simple, stupid
  - KISS
- Summary:
  - Don't solve problems you don't need to
  - Don't introduce unnecessary complexity
  - Prioritize for readability and modularity
  - Don't be clever and/or cute
  - Applies to code architecture and specific code constructs

Software Engineering

Mantras

- Don't repeat yourself
  - DRY
- Summary:
  - Write code only once
  - Simplifies refactoring/incremental development
  - Avoids copy/paste errors

Software Engineering

Mantras

- You aren't gonna need it
  - YAGNI
- Summary:
  - Don't introduce features that are unnecessary
  - Don't write more code then you have to
  - Start small and build from there

Software Engineering

Principles

- Principle of Least Astonishment
- Summary:
  - Be consistent with user's expectations
  - Build on user's intuition
  - Applies to users and developers
    - so both the code and library/program functionality
  - Lowers learning curve

Software Engineering

Principle of Least Astonishment

- Functions/variables should have clear names
  - That should match their functionality!
  - Same for comments
- Functions should not do more than you would think
  - Minimize side effects
- Code should be grouped logically
- Functionality should follow precedence if any exists

Software Engineering

Principles

- Garbage in, garbage out
- Summary:
  - "A system's output quality usually cannot be better than the input quality"
  - So bad input results in garbage output
    - Instead of an error condition
  - Can propagate through the system
  - Can be mitigated by checking the input data

Software Engineering

Principles

- Fault tolerant design
- Summary:
  - Plan for operating failures
    - Running out of memory
    - Data being corrupted
  - Provide fallback modes
  - Important for complex software where minor errors can be common
  - Part of defensive programming

Software Engineering

Principles

- Error tolerant design
- Summary:
  - Plan for user errors
    - "Fault tolerant design" applied to the human component
  - Primarily invalid user input
  - Important for complex software where minor errors can be common
  - Part of defensive programming

Software Engineering

Writing fault/error tolerant code

- Check return values for errors!
  - Many functions have special return values when there are errors, these should usually be checked
  - File accesses
  - scanf()
  - malloc()
- Your code should have special error values
  - LinkedList library
- Program should also return error if failure

Software Engineering

Principles

- Eating your own dogfood
- Summary:
  - When engineers use their own creations, they're generally better
  - More likely that bugs are fixed, features are added because they directly impact the developers
  - In use by all of industry
  - I do it

Software Engineering

Pitfalls

- Premature Optimization
  - "root of all evil"
- Summary:
  - Optimizing code before performance is a critical factor
  - Optimizing reduces readability & modularity
  - Optimization not required for a lot of code
    - See Amdahl's Law
  - See KISS

Software Engineering

Teamwork

- Working as a group is **the** most challenging engineering practice
- Requires:
  - Good communication
- That's it!

Software Engineering

Teamwork

- Pair programming
- Summary:
  - Two developers work side by side: one driving, the other navigating
  - Just like driving:
    - Driver writes code
    - Navigator plans ahead, thinks of edge cases, double-checks driver
  - Requires frequent role switching to be effective!

Software Engineering

Teamwork

- Division of labor
- Summary:
  - Divide work into tasks that can be split between team members
  - Requires coordination to not step on each other's toes
  - Documentation is very important!
  - Can be useful to split testing and development between different people