# Agenda

- Attack Lab
- C Exercises
- C Conventions
- C Debugging
- Version Control
- Compilation
- Demonstration

# Attack Lab...

is due Thursday. Start now if you haven’t yet.

# Some warnings about C

- It’s possible to write bad code. Don’t.
- Watch out for implicit casting.
- Watch out for undefined behavior.
- Watch out for memory leaks.
- Macros and pointer arithmetic can be tricky.
- K&R is the official reference on how things behave.

Implicit casting: if an operation involving both unsigned and signed operand, the bits would be interpreted as unsigned.
When an arithmetic operation involves integer and floating point, the int would be casted to double.

# C-0

```
int foo(unsigned int u) {
    return (u > -1) ? 1 : 0;
}
```

# C-0

```
int foo(unsigned int u) {
    return (u > -1) ? 1 : 0;
}
```

-1 is cast to an unsigned int in the comparison, so the comparison that is happening is actually `u > int_max`. This always returns 0.

# C-1

```
int main() {
    int* a = malloc(100*sizeof(int));
    for (int i=0; i<100; i++) {
        a[i] = i / a[i];
    }
    free(a);
    return 0;
}
```

# C-1

```
int main() {
    int* a = malloc(100*sizeof(int));
    for (int i=0; i<100; i++) {
        a[i] = i / a[i];
    }
    free(a);
    return 0;
}
```

Always remember to initialize the memory retrieved from malloc.

No value in `a` was initialized. The behavior of `main` is undefined.

# C-2

```
int main() {
    char w[strlen("C programming")];
    strcpy(w,"C programming");
    printf("%s\n", w);
    return 0;
}
```

# C-2

```
int main() {
    char w[strlen("C programming")];
    strcpy(w,"C programming");
    printf("%s\n", w);
    return 0;
}
```

`strlen` returns the length of the string not including the null character, so we end up writing a null byte outside the bounds of `w`.

strcpy would keeps putting chars, including the null characters. Thus, the number of bytes allocated is one less than required. The last byte, which is the null character, corrupts the memory of other parts of the program and could cause problem.

# C-3

```
struct ht_node {
    int key;
    int data;
};
typedef struct ht_node* node;

node makeNnode(int k, int e) {
    node curr = malloc(sizeof(node));
    curr->key = k;
    curr->data = e;
    return curr;
}
```

# C-3

```
struct ht_node {
    int key;
    int data;
};
typedef struct ht_node* node;

node makeNnode(int k, int e) {
    node curr = malloc(sizeof(node));
    curr->key = k;
    curr->data = e;
    return curr;
}
```

`node` is a `typedef` to a `struct ht_node` pointer, not the actual struct. So `malloc` could return 4 or 8 depending on system word size.

# C-4

```
char *strcdup(int n, char c) {
    char dup[n+1];
    int i;
    for (i = 0; i < n; i++)
        dup[i] = c;
    dup[i] = ’\0’;
    char *A = dup;
    return A;
}
```

# C-4

```
char *strcdup(int n, char c) {
    char dup[n+1];
    int i;
    for (i = 0; i < n; i++)
        dup[i] = c;
    dup[i] = ’\0’;
    char *A = dup;
    return A;
}
```

`strcdup` returns a stack-allocated pointer. The contents of `A` will be unpredictable once the function returns.

# C-5

```
#define IS_GREATER(a, b) a > b
inline int isGreater(int a, int b) {
   return a > b ? 1 : 0;
}
int m1 = IS_GREATER(1, 0) + 1;
int m2 = isGreater(1, 0) + 1;
```

# C-5

```
#define IS_GREATER(a, b) a > b
inline int isGreater(int a, int b) {
  return a > b ? 1 : 0;
}
int m1 = IS_GREATER(1, 0) + 1;
int m2 = isGreater(1, 0) + 1;
```

`IS_GREATER` is a macro that is not wrapped in parentheses. `m1` will actually evaluate to `0`, since `1 > 0+1 = 0.`

Macro is purely text substitution.

# C-6

```
#define NEXT_BYTE(a) ((char*)(a + 1));

int a1 = 54; // &a1 = 0x100
long long a2 = 42; // &a2 = 0x200
void* b1 = NEXT_BYTE(&a1);
void* b2 = NEXT_BYTE(&a2);
```

# C-6

```
#define NEXT_BYTE(a) ((char*)(a + 1));

int a1 = 54; // &a1 = 0x100
long long a2 = 42; // &a2 = 0x200
void* b1 = NEXT_BYTE(&a1);
void* b2 = NEXT_BYTE(&a2);
```

`b1` is a void pointer to the address `0x104`.
`b2` is a void pointer to the address `0x208`.

# C Workshop

- If you had trouble with the previous exercises, **go!!!**
- Saturday, October 10 in the afternoon. Details TBA.
- Material:
  - Undefined behavior, casting
  - Structs, pointers
  - Memory management
  - Standard library functions
  - Random stuff: macros, typedefs, function pointers, header guards… and anything else you have questions on!

# The C Standard Library

- `#include <stdlib.h>`
- Use it. It is your friend!
- Don’t write code that’s already been written!
  - Your work might have a bug or lack features
- All C Standard Library functions are documented.
  - Use the UNIX `man` command to look up usage

# Robustness

- Code that crashes is bad.
  - Avoid making bad things!
  - Don’t write code with undefined behavior
  - Check for failed system calls and invalid input
- Some errors should be recoverable, others not
  - Proxy Lab is an excellent example of this
- Free memory that you allocate
  - Leaky code will crash (and code that crashes is bad!)
  - Memory leaks will cost you style points

# Robustness: Continued

- CSAPP wrappers check return values of system calls
  - Terminate program when error is encountered
  - `Malloc`, `Free`, `Open`, `Close`, `Fork`, etc.
  - Super duper useful for Proxy & Shell Labs
- Alternatively, check for error codes yourself
  - Useful when you don’t want program to terminate

```
FILE *pfile; // file pointer
if (!(pfile = fopen(“myfile.txt”, “r”))) {
    printf(“Could not find file. Opening default!”);
    pfile = fopen(“default.txt”, “r”);
}
```

# Quick C Tip: `getopt`

- Used for parsing command-line arguments
- **Don’t write your own code for this**. Not worth it.
  - In fact, we actively discourage it
  - Autograder randomizes argument order
  - Try it: `man getopt`

# Style Points

- We read and grade your code for style
  - Style guide: http://cs.cmu.edu/~213/codeStyle.html
  - Vim macro to highlight lines longer than 80 cols:
    - `2mat ErrorMsg '\%80v.'`
  - Emacs users… :

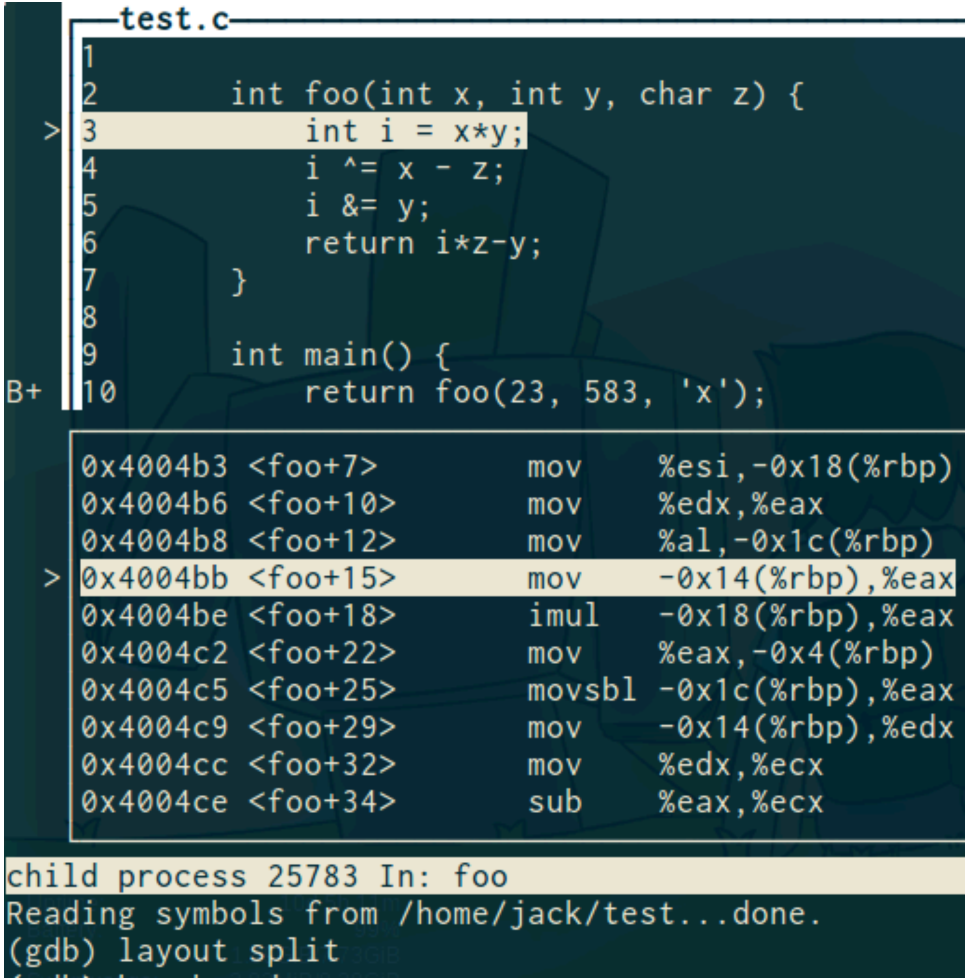
```
(setq whitespace-style '(trailing lines space-
 before-tab indentation space-after-tab)
 whitespace-line-column 80)
```

- View your annotated code on Autolab

# gdb

- Step through C code side-by-side with Assembly
  - Print variables, not just registers and addresses!
  - Break at lines, not just addresses and functions!
- `gdbtui <binary>` is gdb with a less-breakable user interface.
  - Nice for looking at your code during execution
  - Type `layout split` to view Assembly alongside

# gdbtui



```
test.c

        int foo(int x, int y, char z) {
>           int i = x*y;
            i ^= x - z;
            i &= y;
            return i*z-y;
        }

        int main() {
B+          return foo(23, 583, 'x');
```

```
  0x4004b3 <foo+7>          mov    %esi,-0x18(%rbp)
  0x4004b6 <foo+10>         mov    %edx,%eax
  0x4004b8 <foo+12>         mov    %al,-0x1c(%rbp)
> 0x4004bb <foo+15>         mov    -0x14(%rbp),%eax
  0x4004be <foo+18>         imul   -0x18(%rbp),%eax
  0x4004c2 <foo+22>         mov    %eax,-0x4(%rbp)
  0x4004c5 <foo+25>         movsbl -0x1c(%rbp),%eax
  0x4004c9 <foo+29>         mov    -0x14(%rbp),%edx
  0x4004cc <foo+32>         mov    %edx,%ecx
  0x4004ce <foo+34>         sub    %eax,%ecx
```

```
child process 25783 In: foo
Reading symbols from /home/jack/test...done.
(gdb) layout split
```

# `valgrind`

- Best tool for finding...
  - Memory leaks
  - Other memory errors (like double frees)
  - Memory corruption
- Use `gcc` with `-g` to give you line numbers of leaks
- Use `valgrind --leak-check=full` for thoroughness

# Version Control: Your Friend

- You may find it useful to use version control if you are already familiar with it
- If not, that’s okay, it’s not required. Making regular submissions to Autolab can act as a checkpointing system too.

# `gcc`

- GNU Compiler Collection
- Is a C compiler, among other things
- We will give you instructions for compilation in handouts
- `man gcc` if you’re having trouble

# make

- Lab handouts come with a `Makefile`
  - Don’t modify them
- You write your own for Proxy Lab
  - Examples for syntax found in previous labs
- `make` reads `Makefile` and compiles your project
- Easy way to automate tedious shell commands