ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 34

Vince Weaver

https://web.eece.maine.edu/~vweaver

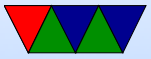
vincent.weaver@maine.edu

29 April 2026

Announcements

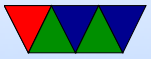
- Maine Day, realized too late that many people wouldn't be here.
- HW#8 won't be happening for various reasons
- Note on Lab#9: I am checking people off Wed+Thurs in lab 2-5

  Note I will be unavailable (at faculty senate) 3pm-4:30 today (Wed)
- If you have partially done labs, upload them for partial credit! If it won't let you upload to brightspace, let me



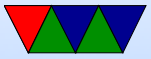
know

- Don't forget Student Evals
- There is a final, Wednesday during final week
  Will review Friday



Recursion / Iterative Functions

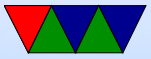
- We've so far looked at iterative functions
- You have a `main()` function and it can call out to other functions which return
- These functions can also call functions
- Functions that don't call any other functions are called `leaf` functions (you can picture your control flow as a tree, and leaves are at the end of branches)
- As an aside, compilers can sometimes optimize/inline leaf functions



Recursion / Calling Yourself

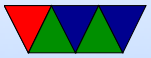
- Is there anything stopping a function from calling itself?

```
int foo(void) {
    ...
    a=foo();
}
```



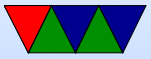
This is Recursion

- Function calling itself
- Worst case what can happen?
  - It can call itself forever
  - Each time will use some resources on stack
  - Eventually you will run out of stack
  - This might not even take very long, usually on Linux stack is limited to 8MB
  - You can find out with `ulimit -s`
  - You can remove the limit if needed



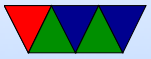
Aside on Stack Space

- 8MB limit can get in the way, local variables go on the stack so if you try to declare a local variable array bigger than 8MB your program might crash in a not very obvious way



Aside: Fork Bombs

- On Linux the `fork()` function starts a new process. After running it two copies of your program are running.
- Usually you'd use `exec()` to change one of them to a different program (this is how Linux launches new programs)
- What happens if instead each new process runs `fork()` again, and each of those does as well...
- Can be a DoS attack and it's sort of hard to defend against it

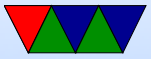


Back to Recursion – Why?

- Why would you recurse?
- Often common example is factorial
- $n! = 1$ if $n = 0$, $n! = n \times (n-1)!\ if n > 0$
- You can think of it as `factorial(n)=n*factorial(n-1)`

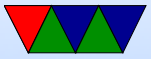
```
int factorial(int n) {
    if (n<=1) return 1;
    else return(n*factorial(n-1));
}
```

- Draw the call stack?



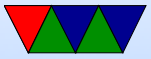
Factorial is tail-end recursion

- Only contains one call to itself, at end
- You can replace that with a simple loop
- `result=1; while(n>1) result*=(n--);`



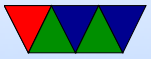
For most trivial examples recursion not needed

- Can be a clever way to solve problems
- Can be dangerous in embedded systems with less resources as hard to predict how much memory will be taken by an algorithm



Recursion – Other Common Examples

- Fibonacci Sequence
- Towers of Hanoi

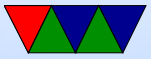


Recursion – Real Life Use

- Some sort algorithms (merge sort, quicksort)
- Writing a compiler. Take C code and parse it
- Like in HW#7 you can have things like:

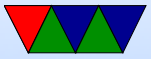
```
z=foo(bar(x+y+bax(3)),bar(y));
```

- You have a parser that parses expressions, but when expressions can contain more expressions you can just recurse to handle it



Command Line

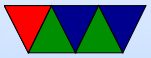
- We've so far looked at embedded code, but also running at the Linux command line
- Why do that and not nice GUI programs (either Windows, MacOS or Linux?)
- It turns out writing proper GUI programs, even just hello-world type, are an order of magnitude more difficult
- Also I am old-fashioned and still do most of my work at the text command line if possible



Command Line Arguments

- You can run programs that take arguments
- You've seen this with gcc at least

```
gcc -O2 -Wall -o hello hello.c
```



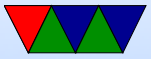
Did you ever wonder how to access those?

- Remember when we define main() it's something like

```
int main(int argc, char **argv)
```

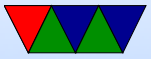
- argc is the number of command line arguments
- It is always at least one. The "first" (0th) argument is the name of the program
- argv is an array of strings, which is why sometimes you see things written

```
int main (int argc, char *argv[])
```



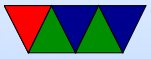
Sample Program

```
int main(int argc, char **argv) {
    int i;

    printf("There are %d command line arguments,argc);

    for(i=0;i<argc;i++) {
        printf("%d: %s\n",i,argv[i]);
    }
    return 0;
}
```

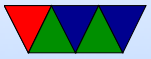


Aside: converting string to int

- What if you pass an integer in? But it's passed as a string.
- How do you convert a string to an int?
- `atoi(string)` – simplest, but problem, what if you pass a non-number to it? Can it report an error? It just returns 0 on error.
- `atof(string)` – same but for floating point
- For error checking might use one of the following:
- `strtol(string,endptr,base)` string to long



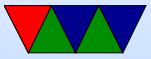
  - Base is 2 to 36 (why 36?), 0 means auto-detect 0x hex, 0b binary, leading 0 octal
  - string is string to convert
  - endptr can be NULL. If a valid pointer, it points to the first invalid character in the conversion
  - can check errno for errors
- `strtod()` – like strol but for double



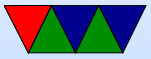
More advanced

- Giving options like `-h` or worse, `--help`
- Can parse yourself, but also `getopt()` will do a lot for your
- Example

```
while ( (c=getopt(argc, argv, "hvo:") ) != -1) {
    switch(c) {
        case 'h': print_help(argv[0],0);
                  break;
        case 'v': print_help(argv[0],1);
                  break;
        case 'o': out_name=strdup(optarg);
                  break;
```



```
        }
    }
    for(i=optind;i<argc;i++) printf("%s\n",argv[i]);
```

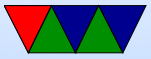


What about Environment Variables?

- You maybe have seen these before
- Variables you can set in your shell that programs can read
- For example, if compiling manually the homeworks

```
PICO_SDK_PATH=../../../pico-sdk cmake
```

- Can see environment variables by running `env`, a lot there
- In the old days you might carefully set these all up so when you logged in it had lots of settings made

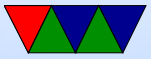


Reading Environment Variables

- The OS sets these up for your program and there's a pointer to it too
- main() gets a pointer to this too though usually you don't include it unless you need it

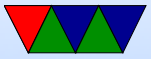
```
int main(int argc, char **argv, char **envp)
```

- Unlike argv you aren't told how many there are, so to find out the last pointer in the array is `NULL`



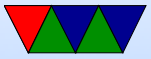
Reading Environment Variables

```
int main(int argc, char **argv, char **envp) {

    int i=0;

    while(envp[i]!=NULL) {
        printf("%d: %s\n",i,envp[i]);
        i++;
    }
    return 0;
}
```



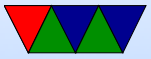
All these pointers, where do they come from?

- It's an OS thing
- Linux it puts them at the top of the stack when your program runs
- Then populates the pointers
- When writing your own OS (ECE531) have to do this yourself
- It means your stack offset can change based on command-line / environment, can actually change



program behavior

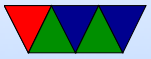
- TODO: show image?



In addition there are AUX vectors

- OS-specific info
- Live above stack with args and env
- Use `getauxval()` to get them, predefined names, things like `AT_NULL`, `AT_RANDOM`, `AT_PAGESZ`
- Uses a union to hold data

```
struct {
    long int a_type;
    union {
        long int a_val;
        void *a_ptr;
        void (*a_fn); // func pointer
    }
```



```
};
```

- Can test: `LD_SHOW_AUXV=1 sleep 1`