ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 14

Vince Weaver

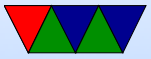
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

27 February 2026

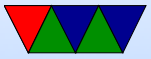
Announcements

- HW#4 was posted
- There is going to be a midterm review for this class at 5pm on Wednesday March 4th in Barrows 130 with pizza
- Lab#5 next week. LEDs. More on Monday.
- Some people asked if there's a lab on Friday for makeup purposes? The answer is no, no Friday lab



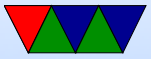
Homework #4 Hints

- For the first set of problems (comparisons) there's no need for an `if`. If it asks for "true if x not equal 0" it is just looking for `(x!=0)` or similar
- Array questions: most common issue is off-by-one errors. In C an array `int a[5]` is indexed from 0..4, not 1..5.

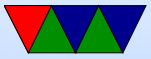


Aside on C syntax

- We've been learning C sort of by example
- You might say, is there a formal syntax definition, so I know when to put a semicolon and when not to, etc?
- There is a formal definition in the C spec. Might cost money to see it
- Mostly used when writing a compiler that has to parse the C files
- Some languages have railroad diagrams, but can't find one for C? Here's one for Pascal

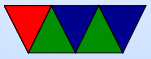


https://www.applefritter.com/files/PascalPosterV2.pdf



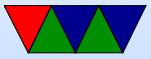
Notes from Last Week

- Remember way back talking about strings
- I got the `strcmp()` explanation backward, it returns 0 (false) on match not non-zero. Which is unusual in C. Rest of my explanation correct



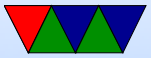
pointers

- We've been talking about variables
- They live in memory
- So far we haven't cared *where* they live in memory
- When we ask what the value of int x is, the compiler tracks that but we don't have to care
- But what if we do want to know?



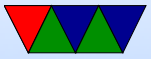
Review: what is computer memory?

- Temporary, volatile storage (goes away when power turned off)
- Your variables (and usually your executable code) live here
- Note this is RAM. DDR4/DDR5/LPDDR. Traditionally comes in sticks called DIMMs but doesn't have to
- This is different than permanent storage (hard-drives/SSD)



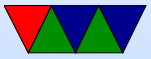
How much memory do you have?

- These days you probably have multiple gigabytes of RAM
- 32-bit systems in theory max out at 4GB but 64-bit can have up to 16 Exabytes (1 billion gigabytes)
- Frontier (recent large supercomputer) has 9 Petabytes, which is 9 million gigabytes
- Memory sizes go up by multiples of 1024: bytes, kilobytes, megabytes, gigabytes, terabytes, petabytes, exabytes, zettabytes, yottabytes.



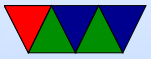
What about smaller computers

- I do retro coding on machines with 64k or even 128 bytes of RAM
- First 32-bit computer my family had was a 386 33MHz with 4MB RAM
- As an undergrad I had a 486 66MHz with 20MB of RAM and you could run full Linux including GUI and web-browser on it
- Pi Pico has 264k of RAM (the Pico 2 has 520k)

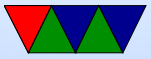


How can you visualize RAM?

- Imagine it starting at 0 and then going up till you hit the limit
- Draw on board. Often draw with 0 at the bottom and the max at the top
- For example imagine a system with 4GB of RAM
- Your program code goes here, as well as all of your variables
- Where exactly your variables end up is interesting and we might talk about it later but not now

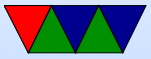


- Note that if you run an OS you are running with something called “virtual memory” which again we’ll skip for now but note your computer is up to something to make your life easier



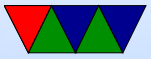
Variable Scope

- We've glossed over this, but there are two types of variables in C
- Global and Local



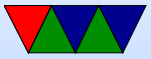
Global Variables

- Declared outside of any functions, and these are visible to any function in the program.
- This can make your life a lot simpler, but it is considered bad form to use them unnecessarily



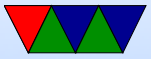
Local Variables

- Sometimes called automatic variables
- Can be declared inside of any block; traditionally done at function level (including `main()` which is just a function too)
- Created on stack when function/block is entered
- Are destroyed when the function/block exits
- Does not remember value across function invocations



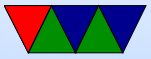
Local Variables Continued

- Cannot be accessed outside the block they are declared in
- *CONFUSING* can have same name as a global variable or a variable in another block. The most inner one takes precedence.
- Each time the block runs, the variable is recreated from scratch, it does not remember values from previous runs



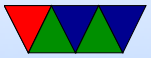
Advanced: Static Local Variables

- If you want a local variable to retain its value across function calls you can declare it `static`
- Something like `static int x;`
- In practice this is making it a global variable but only visible in the current function



Pointers: Location in Memory

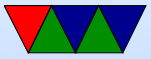
- All of your variables, code, and other things live in memory
- What if you want to find out where?
- Many/most languages (python, java, javascript, etc) specifically hide this because it causes a lot of trouble if you can find out
- C is a systems language so not only can you find out where things are in memory, but it's necessary in some cases



pointers – getting address

- Use the & operator to get the address of a variable
- Note we are not doing an AND operation. This is unary (takes one argument) and means "get the address of"

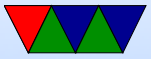
```
int x;
printf("%p\n",&x); // %p means "print a pointer in hex"
```



pointers – declaring pointers

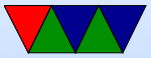
- We can declare pointers

```
int x;
int *ptr; // declaring with * means it is a pointer
ptr=&x;
```



Aside: declaring many pointers

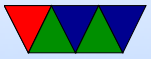
- We said you can say `int *ptr;`
- note it's also valid syntax to do `int* ptr;` but don't use it, it's confusing.
- It comes up when multiple per line `int *x,y,*z;` x and z are pointers, y is plain int.
- If you do `int* x,y,z;` only x is a pointer, rest are plain int



pointers – dereferencing pointers

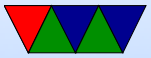
- If we have a pointer, we can use the * operator to follow the pointer into memory and get the value stored there
- This is called de-referencing a pointer

```
int x=5;
int *ptr;
ptr=&x; // ptr is a pointer to x (which is 5)
printf("%p\n",ptr); // prints pointer (address)
printf("%d\n",*ptr); // prints value there (5)
```



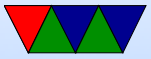
Why Use Pointers?

- Why do we care about what address something is at?
- Memory Mapped I/O:
  - When doing low level code you might want to write to an exact location in memory.
  - Maybe to light up an LED you have to write to address 0x20000000 exactly
  - In C you can declare a pointer to exact memory like that
- Arrays and Strings



- In C all arrays and strings are secretly pointers
- `int a[10];`, a is just an integer pointer to a region of memory that has room for 10 ints.
- We'll revisit this later

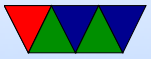
- Passing variables into functions
  - When you call a function, `printf("%d",x);` the variable x itself is not passed to printf, just its value. printf can take this value it gets and do anything it wants with it, but when printf ends that copy goes away, and any changes do not make it back to the caller.
  - A function cannot see the local variables in other



functions

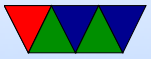
- If you want a function to be able to modify a variable you have, you have to pass into it the *pointer* (location) to the variable, not its value. The function can use the pointer to modify the value through the pointer and so when you return from the function the variable has been changed because

- Allocating memory dynamically, needs functions



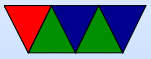
scanf()

- Now know enough to `scanf()`
- Much better way of getting input than just `getchar()`
- Can read into strings, requires knowledge of pointers
- Vaguely similar to `printf()` in reverse



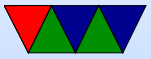
scanf() usage

- like printf, first argument is formatting string
- spaces ignored
- format string like with printf
- if there are characters not in a formatting string, then what you type must match this, if it doesn't the scanf will fail



scanf() example

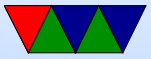
```
int x;
scanf("%d",&x);
printf("%d\n",x);
```



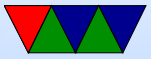
scanf() example – why need pointers?

```
int x=0;
scanf("%d",&x);
printf("%d\n",x);
```

- if called `scanf("%d",x);` would pass in value, 0 to scanf which can't do anything with
- We pass in pointer `&x`
- scanf can then get the integer, and then write the value to the equivalent `*x` which will follow the pointer and put the value into x in our function

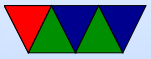


Other scanf codes



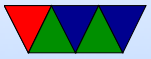
scanf specifiers

%d or %u	signed / unsigned decimal integer
%i	integer (can be decimal/hex/octal)
%x	hexadecimal
%o	octal
%c	character
%s	string
%p	pointer



scanf specifiers – floating point

`%f`	float
`%lf`	double
`%e`	exponent notation
`%g`	



scanf specifiers – other

%%	if you want to read a percent percent
[–]	describes a “scan set”
*	scan in value but don’t write out to variable