ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 35

Vince Weaver

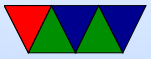
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 May 2026

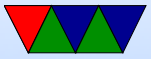
Announcements

- Labs are done. If somehow you missed the deadline e-mail me.
- I will make sure anything in brightspace currently without a grade gets graded. It might not be until next week though. I will send an e-mail when all outstanding labs are graded.
- Don't forget Student Evals. They extended it until Sunday. We're at 37% which isn't bad.
- There is a final, Wednesday May 6th, 1:30pm

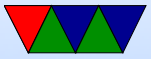


Final Info

- Closed book, notes, no electronics
- Will be similar to the midterms
- Will not be significantly longer than the midterms

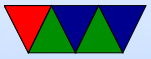


Midterm1 Material



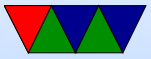
Number Systems

- binary / octal / hex numbers (don't worry about octal)
- Might not ask a specific question but be familiar with how binary and hex works



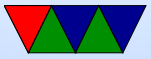
Variable Types

- Variable types, char, short, int, long, float, double
- additional specifiers like signed / unsigned
- Define a variable that can fit 3.14159
  double. float also ok (less precise)
- Define a variable that can hold 255
  unsigned char or larger. Often in C you use "int" by default unless there's a reason to use something smaller



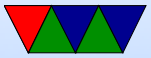
printf()

- `printf()`
- I'm probably not going to ask a specific question, but be sure to know:
  - Takes a string argument
  - This string can have replacement placeholders like `"%d"` which say to substitute in a value from the parameter list
  - For each substitution there needs to be a parameter



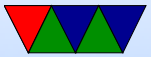
Loops

- Loops for/while/do-while
- How to make an infinite loop
- Loop types:
  - For loops most common
  - While loops mostly equivalent.
  - Do while loops will always run loop body once



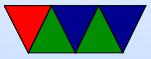
Flow Control – if/else

- Allow taking multiple paths through program based on conditions
- You can nest these arbitrarily deep
  Try to keep things indented properly (problem in labs)



Flow Control – switch/case

- Can replace a long string of if/elses with switch case, where you "switch" on an integer value and have different cases as appropriate
- there's a `default` case
- After each case you need a `break`
- Can have trouble with fallthrough



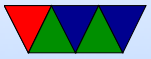
Arrays

- Declaring arrays

```
int a[5];
a[5]=1;
```

- How to initialize array?

  You can use a loop, indexing with a variable
- How to print out all values in array?



Strings

- Array of char

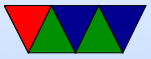
```
char str[10];
```

- C has NUL terminated strings
- To get size use `strlen(str);`
- You cannot compare with equals

```
char str1[]="Hello",str2[]="World";

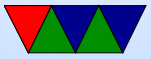
if (str1==str2) printf("Same\n"); // won't work

if (!strcmp(str1,str2)) printf("Same\n"); // proper way
```



Functions

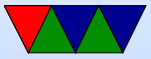
- Sub-routines you can call
- Need to be declared before you can call them
- Optionally can return one value
- Can take an arbitrary number of parameters, passed in by value (usually)
- If you want to pass in a value that allows changing a value in the calling function, need to pass a pointer



Bitwise vs Logic Operators

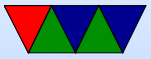
- Bitwise operate on every bit in an integer
- Logic operate in a boolean (true/false) fashion where 0 means false and non-zero means true
- If you use the wrong one compiler might not complain and it might even sort of work but only due to luck

	bitwise	logic
and	&	&&
or	|	||
not	~	!
xor	^	
shift	<< >>	



Bitwise Operators

- OR `|` is generally used to set bits
- AND `&` is generally used to mask, or clear bits (often in conjunction with not)
- XOR `^` is generally used to toggle bits
- NOT `~` will flip all bits.
- SHIFT `<<` or `>>` will shift bits left or right by a count value



Logical Operators

- Boolean Logic, same as bitwise but on a single bit (true or false)
- OR `||` is used to see if either case is true

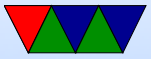
```
if ((x==4) || (y==5))
```

- AND `&&` is only true if both cases are true

```
if ((x==4) && (y==5))
```

- NOT `!` will flip a result from true to false, or false to true

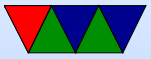
```
if (!(x>4))
```



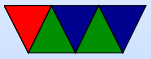
Boolean in C

- No dedicated Boolean type, just 0 or non-zero
- Any non-zero value is considered true

```
int cookies_left=5;
if (cookies_left!=0) printf("There are cookies left\n");
if (cookies_left) printf("There are cookies left\n");
```



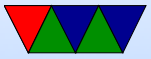
Midterm 2 Material



structs – defining

- Define a struct with a tag-name/template-name/type of grades containing the members:
  - an integer named missed_classes,
  - a double named grade_average,
  - and an array of 9 shorts named quiz_grades

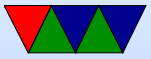
```
struct grades {
  int missed_classes;
  double grade_average;
  short quiz_grades[9];
};
```



structs – declaring

- Declare a struct of type `grades` from above called `final_grades`

```
struct grades final_grades;
```



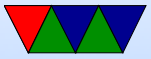
structs – assigning with .

- In the `final_grades` struct defined previously, write the statement needed to set the grade average to 85.3

```
final_grades.grade_average=85.3;
```

- Now set the last element of the quiz_grades array to be 95

```
final_grades.quiz_grades[8]=95;
```



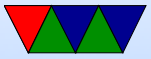
structs – assigning with pointer

- Repeat the previous but instead of `final_grades` being a struct it is a pointer to the struct
- As an aside, to be proper you have to make sure the pointer puts to something so assume it's been malloc()'d with something like this

```
struct grades *final_grades=
                    malloc(sizeof(struct grades));
```

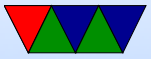
- The big difference is you use the arrow -> (minus + greater than) to "point" to the member rather than a .

```
final_grades->quiz_grades[8]=95;
```



structs – additional background

- Be aware of padding
- Though in real life this doesn't come up much:
  - accessing hardware directly via volatile
  - trying to write data structures directly to disk byte-by-byte
  - Passing structs to Linux and it is strictly enforcing that unknown bytes (including padding bytes) must be zero

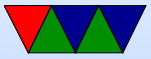


scanf

- You have an integer `x`. You want to read in from the console / keyboard an integer that is typed in

  ```
  scanf("%d",&x);
  ```

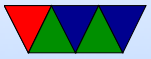
- Note it has to be a pointer you pass in, otherwise scanf can't change it in a way that you can see
- Returns number of successful values read in
- Don't put a `\n` in, so don't do `scanf("%d\n",&x);` it won't do what you expect and it's complex to explain why



pre-processor – constants

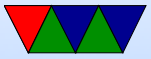
- You want to use the preprocessor to make a constant called `SQRT_2` that has the value `1.4142`

```
#define SQRT_2 1.4142
```



pre-processor – includes

- manpage says you need to include the system `stdio.h` to use a function, what does that look like
- `#include <stdio.h>`
- Remember, use angle brackets for system includes, double quotes if it's a local include



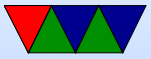
pre-processor – conditional compilation

- Your code defines a pre-processor define like

```
#define SOUND_ENABLED 1
```

- How would you have code that would only play sound if that define is set at compile time?

```
#if (SOUND_ENABLED==1)
    play_sound();
#endif
```



pre-processor – macros

- Define macro PRODUCT that multiplies two variables
- Will this work?

```
#define PRODUCT(x,y) x*y
```

- What happens if someone calls

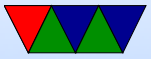
```
a=PRODUCT(q+r,w-h);
```

- That would become

```
a=q+r*w-h;
```

  which due to order of operations is not what you want. Be sure you have sufficient parenthesis!

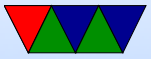
```
#define PRODUCT(x,y) ((x)*(y))
```



pointers – declaring

- We have an integer x;
- Declare a pointer to an integer, the pointer is named `ptr`

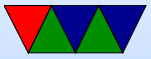
```
int *ptr;
```



pointers – taking an address

- Write the statement that points ptr to the address of x

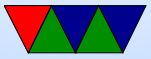
  `ptr=&x;`
- It's OK if you can't write an ampersand, but make it clear somehow that it is one



pointers – dereferencing

- Write a `printf()` statement that prints the value of the integer pointed to by ptr

```
printf("%d\n",*ptr);
// note this is different than printing the
// value of the pointer which is this
printf("%p\n",ptr);
```



malloc / calloc

- We have an integer pointer ptr1

```
int *ptr;
```

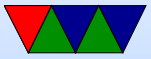
- How can we dynamically allocate an array of 1024 integers using `malloc()`?

```
ptr1=malloc(1024*sizeof(int));
```

- How could we do this using `calloc()`?

```
ptr1=calloc(1024,sizeof(int));
```

- Remember, calloc zeros the memory, malloc doesn't

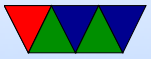


NULL

- If `malloc()` fails, what value does `ptr1` get?
- `NULL` which is a special value that means an invalid pointer
- What happens if you dereference a `NULL` pointer?

```
int *ptr=NULL;
*ptr=5;
```

- If you have an OS this will hopefully crash your program
- If you don't (maybe on a Pi Pico) it might just set some piece of memory you don't own to 5



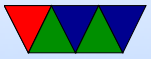
Accessing malloc() Memory

- Assume for this question that the first 3 elements of the array are called the 0th, the 1st, and the 2nd element
- How would we set the second element of ptr to 5 using array notation?

```
ptr1[2]=5;
```

- How would we set the second element to 5 using pointer math?

```
*(ptr1+2)=5;
```



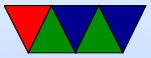
Using free()

- When you are done using memory allocated with malloc/calloc/free how do you free it up?

```
free(ptr);
```

- Should you free a pointer that is NULL or that you have already freed?

  No, that could crash the program



Memory Leaks

- If you have code like this

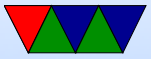
```
int *ptr;
while(1) {
    ptr=malloc(sizeof(int));
    *ptr=5;
    printf("%d\n",*ptr);
}
```

  What can go wrong if you let it run forever?

  Memory Leak

- How can you avoid that problem

  Run `free(ptr);` inside the while loop

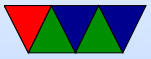


ternary operator

- Ternary operator lets you test an expression, and if it's true then assign the first value (between the ? and the :) and if it was false than assign the second value (between the : and ;)

```
y=x?a:b;
// equivalent to
if (x) y=a;
else y=b;

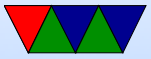
int days=10;
printf("Number of days left is %s than a week\n",
    days<7?"less":"more");
```



sizeof

- Get size of a variable in bytes
- Handy when using malloc() and such
- Examples

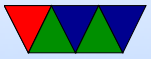
```
x=sizeof(int);          // size of an int
x=sizeof(struct foo);   // size of struct foo
x=sizeof(char *);       // size of char pointer
```



casting

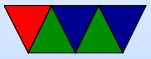
- Tell the compiler to treat a variable of one type as another
- There are few reasons you actually want to do this in modern C
- malloc() returns a void pointer, so to be really correct you can cast it to the right type before assigning

```
int *int_ptr;
int_ptr=(int *)malloc(1024);
```



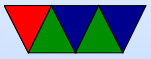
goto

- Can place a label in your code, which is name followed by a colon
- You can then in your code call `goto` which will jump the code execution to that label
- Note: there are many rules, including you can't goto between functions
- Due to structured programming many people frown on using gotos unless strictly necessary



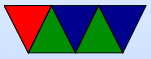
operator precedence

- C has a complex 15-level set of rules about operator precedence / operator ordering rules
- Don't memorize the chart, I will give it to you if needed
- Some examples, don't dwell on this too much as we haven't really had this in the homeworks yet



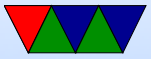
operator precedence example

- See Lecture where we went over this when going over midterm2



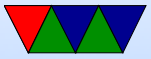
Topics Since Midterm#2

- Note: this material is more nice-to-know stuff, not core C topics
- It will not be emphasized as much on the final



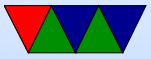
typedef

- Can make your own “type”
- Really just gives a new name to an existing type or struct
- C doesn’t enforce the type safety
- Probably not going to be on the final

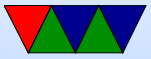


volatile

- Tells the compiler not to optimize reads
  (for speed reasons the compiler might throw out code it thinks doesn't do anything useful)
- If you're reading memory-mapped I/O hardware it might look like reading the same memory twice without an intervening store, which normally would be pointless
- `volatile` says to read the variable again anyway
- Try to limit using `volatile` to embedded systems, if you need it to make normal code work you might be

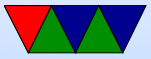


doing something wrong



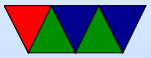
unions

- Lets you have elements of a struct overlap each other to save space
- In modern times with GB of RAM the space saved probably not worth the extra complexity
- See last lecture AUXV for a real-world example from Linux



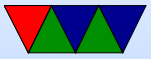
enums

- Let you use names instead of numbers
- ```
  enum color {RED, GREEN, BLUE};
  ```
- You can use the names instead of numbers to make your code more descriptive
- You can also do this with `#define` too which is sometimes more common


  ```

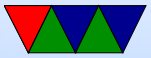
bitfields

- Lets you have a struct where ranges of bits are packed inside of integers
- In theory lets you avoid having to shift/mask manually
- Not very portable, so usually discouraged from using
- This won't be on the final



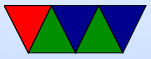
File I/O

- Lets you have a struct where ranges of bits are packed inside of integers
- In theory lets you avoid having to shift/mask manually
- Not very portable, so usually discouraged from using
- This won't be on the final



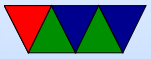
math.h

- Just some helpful math routines
- Some defines for things like Pi
- Trig function (sin/cos/tan), sqrt, exp, log, etc.
- All are subject to floating point limitations
- Also when compiling need to pass the `-lm` option



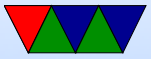
recursion

- Functions can call themselves
- If you're clever you can do useful things with this
- If you're not careful you can run out of stack memory
- Won't be on the final



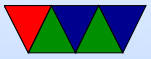
command line arguments

- Most likely not on final
- When running code at command line you can pass in parameters to the program
- These are given to main() as argc (number of arguments) and argv (an array of strings)
- There are routines like getopt() that make parsing easier
- There are also environment variables and AUX vectors you can use to get additional parameters



code quality / git

- Write good code
- Indent well
- COMMENT YOUR CODE
- Write tests
- Write good git commit messages



Writing Good Comments

- Try to give high-level descriptive comments
- Don't just describe what the code is literally doing:

```
x=x+1; // add one to variable x and write back to x
```

- Give something more high-level

```
x=x+1; // move paddle to the right
```