ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 27

Vince Weaver

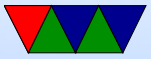
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

8 April 2026

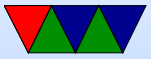
Announcements

- HW#7 not ready, will be due next week
- There will be an ECE review for midterm#2 (with pizza) Barrows Hall 130, Wednesday, April 8th from 5-7 pm
- Second midterm *next class*



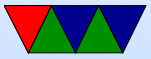
A few extra things on Lab#8

- Is taking a bit longer than last lab
- If you do run out of time, finish up next week. Lab#9 might be shorter, but also we'll have a makeup lab the week after that to get all caught up.



Lab#8 – Copying Files

- When renaming `bounce.c` to `paddle.c` be careful not to accidentally call it `paddle.c.c`
- Windows by default likes to hide file-extensions for some reason so if you try to save it as `paddle.c` it might tack on an extra .c in a non-obvious way
- If that happens the build setup is going to look for `paddle.c` and get the old one with no changes rather than the one you want
- To fix that just re-save it without the extra .c

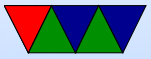


Lab#8 Function Prototypes

- I want you to think about coding when working on the lab, etc, which is why I sometimes use pseudo-code instead of C or only give you partial code or hints
- The lab handout might say use

```
size_t graphics_drawText(char *s,
        uint16_t x, uint16_t y);
```

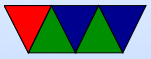
  to draw some text
- In the old days this is what the documentation, or manpage, or web-search would tell you. It's the function



prototype describing how a library routine works

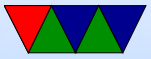
- The types are there to show you what a function returns, and also what parameters it takes
- Ideally you'd know that you remove the types before using, substitute the variables you want to pass in, and if you care about the return value you'd assign it to something

```
result=graphics_drawText(hello_string, 15, 0x23);
```

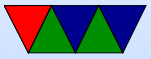


Lab#8 – Keypad

- Don't forget to call the init_keypad() code
- If your keypad code was looping 5..9 and your printf was printing row-5 to print properly, you're going to need to subtract off that 5 when calculating `(row<<4)|col` so you might have something like `((row-5)<<4)|(col-9)`



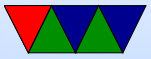
Review of Midterm 2 Topics



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

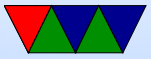
```
struct grades {
  int missed_classes;
  double grade_average;
  short quiz_grades[9];
};
```



structs – declaring

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

```
struct grades final_grades;
```



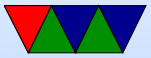
structs – assigning with .

- 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;
```



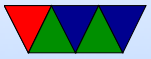
structs – assigning with pointer

- 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));
```

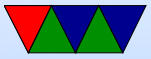
- The big difference is you use the arrow -> (minus + greater than) to "point" to the member rather than a .

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

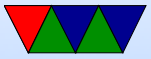


structs – additional background

- Padding in structs
  - Modern systems like variables to be aligned at 32-bit or 64-bit boundaries, so if you have a mix of smaller variables the compiler might "pad" a struct with unused empty space to make the alignment better
  - This might lead to unexpected results from `sizeof()`
  - You can use a packed attribute to force packed structs
- Why we use structs
  - Grouping variables together in their own namespace



- Making it easier to pass groups of variables into functions with one struct rather than each individually

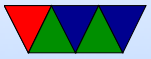


scanf

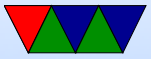
- You have an integer `x`. You want to read in from the console / keyboard an integer that is typed in

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

- Note in C you usually pass by value rather than reference. So if we just passed "x" then the `scanf()` function would just get a copy of the current value of x but no way to modify it so it is changed in the original function
- In C, if you want a function to be able to modify the value of variables in other functions you have to declare



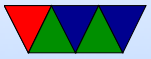
the function to take in pointers, and pass in pointers to the variables. Then the other function can change the values in a way that is visible to the originating function,



pre-processor – constants

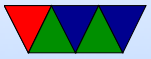
- You want to use the preprocessor to make a constant called `SQRT_2` that has the value `1.4142`

```
#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



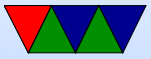
pre-processor – conditional compilation

- 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 a pre-processor macro PRODUCT that multiplies two variables
- Will this work?

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

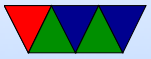
- What happens if someone calls

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

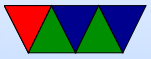
- That would become

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

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



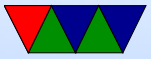
```
#define PRODUCT(x,y) ((x)*(y))
```



pointers – declaring

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

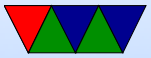
```
int *ptr;
```



pointers – taking an address

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

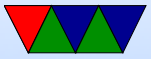
```
ptr=&x;
```



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;
```

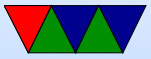
- How can we dynamically allocate an array of 1024 integers using `malloc()`?

```
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

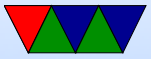


NULL

- 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



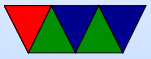
Accessing malloc() Memory

- 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;
```



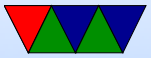
Using free()

- 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

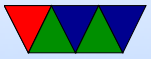
```
int *ptr;
while(1) {
    ptr=malloc(sizeof(int));
    *ptr=5;
    printf("%d\n",*ptr);
}
```

  What can go wrong if you let it run forever?

  Memory Leak

- How can you avoid that problem

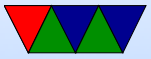
  Run `free(ptr);` inside the while loop



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 ;)

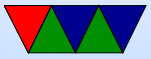
```
y=x?a:b;
// equivalent to
if (x) y=a;
else y=b;

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



ternary operator aside

- Did you notice a bug with that previous example?
- What if days is 7?
- You can do nested ternary operators, even though it's horrifying

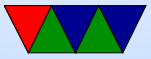
```
printf("%d: Number of days left is %s a week\n",days,
    (days<7)?"less than":(days==7)?
    "same as":"more than");
```



sizeof

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

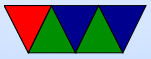
```
x=sizeof(int);          // size of an int
x=sizeof(struct foo);   // size of struct foo
x=sizeof(char *);       // size of char pointer
```



casting

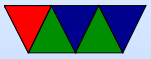
- Tell the compiler to treat a variable of one type as another
- There are few reasons you actually want to do this in modern C
- malloc() returns a void pointer, so to be really correct you can cast it to the right type before assigning

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



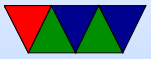
goto

- 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



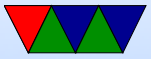
goto example

```
my_label:           // label
    //  ...
    //  ...
    if (x==y) goto my_label;
```



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



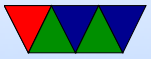
operator precedence example

- Put in parenthesis to match the operator ordering

```
y=1+x*3;
```

- Result:

```
y=1+(x*3);
```



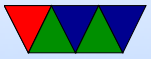
operator precedence example 2

- This is a common gotcha because `==` has higher precedence than `&` for historical reasons
- Put in parenthesis to match the operator ordering

```
if (x==y&3)
```

- Result:

```
if ((x==y)&3)  // how C parses, prob not what you wanted
```



operator precedence example 3

- Another common gotcha shift is lower precedence than addition
- Put in parenthesis to match the operator ordering

```
y=x+1<<8;
```

- Result:

```
y=(x+1)<<8;  // how C parses, prob not what you wanted
```

operator precedence example 4

- Another common gotcha, invert is higher than shift
- Put in parenthesis to match the operator ordering

```
y=~1<<8;
```

- Result:

```
y=(~1)<<8;  // how C parses, prob not what you wanted
```