ECE 177 – Programming I: From C Foundations to Hardware Interaction Lecture 30

Vince Weaver

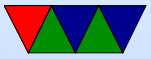
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

17 April 2026

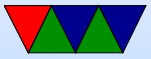
Announcements

- HW#7 finally posted, due next Friday
- Talk about research a bit (symposium)
  Research jobs at UMaine
  Also next year they will be hiring lab TAs for this class



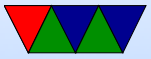
C Reserved Words

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			



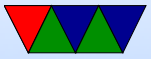
C Reserved Words

- Note this is fairly few
- C++ is a superset of C in a way, but has some extra ones
  - This can cause trouble when re-using C code as C++
  - The most troublesome one is `new`



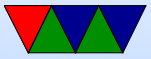
auto

- This is one of the remaining keywords in C
- It's useless which is why we haven't mentioned it yet
- Declares variable as local to the current block
- `auto int x;`
- This is the default in C anyway
- This exists because it was inherited from the previous B language



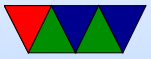
auto new behavior

- If you use C++ you might know auto got reused to mean something else
- It means automatically pick a type for a variable based on how you are initializing it
- So `auto f=1.5;` would automatically make it a float
- Less pointless: something like `auto f=init_something()` so you can change the type that returns and all variables will automatically get updated if declared auto
- With C23 the C behavior now can be more like this, but



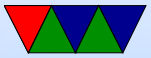
with lots of restrictions

- You can still get the old behavior with `auto int x;`

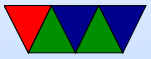


extern – for functions

- This is used with function prototypes
- Says the function might be defined in a separate file
- When building an executable the linker will note you are using an extern function and make sure it points to the right place
- `extern` not used very often because this is the default with a function prototype, you can leave it off
- Header files it's good practice to use extern
- If you look in `stdio.h` you'll see printf is declared

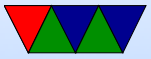


```
extern int printf(...
```



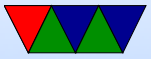
extern – for variables

- Can also be used to pre-declare global variables that will be in a separate file
- `extern int something;`
- You'll have to declare the variable at least once as non-extern



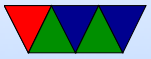
File I/O

- Reminder of what a file is
  - Set of bytes on your computer that you control
  - Don't let them take it away from you
- Harder to try out on Pi Pico as it's not set up with a full OS / filesystem
- Functions aren't part of C compiler, but part of C library



Simple File I/O

- Based on low-level Linux/UNIX
- On Linux/UNIX everything is a file
  Including device files as you'll see in ECE471
- System calls that call into the Operating System Kernel
- Based around having an opaque number called a "file descriptor" representing an open file

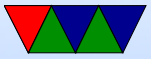


open()

- Use `open()` to open a file

```
fd=open("file.c",O_RDONLY);
```

- First argument string representing filename
- Second argument describes how to open it, a bunch RDONLY, WRONLY, RDWR also things like writing
- Optional third argument is permissions for file if creating for writing
- Return value is file descriptor
- Have to `#include <fcntl.h>`

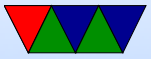


What if open() Returns an Error

- Syscalls return -1 on error
- To get more details can check `errno`
- `#include <errno.h>`
- This returns things like `EINVAL` or `EACCESS`
- Can use `strerror()` to get string

```
printf("%s\n",strerror(errno));
```

- Can also use `perror()`

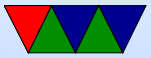


read()

- Reading from file
- Use `read()` to read in data from open file descriptor

```
result=read(fd,buffer,length);
```

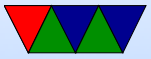
- First argument file descriptor
- Second argument pointer to a character buffer to read into
- Third argument is bytes to read
- Return value is bytes successfully read
- Have to `#include <unistd.h>`



write()

- Write to file
- Use `write()` to write data to open file descriptor

  ```
  result=write(fd,buffer,length);
  ```
- First argument file descriptor
- Second argument pointer to a character buffer to write out
- Third argument is bytes to write
- Return value is bytes successfully written
- Have to `#include <unistd.h>`

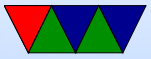


close()

- Use `close()` to close an open file descriptor

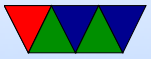
```
result=close(fd);
```

- First argument file descriptor
- Have to `#include <unistd.h>`



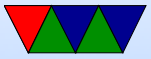
other low-level

- `lseek()` – seek position in file
- `sync()` – sync to filesystem
- See next lecture for some more



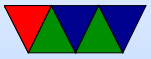
stdin/stdout/stderr

- By default on Linux get three file descriptors at start
- stdin (fd=0) can read from console
- stdout (fd=1) write to console
- stderr (fd=2) also goes to console, but you can filter it to easily separate out error messages



Buffered Stream I/O

- open/read/write are very low level
- They make it hard to do things like read in integers or lines of text
- Buffered I/O better for performance, has buffer and only calls out to write() once a lot has been written (or a linefeed)
  syscalls are expensive
- These are C library routines, deep inside they still do open/read/write

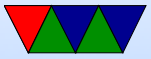


FILE pointer

- for buffered I/O you declare a FILE pointer

```
FILE *fff;
```

- This is actually a typedef'd pointer to a struct (that holds things like the file descriptor) but you're not supposed to mess with the struct directly

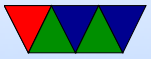


fopen()

- Use `fopen()` to open a file

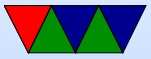
```
fff=fopen("file.c","r");
```

- First argument string representing filename
- Second argument describes how to open it r is read only, w is write only, r+ is read/write. a append
  On windows might need b for binary (vs text) RDONLY, WRONLY, RDWR also things like writing
- Have to `#include <stdio.h>`



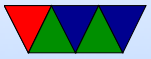
fopen() errors

- Return value is FILE pointer, or NULL if fails to open
- Can still check errno to see why it failed



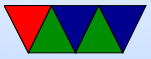
fread()

- Reading from file `result=read(buffer,length,file);`
- First argument pointer to a character buffer to read into
- Second argument is bytes to read
- Third argument is FILE ptr
- Return value is bytes successfully read
- Have to `#include <stdio.h>`



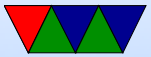
fwrite()

- Write to file `result=fwrite(buffer,length,file);`
- Have to `#include <stdio.h>`



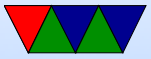
fclose()

- Use `fclose()` to close an open file `result=fclose(fff);`



Other FILE output routines

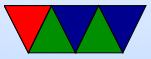
- `fprintf()` – like printf but first arg a file
- `fputc()` – put char
- `fputs()` – put string



Other FILE input routines

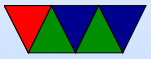
- `fscanf()` – like scanf but first arg a file
- `fgetc()` – read next char
- `getchar()` – like fgetc(stdin);
- `fgets()` – get line of text to new line (includes new line)

  Aside on `gets()` the forbidden stdin-reading function



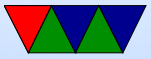
Other FILE useful routines

- `fseek()` – seek in a file
- `rewind()` – go to beginning of file
- `fsync()` – sync to filesystem
- `feof()` – see if hit end of file



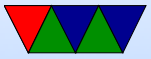
EOF (end of file)

- In C has value -1 typically
- When writing out from stdin can initiate end of file
  - Linux: control-D
  - Windows: control-Z
- On Linux t least this isn't actually at the end of files, the filesystem metadata tracks how many bytes are in a file



stdin/stdout/stderr

- predefined values you can use for writing
- `stdin` can read from console
- `stdout` write to console
- `stderr` also goes to console
- `fprintf(stderr,"Error message\n");`



filtering stdout

- On Linux can re-direct output
- `cat file.c > out`
- Can remap stderr, on bash can

  `./program > out 2> err`
- Can also take file and set as stdin `sort < file.c`
- Can also pipe output from one program into another

  `ls | sort`