string.h

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

Announcements

- Homework 9 due Monday on Gradescope
- Final Exam Thursday April 30, 2026 at 7pm

string.h manual page and examples

strcmp(), mcp(), strcat()

printf

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

a pointer to a format string (in register rdi)

telling the type checker: I don't care what's coming next.

We have calling conventions. The compilers will put the parameters in the right places, like registers, floating point registers or stack

(man 3 printf : check format string).

1. Format String Structure

The manual says:

A *format string* contains:

- **Ordinary characters:**
  These are copied *exactly as-is* into the output.
- **Conversion specifications:**
  These begin with `%` and tell `printf`:
  - that an argument must be consumed
  - how to interpret that argument
  - how to convert it into characters for output

The format string must contain *very specific information*, because without it:

`printf` would not know what arguments to read, in what order, or where in registers they live.

2. Components of a Conversion Specification

Each `%` item can include:

- optional **flags**
- optional **field width**
- optional **precision**
- optional **length modifier**
- a required **conversion specifier** (the final character)

"The simplest form is just `%` followed by a specifier character."

Here are a few `printf` examples:

- Simplest form: `printf("%d", 42);`
  - `%d` is the conversion specifier
  - prints: `42`
- With field width: `printf("%5d", 42);`
  - width is `5`
  - prints: `42`
- With precision: `printf("%.2f", 3.14159);`
  - precision is `.2`
  - prints: `3.14`
- With flags: `printf("%-8d", 42);`
  - `-` is a flag for left alignment
  - prints: `42`
- With length modifier: `printf("%ld", 123456789L);`
  - `l` is the length modifier
  - `%d` is still the specifier

One full example including several parts is:

`printf("%-8.2f", 3.14159);`

printf("%4.2f", 3.14159);
(%8.2f, 3.141);

This includes:

- flag: `-`
- field width: `8`
- precision: `.2`
- conversion specifier: `f`

It prints `3.14` left-aligned in a field of width 8.

```c
#include <stdio.h>

int main() {
    printf("23534624754765☺ç–½8");
    printf("x   x  \n\t");
    printf("\n------\n");
    int x = -2130;
    printf("A number: %d <=- like that\n", x);
    printf("A number: %o <=- like that\n", x);
    printf("A number: %u <=- like that\n", x);
    printf("A number: %x <=- like that\n", x);
    printf("A number: %X <=- like that\n", x);
    printf("%d + %d = %d (%s)\n", 2, 3, 2+3, "yay", 34);
    return 0;
}
```

printf prints exactly the bytes you give it
It does not automatically add a newline (unlike puts)

So we need to add '\n' to print a new line.

Conversion specifiers: a percentage sign, followed by what type of the thing is.

1 . Setting up a negative integer

```
int x = -2130;
```

We will print this same integer using different conversion specifiers.

2 . `%d`: signed decimal

```
printf("A number: %d <=-- like that\n", x);
```

Explanation:
`%d` prints the value **as a signed decimal integer**.
So the result is simply:

```
A number: -2130 <=-- like that
```

3 . `%o`: unsigned octal

```
printf("A number: %o <=-- like that\n", x);
```

Explanation:
`%o` interprets the bits of the int as an **unsigned integer**, then prints them in **base 8**.
Because `x` is negative, its bits are in **two's complement**, so interpreted as *unsigned*, it becomes a very large number.

Example concept:

```
-2130 (signed) → 0xFFFFF780 (unsigned
interpretation)
then printed in octal
```

This is why the result is a strange large octal number.

4 . `%u`: unsigned decimal

```
printf("A number: %u <=-- like that\n", x);
```

Explanation:
Again, interpret the bits as an **unsigned int**, then print in **decimal**.
This produces a huge number near `2^32`.

5 . `%x` and `%X`: hexadecimal

```
printf("A number: %x <=-- like that\n", x);
printf("A number: %X <=-- like that\n", x);
```

Explanation:

- `%x` prints the **hexadecimal representation** using lowercase `a-f`.
- `%X` prints the same value using uppercase `A-F`.

Since `x` is negative, C prints the two's-complement form:

Example (conceptually):

```
x = -2130
binary (32-bit) = FFFFF780
%x prints   fffff780
%X prints   FFFFF780
```

6 . Multiple arguments example

```
printf("%d + %d = %d (%s)\n", 2, 3, 2+3, "yay",
34);
```

! The format string only has 4 specifiers:

- `%d`
- `%d`
- `%d`
- `%s`

But the programmer passed **5 arguments**:

`2, 3, 2+3, "yay", 34`

Explanation:

- The first four arguments match the four format specifiers.
- The last argument (`34`) has **no corresponding % placeholder**.
- The compiler will **warn** but the program will still run.
- Extra arguments are simply **ignored** by `printf`.

Output:

```
2 + 3 = 5 (yay)
```

Passing Too Many Arguments:

```
useprintf-canned.c:13:53: warning: data argument not
used by format string [-Wformat-extra-args]
```

Passing Too Few Arguments:

```
useprintf-canned.c:13:28: warning: more '%' conversions
than data arguments [-Wformat-insufficient-args]
   13 |     printf("%d + %d = %d (%s)\n", 2, 3, 2+3);
```

Why the Compiler Warns for `printf`

The compiler *special-cases* `printf` because it's extremely common and extremely important.

Normally, C variadic functions **cannot** be type-checked. But for `printf`, the compiler actually **parses the format string** and checks argument count/mismatch.

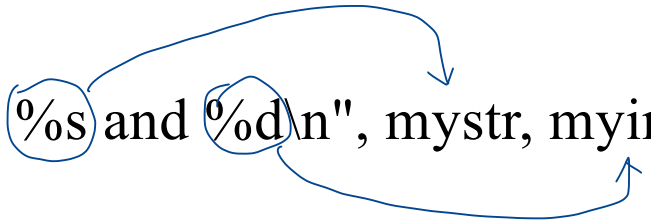
printf

Could you write printf?

printf

```
int printf(const char *format, ...);
```

```
printf("hi: %s and %d\n", mystr, myint);
```



You'd scan the format string one character at a time.

When you see %, you parse the specifier.

That tells you the type of the next argument.

You fetch the argument and convert it (e.g., convert int → its decimal characters).

You output everything to some destination.

"Where does the output actually go?"

`printf` ultimately sends characters to stdout**, which is a file-like object.**

To do that, `printf` must eventually call **the system call `write`**.

stdio.h manual page

1. What is `<stdio.h>`?

`<stdio.h>` is the **standard buffered input/output library** in C.

1. **Standard** (commonly accepted, default, expected way to do IO)
2. **Buffered** (IO is not instantaneous; it is collected in buffers)
3. **Input/Output** (reading from user / sending characters to the user)

2. Why is IO "weird"? The role of buffering

IO feels strange, especially in assignments, because **it is buffered**.

Why buffering exists

When you type something:

- OS must detect the keystroke
- translate the keyboard code to a character based on locale
- figure out which program should receive the input
- deliver that input to the program
- the program must ask the OS for it

This is expensive.
So instead of handling characters **one by one**, `<stdio.h>`:

- reads **a chunk** at once (maybe hundreds or thousands of bytes)
- stores it in a **buffer**
- future reads simply take characters from this buffer

This makes IO faster and more efficient.

Why buffering causes weird behavior

- Input may not appear until *Enter* is pressed
- Sometimes input "lags" or seems not to arrive
- Reads may get an entire line at once
- Output may not appear until the buffer flushes

This is normal.

3. Why is `FILE` hidden? (Encapsulation in C Standard Library)

`<stdio.h>` defines a type:

```
typedef struct _IO_FILE FILE;
```

…but the **contents of the structure are not shown**.

- This lets library authors change how `FILE` works internally **without breaking your code**
- This is similar to **Java encapsulation**: private fields + public methods
- As a programmer, **you are not supposed to know the internals**
- You only receive **a pointer to FILE** and use library functions to operate on it

If you think you need to know what's inside `FILE`, you're probably doing something wrong.

4. What's actually inside `<stdio.h>`?

4.1 Standard streams

These macros are provided automatically:

```
stdin
stdout
stderr
```

- `stdin` – default input (keyboard)
- `stdout` – default output (prints to console)
- `stderr` – error output (unbuffered or line-buffered)

Many functions operate on these by default if no `FILE*` is provided.

4.2 The large set of IO functions

Most start with **`f`** and take a `FILE*`:

- `fopen`
- `fclose`
- `fread`
- `fwrite`
- `fprintf`
- `fscanf`
- `fseek`
- `ftell`
- `feof`, `ferror`
- etc.

Whenever you see a function that takes a `FILE*`, it means:
"Which file / which stream do you want to operate on?"

Functions that do NOT start with `f`

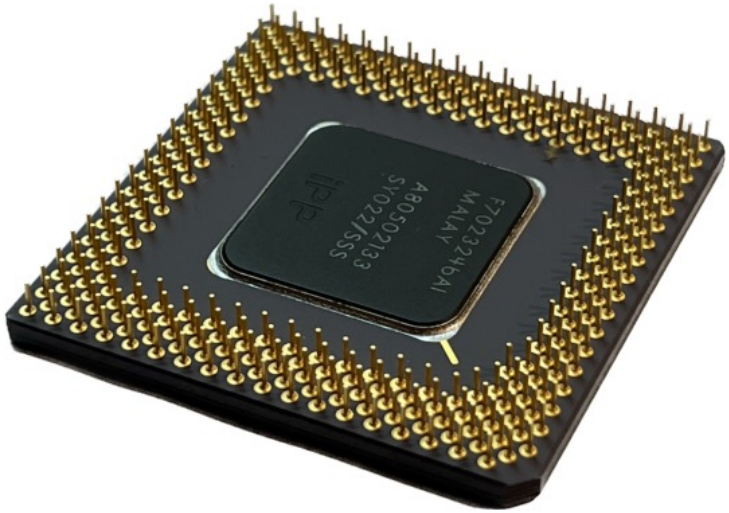
Examples:

- `printf` → prints to **stdout**
- `putchar` → sends one character to **stdout**
- `getchar` → reads one character from **stdin**

If there is **no `FILE*` parameter**, it is using a **default stream**.

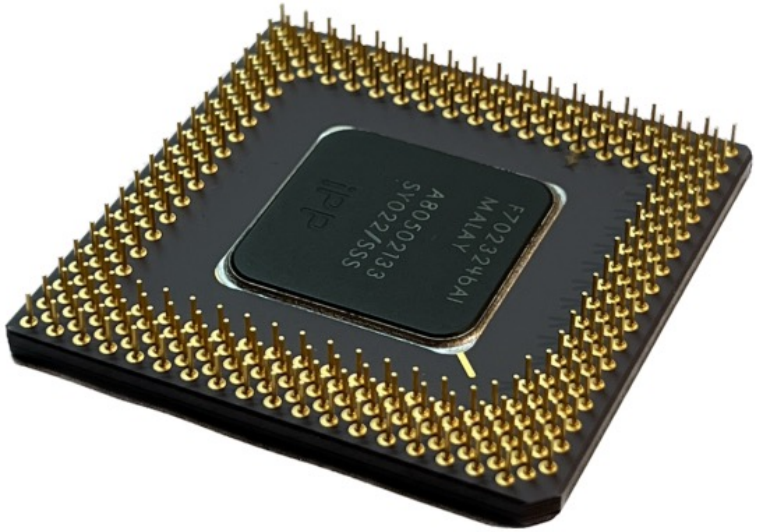
5. Why the library is designed this way

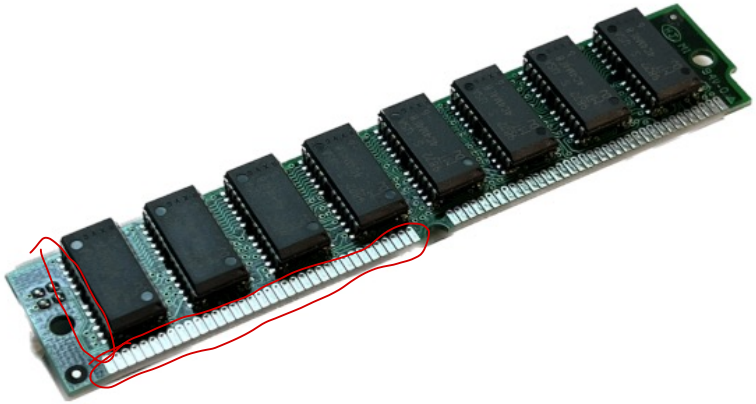
- The C Standard Library provides **a simple, expected, portable** way to do IO.
- Encapsulation lets the library evolve without breaking your code.
- IO is buffered because interacting with the OS character-by-character is extremely expensive.



CPU. (20 years old).

Those pins exist because the CPU must communicate with the outside world. Some pins carry power, but most are used to talk to other parts of the computer.

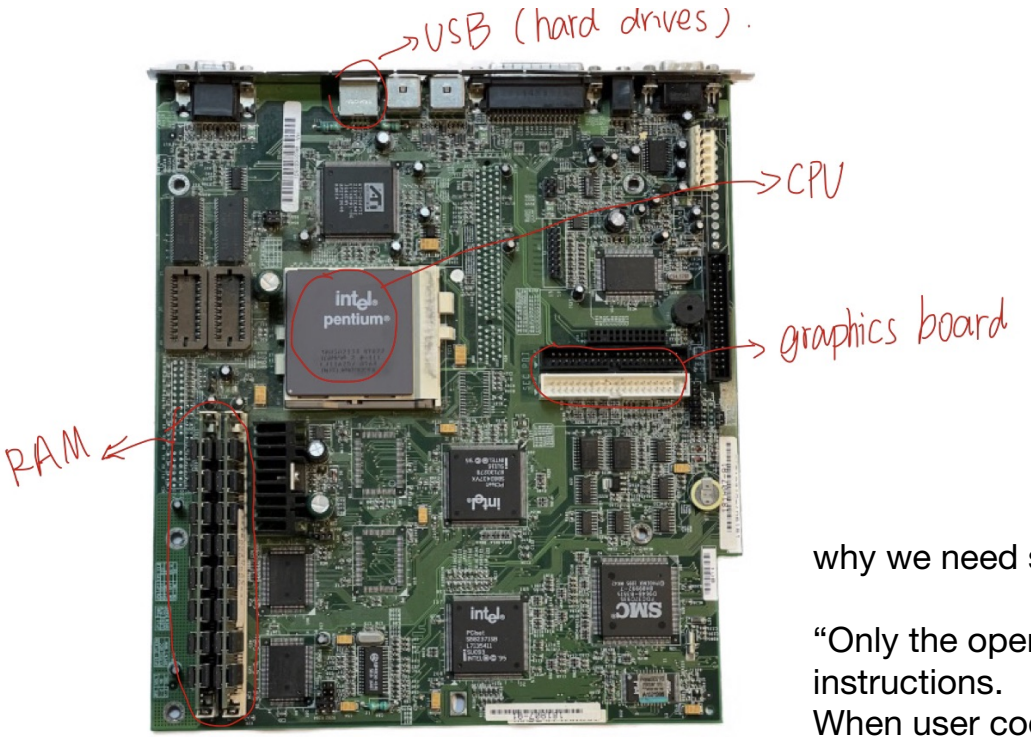




RAM. Notice it also has rows of pins.

Some of the CPU's pins are specifically for communicating with memory. This is how the CPU reads instructions, loads data, and stores results.

But remember, not all pins are for memory — the CPU still needs to talk to many other things."



Here is a full motherboard.
You can see the CPU, the RAM slots, and connectors for hard drives, USB, network cards, graphics, and more.
The CPU must communicate with all of these components — not just memory.
And here's the key point:
We have never discussed instructions that let us talk to these devices.
Those instructions exist, but they're extremely dangerous.
A user-level program is not allowed to execute them — the hardware blocks it — because a tiny bug could accidentally format your hard drive or send invalid commands to a device.

why we need system calls:

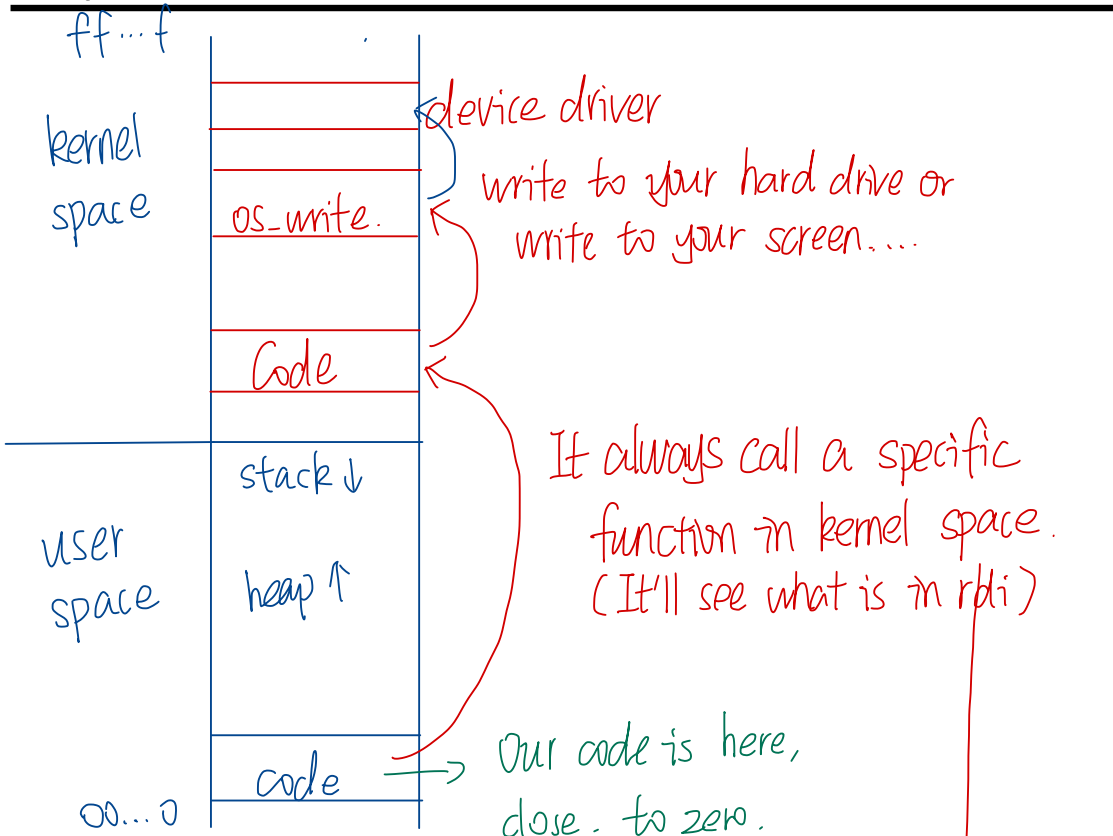
"Only the operating system (the kernel) is allowed to use those instructions.
When user code needs to talk to a device (like printing to the terminal), it must call into the kernel through a system call, like write()."

kernel is allowed to call these special assembly instructions that do things like write to the hard drive, write to the terminal, open/close the files, that kind of thing.

Syscalls

How do I get the kernel to do things?

how do we, as user programs, get the kernel to do things for us?



write.

argument checking

syscall → a special function call.

return value checking

return.

→ It always call the exactly same function

But we are not allowed to read/write in the kernel space. Syscall is allowed to do this. When I run this instruction, it becomes the operation system.