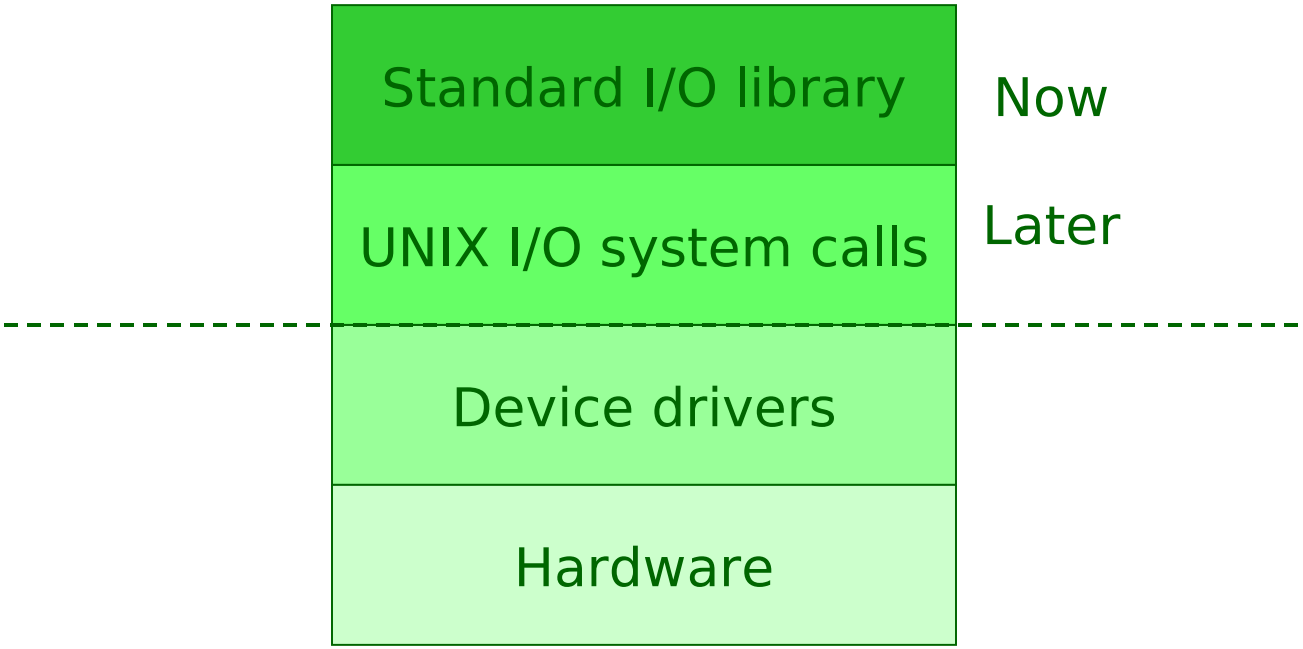
C’s Standard I/O Library

Alan L. Cox
alc@cs.rice.edu

Layers of I/O Abstraction



Standard I/O vs. UNIX I/O:

- **Only the common features**
- **Easier to use**

Character Output

```
#include <stdio.h>

int
main(void)
{

  putchar('H');
  putchar('e');
  putchar('l');
  putchar('l');
  putchar('o');
  putchar('\n');

  return (0);
}
```

Won't continually remind you of this

```
UNIX% ./hello
Hello
UNIX%
```

Cumbersome to output one character at a time

- **Still useful in some situations**

Difficult to format output

String Output

```
#include <stdio.h>

int
main(void)
{

  puts(“Hello\n”);

  return (0);
}
```

```
UNIX% ./hello
Hello
UNIX%
```

Better than a character at a time

- Still difficult to format output

Formatted Output

```
void
hello(char *name, int hour, int min)
{

  printf("Hello, %s, it's %d:%d.\n", name, hour, min);
}

int
main(void)
{

  hello("Alan", 2, 55);

  return (0);
}
```

Conversion specifications indicate the type of argument

`man 3 printf` **for complete list**

```
UNIX% ./hello
Hello, Alan, it's 2:55.
UNIX%
```

Character Input

```
#include <ctype.h>
#include <stdio.h>
/* Convert input to lowercase */
int
main(void)
{
  int c;

  c = getchar();
  while (c != EOF) {
    c = isupper(c) ? tolower(c) : c;
    putchar(c);
    c = getchar();
  }
  return (0);
}
```

Why int instead of char?

EOF is not a character, getchar returns this value at the end-of-file

We could implement these, but they are part of the standard library

String Input

```
#include <stdio.h>
/* Echo */
int
main(void)
{
  char input[100];

  while (gets(input) != NULL) {
    puts(input);
  }

  return (0);
}
```

What if input string > 99 characters? (+ terminator)
Don't use `gets`!

Returns NULL at the end-of-file

Formatted Input

```
int
main(void)
{
  char  name[50];
  int   num;
  float rate;
  int   dollars, cents;

  scanf("%d %s %f", &num, name, &rate);

  dollars = (int)rate;
  cents   = (int)((rate - (float)dollars) * 100);

  printf("%s is employee #%d and makes $%d.%02d per hour.\n",
         name, num, dollars, cents);

  return (0);
}
```

What if input string > 49 characters? (terminator)

Same conversion specifications as printf

Don't use scanf with strings!

Pointers – Why?

Why not &name?

Ugly C typecasting

More formatting

```
Input: 36 Alan 90.5E-1
Output: Alan is employee #36 and makes $9.05 per hour.
```

UNIX Shell Redirection

Can redirect keyboard & screen I/O to/from files

- I/O works with keyboard and screen by default
- Use redirection to read from/print to a file

```
UNIX%  program < infile
UNIX%  program > outfile
UNIX%  program >& outerrfile
UNIX%  program < inputfile > outputfile
```

other options depend on shell

What if you want to read/write multiple files?

Basic File Operations: Open & Close

```
FILE *fopen(const char *filename,
            const char *mode);
```

FILE * = file "handle"

"r" – read
"w" – write
"a" – append
…

```
int fclose(FILE *stream);
```

0 iff successful

Open files automatically closed upon exit

Built-in Files

`stdin` – keyboard input

`stdout` – screen output – normal output

`stderr` – screen output – error output

No need to open/close

Why two?

Writing to Files

```
int putc(int c, FILE *stream);
  - returns EOF if error
  - putchar specialized to stdout

int fputs(const char *s, FILE *stream);
  - returns the # of chars written (or EOF)
  - puts specialized to stdout

int fprintf(FILE *stream,
            const char *format,
            … );
  - returns the # of chars written
  - returns negative value on error
  - printf specialized to stdout
```

Reading from Files

```
int getc(FILE *stream);
  - returns character (or EOF)
  - getchar specialized to stdin

char *fgets(char *s, int n, FILE *stream);
  - returns s (or NULL)
  - gets specialized to stdin
  - max length argument (n) makes this safe!

int fscanf(FILE *stream,
           const char *format,
           … );
  - returns the # of matched and assigned inputs
  - returns EOF on error
  - scanf specialized to stdin
```

End of File

Need a way to detect when everything in a file has been read

```
int feof(FILE *stream);
```

True/false

```
int c = getc(FILE *stream);
if (c == EOF) …
```

Not a **char**

Example: Character I/O

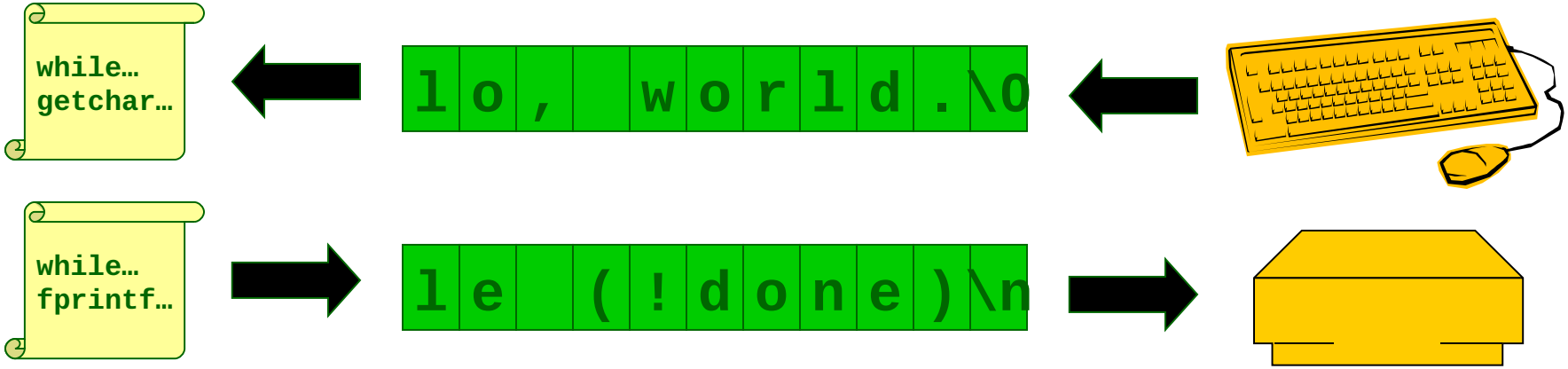
```
FILE *infile  = fopen(infilename, “r”);
FILE *outfile = fopen(outfilename, “w”);
int c;

/* Verify that fopen() succeeded. */
while (!feof(infile) {
   c = getc(infile);
   putc(c, outfile);
}
fclose(infile);
fclose(outfile);
```

Buffering

Streams are buffered

Manages data being produced & consumed at different rates

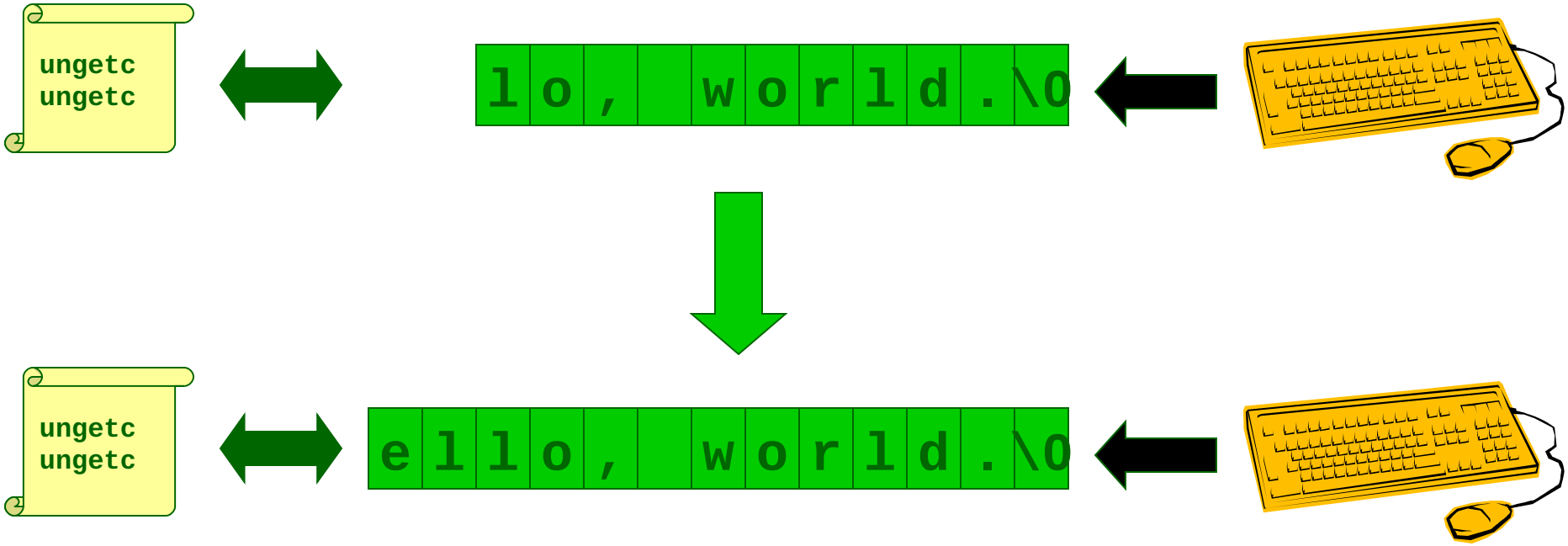
Transfer data in blocks, for efficiency



Main exception: **`stderr`** unbuffered

Buffering

```
int ungetc(int c, FILE *stream);
```



Buffering & Flushing

Flush **= empty buffer to output file**

When:

- **Internal buffer full**
- **Closing a file**
- **If line-buffered (e.g., stdout) → writing newline**

Mainly important in case of errors/interrupts

```
int fflush(FILE *stream);
```

Seeking

Can also write/read via random access:

```
int fseek(FILE *stream,
          long offset,
          int whence);
long ftell(FILE *stream);
```

Encodes “from here”, “from beginning”, or “from end”

offset

Or if a long isn’t big enough:

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

“Writing” & “Reading” To/From a String

```
int sprintf(char *s,
            const char *format,
            …);
int snprintf(char *s,
             size_t n,
             const char *format,
             …);
int sscanf(const char *s,
           const char *format,
           …);
```

What's Wrong With This Code?

```
char  *s = "Hello";

gets(s);
```

Doesn't check maximum string size.

- E.g., if user enters **"123456789"**, writes unallocated memory
- **<u>Never</u>** use **gets()** – use **fgets()** or repeated **getc()**

Basis for classic *buffer overflow attacks*:

- 1988 Internet worm
- Modern attacks on Web servers
- If overflows a stack-allocated buffer, can change return address!

What's Wrong With This Code?

```
int  value;
char s[8];
…
scanf("%d %s", value, s);
```

&value

scanf() expects pointers

(Be aware when functions take pointer args!)

Like **gets()**, could overflow buffer

Don't use **scanf()** for strings

Writing & Reading Words

```
int putw(int w, FILE *stream);
int getw(FILE *stream);
```

Compare:

```
putw(514,file1);
fprintf(file2,"%d",514);
```

? Results? ?

file1:

0x00	0x00	0x01	0x02

assumes big-endian

514 = 0000000000010010

file2:

0x35	0x31	0x33	0x00
'5'	'1'	'4'	'\0'

Writing & Reading Bytes

```
size_t fwrite(const void *ptr,
              size_t size,
              size_t nitems,
              FILE *stream);
size_t fread(void *ptr,
             size_t size,
             size_t nitems,
             FILE *stream);
```

More details

- I have only outlined the functions' purposes
- There are a handful of other functions
- See reference book, man pages, or web for details

Next Time

Memory Allocation