CSc 352
C - char arrays and reading input, make

Benjamin Dicken

Strings?

- The concept of a “String” as a type does not exist in the base C language. A “String” in C is an ***array of type char***
- Terminated by a NULL (specified as `'\0'`)
- The functionality of strings such as concatenation, copying, etc happens through standard library functions `<string.h>`
- C arrays (and therefore C strings) do NOT have automatic bound checking for indexes

(I’ll cover arrays later - for now just focus on “strings”)

Define a new char[ ]

```
char x[] = "abcdefg";
char x[8] = "abcdefg";
char x[8] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};
```

Print a char[ ]

Note the %s for “string” and the %c for character

```
printf("%s\n", x);
for (int i = 0; i < sizeof(x); i++) {
    printf("%c", x[i]);
}
printf("\n");
```

Read a string from standard input:

```
char x[32];
scanf("%31s", x);
printf("%s", x);
```

**Why is the char[ ] length 32?
and what is the %31s for?**

How to compare strings:

```
???
```

Compare two strings

Write a C program that:

- Asks the user to enter two words
- Determines which would come first in a dictionary

Characters

- What exactly *is* a char(acter) array?
- A character is (generally) a byte, or 8 bits, of general information
- Can be interpreted as a *number* or a *character* (ASCII)

Characters

- What exactly *is* a char(acter) array?
- A character is (generally) a byte, or 8 bits, of general information
- Can be interpreted as a *number* or a *character* (ASCII)

```
$ man ascii
```

01011010 == 90

and

01011010 == 'Z'

What will it print?

```c
#include <stdio.h>

int main() {
  char x[] = "Thessalonica";
  int y = x[2] + x[4];
  int z = x[5] + x[1];
  if (y > z) { printf("GREATER\n"); }
  else       { printf("LESS\n"); }
  return 0;
}
```

Read strings from standard input repeatedly

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  char buffer[32];
  while(scanf("%s", buffer) != EOF) {
    int i = 0;
    while (buffer[i] != '\0') { i += 1; }
    printf("INPUT length %d WAS: %s\n", i, buffer);
  }
  return 0;
}
```

After a few runs, CTRL-D to send EOF

What is this code going to do?

Alternative options for reading

Notice the 31

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  char buffer[32];
  while(scanf("%31s", buffer) != EOF) {
    int i = 0;
    while (buffer[i] != '\0') { i += 1; }
    printf("INPUT length %d WAS: %s\n", i, buffer);
  }
  return 0;
}
```

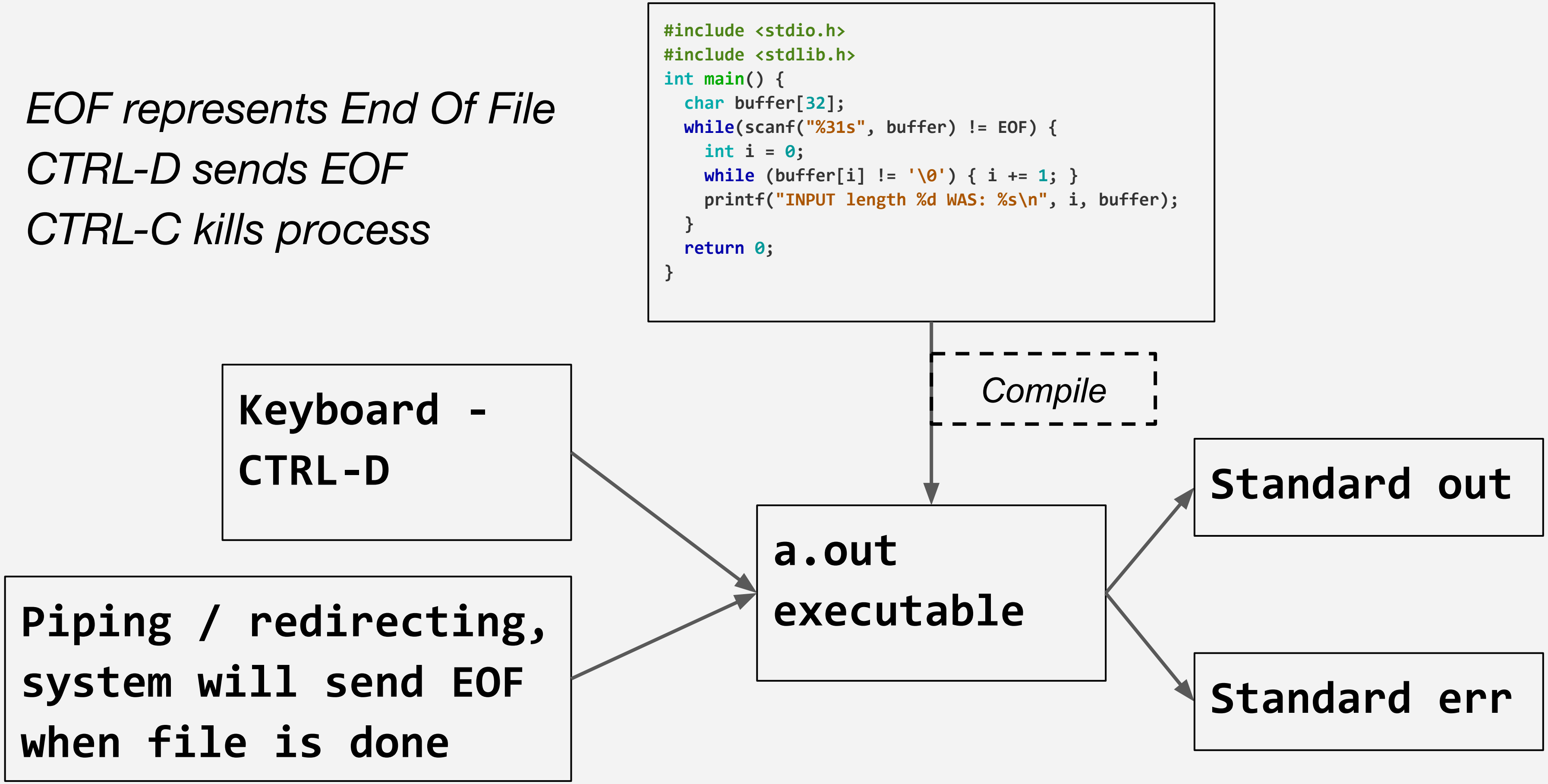
Alternative options for reading

```c
char buffer[32];
while(fgets(buffer, 31, stdin) != NULL) {
    printf("%s", buffer);
}
```

EOF represents End Of File

CTRL-D sends EOF

CTRL-C kills process



```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  char buffer[32];
  while(scanf("%31s", buffer) != EOF) {
    int i = 0;
    while (buffer[i] != '\0') { i += 1; }
    printf("INPUT length %d WAS: %s\n", i, buffer);
  }
  return 0;
}
```

Piping / redirecting, system will send EOF when file is done

Characters and char* literals

- Scanf returns -1 as EOF
- Can return other non-zero codes though too!
- How to tell if an error, or EOF?

Characters and char* literals

- C differentiates between a character and a string (char array) literal
- Single-quotes are used for chars
- Double-quotes for literals

Which of these are valid?

```
char words[] = "one small token";
char more_words[] = 'the large hill over there';
char letter_1 = "a";
char letter_2 = 'b';
```

Activity

Counting Cases

Write a C program that:

- Continuously reads in standard input until end / EOF
- Keeps a count of digits, lower-case, and upper-case letters
- Reports the total count
- (Ignore special symbols, spaces, etc)

For reference

```
char buffer[32];
while(fgets(buffer, 31, stdin) != NULL) {
    printf("%s", buffer);
}
```

Test your program thoroughly

- Ensure your output matches exactly what spec says
- Test with MORE test cases that what the spec says
- Handle edge cases (if applicable)
- Use **`sbt.py`**

Basic Make

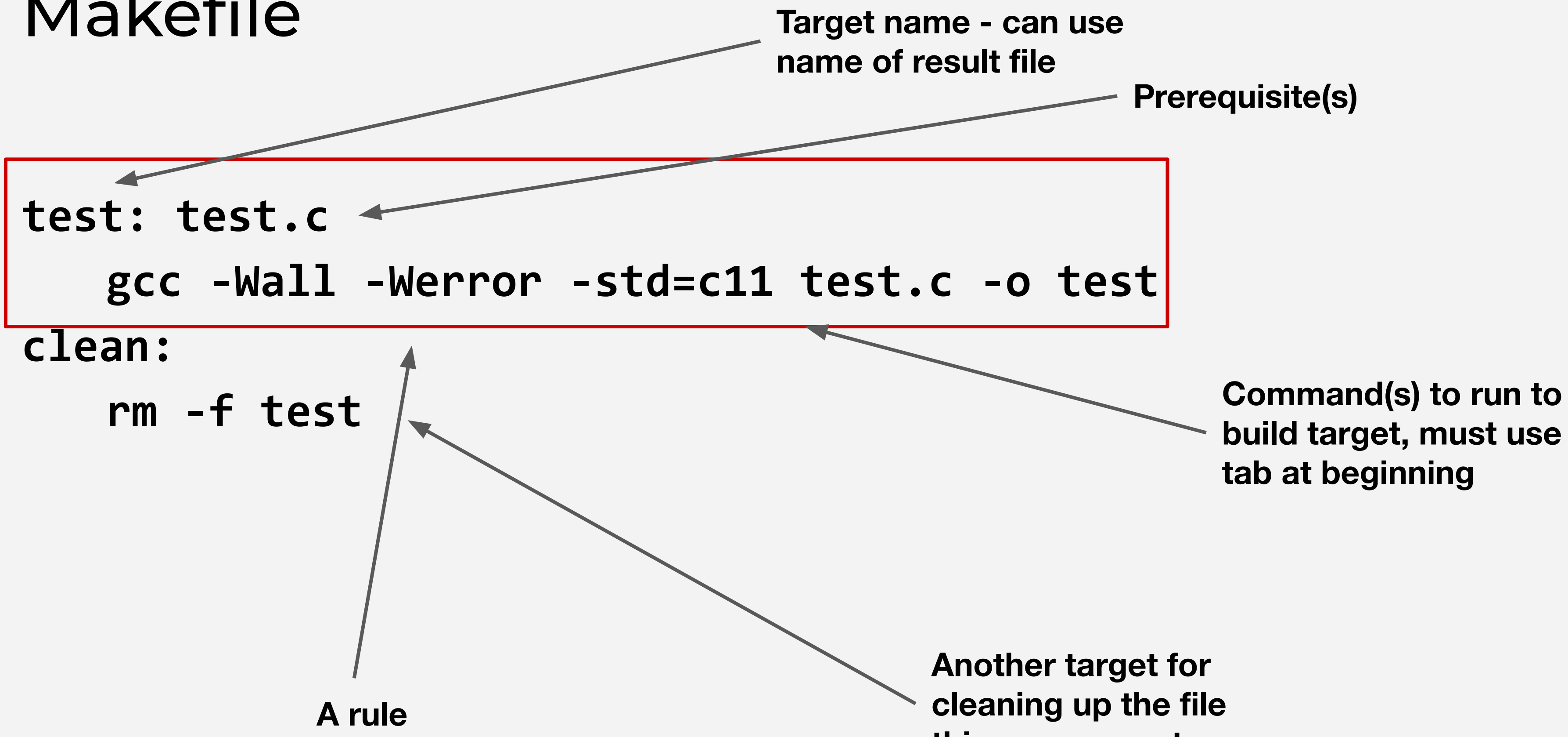
Make is a unix tool (available on lectura) that can be used to manage the compiling / building of programs

For Now, very basic overview of how it works, just so that you can use it to save you a bit of time :)

Makefile

```
test: test.c
    gcc -Wall -Werror -std=c11 test.c -o test
clean:
    rm -f test
```

Makefile



```
test: test.c
	gcc -Wall -Werror -std=c11 test.c -o test
clean:
	rm -f test
```

Target name - can use name of result file
Prerequisite(s)
Command(s) to run to build target, must use tab at beginning
A rule
Another target for cleaning up the file this can generate

Running Make

```
$ ls
makefile  test.c
$ make
gcc -Wall -Werror -std=c11 test.c -o test
$ ls
makefile  test   test.c
$ make clean
rm -f test
$
```