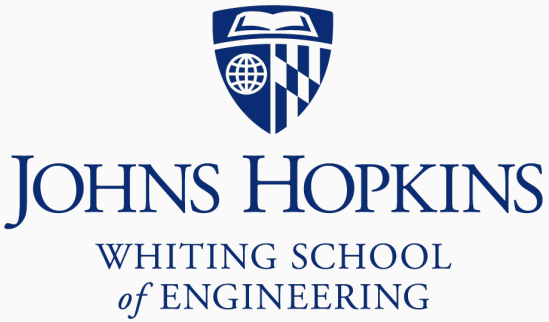
Reading input

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Reading input

`printf` writes formatted output strings

`scanf` reads formatted *input* strings

Both use "format tags" we've already seen

- `%d`: `int`
- `%u`: `unsigned int`
- `%f`: `float` or `double`
- `%.3f`: `float` / `double`, up to 3 decimal places
- `%e`: `float` / `double`, scientific notation

scanf

```c
#include <stdio.h>
int main() {
    int a = 0;
    float b = 0;
    // read an integer, and a floating-point
    // number from stdin, separated by space
    scanf("%d%f", &a, &b);
    printf("Read in a=%d, b=%.1f\n", a, b);
    return 0;
}
```

```
$ gcc formatted_io_eg1.c -std=c99 -pedantic -Wall -Wextra
$ echo "7 99.9" | ./a.out
Read in a=7, b=99.9
```

scanf

`scanf("%d%f", &a, &b)`: read an integer and a floating-point number separated by whitespace; store integer in `a` and floating-point in `b`

"Whitespace" means spaces, tabs & newlines. `scanf` allows any whitespace characters to separate items.

We write `&a, &b` instead of `a, b` because `scanf` *modifies* them

- `&` means "address-of"; more on this when we discuss pointers

scanf

What does echo "7 99.9" | ./a.out do?

```
$ gcc formatted_io_eg1.c -std=c99 -pedantic -Wall -Wextra
$ echo "7 99.9" | ./a.out
Read in a=7, b=99.9
```

Prints “7 99.9” and runs ./a.out, sending “7 99.9” to its “standard in” file

scanf

scanf returns number of items successfully parsed

It's good practice to *check this*; if it's not what you expect, something's wrong:

```c
// if result is not 2, something unexpected happened
if(scanf("%d%f", &a, &b) != 2) {
    puts("Parsing error");
    return 1; // non-zero return value
}
```

scanf

Reading a string with `scanf("%s", array)` is *unsafe* because the string may be longer than the array

```c
#include <stdio.h>

int main() {
    // I *think* the string will be no longer than
    // 10 chars (incl. terminator)
    char str[10];
    int nread = scanf("%s", str);
    if(nread != 1) {
        puts("Parsing error");
        return 1;
    }
    return 0;
}
```

scanf

This is fine:

```
$ gcc scanf_str.c -std=c99 -pedantic -Wall -Wextra
$ echo "hello" | ./a.out
```

This overflows the buffer:

```
$ echo "hellohellohello" | ./a.out
```

Unpredictable things can happen when you overflow a buffer

- On my computer, it crashes
- Can also result in security vulnerabilities

scanf

Instead, include a number between % and s to put a limit on how many characters to read

```c
#include <stdio.h>

int main() {
    char str[10];
    int nread = scanf("%9s", str);
    //                 ^^^
    if(nread != 1) {
        puts("Parsing error");
        return 1;
    }
    printf("I read: %s\n", str);
    return 0;
}
```

scanf

```
$ gcc scanf_str2.c -std=c99 -pedantic -Wall -Wextra
$ echo "hellohellohello" | ./a.out
I read: hellohell
```

"%9s" means "read the string up to the ninth character"

We use "%9s" instead of "%10s" because the number doesn't count the null terminator, which scanf also writes

scanf

More scanf details:

en.wikipedia.org/wiki/Scanf_format_string

Character-by-character

putchar writes a single character to standard output

getchar reads a single character from standard input

Character-by-character

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

int main() {
    // read character-by-character until we get newline
    int c = 0;
    while((c = getchar()) != '\n') {
        // write uppercase version of character
        putchar(toupper(c));
    }
    return 0;
}
```

```
$ gcc cap_chars.c -std=c99 -pedantic -Wall -Wextra
$ echo "hello" | ./a.out
HELLO
```

Line-by-line

We'll defer discussion of line-by-line input until we discuss reading and writing from files