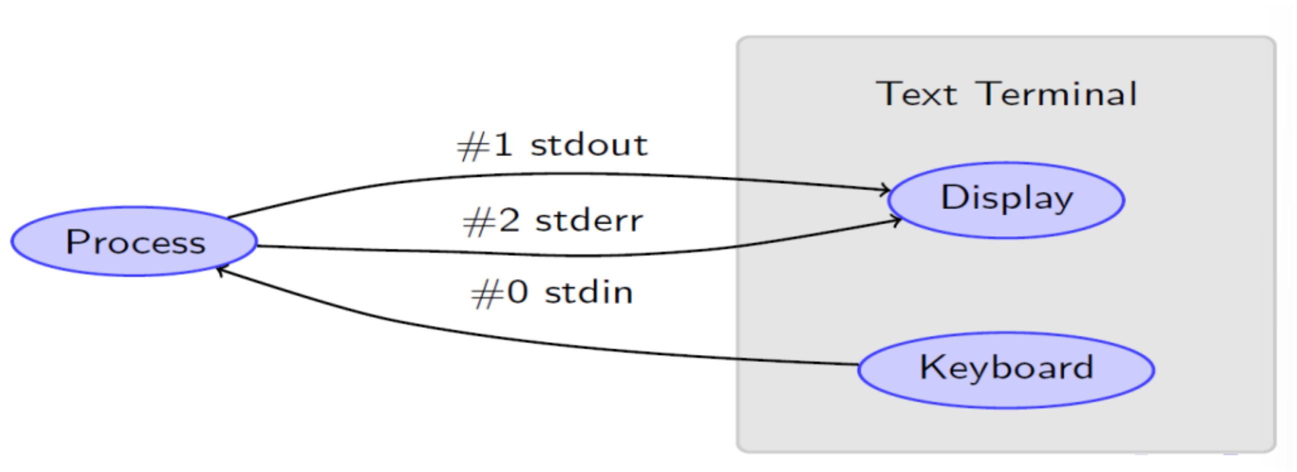
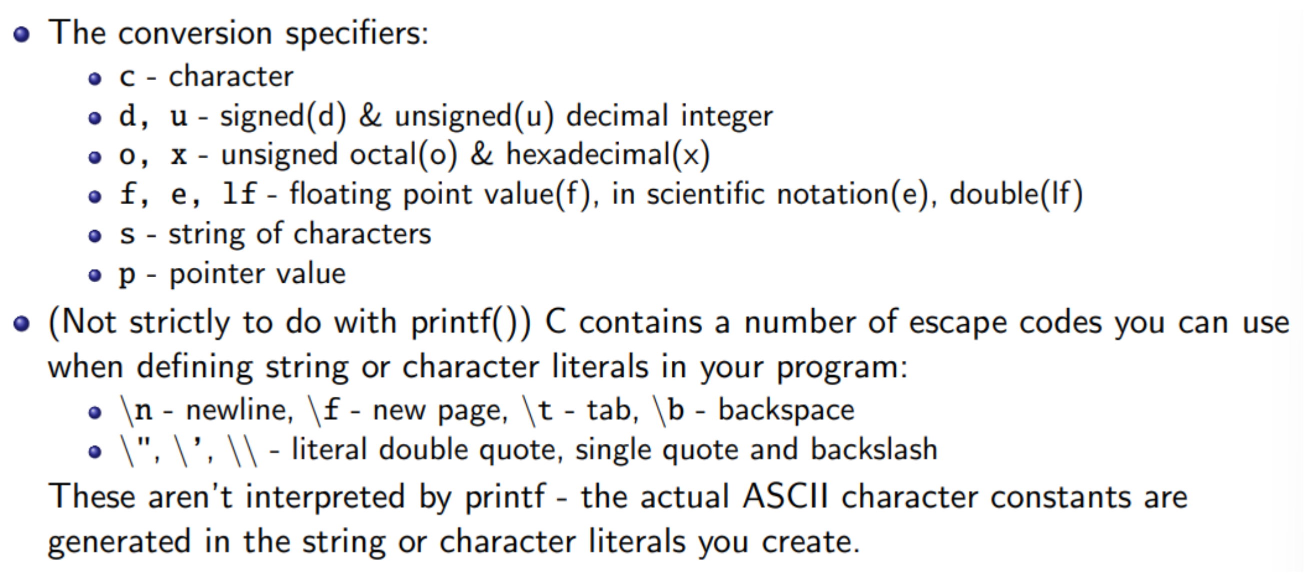
# Simple Input and Output

C programs have 3 file-handles pre-opened from which data can be read or written:

- Standard input ( `stdin` ) – usually the keyboard
- Standard output ( `stdout` ) – usually the terminal
- Standard error ( `stderr` ) – used for error or diagnostic information



## C Output ( `printf` )



- The conversion specifiers:
  - c - character
  - d, u - signed(d) & unsigned(u) decimal integer
  - o, x - unsigned octal(o) & hexadecimal(x)
  - f, e, lf - floating point value(f), in scientific notation(e), double(lf)
  - s - string of characters
  - p - pointer value
- (Not strictly to do with printf()) C contains a number of escape codes you can use when defining string or character literals in your program:
  - \n - newline, \f - new page, \t - tab, \b - backspace
  - \", \', \\ - literal double quote, single quote and backslash

  These aren't interpreted by printf - the actual ASCII character constants are generated in the string or character literals you create.

Examples:

```
#include <stdio.h>
int main(void) {
        int i = 72;
        printf("An integer: %d\n", i);
        double d = 72.1;
        float f = 3.1;
        printf("A double: %lf, a float: %f, natural format: %g,
%g\n", d, f, d, f);
        char s[] = "The quick brown fox...";
        printf("A string: %s\n", s);
        char x = 'A', y = 'B', z = '\n';
        printf("Two characters and a newline: %c%c%c", x, y, z);
}
```

## Security of `printf`

If you have some string `str` supplied by an untrusted source, never do `printf(str)`

This allows attackers to supply conversion specifiers to `printf` and gives them the means to attack your program

Instead use `printf("%s", str)`

## Simple I/O Functions

Beyond `printf()`, the C standard library header file `stdio.h` provides the file descriptors `stdin`, `stdout`, `stderr` and a set of functions to read/write from them

Some functions require you supply a file descriptor, others use one of the standard ones implicitly

Here are some of the simpler functions:

### `getchar()`

Retrieves a single character from standard input

```
#include <stdio.h>
int main(void) {
        int c;
        c = getchar();
        printf("You typed the character %c\n", c);
        return 0;
}
```

Suppose we want to read a character, then skip all characters up to and including the next newline. We can do:

```
int c = getchar();
while(getchar() != ’\n’) /*EMPTY BODY*/;
```

`getchar()` can also return an error value called EOF which indicates there’s no more input

On many platforms EOF is defined as -1, this is why `getchar()` returns an int not a char

A natural companion to `getchar()` is

## `putchar()`

Writes a single character to standard output
For instance, you can copy all data from `stdin` to `stdout` by:

```
int c;
while( (c = getchar()) != EOF ) {
        putchar(c);
}
```

## `scanf()`

Used to read data

The following code shows how to read an integer from `stdin`:

```
int i;
int ret = scanf("%d", &i);
if( ret == 1 ) {
        // ok, use i
} else {
        // scanf failed to read an int
}
```

We’re passing `&i` into `scanf()` to allow `scanf` to store the integer value that is read from stdin into `i`

`scanf` returns the number of items matched (or values less than zero to indicate error conditions)

We can also use `scanf()` to read a string from standard input (until the next whitespace character or newline)

Since arrays basally decay to pointers, **we don’t need to use the & operator**

For Example:

```
char buffer[50];
int ret = scanf("%s", buffer);
if( ret == 1 ) {
        // ok, deal with buffer
} else {
        // scanf failed to read a line
}
```

This is **bad!**

We don't know how long the string might be, so it’s possible to overrun the end of the supplied buffer

In this case, there are safer functions to input strings, so **don't use `scanf()` with the "%s" modifier**

## The limitations of `scanf()`

Distinguish safe uses of `scanf` from unsafe ones, and use safer alternatives to the latter

One alternative is to read each line (taking care not to overrun your buffer) and then use string manipulation routines to split the line into pieces and check whether the sequence of pieces make sense

The C library function `sscanf()` may then prove useful - it's a version of `scanf()` that reads from a string (not a file)

## `fgets()`

`fgets()` reads a string from an input stream (e.g.: `stdin`) into a buffer until the next newline, end of file, or the buffer is full. So it's really a function to read as much of the next input line as will fit into a string buffer

To read a line from stdin we can do the following:

```
#define BUFSIZE 100
char buf[BUFSIZE];
fgets(buf, BUFSIZE, stdin);
```

The BUFSIZE parameter is the maximum number of characters that `fgets` will write to the buffer. `fgets` will always terminate the string buffer with '`\0`'

If a newline is read, it is also stored into the buffer. Note that if the line of input is too long, the remainder of it (including the newline) remains unread on the input stream (stdin)

Note that fgets() can fail to read any input at all, and return NULL (read man fgets for the details). So we should check that the return value of `fgets()` is not NULL

## `fgets()` with newline elimination

A standard idiom is to call `fgets()` and then remove the final newline (if present):

```
if( fgets( buf, BUFSIZE, stdin ) != NULL ) {
        int len = strlen(buf);
        if( buf[len-1] == '\n' ) {
                buf[len-1] = '\0';
        }
}
```

This leads to an idiomatic for each line on `stdin` snippet:

```
while( fgets( buf, BUFSIZE, stdin ) != NULL ) {
        int len = strlen(buf);
        if( buf[len-1] == '\n' ) {
                buf[len-1] = '\0';
        }
        printf( "read line '%s'\n", buf );
}
```

Note that this, by not dealing with overlong lines, treats each BUFSIZE-1 char section of an overlong line as a separate line, which is rather weird.

In action, with BUFSIZE as 20:

```
./foreachline1
hello
read line 'hello'
abcdefghijklmnopqrstuvwxyz
read line 'abcdefghijklmnopqrs'
read line 'tuvwxyz'
```

Alternatively, you might want to discard the rest of an overlong input line:

```
while( fgets( buf, BUFSIZE, stdin ) != NULL ) {
        int len = strlen(buf);
        if( buf[len-1] != '\n' ) {
                while(getchar() != '\n') /*EMPTY BODY*/;
        } else {
                buf[len-1] = '\0'; len--;
        }
        printf( "read line '%s' length %d\n", buf, len );
}
```

In action, with BUFSIZE as 20:

```
./foreachline2
hello
read line ’hello’ length 5 abcdefghijklmnopqrstuvwxyz
read line ’abcdefghijklmnopqrs’ length 19
```

## Printing to standard error

There’s a version of `printf` called `fprintf` used for printing to files. We can print to `stdout` and `stderr` as follows:

```
#include <stdio.h>
int main( void ) {
        fprintf(stdout, "Hello standard out!\n");
        fprintf(stderr, "Hello standard error!\n");
        return 0;
}
```

We can test this from any bash-like Unix shell by discarding one or the other:

```
$ gcc -Wall fprintf.c -o fprintf
$ ./fprintf 1>/dev/null
$ ./fprintf 2>/dev/null
```

### `fputc()`

Similarly, `fputc( ch, file )` is like `putchar()` to an open writeable file

So the following is a laborious way of printing "hi!" to `stderr`:
`fputc( ’h’, stderr ); fputc( ’i’, stderr ); fputc( ’!’, stderr );`