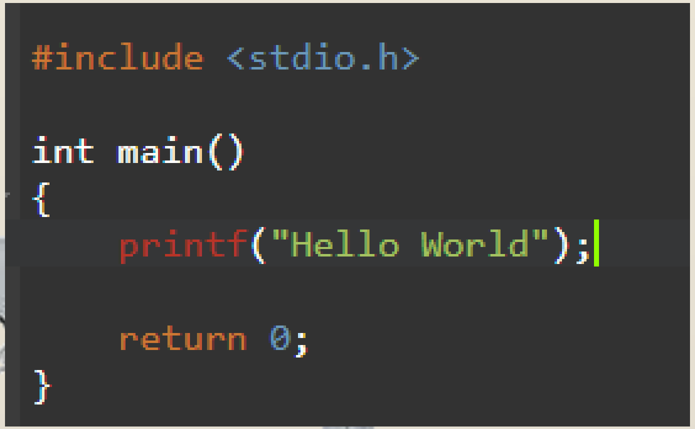


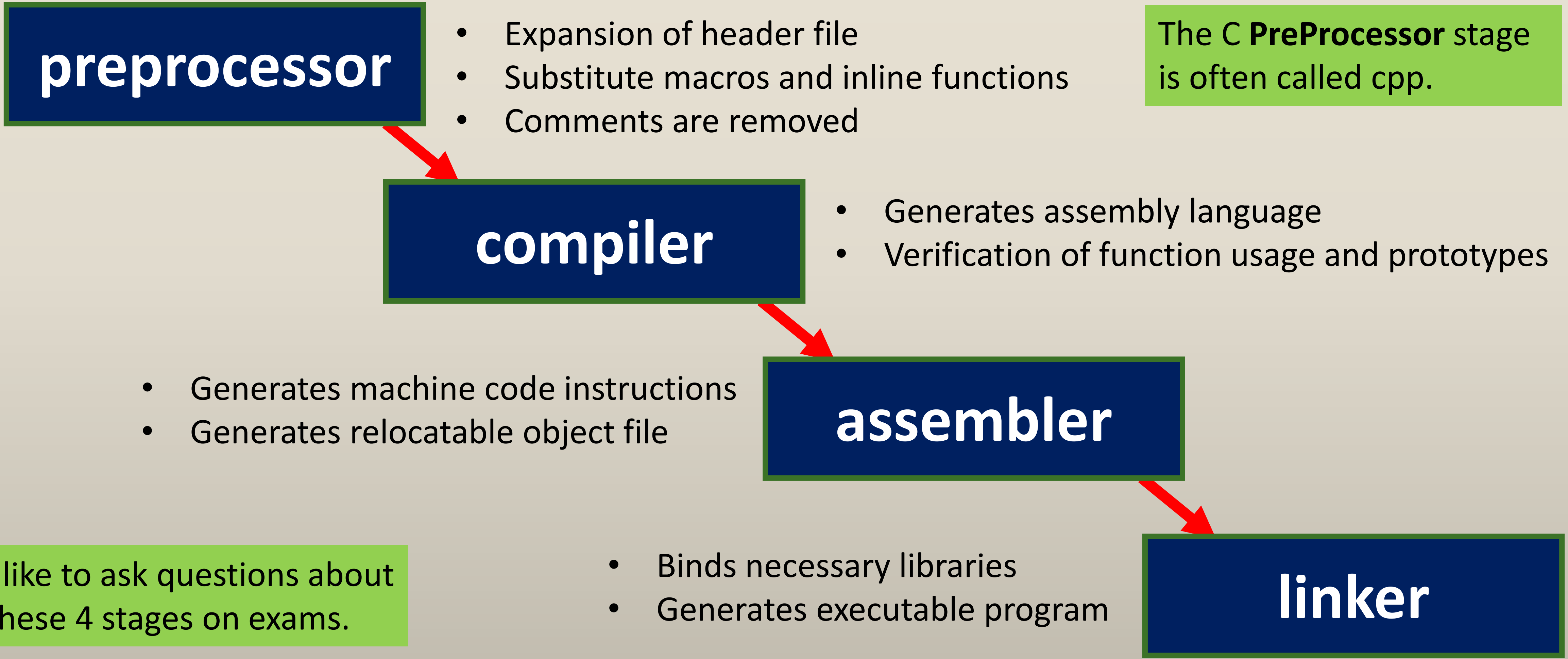
```
#include <stdio.h>

int main()
{
    printf("Hello World");

    return 0;
}
```

CS 333

Intro to Operating Systems
C Programming Language
The C-quel

`continue;`

The 4 Stages of Compilation of a C Program



preprocessor

- Expansion of header file
- Substitute macros and inline functions
- Comments are removed

The C **PreProcessor** stage is often called cpp.

compiler

- Generates assembly language
- Verification of function usage and prototypes

assembler

- Generates machine code instructions
- Generates relocatable object file

linker

- Binds necessary libraries
- Generates executable program

I like to ask questions about these 4 stages on exams.

The C Preprocessor – cpp

- Lines starting with a **#** character are interpreted by the C preprocessor as **preprocessor directives**.
- The C preprocessor is so much more than just header file inclusion.
- It performs macro expansion and conditional compilation.

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

If your use of cpp is limited to this, you are really missing out on a powerful feature of the C compiler.

About Those Include Files

Notice the `.h` on the end of the `#include` preprocessor directive.

```
#include <stdio.h>

#include "my_include.h"

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

System level include files **ALWAYS PRECED** your include files and are surrounded by less-than/greater-than symbols.

Your include files (the ones you create for your development) **ALWAYS FOLLOW** the system include files and are surrounded by double quotes.

String Literal Concatenation

A minor function of the preprocessor is joining string literals together, "string literal concatenation" – automatically turning code like

```
printf("A long string " "with a longer string " "\n");
```

String literals can be separated by spaces.

Into

```
printf("A long string with a longer string \n");
```

Conditional Compilation

The use of `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` can make editing and debugging your code much easier.

```
#ifdef DEBUG
  fprint(stderr, "debug message\n");
#endif // DEBUG


#if DEBUG >= 2
  fprint(stderr, "debug message 2\n");
#endif // DEBUG > 2
```

```
#ifdef DEBUG
  fprint(stderr, "debug message\n");
#else // not DEBUG
  fprint(stderr, "happy message\n");
#endif // DEBUG
```

Use of these can really make your life a lot better. I highly recommend them.

Simple Macros

The use of simple macros in C can easily be compared to the use of `const`, and is in some ways not as good as using const.

However, in this class, we’ll be using a lot of macros.

```
#define MY_NAME "R. Jesse Chaney"

printf("My name is %s\n", "R. Jesse Chaney" );

printf("My name is %s\n", MY_NAME );
```

If you see this, **DON’T do this!**

Do the right/smart/proper/cool thing.
It’s a macro, use it like one.

Almost Simple Macros

Here is one of my favorite uses of macros.

```
#ifdef NOISY_DEBUG
# define NOISY_DEBUG_PRINT fprintf(stderr, "%s %s %d\n" \
      , __FILE__, __func__, __LINE__ )
#else // not NOISY_DEBUG
# define NOISY_DEBUG_PRINT
#endif // NOISY_DEBUG
```

The line continuation character for macros.

Special macros defined by cpp or the compiler.

Define the macro to be nothing if NOISY_DEBUG is not defined.

I can sprinkle the macro `NOISY_DEBUG_PRINT` generously in my code. Any time I `#define NOISY_DEBUG`, I will see a trace of line numbers generated to `stderr`.

If I don’t `#define NOISY_DEBUG`, no debugging messages are generated.

Send all *diagnostic* messages to `stderr`, not `stdout`.

Send all *diagnostic* messages to `stderr`, not `stdout`.

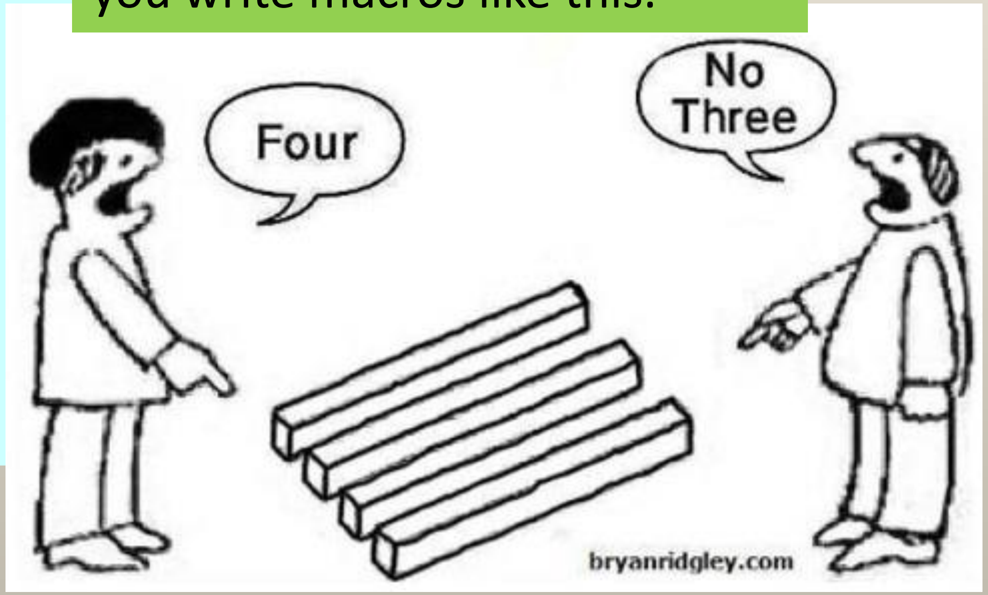
We will describe stderr in just a moment…

Macros Like Functions

It is possible to define macros that **take arguments**. They will look like function calls when used. Here are a couple simple common ones.

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

int i1 = -7, i2 = 10;
float f1 = -7.0, f2 = 10.0;

i1 = ABSOLUTE_VALUE(i1);
f1 = ABSOLUTE_VALUE(f1);
i1 = MIN(i1,i2);
f1 = MIN(f1,f2);
```

Use LOTS of parentheses when you write macros like this.



The assert() Macro

```
#include <assert.h>

void assert(scalar expression);
```

If expression is false (compares equal to zero), assert() prints an error message to standard error and terminates the program by calling abort(3).

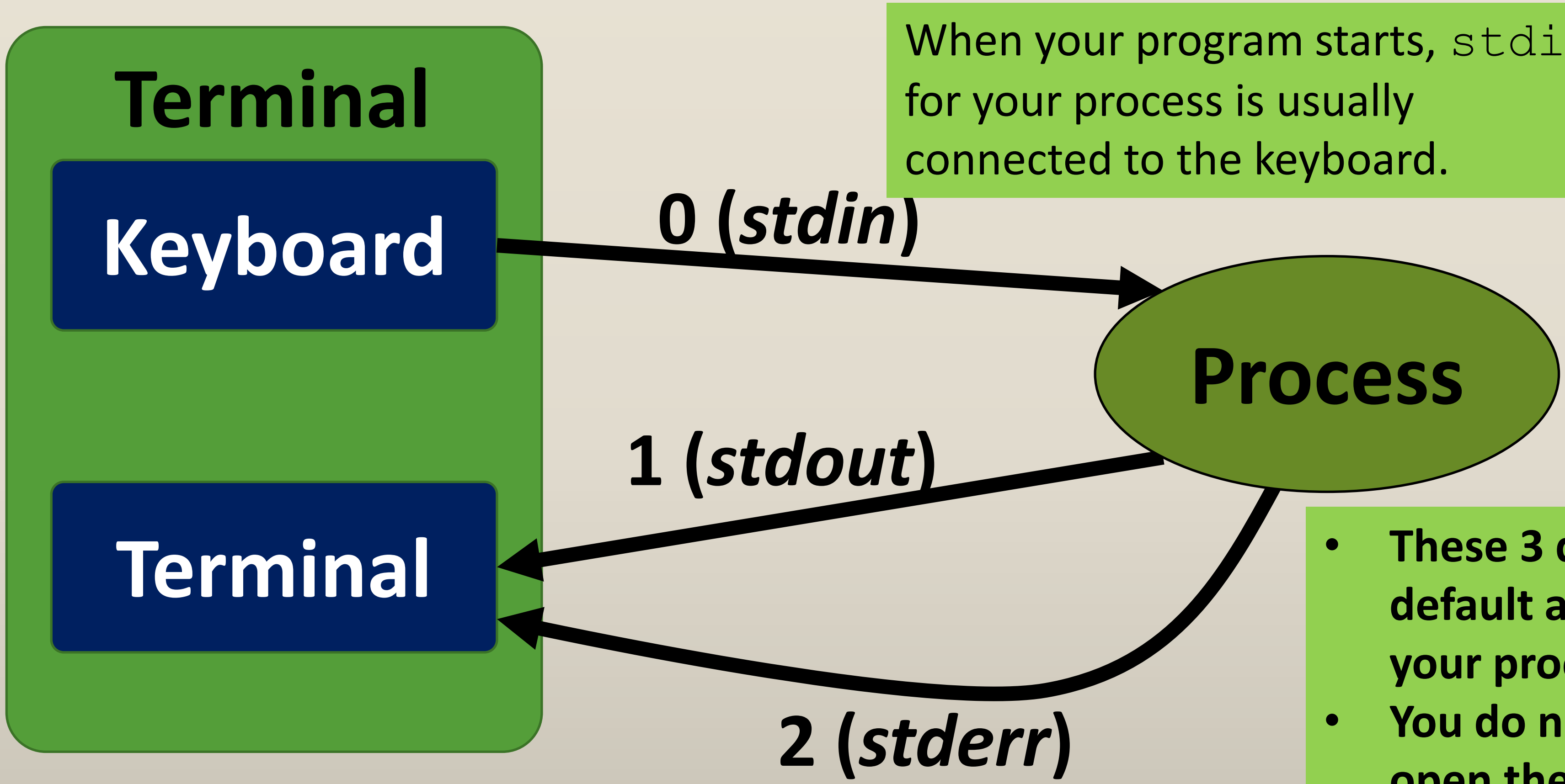
- The assert macro provides a convenient way to abort the program while printing a message about where in the program the error was detected.
- You can disable the error checks performed by the assert() macro by recompiling with the macro NDEBUG defined.

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

int test_assert( int x ) {
    assert( x <= 4 );
    return x;
}

int main( ) {
    int i;

    for ( i = 0 ; i <= 9 ; i++ ) {
        test_assert( i );
        printf( "i = %d\n", i );
    }
    return 0;
}
```

Use of the `assert()` macro. If the value of x is less than or equal to 4, nothing happens.

If the value of x is greater than 4, a message will be printed and the program will abort.

Standard I/O



When your program starts, `stdin` for your process is usually connected to the keyboard.

When your program starts, `stdout` and `stderr` are usually connected to the terminal display.

- **These 3 devices are by default already open when your process begins.**
- **You do not have to explicitly open them.**
- **You also do not have to close them before your process terminates.**

The C `stdio.h` Header File

- In order to use the C input/output functions, you must include the `stdio.h` file.

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

- Do notice that you must include the .h portion of the header file name in the `#include` statement.

- **The C Book** has a nice description of this header file with some useful information about the general I/O mode in C.

  http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html

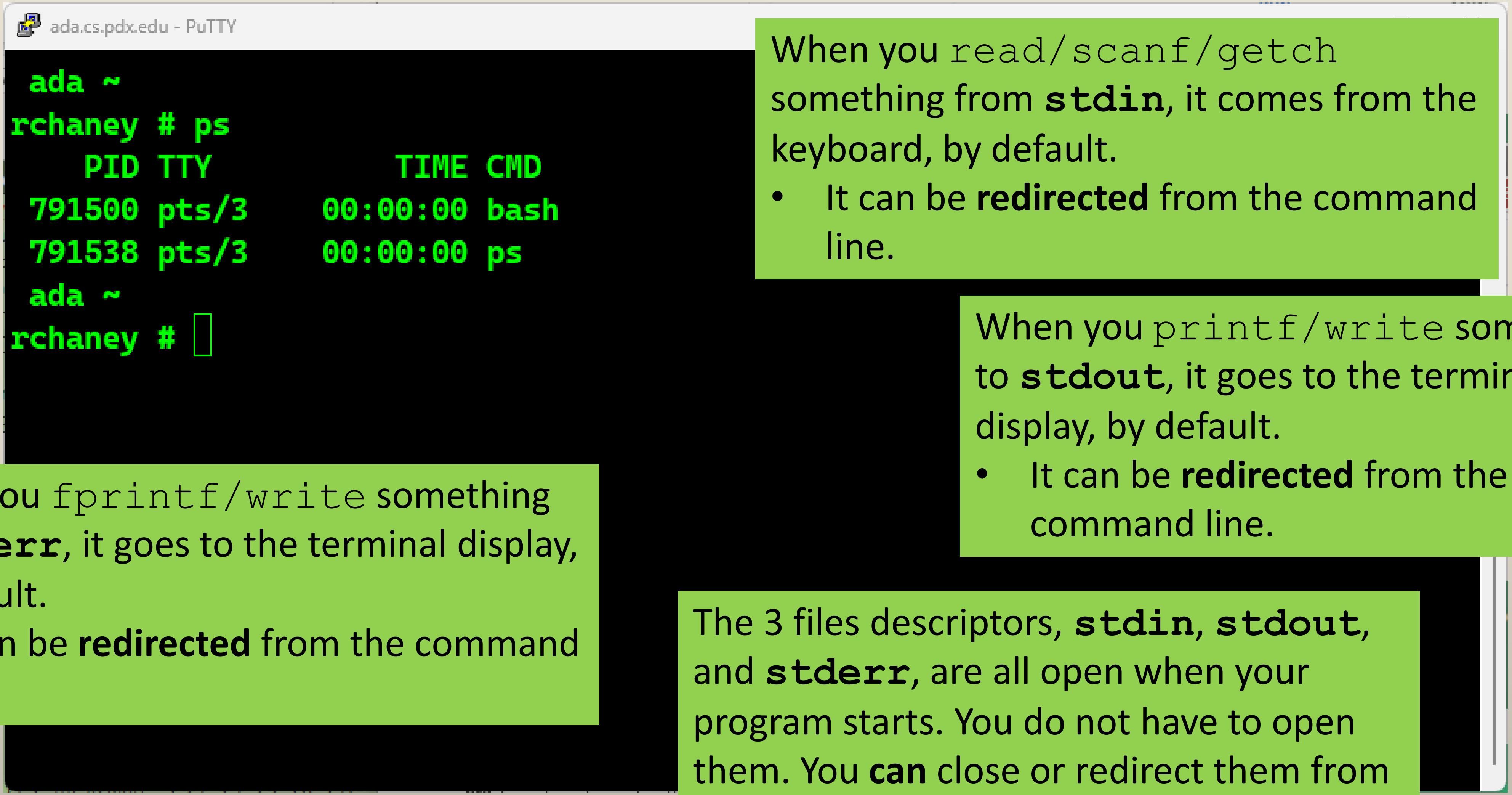
Standard I/O

FD	Macro	Stream	Device
0	STDIN_FILENO	*stdin*	keyboard
1	STDOUT_FILENO	*stdout*	terminal
2	STDERR_FILENO	*stderr*	terminal

We'll talk about these macros throughout the term.

FD stands for file descriptor.

Standard I/O



```
 ada ~
rchaney # ps
    PID TTY          TIME CMD
 791500 pts/3    00:00:00 bash
 791538 pts/3    00:00:00 ps
 ada ~
rchaney # 
```

When you `read/scanf/getch` something from **`stdin`**, it comes from the keyboard, by default.

- It can be **redirected** from the command line.

When you `printf/write` something to **`stdout`**, it goes to the terminal display, by default.

- It can be **redirected** from the command line.

When you `fprintf/write` something to **`stderr`**, it goes to the terminal display, by default.

- It can be **redirected** from the command line.

The 3 files descriptors, **`stdin`**, **`stdout`**, and **`stderr`**, are all open when your program starts. You do not have to open them. You **can** close or redirect them from within your program.

The C `printf()` Function

- A common statement you'll use to send text to the **terminal window** will be the `printf()` statement.
- The `printf()` statement **uses the already open `stdout`** file stream.
- While it is possible for `stdout` to not be open when your program starts, it is unlikely and takes a lot of effort (and reason).

- For your programs, you can assume `stdout` is open at the beginning of the program.
- It is also possible to **redirect** `stdout` to something other than the terminal display. We will cover this.

The C `printf()` Function

Syntax:

```
printf(<format_string>, <arg_list>);
```

- The `printf` function **requires a formatting** string for all variables you pass to the function.
- If formatting string contains any format specifiers an argument list must be supplied.
- There must be a matching argument for every format specifier in the format string.

The C `printf()` Function

`printf` function – for printing formatted output to the screen

- **`printf( "Hello World" );`**
- Supports use of **format specifiers** to determine the type of the data print.

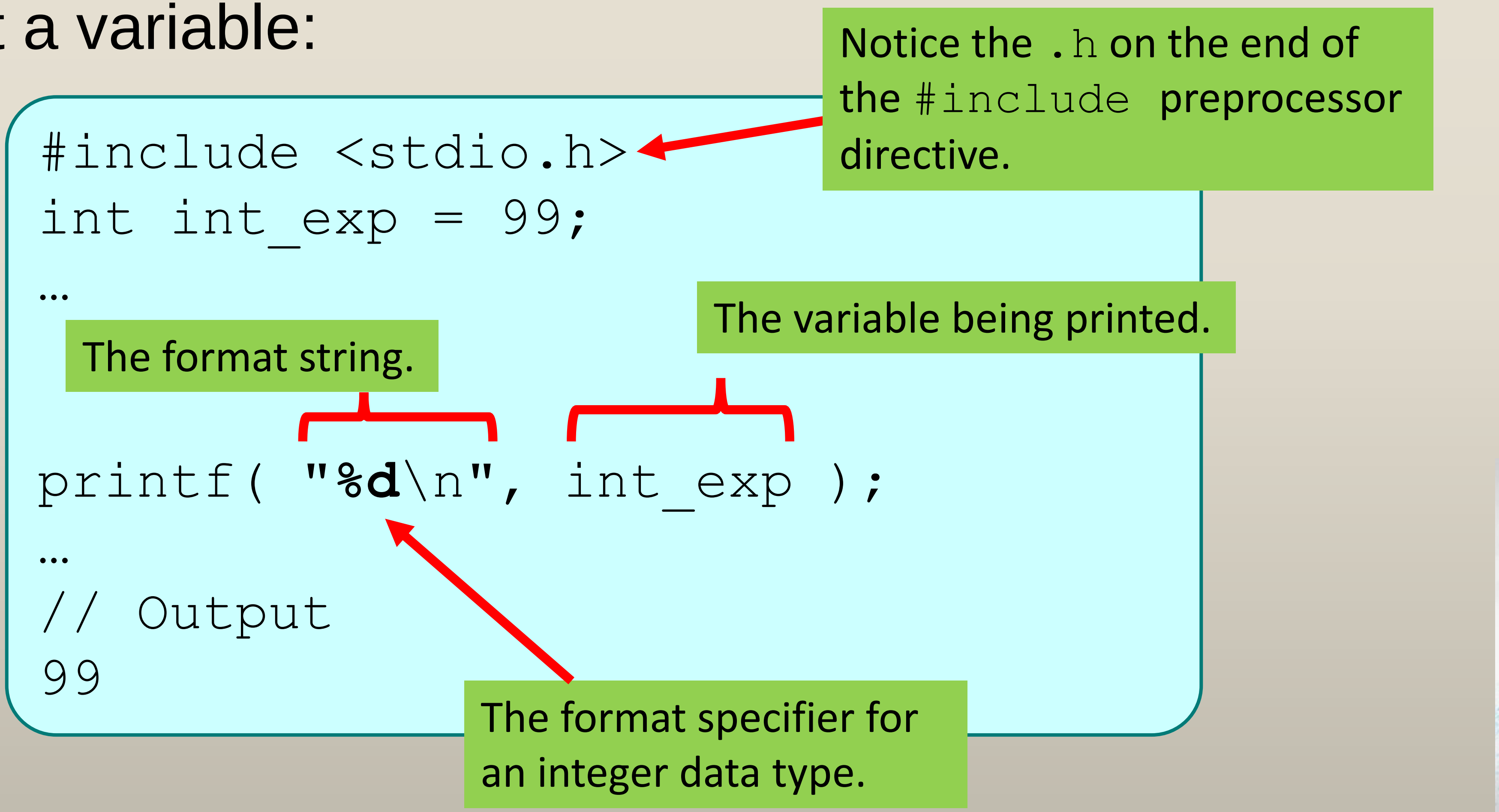
Data Type	Format Specifier	Example
`int`	**`%d or %i`**	123
`int`	`%o`	unsigned octal value
`int`	`%x` or `%X`	unsigned hex value
`float`	`%f`	3.1400
`double`	`%lf`	12.4567878
`char`	`%c`	A
string	`%s`	Hello
pointer	`%p`	0x12345678

You **WANT** to remember these!!!

The C printf() Function

Requires the header file <stdio.h>

Print a variable:



```
#include <stdio.h>
int int_exp = 99;
…

printf( "%d\n", int_exp );
…
// Output
99
```

Notice the .h on the end of the #include preprocessor directive.

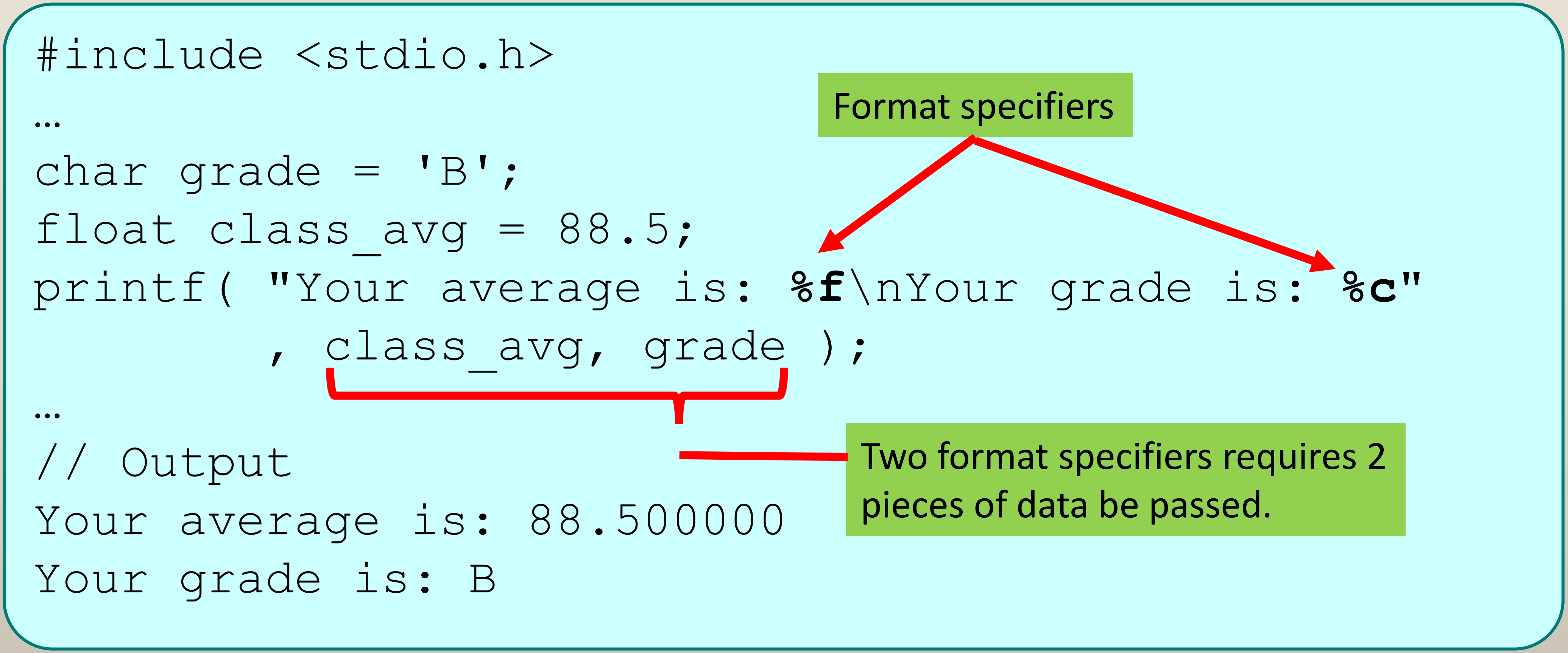
The format string.

The variable being printed.

The format specifier for an integer data type.

The C `printf()` Function

Printing multiple pieces of data:



```
#include <stdio.h>
…
char grade = 'B';
float class_avg = 88.5;
printf( "Your average is: %f\nYour grade is: %c"
        , class_avg, grade );
…
// Output
Your average is: 88.500000
Your grade is: B
```

Format specifiers

Two format specifiers requires 2 pieces of data be passed.

Formatted Output with `printf()`

The general form for a format specifier is:

`%<flags><field width><precision><length>conversion`

Examples:

- `%5d` Print an integer with 5 total spaces, right justified
- `%6.2f` Print a decimal with 6 total places, including the decimal, with 2 places to the right of the decimal point.
- `%-25s` Print the string **left** justified with a total width of 25

Formatted Output with `printf()`

```c
#include <stdio.h> // Needed for printf
int main( void )
{   int int_exp = 123;
    float float_exp = 98.7653F;
    // Print an integer with 5 total spaces, right aligned
    printf( "%5d\n", int_exp );
    // Print a decimal with 6 total places including the decimal
    // with two places to the right
    printf( "%6.2f\n", float_exp );

    // Print the string literal left justified with a total width of 25
    printf( "%-25s\n", "Calvin" );
    // Print with 4 total spaces and two to the right of the decimal point
    printf( "%4.2f\n", .346 );
}
// Output
  123
 98.77
Calvin
0.35
```

The C `scanf()` Function

- `scanf()` – **reads from the keyboard** (or what ever is connected to `stdin`)
- Uses format specifiers previously discussed.
- Formatting string should **only** contain format specifiers and spaces.
- Each argument, **except for strings**, must be prefixed with an ampersand (&).
- The ampersand used in this context is called the **“address of”** operator.

The C scanf() Function

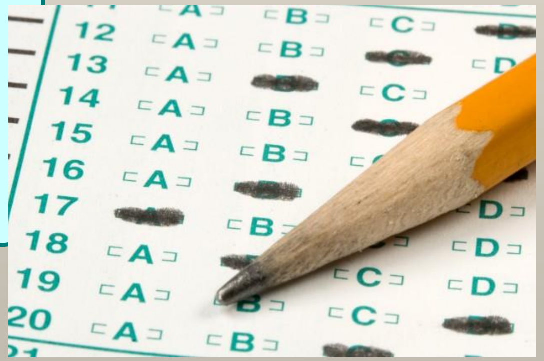
- Reading a value from the keyboard

```
int score = 0;

scanf( "%d", &score ); // Don't forget the &
```

- Reading multiple values from the keyboard

```
int score1 = 0, score2 = 0;

printf( "Enter two scores: " );
scanf( "%d %d", &score1, &score2 );   // Place a space
                                      // between each
                                      // specifier
```

2 format specifiers requires 2 pieces of data be passed. Notice that each has an & in front of the variable name.



The C `fopen()` Function

- Sometimes you need to read from or write to more than just the keyboard or terminal. **You need to read and write files**.
- Then, you need to use the `fopen()` call to open the file, before you use it.
- The `fopen()` call returns a **`FILE*`** type.
- If `fopen()` fails, it returns a `NULL` pointer.

```
#include <stdio.h>
FILE *fopen(const char *pathname
    , const char *mode);
```

The name of the file to be opened, **as a string**.

Notice that mode is a `char *`, not just a `char`.

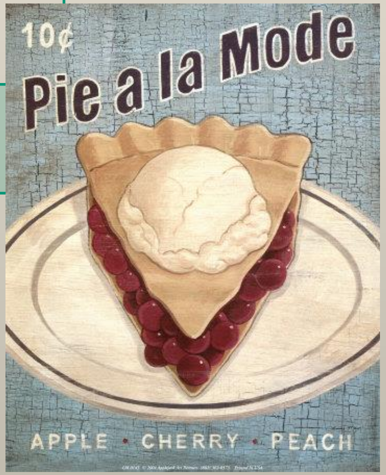
The C `fopen()` Function Modes

- The most common modes you'll use for the call to `fopen()` are listed in the table below.
- There are many other modes, especially for handling **binary files**.

Mode	Type of File	Read	Write	Create	Truncate
`"r"`	text	Yes	No	No	No
`"w"`	text	No	Yes	Yes	Yes
`"a"`	text	No	Yes	Yes	No

Notice the double quotes, NOT single quotes.

For a file opened in append mode, **all** writes will occur at the end of the file, regardless of attempts to move the file position indicator with `fseek()`.



The C `fclose()` Function

- Of course, once you are done with a file (reading, writing, or both), you'll need to close it.
- Closing a file frees up important resources within the kernel.
- The kernel has a limited number of open files it can manage.
- Your process also has a limited number of open files it is allowed to have at any one time.
- **Don't pass a `NULL` pointer to `fclose()`.**

```
#include <stdio.h>
int fclose(FILE *stream);
```

The kind of thing returned from `fopen()`.

The C `fopen()` Function

```
#include <stdio.h>

int main( void )
{
    FILE *fp;
    char file_name[] = "file does not exist";

    fp = fopen(file_name, "r");
    if ( NULL == fp )
    {
        // file failed to open
    }
    else
    {
        // process file
        fclose( fp );
    }
}
```

Needed to access the `fopen()` and `fclose()` calls.

If `fopen()` fails, it returns a `NULL` pointer value.

Close the file when you are done with it.

File I/O on Opened Files

- Once you've (successfully) opened the file, you'll need to perform operations on it.
- You can use the **f**`printf()` and **f**`scanf()` calls.
- The **f**`printf()` and **f**`scanf()` calls behave exactly like the `printf()` and `scanf()` calls, except that the first parameter to **f**`printf()` and **f**`scanf()` is the open `FILE *` returned from a (successful) call to `fopen()`.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

The kind of thing returned from `fopen()`.

```
… … …

    FILE *fp = NULL;
    int a, b, c;
    char str[50];
    fp = fopen(file_name, "r");
    if ( NULL == fp )
    {
       // file failed to open
    }
    else
    {
       fscanf(fp, "%i %i %i %s", &a, &b, &c, str );
       fclose( fp );
    }

… … …
```

Notice how the & is required for non-stringy variables and is not for the character array (aka string).

```
… … …

    FILE *fp = NULL;
    int a = 1, b = 2, c = 3;
    fp = fopen(file_name, "w");
    if ( fp == NULL )
    {
        // file failed to open
    }
    else
    {
        fprintf(fp, "%06i %-7i %10i", a, b, c );
        fclose( fp );
    }

… … …
```

What does the zero in front of the width specifier do?

http://www.cplusplus.com/reference/cstdio/printf/

Other Functions for I/O

In addition to the `printf`, `fprintf`, `scanf`, and `fscanf` functions, there are a couple additional I/O functions that work with **strings**. These are useful when you know you will work with an entire line from the file (or keyboard). Lines from a file are delimited by a newline (`\n`) for `fgets()`.

```
fgets()        // get an entire line from a file
fputs()        // write the string to stream
```

The `fgets()` function can read from `stdin` and `fputs()` can write to `stdout`.

OTHER

Let’s Take Exceptions

A C++/Java/Python feature that you may miss while programming in C are exceptions.

Exceptions are a terrific language construct in C++/Java/Python to **manage anomalous conditions**.

In C, you will need to **check the return value** of the functions you call and determine if an error occurred and how to manage it.

The `perror()` Function

```
#include <stdio.h>

void perror(const char *s);
```

The perror() function is very useful.
Get used to using it.

The `perror()` function produces a message to **standard error describing the last error encountered during a call to a system or library function**.

```
…
perror( "Cannot open file " FILE_NAME );
```

Redirection of `stdin/stdout/stderr`

The shell (`bash` or other) and many UNIX commands take their input from standard input (`stdin`), write output to standard output (`stdout`), and write error output to standard error (`stderr`).

- By default, standard input is connected to the terminal keyboard and standard output and error to the terminal display.
- **The way of indicating an end-of-file on the standard input, a terminal, is usually `<Ctrl-d>`.**
- I've mentioned that you can redirect `stdin/stdout/stderr`.

Basic Redirection Operators

You will see these a LOT during the term.

Character	Action
>	Redirect standard output
2>	Redirect standard error
2>&1	Redirect standard error to standard output
<	Redirect standard input
|	Pipe standard output to another command
>>	Append to standard output

You use these on the command line in the shell.

Some simple examples:

```
$ who > names
```

- Redirect standard output from the `who` command to a file named `names`. All `printf()` calls will automatically go into the `names` file.

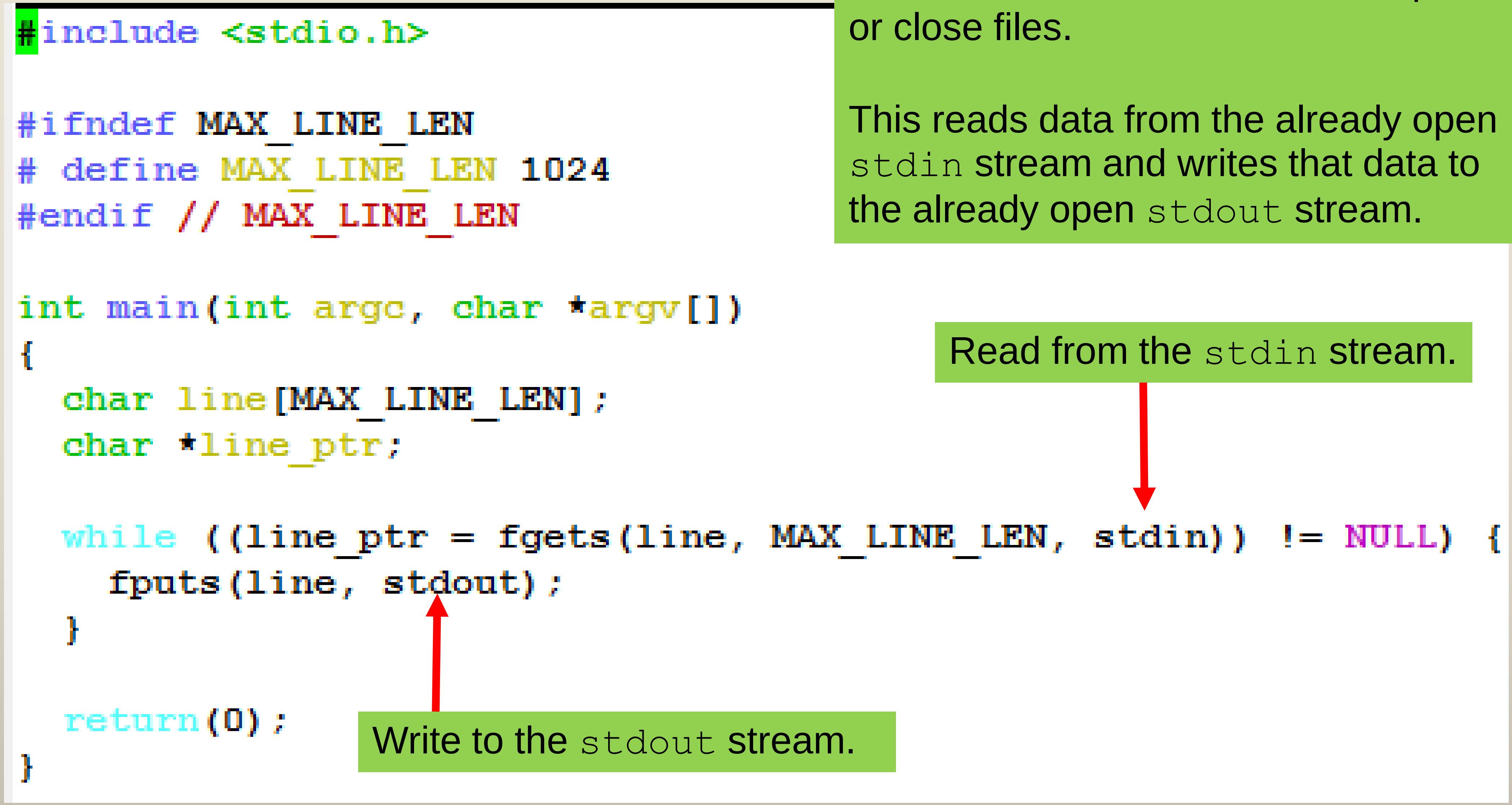
```
$ cat < file.txt
```

- Redirect the file `file.txt` as the `stdin` to the `cat` command. All calls to `scanf()/gets()` come from the `file.txt` file.

```
$ who | wc
```

- The `stdout` from the `who` command is sent to the pipe (**the | character**) and is redirected as `stdin` for the `wc` command.

When used on UNIX/Linux command line, the vertical bar character is called a pipe.



```
#include <stdio.h>

#ifndef MAX_LINE_LEN
# define MAX_LINE_LEN 1024
#endif // MAX_LINE_LEN

int main(int argc, char *argv[])
{
  char line[MAX_LINE_LEN];
  char *line_ptr;

  while ((line_ptr = fgets(line, MAX_LINE_LEN, stdin)) != NULL) {
    fputs(line, stdout);
  }

  return(0);
}
```

Notice that there are no calls to open or close files.

This reads data from the already open `stdin` stream and writes that data to the already open `stdout` stream.

Read from the `stdin` stream.

Write to the `stdout` stream.

Can be found in ~chaneyr/Classes/cs344/src/cat/my_cat1.c

Examples how you can run my_cat1

```
./my_cat1 < passwd
cat passwd | my_cat1
```

You can look at the source code.

You may also want to look at the source to my_cat2.c. The my_cat2.c program will allow you to have multiple files to cat on the command line.

```
./my_cat2 passwd my_cat1.c my_cat2.c
```

The C Programming Language

Some basic C capabilities:

- Structures and `typedef`
- Scope and extent
- Pointers
- Strings
- The C Preprocessor (aka `cpp`)
  - conditional compilation
  - macros
- stdio, `printf`, `fgets`, and buddies.