Basic I/O

- There is no input or output defined in C itself
- Character based (no format specifiers) – character by character I/O
  - getchar() - input
  - putchar(c) - output
- Formatted - standard I/O
  - scanf(stuff goes in here) - input *** white space is important!!!
  - printf(stuff goes in here) - output
  - Format Specifiers (% before specifier) – see next slide

```c
#include <stdio.h>
int main(void) {
    int i = 65; /* what if 258 instead of 65? */
    char a;
    printf("i=\n",i);
    printf("output with a putchar ");
    putchar(i);
    a = (char) i;
    printf("a=\n",a);
    getchar();
    return(0); } /* check.c */
```

```c
#include <stdio.h>
int main() { /* check1.c */
    int x;
    scanf("%d\n", &x);
    printf("x=%d\n", x);
}
```

Q. Why are pointers given to scanf?
A. Needs a pointer to the variable if it is going to change the variable itself i.e. assign a value to x.
When programming in C, you use conversion characters — the percent sign and a letter, for the most part — as placeholders for variables you want to display. The following table shows the conversion characters and what they display:

<table>
<thead>
<tr>
<th>Conversion Character</th>
<th>Displays Argument (Variable’s Contents) As</th>
</tr>
</thead>
<tbody>
<tr>
<td>%c</td>
<td>Single character</td>
</tr>
<tr>
<td>%d</td>
<td>Signed decimal integer (int)</td>
</tr>
<tr>
<td>%e</td>
<td>Signed floating-point value in E notation</td>
</tr>
<tr>
<td>%f</td>
<td>Signed floating-point value (float)</td>
</tr>
<tr>
<td>%g</td>
<td>Signed value in %e or %f format, whichever is shorter</td>
</tr>
<tr>
<td>%i</td>
<td>Signed decimal integer (int)</td>
</tr>
<tr>
<td>%o</td>
<td>Unsigned octal (base 8) integer (int)</td>
</tr>
<tr>
<td>%s</td>
<td>String of text</td>
</tr>
<tr>
<td>%u</td>
<td>Unsigned decimal integer (int)</td>
</tr>
<tr>
<td>%x</td>
<td>Unsigned hexadecimal (base 16) integer (int)</td>
</tr>
<tr>
<td>%%</td>
<td>(percent character)</td>
</tr>
</tbody>
</table>
Answers: check.c and check1.c

```
#include <stdio.h>
int main(void) {
    int i = 65;
    /* what if 258 instead of 65? */
    char a;
    printf("i=%d\n",i);
    printf("output with a putchar ");
    putchar(i);
    printf("\ni=%i\n",i);
    a = (char) i;
    printf("\na=%c\n",a);
    i=getchar();
    printf("\n\ni=%c\n",i);
    printf("\n\ni=0x%x\n",i);
    printf("\n\ni=%d\n",i);
    return (0);
}
```

```
#include <stdio.h>
#define PI 3.14159265358979323846
int main() {
    int x;
    scanf("%d", &x); /* why need & ? */
    printf("%d\n", x);
    float var;
    scanf("%f",&var);
    scanf("%d",&var);
    scanf("%lf", &var);
    int first, second;
    scanf("enter value ", &var);
    scanf("%d%d", &first, &second);
    int i, j;
    scanf(" %d %*d %*d %d ", &i, &j)
    return 0;  }
```
## Printf formatted output conversions

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument type; Printed As</th>
</tr>
</thead>
<tbody>
<tr>
<td>d,i</td>
<td>int; signed decimal notation.</td>
</tr>
<tr>
<td>o</td>
<td>int; unsigned octal notation (without a leading zero).</td>
</tr>
<tr>
<td>x,X</td>
<td>unsigned int; unsigned hexadecimal notation (without a leading 0xor 0X), using abcdef for 0x or ABCDEFfor 0X.</td>
</tr>
<tr>
<td>u</td>
<td>int; unsigned decimal notation.</td>
</tr>
<tr>
<td>c</td>
<td>int; single character, after conversion to unsigned char</td>
</tr>
<tr>
<td>s</td>
<td>characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.</td>
</tr>
<tr>
<td>f</td>
<td>double; decimal notation of the form [-]mmm.ddd, where the number of d’s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.</td>
</tr>
<tr>
<td>e,E</td>
<td>double; decimal notation of the form [-]m.dddde+/xx or [-]m.ddddeE+/xx, where the number of d’s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.</td>
</tr>
<tr>
<td>g,G</td>
<td>double; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal point are not printed.</td>
</tr>
</tbody>
</table>
Decimal & Floating point

- **%d**: print as decimal integer
- **%6d**: print as decimal integer, at least 6 characters wide
- **%f**: print as floating point
- **%6f**: print as floating point, at least 6 characters wide
- **%.2f**: print as floating point, 2 characters after decimal point
- **%6.2f**: print as floating point, at least 6 wide and 2 after decimal point

- Width of the whole number portion
  - Is for decimal integers
- The character width for float
  - Includes the decimal point position
Printf examples

causes the values of the two integers fahr and celsius to be printed, with a tab (\t) between them
   printf("%d\t%d\n", fahr, celsius);

to print the first number of each line in a field three digits wide, and the second in a field six digits wide
   printf("%3d %6d\n", fahr, celsius);

Each % construction in the first argument of printf is paired with the corresponding second argument, third argument, etc.; they must match up properly by number and type, or you will get wrong answers.

printf("\na=%f\nb=%f\nc=%f\nPI=%f", a, b, c, d);

c = a + b;
printf("%d + %d = %d\n", a, b, c);
Printf and Scanf

- Both formatted I/O
- Both sent to “standard I/O” location

Printf
- Converts values to character form according to the format string

Scanf
- Converts characters according to the format string, and followed by pointer arguments indicating where the resulting values are stored
Scanf requires two inputs:
- String argument - with format specifiers
- Set of additional arguments (pointers to variables)
- Consists of % at the beginning and a type indicator at the end
- Skips over all leading white space (spaces, tabs, and newlines) prior to finding first input value
- In between options:
  - * = used to suppress input
  - maximum field-width indicator
  - type indicator modifier
- Input stops when:
  - End of format string
  - Input read does not match what the format string specifies i.e. pointer arguments MUST BE the right type
  - The next call to scanf resumes searching immediately after the last character already converted.
- Return value = # of values converted

<table>
<thead>
<tr>
<th>FORMAT</th>
<th>MEANING</th>
<th>VARIABLE TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>%d</td>
<td>read an integer value</td>
<td>int</td>
</tr>
<tr>
<td>%ld</td>
<td>read a long integer value</td>
<td>long</td>
</tr>
<tr>
<td>%f</td>
<td>read a real value</td>
<td>float</td>
</tr>
<tr>
<td>%lf</td>
<td>read a double precision real value</td>
<td>double</td>
</tr>
<tr>
<td>%c</td>
<td>read a character</td>
<td>char</td>
</tr>
<tr>
<td>%s</td>
<td>read a character string from the input</td>
<td>array of char</td>
</tr>
</tbody>
</table>
Scanf examples

```c
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
Input:
01/29/64

int anInt;
scanf("%i%%", &anInt);
Input:
23%
anInt==23

int anInt; long l;
scanf("%d %ld", &anInt, &l);
Input:
-23 200
anInt==−23
l==200

double d;
scanf("%lf", &d);
Input:
3.14
d==3.14

string s;
scanf("%9s", s);
Input:
VeryLongString
s=="VeryLongS"
Why no & before s?

int anInt;
scanf("%*s %i", &anInt);
Input:
Age: 29
anInt==29 result

int anInt, anInt2;
scanf("%2i", &anInt);
scanf("%2i", &anInt2);
Input:
2345
anInt==23
anInt2==45

int anInt;
scanf("%i%%", &anInt);
Input:
23%
anInt==23

NOTE: pressing the enter key means you have entered a character...
```
## more Scanf examples

<table>
<thead>
<tr>
<th>Letter</th>
<th>Type of Matching Argument</th>
<th>Auto-skip; Leading White-Space</th>
<th>Example</th>
<th>Sample Matching Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>% (a literal, matched but not converted or assigned)</td>
<td>no</td>
<td>int anInt; scanf(&quot;%i%%&quot;, &amp;anInt);</td>
<td>23%</td>
</tr>
<tr>
<td>d</td>
<td>int</td>
<td>yes</td>
<td>int anInt; long l; scanf(&quot;%d %ld&quot;, &amp;anInt, &amp;l);</td>
<td>-23 200</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>yes</td>
<td>int anInt; scanf(&quot;%i&quot;, &amp;anInt);</td>
<td>0x23</td>
</tr>
<tr>
<td>o</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%o&quot;, &amp;aUInt);</td>
<td>023</td>
</tr>
<tr>
<td>u</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%u&quot;, &amp;aUInt);</td>
<td>23</td>
</tr>
<tr>
<td>x</td>
<td>unsigned int</td>
<td>yes</td>
<td>unsigned int aUInt; scanf(&quot;%d&quot;, &amp;aUInt);</td>
<td>1A</td>
</tr>
<tr>
<td>a, e, f, g</td>
<td>float or double</td>
<td>yes</td>
<td>float f; double d; scanf(&quot;%f %lf&quot;, &amp;f, &amp;d);</td>
<td>1.2 3.4</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>no</td>
<td>char ch; scanf(&quot; %c&quot;, &amp;ch);</td>
<td>Q</td>
</tr>
<tr>
<td>s</td>
<td>array of char</td>
<td>yes</td>
<td>char s[30]; scanf(&quot;%29s&quot;, s);</td>
<td>hello</td>
</tr>
<tr>
<td>n</td>
<td>int</td>
<td>no</td>
<td>int x, cnt; scanf(&quot;X: %d%n&quot;, &amp;x, &amp;cnt);</td>
<td>X: 123 (cnt==6)</td>
</tr>
<tr>
<td>[</td>
<td>array of char</td>
<td>no</td>
<td>char s1[64], s2[64]; scanf(&quot;%[^\t] %[^\t]&quot;, s1, s2);</td>
<td>Hello World field1 field2</td>
</tr>
</tbody>
</table>
You can use this function even with spaces in the input:

```c
scanf(" %[^\n]s", a);
```
File function summary

- **Open/Close files**
  - `fopen()` – open a stream for a file
  - `fclose()` – closes a stream

- **One character at a time:**
  - `fgetc()` – similar to `getchar()`
  - `fputc()` – similar to `putchar()`

- **One line at a time:**
  - `fprintf()`/`fputs()` – similar to `printf()`
  - `fscanf()`/`fgets()` – similar to `scanf()`

- **File errors**
  - `perror()` – reports an error in a system call
Text Streams

- Files accessed through the FILE mechanism provided by `<stdio.h>`
- Text streams are composed of lines.
- Each line has zero or more characters and are terminated by a new-line character which is the last character in a line.
- Conversions may occur on text streams during input and output.
- Text streams consist of only printable characters, the tab character, and the new-line character.
- Spaces cannot appear before a newline character, although it is implementation-defined whether or not reading a text stream removes these spaces.
**File constants `<stdio.h>`**

- **FILE** – a variable type suitable for string information for a file stream
- **fpos_t** – a variable type suitable for starting any position in a file
- **NULL** – value of a null pointer constant
- **EOF** – negative integer which indicates end-of-file has been reached
- **FOPEN_MAX** – integer which represents the maximum number of files that the system can guarantee that can be opened simultaneously
- **FILENAME_MAX** – integer which represents the longest length of a char array
- **stderr/stdin/stdout** – pointers to FILE types which correspond to the standard streams
When a program begins, there are already three available streams which are predefined and need not be opened explicitly and are of type “pointer to FILE”

- standard input
- standard output
- standard error

Files are associated with streams and must be opened to be used.

- The point of I/O within a file is determined by the file position.
- When a file is opened, the file position points to the beginning of the file (unless the file is opened for an append operation in which case the position points to the end of the file).
- The file position follows read and write operations to indicate where the next operation will occur.
- When a file is closed, no more actions can be taken on it until it is opened again.
- Exiting from the main function causes all open files to be closed.
Open/Read a File – one char at a time

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

int main() {
    char ch;
    FILE *fp;
    fp = fopen("lab2p2in","r"); // read mode
    if( fp == NULL ) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }
    printf("The contents of the file is :- \n\n");
    while( ( ch = fgetc(fp) ) != EOF )
        printf("%c",ch);
    fclose(fp);
    return 0; }

C programming code to open a file and print its contents to the screen, one character at a time.
//fileio1.c

(1) fgetc returns the value of an int that is converted from the character
(2) What happens if delete lab2p2in file? i.e. it can’t be found to open?
FILE *fopen(const char *filename, const char *mode);

Mode... (lots more!)
- r – read text mode
- w – write text mode (truncates file to zero length or creates a new file)
- If the file does not exist and it is opened with read mode (r), then the open fails → need to check for this

Declaration: int fclose(FILE *stream);
- Closes the stream.
- If successful, it returns zero.
- On error it returns EOF.

perror
- void perror(const char *str);
  - Prints a descriptive error message to stderr. First the string str is printed followed by a colon then a space (your error message). Then an error message based on the current setting of the variable errno is printed (system error message).
fgetc and fputc

Declaration: `int fgetc(FILE *stream);`
- Gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.
- On success the character is returned.
- If the end-of-file is encountered, then EOF is returned and the end-of-file indicator is set.
- If an error occurs then the error indicator for the stream is set and EOF is returned.

Declaration: `int fputc(int char, FILE *stream);`
- Writes a character (an unsigned char) specified by the argument char to the specified stream and advances the position indicator for the stream.
- On success the character is returned.
- If an error occurs, the error indicator for the stream is set and EOF is returned.
#include<stdio.h>
#include<stdlib.h>

int main() {
    char ch, chout;
    FILE *fpin, *fpout;
    fpin = fopen("lab2p2in","r");   // read mode
    fpout = fopen("lab2p2inout","w"); // write mode
    if( fpin == NULL ) {
        perror("Error while opening the input file.\n");
        exit(EXIT_FAILURE); }
    if (fpout == NULL ) {
        perror("Error while opening the output file.\n");
        exit(EXIT_FAILURE); }
    while( ( ch = fgetc(fpin) ) != EOF && chout != EOF )
        chout = fputc(ch,fpout);   // ret char if success ow EOF
    fclose(fpin);
    fclose(fpout);
    return 0; }

C programming code to open a file and print its contents to the another file, one character at a time.
//fileio2.c
Lab2p2 excerpt example

```c
FILE *infp, *outfp;
char * mode = "r";
char outfile[] = "lab2p2out";
char input[101], save_first_letter;
char * inptr;
int first_letter = TRUE, n=101;
infp = fopen("lab2p2in","r");
if (infp == NULL){
    printf(stderr, "can't open input file lab2p2in!\n");
    exit(EXIT_FAILURE); }
outfp = fopen(outfile,"w");
if (outfp == NULL) {
    printf(stderr, "Can't open output file %s!\n", outfile);
    exit(EXIT_FAILURE); }
fgets(input,n,infp);
while (!feof(infp))
    { // etc
        fgets(input,n,infp);
    }
//close files
```

- **fgets(buffer, size, stdin)**;
  - **buffer** is the location of your string storage space or buffer.
  - **size** is the number of characters to input. This sets a limit on input.
  - Note that **fgets()** also reads in the carriage return (enter key; newline character) at the end of the line. That character becomes part of the string you input.
- **fscanf(infp,"%s",input)**;
- **while (!feof(infp))**
fgets vs fscanf

**Declaration:** `char *fgets(char *str, int n, FILE *stream);`

- Reads a line from the specified stream and stores it into the string pointed to by `str`.
- It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.
- **The newline character is copied to the string.**
- A null character is appended to the end of the string.
- On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

**Declaration:** `int fscanf(FILE *stream, const char *format, ...);`

- Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied.
- On success the number of input fields converted and stored are returned. If an input failure occurred, then EOF is returned.
- Returns EOF in case of errors or if it reaches `eof`
fprintf and feof

Declaration:

int fprintf(FILE *stream, const char *format, ...);

- Sends formatted output to a stream
- Just like printf, but puts file pointer as first argument

In lab2p2:

> fprintf(outfp, “Your sipher coded message is %s\n”, input);

Declaration: int feof(FILE *stream);

- Tests the end-of-file indicator for the given stream
  - If the stream is at the end-of-file, then it returns a nonzero value. If it is not at the end of the file, then it returns zero.
File I/O example

See handout

lab3chin

lab3chin2

1 abc 2
3 def 4
5 ghi 6
7 jkl 8
9 mno 10