



# INPUT / OUTPUT AND FILES



# sprintf() and sscanf()

- The functions sprintf() and sscanf() are string versions of the functions printf() and scanf() respectively. They have the same parameter rules and format specifications as the functions printf and scanf

```
sprintf( char *s, const char *format, ...);
```

```
/* fills s using format and other params */
```

```
sscanf( char *s, const char *format, ...);
```

```
/* retrieves data from s, based on format */
```

# sscanf()

- The function sscanf is used for reading data from a character string
- A call to this function takes the general form  
    sscanf(string, format\_string, &arg1, &arg2, ...)
  - string refers to the character string to read from
  - format\_string refers to a character string containing certain required formatting information
  - arg1, arg2, etc., are arguments that represent the individual input data items

# sprintf()

- The function sprintf is used for writing data to a character string
- A call to this function takes the general form  
    sprintf(string, format\_string, arg1, arg2, ...)
  - string refers to the character string to write
  - format\_string refers to a character string containing certain required formatting information
  - arg1, arg2, etc., are arguments that represent the individual input data items

# Example

- Format strings in `sscanf()` specify how input will be formatted

```
int hour, min;  
sscanf(str, "%d:%d", &hour, &min);
```
- Most useful when reading from file, not good for user input (users often type slightly wrong things)

## Example 2

```
sprintf( str, "%s", "Hello World\n");
```

- str will contain the string "Hello World\n". Following '\n' will be, '\0', the string termination character

```
sprintf (str, "Square of four is %d and square  
of six is %d\n",    4*4, 6 * 6);
```

- str will hold the string "Square of four is 16 and square of six is 36\n"

## Example 3

```
char in_string[ ] = "brown fox";  
char out_string[100], word1[24], word2[24];  
  
sscanf(in_string, "%s %s", word1, word2);  
sprintf(out_string, "%s %s\n", word2, word1);
```

Note the space in the formatting string "%s %s"!

## Example 4

```
str = "25.4 13.7";
```

```
formatstring = "%lf %lf"
```

```
sscanf(str,formatstring, &double1, &double2);
```

```
/* or */
```

```
sscanf("25.4 13.7", "%lf %lf", &double1, &double2);
```

- Either way, double1 will be 25.4, and double2 13.7





# Files

- Files can be thought of as a stream of characters
- It is common to store information in files. Most people get the mental picture of a folder with papers. The collection of folders would be kept in a file cabinet
- This notion carries over to computers. In computers a file is a collection of information held in one logical unit, usually on disk. E.g.
  - The gcc compiler is a file
  - Microsoft Office documents, spreadsheets, etc, are all files. Even the programs Word and Excel are files
  - Your C source code is a file

# Files 2

- On most operating systems, files have properties
- In Windows/DOS some of the following properties are listed for a document (".doc")  
fileType
  - location
  - size
  - Date Created
  - Date Modified
  - Date Last Accessed
  - Name
  - ...



# Files 3

- All the programs you have seen so far, get their input from the keyboard and write output to the screen
- There are more general ways of using files
- If you have used Windows applications, you have seen the menu option to open a file. You give it the name of the file and it goes out to the disk and reads it. This becomes the data your application will use.



# Standard files in C

The compiler produces code to open these three files before the main function is invoked. The I/O macros and functions you have used so far read from stdin and write to stdout. These include getchar, putchar, scanf, and printf



# File properties

- Files have several important properties
  - They have a name (e.g. afile.c)
  - They must be opened and closed
  - They can be written to, or read from, or appended to



# Files 4

- When a computer program opens a file, no data is read from the file
- The open request is given to the operating system. The operating system finds where the file is stored and issues a “file handle”. When the program wants to read or write to the file at this point, it uses the file handle (also known as a file descriptor) as a parameter to an operating system request
- When a file is opened, the logical name is translated into a physical location on a specific device (e.g. a disk drive)



# I/O streams in C

- File handling in C is provided by a number of functions in the standard library
- All input/output is based on the concept of a stream
- There are two types of stream
  - text stream
  - binary stream

# I/O streams in C 2

- A text stream is a sequence of characters composed into lines, each line is terminated with a newline (`\n`) character
- A binary stream is a sequence of unprocessed bytes
  - newline has no significance





# Files in C

- In C, each file is simply a sequential stream of bytes. C imposes no structure on a file
- A file must first be opened properly before it can be accessed for reading or writing. When a file is opened, a stream is associated with the file
- Successfully opening a file returns a pointer to (i.e., the address of) a file structure, which contains a file descriptor and a file control block



# Files in C 2

- FILE is the name of a struct type defined in `stdio.h` that represents information about a file stream
- Normally accessed via pointers of `FILE *` type
- These pointers can be created using `fopen()`

# Files in C 3

- To define a pointer to a file we write  
`FILE *file_name;`

- The statement

`FILE *fptr1, *fptr2 ;`

declares that `fptr1` and `fptr2` are pointer variables of type `FILE`. They will be assigned the address of a file descriptor, that is, an area of memory that will be associated with an input or output stream

- (Whenever you are to read from or write to the file, you must first open the file and assign the address of its file descriptor (or structure) to the file pointer variable)

# fopen()

- A function call of the form

`fopen(file_name, mode)`

opens the named file in a particular mode and returns a file pointer

- In general, the function `fopen()` prepares a file for use

`FILE *fopen(char *file_id, char *mode);`

- `file_id` is the path, file name, and extension
- `mode` is how we intend to use it
- The return value is the file pointer

`FILE *payfile;`

`payfile = fopen("payroll.txt", "r");`

# Modes for opening files

- "r" open text file for reading
  - "w" open text file for writing
  - "a" open text file for appending
  - "rb" open binary file for reading
  - "wb" open binary file for writing
  - "ab" open binary file for appending
- 
- If the file is to be opened for both reading and writing use "r+" or "w+"

# fclose()

- A function call of the form  
`fclose(file_handle)`  
closes the file. If the file is successfully closed, zero is returned
- Generally, the function `fclose()` closes a file  
`int fclose(FILE *file);`
- Files are automatically closed at normal program termination, including `exit()`
  - They are not closed when the program crashes

# Example (dbl\_space)

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

```
void double_space(FILE *ifp, FILE *ofp);
void prn_info(char *pgm_name);
```

```
int main(int argc, char **argv)
{
    FILE *ifp, *ofp;


    if (argc != 3) {
        prn_info(argv[0]);
        exit(1);
    }
```

```
ifp = fopen(argv[1], "r");    /* open for reading */
ofp = fopen(argv[2], "w");    /* open for writing */
double_space(ifp, ofp);
fclose(ifp);
fclose(ofp);
return o;
}

void double_space(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF) {
        putc(c, ofp);
        if (c == '\n')
            putc('\n', ofp);    /* found newline - duplicate it */
    }
}
```





```
void prn_info(char *pgm_name)
{
    printf("\n%s%s%s\n\n%s%s\n\n",
        "Usage: ", pgm_name, " infile outfile",
        "The contents of infile will be double-spaced ",
        "and written to outfile.");
}
```

dbl\_space file1 file2

(reads from file1, writes to file 2, contents the same except for \n being duplicated in file2)

# Accessing files 3


- There are two ways of accessing a file, either
  - Sequential access
    - We usually access files sequentially
    - More details follow
- or
  - Random/direct access
    - Can randomly decide where to start reading/writing a file

# Sequential access

- Reading characters from a file
  - Use function fscanf for multiple variable reads  
`int fscanf (FILE *fp, "format str", var list)`
  - Syntax similar to scanf, only that we have an extra parameter, which is the file pointer
  - E.g:  
`int x;`  
`fscanf (fp, "%d", &x);`



# fscanf()

- On success this returns the number of successfully matched/assigned input fields
  - On failure it returns EOF
  - Format specifications same as scanf ( )
  - Found in the header file stdio.h
  - File must be open to use fscanf
- 

# fscanf() 2

- Use fscanf() to read from a file

```
int fscanf(FILE *file, char *control, arguments);
```

- Example

```
fscanf(payfile, "%s %f", name, pay);
```

# Sequential access 2

- Reading single characters
  - Use function `fgetc`, found in `stdio.h`  
`int fgetc (FILE *fp)`
  - Reads the next character from `fp`
  - Increments file offset automatically
  - On success, returns value of the character read
  - On failure, returns EOF
  - E.g.  
`char x;`  
`x = fgetc (fp);`

# Single character access

- Single character access from the keyboard and to the screen with `getchar()` and `putchar()`
- With files it is `getc()` and `putc()`
  - `int getc(FILE *file);`  
`int putc(int character, FILE *file);`
  - Both return EOF if an error occurs

# Sequential access 3

- Reading a string

- Use the fgets function, found in stdio.h

`char *fgets (char *str, int n, FILE *fp)`

- Reads a string of n characters from the file pointer fp and stores it in str
- Stops reading on new line ('\n') or n characters
- Reads white space and treats them as part of the string → unlike scanf



# fgets()

- Automatically appends a NULL character at the end of the string 😊
- The string str must be large enough to hold n characters + NULL
- File must be open to use fgets

```
char str [25]; fgets (str, 24, stdin);
```

- stdin is a pointer to standard input stream.

# fprintf()

- Writing characters to a file
  - Use function fprintf for multiple variable writes

```
int fprintf (FILE *fp, "format str", var list)
```
  - Syntax similar to printf, but with an extra parameter, which is the file pointer
  - E.g.

```
int x = 10;
fprintf (fp, "%d", x);
```

# fprintf() 2

- On success returns number of bytes written
- On failure returns EOF
- Format specifications same as printf()
- Found in the header file stdio.h
- File must be open to use fprintf

# fprintf() 3

- Use fprintf() to write to a file

```
int fprintf(FILE *file, char *control, arguments);
```

- Example

```
fprintf(payfile, "Name: %s, Pay: %.2f\n", name, pay);
```

- These are equivalent

```
fprintf(stdout, "hello, %s", name);
```

```
printf("hello, %s", name);
```

# Examples

```
/* input an integer and a float from a file  
   previously opened using fopen with the  
   return value stored in fp1 */
```

```
fscanf(fp1,"%d%f",&x,&y);
```

```
/*output to the file previously opened with the  
   return value stored in fp2*/
```

```
fprintf(fp2,"Final total was %d",total);
```

# fputc()

- Writing single characters
  - Use the function fputc, found in stdio.h

```
int fputc (int c, FILE *fp)
```
  - Writes the next character to fp
  - Increments file offset automatically
  - On success, returns integer value of characters written
  - On failure, returns EOF
  - E.g

```
char x = 'c' ;  
fputc (x, fp);
```

# fputs()

- Writing a string
  - Use the fputs function, found in stdio.h  
`char *fputs (char *str, FILE *fp)`
  - Writes a NULL terminated string str to fp
  - Does not append new line character
  - Terminating NULL is not copied

# fputs() 2

- On success, returns a non-negative number
- On error, returns EOF
- File must be open to use fputs



# Simple example code

```
#include <stdio.h>
main () {
FILE *fp1,*fp2,*fp3;
char filename[32];
char result[30]="";
int letter;
fp1=fopen("c:/temp/mydata.txt","r");
fp2=fopen("c:/temp/results.txt","w");
```

## Simple example code 2

```
printf("Name of output file:");  
scanf("%s",filename);  
fp3=fopen(filename,"a");  
fgets(result,14,fp1);  
printf("The first file contains:%s",result);  
fputs(result,fp2);  
letter='a';  
fputc(letter,fp3);  
fclose(fp1);fclose(fp2);fclose(fp3);
```

# Reading and writing

- Reading and Writing arrays of bytes
  - Use the functions fread and fwrite
  - Found in the header file stdio.h
- Both of them takes four parameters
  - a pointer to the array's first element (void\*)
  - the size (in bytes) of an element
  - the number of elements in the array
  - a file pointer

# Example fread

```
int main( void ) {  
    FILE * fp;  
    int x[5], i;  
  
    if ((fp = fopen("c:\\test.txt", "rb")) == NULL) {  
        printf("Error opening file!");  
        exit(1);  
    }  
  
    fread(x, sizeof(int), 5, fp);  
    for (i = 0; i < 5; i++)  
        printf("%d ", x[i]);  
    return 0;  
}
```

# Example fwrite

```
int main( void ) {  
    FILE * fp;  
    int x[5] = {1, 2, 3, 4, 5};  
  
    if ((fp = fopen("c:\\test.txt", "wb")) == NULL) {  
        printf("Error opening file!");  
        exit(1);  
    }  
  
    fwrite(x, sizeof(int), 5, fp);  
    return 0;  
}
```

# Accessing files 4

- Remember
- There are two ways of accessing a file, either
  - Sequential access (we just seen this)
- or
  - Random/direct access
    - Can randomly decide where to start reading/writing a file
    - `fseek()` and `ftell()` are used for this

# Standard library functions

- File access  
fopen, fclose, fflush, freopen
- Operations on files  
remove, rename
- Character input/output  
fgetc, getc, getchar, ungetc, fgets, gets, fputc, putc, putchar, fputs, puts
- Formatted input/output  
fscanf, scanf, sscanf, fprintf, printf, sprintf