General Input and Output

Darin Brezeale

The University of Texas at Arlington

Connecting to Files

In order to read or write to a file, we need to make a connection to it. For this we will use the following functions:

`fopen()` – makes the connection to a file
`fclose()` – releases the connection to a file

Note that `fopen()` and `fclose()` are declared in `stdio.h`.

Connecting to Files

When we make a connection to a file, we need a variable name to associate with it. This requires that we create a variable of type `FILE *`. This variable is the *file pointer*.

Example:

```
    FILE* newfile;
```

We could be working with multiple files; each would have its own file pointer.

Connecting to Files

When we use `fopen()` to make a connection to a file, we need to provide it with two things:

1. the name of the file
2. the mode for accessing the file

Example:

```
#include <stdio.h>
int main(void)
{
    FILE* newfile;  /* create file pointer */

    /*  format is fopen(filename, mode)  */
    newfile = fopen( "somefile.txt", "r" );
        /* do something with the file here */
    fclose( newfile );  /* release file */
}
```

File Access Modes

mode	purpose	file to use
r	read	use existing
w	write	create new, destroy existing
a	write to end	create new, use existing
r+	read & write	use existing
w+	read & write	create new, destroy existing
a+	read, write to end	create new, use existing

Simple Error Checking

When attempting to access files, there are many opportunities for problems:

- a file we wish to read may not exist
- a file we wish to write to may be in use by another program

Therefore, we should do some basic error checking when initiating access.

Simple Error Checking

If we are unable to open a file, we will get a NULL pointer.

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

int main(void)
{
    FILE* newfile;

    if ( (newfile = fopen("somefile.txt", "r" )) == NULL )
    {
        printf("this file could not be opened for reading\n");
        exit(1);  /* we should exit if there is an error */
    }

    fclose( newfile );
}
```

File Input and Output

There are many functions in the Standard C Library for reading and writing to files. Before discussing them, let's look again at `printf()` and `scanf()`.

printf()

We've already been using printf() in all of our programs.

Example:

```
    printf("the value of x is %d", x);
```

This is just a function call with two parameters: a string and an int.

printf()

The first (if there are more than one) parameter to be passed to `printf()` should be a string. If we are only passing a string to `printf()`, then we can pass it as a variable like we have with other functions.

Example:

```
#include <stdio.h>

int main(void)
{
    int x = 5;
    char text[] = "this is a string\n";

    printf(text);
}
```

printf()

We can also use a pointer to a string.

Example:

```
#include <stdio.h>

int main(void)
{
    int x = 5;
    char text1[] = "x is greater than 4\n";
    char text2[] = "x is not greater than 4\n";
    char *ptr;

    if (x > 4)
        ptr = text1;
    else
        ptr = text2;

    printf(ptr);
}
```

`scanf()`

We have used `scanf()` in a few programs to read values that were stored as `int`s. `scanf()` allows us to read other variable types as well. `scanf()` has format specifiers, such as

%d	int
%f	float
%lf	double
%c	char
%s	string

Note how the format specifier for a double is %lf, not %f as for `printf()`.

scanf()

We can read multiple values at once, using multiple format specifiers:

Example:

```
int some_int;
double some_double;

printf("provide an int and a double\n");
scanf("%d%lf", &some_int, &some_double);
```

See `example-io-scanf.c` on the course webpage.

fgets()

The Standard C library includes functions for reading strings, either from the keyboard or from a file.

One such function is `fgets()`. A call to `fgets()` has the following form:

```
fgets(array_name, array_size, source_of_input)
```

If the call to `fgets()` is successful, it returns the address of the array. Otherwise, it returns NULL.

`fgets()`

What we need to know about `fgets()` is:

- We should make sure the array for storing the input is large enough to hold all of the characters plus a terminating $\backslash 0$.
- A terminating $\backslash 0$ is automatically added to our input, either when we press the Enter key or we reach the end of our allocated space.
- Basic error checking is performed by checking if NULL was returned.
- `fgets()` is in `stdlib.h`

fgets()

Example:

```
char input[101];  /* we assume we won’t need to store more
                     than 100 characters */
char *ptr;

printf("enter a string of text to be printed\n");

/* stdin here means ’standard input’, which in this case is
   what the user types on the keyboard */
ptr = fgets(input, 101, stdin);
```

See `example-io-strings.c` on the course webpage.

fputs()

The Standard C library includes functions for writing strings, either to the screen or a file.

One such function is `fputs()`. A call to `fputs()` has the following form:

```
    fputs(array_name, destination)
```

fputs()

Example:

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

int main(void)
{
    char text[101] = "line one\nline two\nline three\n";

    fputs(text, stdout);  /* stdout here means ’standard out’,
                             which is usually the screen     */
}
```

produces

```
line one
line two
line three
```

File I/O

We can use `fgets()` and `fputs()` with files.

See `example-io-files.c` on the course webpage.

File I/O

`printf()` provides formatted output to stdout (i.e., the screen); `scanf()` provides formatted input from stdin (i.e., the keyboard). The equivalent for files are performed by `fprintf()` and `fscanf()`.

`fprintf()` and `fscanf()` have forms similar to `printf()` and `scanf()` except that we must also include a file pointer.

See `example-io-files2.c` and `example-io-files3.c` on the course webpage.

`fscanf()` vs `fgets()`

What is the difference between `fscanf()` and `fgets()`?

`fscanf()` expects us to know the format of the input, for example, a string followed by two integers.

`fgets()` just gets a string, which we must then process if we wish to break it into parts.

Summary of I/O Functions

Reading from the keyboard:

```
    fgets(input_array, buffer_size, stdin)
```

Reading from a file:

```
    fgets(input_array, buffer_size, pointer_to_file)
```

Writing to the screen:

```
    fputs(input_array, stdout)
```

Writing to a file:

```
    fputs(input_array, pointer_to_file)
```

Summary of I/O Functions

To perform formatted input and output, we have the following functions:

Reading from the keyboard:

```
scanf(string, variable(s))
```

Reading from a file:

```
fscanf(file_pointer, string, variable(s) or expression(s))
```

Writing to the screen:

```
printf(string, variable(s))
```

Writing to a file:

```
fprintf(file_pointer, string, variable(s) or expression(s))
```

Formatted I/O with strings

We can also perform formatted I/O with strings using `sprintf()` and `sscanf()` (note the beginning letter s).

```
char text[30];
char name[] = "something";
char first[20];
int second;

sprintf(text, "%s %d", name, 42);
printf("%s\n", text);

sscanf(text, "%s %d", first, &second);
printf("%d %s\n", second, first);
```

produces

```
something 42
42 something
```

Command-line Parameters

Sometimes we don’t know the name of the file(s) to read or write until we run a program. Since `main()` is a function, we can pass variables to it just as we have other functions.

We do this using

```
    int main ( int argc, char *argv[])
```

where `argc` is the number of command-line parameters and `argv` is an array of pointers to each command-line parameter.

Command-line Parameters

Example 1:

```
somefile.exe input.txt
```

Here `argc` = 2, `argv[0]` = somefile.exe, and `argv[1]` = input.txt.

Example 2:

```
hw.exe input.txt output.txt
```

Here `argc` = 3, `argv[0]` = hw.exe, `argv[1]` = input.txt, and `argv[2]` = output.txt.