CSE 1320

Week of 3/25/2019

Instructor : Donna French

Command Line Parameters

Command line parameters are a sequence of strings used to pass information to a C program at execution time.

They appear as strings on the command line after the name of the program when it is executed.

These strings are separated by blanks, tabs, or other whitespace.

Command line parameters are available throughout the time that the program is executing.

Command Line Parameters

Command line parameters are accessed via arguments to `main()`

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

`argc` and `argv` are traditional names but can be anything

`argc` contains the count of parameters on the command line. The name of the program is the first command line parameter and it is part of the count so `argc` is always at least one.

Command Line Parameters

`argv` is an array of pointers to `char`s

the pointers point to the strings that appear on the command line

the array is indexed by `0` to `argc – 1` and terminated with a `NULL` pointer

Command Line Parameters

Running a program with command line parameters

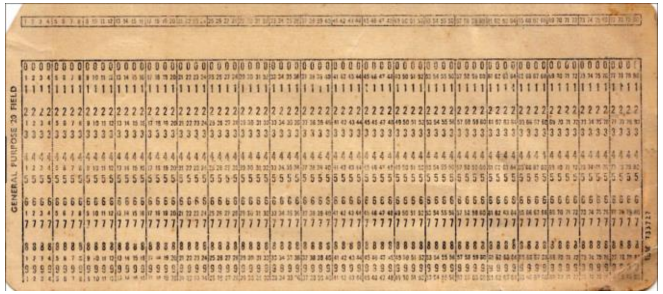
```
    a.out clp1 clp2 clp3
```

Running a program in debug with command line parameters

```
    gdb --args a.out clp1 clp2 clp3
```

argcargv1Demo.c

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

int main(int argc, char *argv[])
{
     int i;
     char filename1[20] = {};
     char filename2[20] = {};

     strcpy(filename1, *(argv + 1));
     strcpy(filename2, *(argv + 2));

     printf("filename1 is %s and filename2 is %s\n",
     filename1, filename2);

     return 0;
}
```



File Handling in C

Storage of data in memory is temporary.

Files are used for data persistence – permanent retention of data.

Computers store files on secondary storage devices such as hard disks, CDs, DVDs, flash drives and tapes.

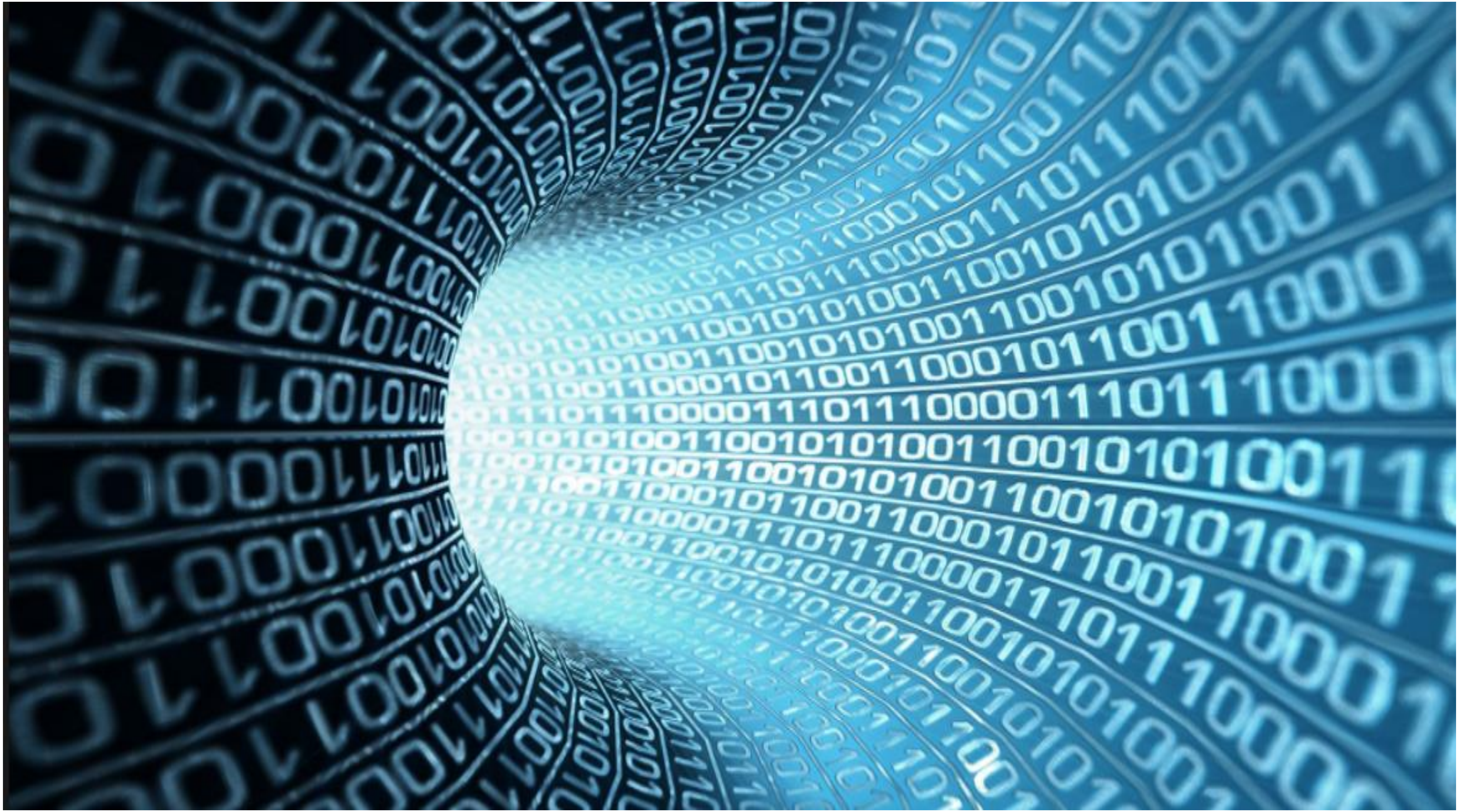
File Handling in C

C does not impose structure on a file. A file is just a sequence of data.

The concept of a record in a file does not exist.

The application using the file imposes the structure/record on the file.

For example, a Word document is just a sequence of data that Word knows how to interpret and display and manipulate.



File Handling in C

C views each file as simply a sequence of bytes.

Each file ends either with an end-of-file marker (EOF) or at a specific byte number recorded in an operating-system-maintained administrative database.

When a file is opened, a file handle is associated with that file and is used by the C program to refer to the file.

File Handling in C

When a file is to be opened for use in a program, the programmer must declare a new variable of type `FILE *`.

```
FILE *myfile, *yourfile;
```

```
(gdb) p myFile
$1 = (FILE *) 0x0
```

Definition of `FILE *`

```
(gdb) ptype myFile
```

```
type = struct _IO_FILE {
    int _flags;
    char *_IO_read_ptr;
    char *_IO_read_end;
    char *_IO_read_base;
    char *_IO_write_base;
    char *_IO_write_ptr;
    char *_IO_write_end;
    char *_IO_buf_base;
    char *_IO_buf_end;
    char *_IO_save_base;
    char *_IO_backup_base;
    char *_IO_save_end;
    struct _IO_marker *_markers;
    struct _IO_FILE *_chain;
    int _fileno;
    int _flags2;
    __off_t _old_offset;
    short unsigned int _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];
    _IO_lock_t *_lock;
    __off64_t _offset;
    void *__pad1;
    void *__pad2;
    void *__pad3;
    void *__pad4;
    size_t __pad5;
    int _mode;
    char _unused2[20];
} *
```

File Handling in C

C library function, `fopen()`, is then used to connect these declarations with the files on disk. Using `fopen()` makes a file available for use by the program.

```
myfile = fopen("filename", "mode");
```

```
myFile = fopen("it.txt", "r");
```

We will refer to `myfile` as the file's handle or file handle.

```
(gdb) p myFile
$2 = (FILE *) 0x601010
```

```
Breakpoint 1, main () at file2Demo.c:10
10              myFile = fopen("it.txt", "r");
(gdb) shell ls -l /proc/7661/fd
total 0
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 0 -> /dev/pts/31
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 1 -> /dev/pts/31
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 2 -> /dev/pts/31
lr-x------ 1 frenchdm staff 64 Nov  3 16:00 3 -> pipe:[124469163]
l-wx------ 1 frenchdm staff 64 Nov  3 16:00 4 -> pipe:[124469163]
lr-x------ 1 frenchdm staff 64 Nov  3 16:00 5 -> pipe:[124469164]
l-wx------ 1 frenchdm staff 64 Nov  3 16:00 6 -> pipe:[124469164]
(gdb) step
12              return 0;
(gdb) shell ls -l /proc/7661/fd
total 0
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 0 -> /dev/pts/31
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 1 -> /dev/pts/31
lrwx------ 1 frenchdm staff 64 Nov  3 16:00 2 -> /dev/pts/31
lr-x------ 1 frenchdm staff 64 Nov  3 16:00 3 -> pipe:[124469163]
l-wx------ 1 frenchdm staff 64 Nov  3 16:00 4 -> pipe:[124469163]
lr-x------ 1 frenchdm staff 64 Nov  3 16:00 5 -> pipe:[124469164]
l-wx------ 1 frenchdm staff 64 Nov  3 16:00 6 -> pipe:[124469164]
lr-x------ 1 frenchdm staff 64 Nov  3 16:00 7 -> /home/f/fr/frenchdm/it.txt
```

File Handling in C

```
myfile = fopen("filename", "mode");
```

mode of a file determines whether the file is opened for reading, writing or a combination of the two

`"r"`	open an existing file for reading only
`"w"`	open new file for writing only
`"a"`	open a file for appending (writing at the end of the file)
`"r+"`	open an existing file for update (reading and writing)
`"w+"`	open (create) a new file for update
`"a+"`	open a (new or existing) file for reading and appending

Formatted Input and Output

`printf()` and `scanf()`

do formatted input and output to and from the terminal

`fprintf()` and `fscanf()`

do formatted input and output to and from a file

`sprintf()` and `sscanf()`

write and read to and from a string

`printf()` and `scanf()` families

Formatted Input and Output

`fscanf()` and `fprintf()`

```
fscanf(fp, control_string, args, …)

fprintf(fp, control_string, args, …)
```

`fp`	file handle (`FILE *`) - associated with an open file
`control_string`	conversion specifier
`args`	argument to conversion specifier

```
a.out FILENAME=it.txt MODE=w+ ACTION=WRITE
```

```
int main(int argc, char *argv[])
{
   char filename[100] = {};
   char mode[2] = {};
   char action[10] = {};
   int debug = 0;
   char arg_value[100] = {};

   char buffer[100] = {};
   FILE *MyFile;

   get_command_line_parameter(argv, "DEBUG=", arg_value);
   if (!strcmp(arg_value, "ON"))
      debug = 1;

   get_command_line_parameter(argv, "FILENAME=", arg_value);
   strcpy(filename, arg_value);

   get_command_line_parameter(argv, "MODE=", arg_value);
   strcpy(mode, arg_value);

   get_command_line_parameter(argv, "ACTION=", arg_value);
   strcpy(action, arg_value);
```

```
(gdb) p debug
$1 = 0
(gdb) p filename
$2 = "it.txt", '\000'
<repeats 93 times>
(gdb) p mode
$3 = "w+"
(gdb) p action
$4 =
"WRITE\000\000\000\000"
```

fprintfDemo.c

```
void get_command_line_parameter(char *argv[], char ParamName[], char ParamValue[])
{
        int i  = 0;

        while (argv[++i] != NULL)
        {
                if (!strncmp(argv[i], ParamName, strlen(ParamName)))
                {
                        strcpy(ParamValue, strchr(argv[i], '=') + 1);
                }
        }

        return;
}
```

Part before the =

Part after the =

```
a.out FILENAME=it.txt MODE=w+ ACTION=WRITE
```

```
(gdb) p debug
$1 = 0
(gdb) p filename
$2 = "it.txt", '\000'
<repeats 93 times>
(gdb) p mode
$3 = "w+"
(gdb) p action
$4 =
"WRITE\000\000\000\000"
```

```
get_command_line_parameter(argv, "DEBUG=", arg_value);

get_command_line_parameter(argv, "FILENAME=", arg_value);

get_command_line_parameter(argv, "MODE=", arg_value);

get_command_line_parameter(argv, "ACTION=", arg_value);
```

fprintfDemo.c

```
MyFile = fopen(filename, mode);

if (MyFile == NULL)
{
   printf("  %s did not properly open...exiting\n", filename);
   exit(0);
}
else if (debug)
   printf("File %s opened\n", filename);
```

fprintfDemo.c

```
Breakpoint 2, main (argc=5, argv=0x7fffffffe848) at fprintfDemo.c:45
45              MyFile = fopen(filename, mode);
(gdb) p filename
$2 = "it.txt", '\000' <repeats 93 times>
(gdb) p MyFile
$3 = (FILE *) 0x0
(gdb) step
47              if (MyFile == NULL)
(gdb) step
52              else if (debug)
(gdb) p MyFile
$4 = (FILE *) 0x601010
(gdb) p debug
$5 = 1
(gdb) step
53                      printf("File %s opened\n", filename);
(gdb)
File it.txt opened
```

fprintfDemo.c

```
(gdb) p action
$6 = "WRITE\000\000\000\000"

55                      if (!strcmp(action, "WRITE"))
56                      {
57                              printf("Enter a string ");
58                              fgets(buffer, sizeof(buffer), stdin);
59                              fprintf(MyFile, "%s", buffer);
(gdb)
60                      }
```

fprintfDemo.c

```
Breakpoint 2, main (argc=5, argv=0x7fffffffe848) at
fprintfDemo.c:55
55                if (!strcmp(action, "WRITE"))
(gdb) step
57                        printf("Enter a string ");
(gdb)
58                        fgets(buffer, sizeof(buffer)-1, stdin);
(gdb)
Enter a string Hello again
59                        fprintf(MyFile, "%s", buffer);
(gdb) p buffer
$1 = "Hello again\n", '\000' <repeats 87 times>
```

Using `stdout` instead of a file name will write out to the screen since `stdout` is the standard out which is currently tied to the screen.

```
[frenchdm@omega ~]$ more it.txt
Hello again
```

fprintfDemo.c

```
Breakpoint 2, main (argc=5, argv=0x7fffffffe858) at fprintfDemo.c:63
63              if (!strcmp(action, "READ"))
(gdb) p action
$1 = "READ\000\000\000\000\000"
(gdb) step
65                  fscanf(MyFile, "%s", &buffer);
(gdb) p MyFile
$2 = (FILE *) 0x602010
(gdb) p buffer
$3 = '\000' <repeats 99 times>
(gdb) step
66                  printf("Read \"%s\" from the file\n", buffer);
(gdb) p buffer
$4 = "Hello", '\000' <repeats 94 times>
(gdb) c
Continuing.
Read "Hello" from the file
```

```
[frenchdm@omega ~]$ more it.txt
Hello again
```

`fscanf()` will handle the blank just like `scanf()`

– stops reading when it encounters one

fprintfDemo.c

```
Breakpoint 2, main (argc=5, argv=0x7fffffffe858) at fprintfDemo.c:69
69              fclose(MyFile);
(gdb) shell ls -l /proc/26669/fd
total 0
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 0 -> /dev/pts/32
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 1 -> /dev/pts/32
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 2 -> /dev/pts/32
lr-x------ 1 frenchdm staff 64 Nov  3 19:39 3 -> pipe:[124773434]
l-wx------ 1 frenchdm staff 64 Nov  3 19:39 4 -> pipe:[124773434]
lr-x------ 1 frenchdm staff 64 Nov  3 19:39 5 -> pipe:[124773435]
l-wx------ 1 frenchdm staff 64 Nov  3 19:39 6 -> pipe:[124773435]
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 7 -> /home/f/fr/frenchdm/it.txt
(gdb) step
71              return 0;
(gdb) shell ls -l /proc/26669/fd
total 0
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 0 -> /dev/pts/32
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 1 -> /dev/pts/32
lrwx------ 1 frenchdm staff 64 Nov  3 19:39 2 -> /dev/pts/32
lr-x------ 1 frenchdm staff 64 Nov  3 19:39 3 -> pipe:[124773434]
l-wx------ 1 frenchdm staff 64 Nov  3 19:39 4 -> pipe:[124773434]
lr-x------ 1 frenchdm staff 64 Nov  3 19:39 5 -> pipe:[124773435]
l-wx------ 1 frenchdm staff 64 Nov  3 19:39 6 -> pipe:[124773435]
```

fprintfDemo.c

Formatted Input and Output

sscanf() and sprintf()

```
sscanf(buffer, control_string, args, …)

sprintf(buffer, control_string, args, …)
```

buffer	buffer in memory
control_string	conversion specifier
args	argument to conversion specifier

Formatted Input and Output

`sscanf()` **and** `sprintf()`

```
sprintf(buffer, "%s %s has student id %s ",
        first_name, last_name, id);

sscanf(buffer, "%s %s %*s %*s %*s %s", a, b, c);
```

```
char buffer[100] = {};
char first_name[50] = {};
char last_name[50] = {};
char id[10] = {};
char a[50] = {};
char b[50] = {};
char c[10] = {};

printf("Enter first name ");
scanf("%s", &first_name);
printf("\nEnter last name ");
scanf("%s", &last_name);
printf("\nEnter id ");
scanf("%s", &id);
```

```
Enter first name Fred

Enter last name Flintstone

Enter id 1000000001
```

sprintfDemo.c

```
Breakpoint 2, main () at sprintfDemo.c:23
23              sprintf(buffer, "%s %s has student id %s ",
24                      first_name, last_name, id);
(gdb) p first_name
$1 = "Fred", '\000' <repeats 36 times>"\377, \265\360\000\000\000\000\000",
<incomplete sequence \302>
(gdb) p last_name
$2 =
"Flintstone\000\000\000\000\000\000\000\347\377\377\001\000\000\000\340\366\252\252\25
2*\000\000`䪪\252*\000\000\020\350\377\377\377\177\000\000\000", <incomplete sequence
\340>
(gdb) p id
$3 = "1000000001"
(gdb) step


(gdb) p buffer
$4 = "Fred Flintstone has student id 1000000001
\000\000\000\000\000\000\001\000\000\000\000\000\303\000\307\006@", '\000' <repeats 13
times>"\300, \313!\311>\000\000\000\220\006@", '\000' <repeats 13 times>"\220,
\350\377\377"
```

sprintfDemo.c

%*s tells sscanf() to skip the characters between whitespaces.

```
Fred Flintstone has student id 1000000001

27                sscanf(buffer, "%s %s %*s %*s %*s %s", a, b, c);
(gdb) p a
$5 = "\000\000\000\000\000\000\000\000\002\223\000\311>", '\000' <repeats 11
times>"\340, \366\252\252\252*\000\000\001", '\000' <repeats 15 times>, "\001"
(gdb) p b
$6 = "\000\000\000\000\001\000\000\000\227\a\000\000\001", '\000' <repeats 11 times>,
"`翄\252*\000\000\340\347\377\377\377\177\000\000\220\347\377\377\377\177\000\000.N"
(gdb) p c
$7 = "@\374@\311>\000\000\000\250\002"
(gdb) step
28                printf("First name = %s\nLast Name = %s\nID = %s\n\n", a, b, c);
(gdb) p a
$8 = "Fred\000\000\000\000\002\223\000\311>", '\000' <repeats 11 times>"\340,
\366\252\252\252*\000\000\001", '\000' <repeats 15 times>, "\001"
(gdb) p b
$9 = "Flintstone\000\000\001", '\000' <repeats 11 times>, "`翄
\252*\000\000\340\347\377\377\377\177\000\000\220\347\377\377\377\177\000\000.N"
(gdb) p c
$10 = "1000000001"
```

sprintfDemo.c

Error Detection with the `printf()` and `scanf()` Families

The `printf()` family will return an `int` value indicating the total number of characters written by each particular call.

The `scanf()` family will return an `int` indicating the number of conversions that were made which should match the conversion specifications in its control string.

Depending on the criticality of your application, adding this level of error checking may not be worth the added complexity.

File Handling in C

Part of the structure defined in the `typedef FILE` is a value that tracks the current position in the file.

We will refer to that as the ***file pointer***.

The file pointer moves as reads and writes are done.

File Handling in C
Two Types of Access

Sequential Access

When a file is opened, reading (or writing) starts at the beginning of the file and proceeds through the file in a sequential manner.

Whenever a read is done, the file pointer moves to point to the next element in the file to be read.

Random Access

Allows the reading of the records in any order.

Random Access in Files

Two library functions in the standard C library help with random access of files

```
fseek(fp, offset, start);
```

`fp` file handle (`FILE *`) - associated with an open file

`offset` variable of type long that represents the byte offset or number of bytes that the pointer is to be moved

`start` indicates the beginning position for the file pointer

```
ftell(fp);
```

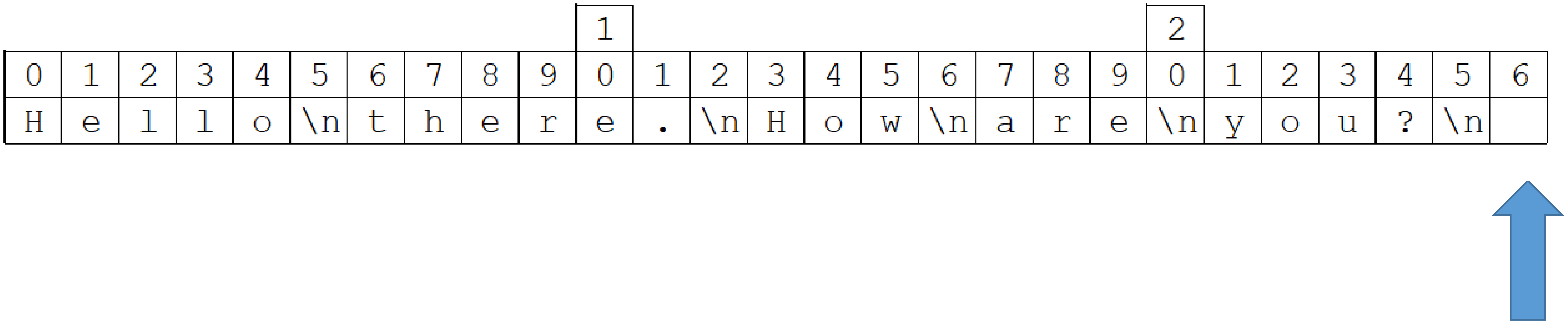
`fp` file handle (`FILE *`) – associated with an open file

returns the current byte offset from the beginning of the file

```
for (i = 0; i < 5; i++)
{
   printf("Enter string %d ", i);
   fgets(buffer, sizeof(buffer), stdin);
   fprintf(MyFile, "%s", buffer);
}
```

```
Enter string 0 Hello
Enter string 1 there.
Enter string 2 How
Enter string 3 are
Enter string 4 you?
```

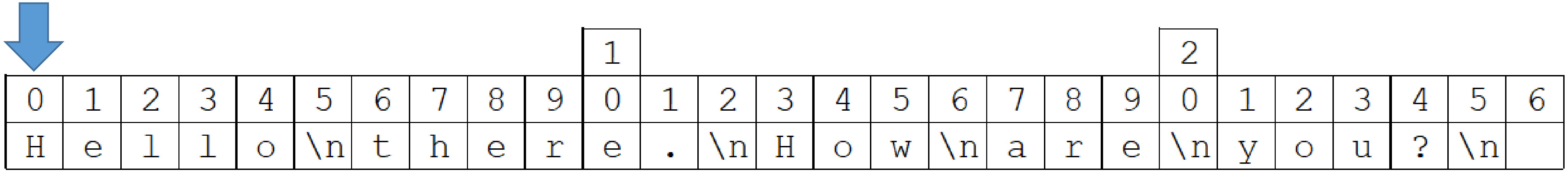
										1										2					
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
H	e	l	l	o	\n	t	h	e	r	e	.	\n	H	o	w	\n	a	r	e	\n	y	o	u	?	\n



										1										2						
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
H	e	l	l	o	\n	t	h	e	r	e	.	\n	H	o	w	\n	a	r	e	\n	y	o	u	?	\n	

The file pointer is sitting at position 26. `ftell()` can return the file pointer's location.

```
Breakpoint 4, main (argc=4, argv=0x7fffffffe868) at fseekDemo.c:62
62                  printf("ftell() = %d\n\n", ftell(MyFile));
(gdb) step
ftell() = 26
```

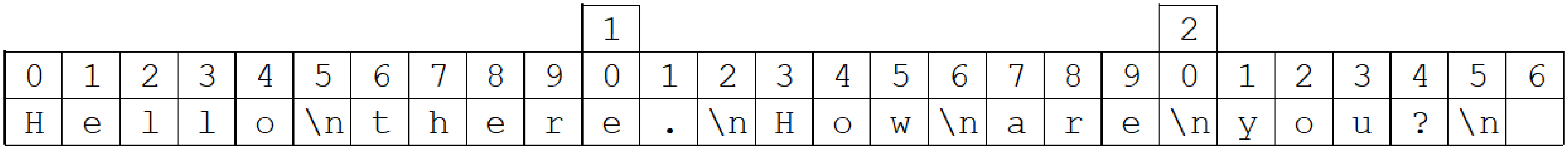


										1										2						
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
H	e	l	l	o	\n	t	h	e	r	e	.	\n	H	o	w	\n	a	r	e	\n	y	o	u	?	\n	

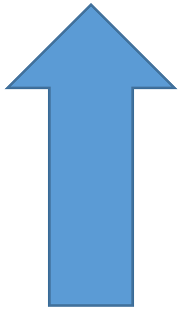
```
fseek(MyFile, 0, 0);

for (i = 0; i < 5; i++)
{
   printf("ftell() = %d\t", ftell(MyFile));
   fscanf(MyFile, "%s", &buffer);
   printf("Printing string %d from file : %s\t", i, buffer);
   printf("ftell() = %d\n", ftell(MyFile));
}
```

fscanf() stops at \n

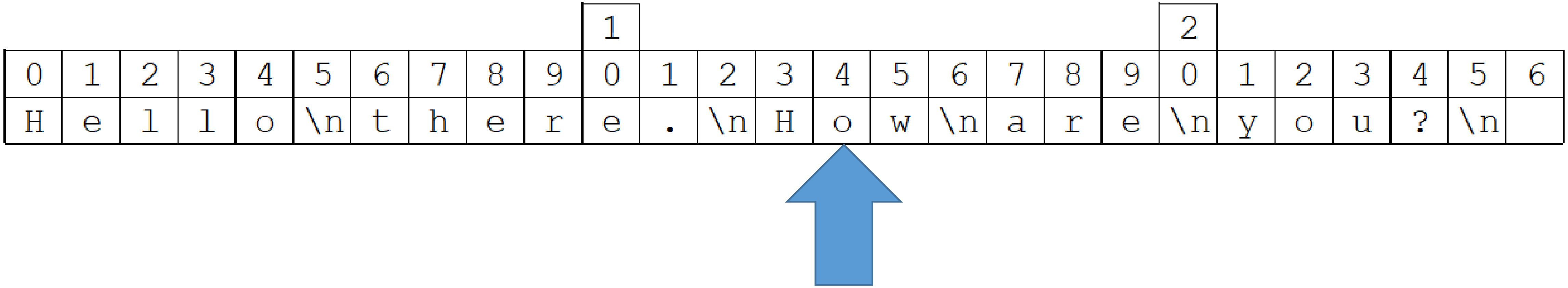
```
ftell() = 0     Printing string 0 from file : Hello     ftell() = 5
ftell() = 5     Printing string 1 from file : there.    ftell() = 12
ftell() = 12    Printing string 2 from file : How       ftell() = 16
ftell() = 16    Printing string 3 from file : are       ftell() = 20
ftell() = 20    Printing string 4 from file : you?      ftell() = 25
```



										1										2						
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
H	e	l	l	o	\n	t	h	e	r	e	.	\n	H	o	w	\n	a	r	e	\n	y	o	u	?	\n	



```
75                  printf("Enter an offset of fseek() ");
(gdb)
76                  scanf("%ld", &offset);
(gdb)

Enter an offset of fseek() 21

80                          fseek(MyFile, offset, SEEK_SET);
(gdb) step
81                          fscanf(MyFile, "%s", &buffer);
(gdb)
82                          printf("Printing string from file : %s\n\n", buffer);

Printing string from file : you?
```



										1										2						
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6
H	e	l	l	o	\n	t	h	e	r	e	.	\n	H	o	w	\n	a	r	e	\n	y	o	u	?	\n	

```
84                              printf("Enter an offset of fseek() ");
(gdb) step
85                              scanf("%ld", &offset);
(gdb)
Enter an offset of fseek() 14
87                    while (offset != -1);
(gdb)
80                              fseek(MyFile, offset, SEEK_SET);
(gdb)
81                              fscanf(MyFile, "%s", &buffer);
(gdb)
82                              printf("Printing string from file : %s\n\n", buffer);
(gdb)
Printing string from file : ow
```

Random Access in Files

Defines from `stdio.h` that can be used with file access

```
#define SEEK_SET        0       /* Seek from beginning of file.  */
#define SEEK_CUR        1       /* Seek from current position.  */
#define SEEK_END        2       /* Seek from end of file.  */
```

Error Handling with the ANSI C Library

Some error handling functions are provided in the ANSI C library.
Handle errors that occur when reading from or writing to a file. Must include `errno.h`.

`ferror(FILE *)`

checks to see if an error has occurred on the file
returns a nonzero value if an error occurred; otherwise, 0

`perror(char string)`

will display the string followed by a colon, a space and an error message

`clearerr(FILE *)`

clears the error condition for the file

```
a.out FILENAME=ABC
```

File ABC does not exist and no MODE was passed in

```
Breakpoint 2, main (argc=1, argv=0x7fffffffe898) at errorDemo.c:44
44                  MyFile = fopen(filename, mode);
(gdb) step
45                  if (MyFile == NULL)
(gdb)
47                          perror("fopen() failed ");
(gdb) p MyFile
$1 = (FILE *) 0x0
(gdb) step

fopen() failed : Invalid argument
```

errorDemo.c

```
a.out FILENAME=ABC MODE=w
```

File ABC does not exist
open new file for writing only

```
Breakpoint 2, main (argc=3, argv=0x7fffffffe878) at errorDemo.c:44
44                      MyFile = fopen(filename, mode);
(gdb) step
45                      if (MyFile == NULL)
(gdb)
51                      fgets(buffer, 100, MyFile);
(gdb)
53                      if (ferror(MyFile))
(gdb)
55                              perror("fgets() error1 ");
(gdb)
```

fgets() is trying to read from the file
but it was opened for writing only

```
fgets() error1 : Bad file descriptor
```

errorDemo.c

```
if (ferror(MyFile))
{
        perror("fgets() error1 ");
}

if (CE)
        clearerr(MyFile);

if (ferror(MyFile))
{
        perror("fgets() error2 ");
}
```

```
[frenchdm@omega ~]$ a.out FILENAME=ABC MODE=w
fgets() error1 : Bad file descriptor
fgets() error2 : Illegal seek
```

Did not run with `CE=ON` so `CE` was not set to 1 so `clearerr()` was not called

```
[frenchdm@omega ~]$ a.out FILENAME=ABC MODE=w CE=ON
fgets() error1 : Bad file descriptor
```

Did run with `CE=ON` so `CE` was set to 1 so `clearerr()` was called

errorDemo.c