Programming in Modern C++

Module M41: Input-Output: File Handling in C

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Many diagrams in this module are taken from *Computer Science: A Structured Programming Approach Using C*

Weekly Recap

- Introduced the concept of exceptions
  - Discussed error handling in C with various language features and library support in C for handling errors
  - Discussed exception (error) handling in C++ with `try-throw-catch` feature in C++ for handling errors
- Introduced the templates in C++
  - Discussed function templates as generic algorithmic solution for code reuse
  - Discussed class templates as generic solution for data structure reuse
  - Explained partial template instantiation and default template parameters
  - Demonstrated templates on inheritance hierarchy
- Introduced Function Objects or Functors
  - Illustrated functors with several simple examples and examples from STL

Module Objectives

- Understand file handling and I/O in C
- To understand Text and Binary I/O

Module Outline

1. Weekly Recap
2. Standard Library for I/O
3. Files and Streams
   - File Open / Close
4. Formatted I/O
   - Output
   - Read
5. Unformatted I/O
6. Direct IO
7. File Positioning
8. Module Summary

Standard Library for I/O

Sources:

- Computer Science: A Structured Programming Approach Using C
- C file input/output, Wikipedia

Standard C I/O Functions

- The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library header `<stdio.h>`
- **Categories of I/O Functions**
  - File Open/Close
  - Formatted Input/Output
  - Character Input/Output
  - Line Input/Output
  - Block Input/Output
  - File Positioning
  - System File Operations
  - File Status

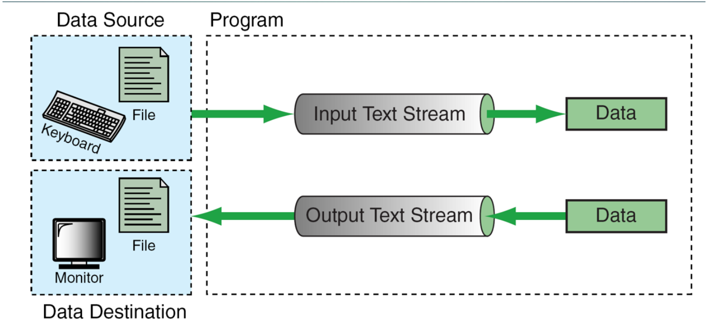
Source: C file input/output, Wikipedia

Files and Streams

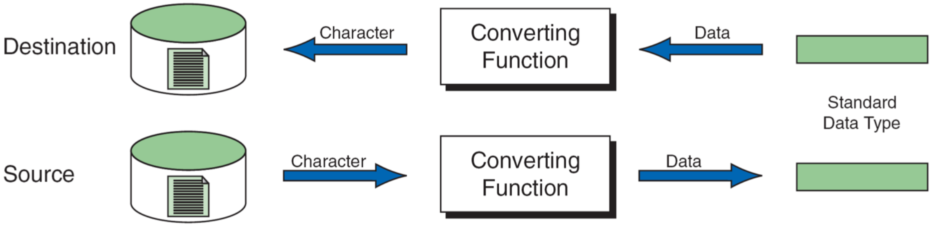
Files and Streams

Files and Streams

- *A* ***file*** *is an external collection of related data treated as a unit*. The primary purpose of a file is to keep a record of data. Since the contents of primary memory are lost when the computer is shut down, we need files to store our data in a more permanent form
- *Data is input to and output from a* **stream**. A stream can be associated with a physical device, such as a terminal, or with a file stored in auxiliary memory



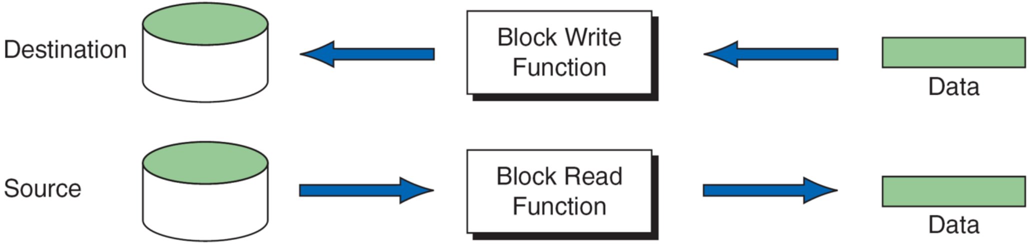
Reading and Writing Text Files

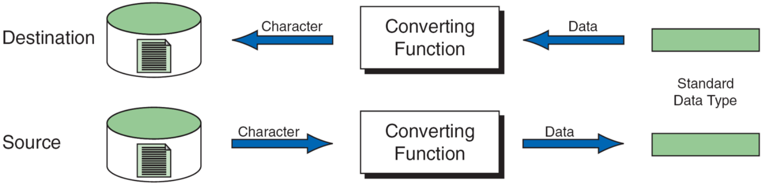


Text files are used for:

- Formatted input/output functions
- Character input/output functions
- String input/output functions

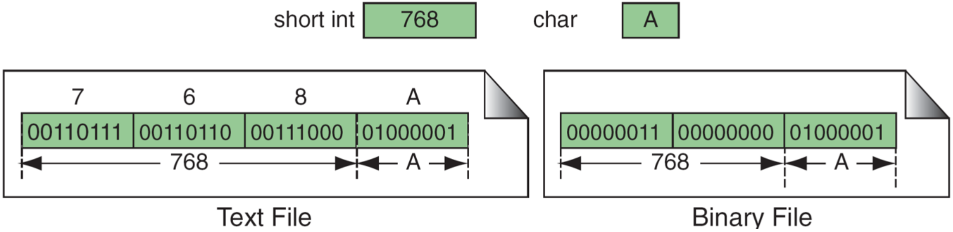
Binary File: Block Input and Output





Text and Binary Files

- ascii(7) = 55 = 0b 0011 0111
- ascii(6) = 54 = 0b 0011 0110
- ascii(8) = 56 = 0b 0011 1000
- ascii('A') = 65 = 0b 0100 0001
- 768 = 0b 0000 0011 0000 0000



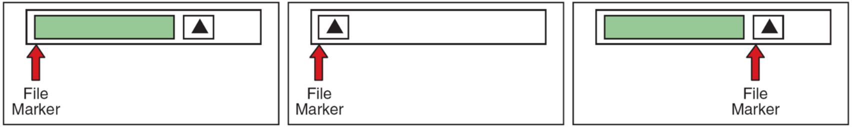
- Text files store data as a sequence of characters
- Binary files store data as they are stored in primary memory

File Modes

Mode	`r`	`w`	`a`	`r+`	`w+`	`a+`
Open state	read	write	write	read	write	write
Read allowed	yes	no	no	yes	yes	yes
Write allowed	no	yes	yes	yes	yes	yes
Append allowed	no	no	yes	no	no	yes
File must exist	yes	no	no	yes	no	no
Contents of existing file lost	no	yes	no	no	yes	no

For read/write of binary files, use '`b`' with one of the above modes

File Opening Modes

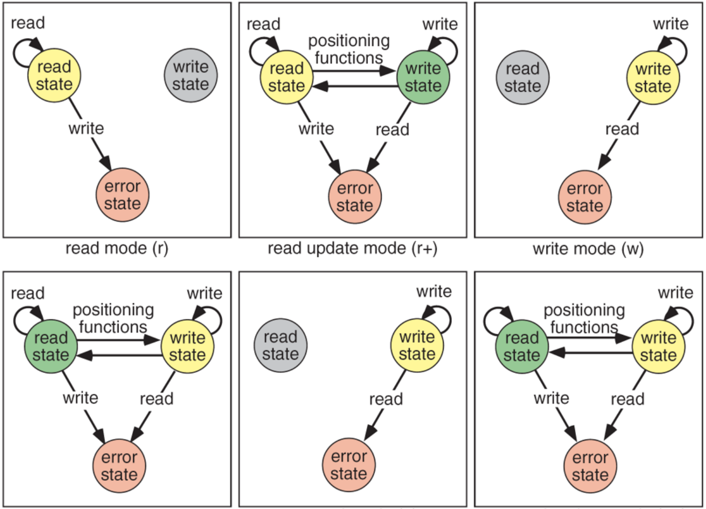


Read Mode (r, r+)

Write Mode (w, w+)

Append Mode (a, a+)

File States and Transitions



read mode (r)

read update mode (r+)

write mode (w)

write update mode (w+)

append mode (a)

append update mode (a+)

File Open and Close

- To write to or read from a file, we need to **open** it:

```
FILE *fopen(const char *filename, const char *mode);

// Each FILE object denotes a C stream and keeps the state during I/O
// filename: file name to associate the file stream to
// mode: null-terminated character string determining file access mode
```

  - If successful, returns a pointer to the new file stream. The stream is fully buffered unless filename refers to an interactive device.
  - On error, returns a null pointer
- On successful opening, we **write** and/or **read** data using I/O functions
- Once the write or read file over, we need to **close** it:

```
int fclose(FILE *stream); // stream: the file stream to close
```

  - Returns 0 on success, `EOF` otherwise // `EOF` is special End-of-File marker
  - Closes the file stream, flushes unwritten buffered data, and discards unread buffered data
  - The stream is no longer associated with a file, the buffer is disassociated and deallocated
  - The behavior is undefined if the value of the pointer stream is used after `fclose` returns

File Open and Close

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        // ...
        FILE* spTemps; // Declarations for file handler

        // ...

        if ((spTemps = fopen("TEMPS.DAT", "r")) == NULL) {
                printf("\aERROR opening TEMPS.DAT\n");
                exit(100);
        } // if open

        // Perform I/O

        if (fclose(spTemps) == EOF) {
                printf("\aERROR closing TEMPS.DAT\n");
                exit(102);
        } // if close

        // ...
} // main
```

Formatted I/O

Formatted I/O: File Write and Read

- To **write**, we use:

```
int printf(const char *format, ... );                // Writes to output stream stdout
int fprintf(FILE *stream, const char *format, ... ); // Writes to output stream stream
int sprintf(char *buffer, const char *format, ... ); // Writes to a string buffer

// format: A null-terminated multibyte string specifying interpretation of the data
// ...: arguments specifying data to print - a variadic function
// stream must be open before writing (stdout stays open) or reading (stdin stays open)
// buffer must be allocated before writing
```

  - If successful, number of characters transmitted to the output stream or number of characters written to buffer (not counting the terminating null character) is returned
  - A negative value is returned for an output error or an encoding error

- To **read**, we use:

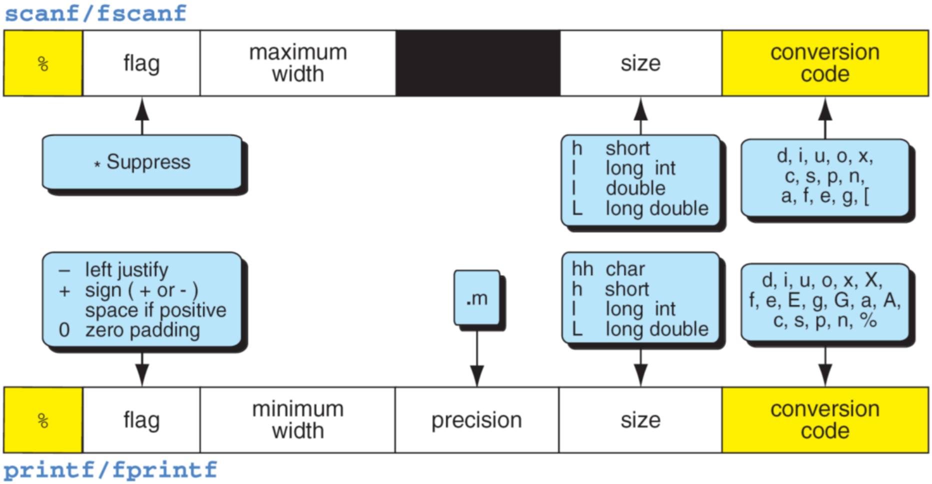
```
int scanf(const char *format, ... );                // Reads from input stream stdin
int fscanf(FILE *stream, const char *format, ... ); // Reads from input stream stream
int sscanf(char *buffer, const char *format, ... ); // Reads from a string buffer
```

  - Number of receiving arguments successfully assigned (which may be zero in case a matching failure occurred before the first receiving argument was assigned), or `EOF` if input failure occurs before the first receiving argument was assigned

Side Effect and Value of Formatted I/O Function

- `printf`, `fprintf` etc.
    - **Side Effect**
        - Converts internal data, as required, to strings of characters and writes the converted values to a file, which may be the standard output or error file
    - **Value**
        - Returns the number of characters written to the output file. In case of an error, it returns EOF
- `scanf`, `fscanf` etc.
    - **Side Effect**
        - Reads and converts a stream of characters from the input file, and stores the converted values in the list of variables found in the address list
    - **Value**
        - Returns the number of successful data conversions. If end of file is reached before any data are converted, it returns EOF

Format (Conversion) Specifications



scanf/fscanf

%	flag	maximum width		size	conversion code

flag: * Suppress

size:
- h short
- l long int
- l double
- L long double

conversion code: d, i, u, o, x, c, s, p, n, a, f, e, g, [

flag:
- – left justify
- + sign (+ or -)
- space if positive
- 0 zero padding

precision: .m

size:
- hh char
- h short
- l long int
- L long double

conversion code: d, i, u, o, x, X, f, e, E, g, G, a, A, c, s, p, n, %

%	flag	minimum width	precision	size	conversion code

printf/fprintf

Print Built-in Type Data

```
#include <stdio.h>

int main() {
    int i = 17;
    long l = 0x012a78cb; // 19560651
    long long unsigned int i64 = 0x012a78cb2597ac3d; // 84012356964166717
    float f = 15.0 / 7;
    double d = 15.0 / 7;
    char c = ’x’;
    const char *s = "ppd";
    int *p = &i;

    printf("%d\n", i);        // dec      // 17
    printf("%x\n", i);        // hex      // 11
    printf("%o\n", i);        // oct      // 21
    printf("%ld\n", l);       // long     // 19560651
    printf("%llu\n", i64);    // int 64   // 84012356964166717
    printf("%f\n", f);        // float    // 2.142857
    printf("%lf\n", d);       // double   // 2.142857
    printf("%c\n", c);        // char     // x
    printf("%s\n", s);        // string   // ppd
    printf("%p\n", p);        // pointer  // 0x7ffc28102988
}
```

Print User-defined Type Data

```
#include <stdio.h>

typedef struct Complex {
        double re, im;
} Complex;

int main() {
    Complex c1 = { 2.5, 7.3 }, c2 = { 4.3, 8.9 };

    printf("(%lf, %lf)", c1.re, c1.im);     // Need to print component-wise
    printf("; ");
    printf("(%lf, %lf)\n", c2.re, c2.im);   // Need to print component-wise

    printf("(%lf, %lf)", c1);   // warning: format '%lf' expects argument of type 'double',
                                // but argument 2 has type 'Complex' {aka 'struct Complex'}
                                // warning: format '%lf' expects a matching 'double' argument
    printf("; ");
    printf("(%lf, %lf)\n", c2); // Same as above

}
```

```
(2.500000, 7.300000); (4.300000, 8.900000)
(2.500000, 7.300000); (4.300000, 8.900000)
```

- MSVC++ does not produce the warnings

Flags, Sizes, and Conversion Code for printf family

Argument Type	Flag	Size Specifier	Code
integer	−, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	d, i
unsigned int	−, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	u
integer (octal)	−, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	o
integer (hex)	−, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	x, X
real	−, +, 0, #, space	none (double), l (double), L (double)	f
real (scientific)	−, +, 0, #, space	none (double), l (double), L (double)	e, E
real (scientific)	−, +, 0, #, space	none (double), l (double), L (double)	g, G
real (hex)	−, +, 0, #, space	none (double), l (double), L (double)	a, A
character	−	none (char), l (wchar_t)	c
string	−	none (char string), l (wchar_t string)	s
pointer			p
integer (for count)		none (int), h (short), l (long)	n
to print %			%

Flag Formatting Options

Flag Type	Flag Code	Formatting
Justification	none	right justified
	–	left justified
Padding	none	space padding
	0	zero padding
Sign	none	positive value: no sign negative value: –
	+	positive value: + negative value: –
	space	positive value: space negative value: –
Alternate	#	print alternative format for scientific, hexadecimal, and octal

Read Built-in Type Data

```
#include <stdio.h>
int main() {
    int i; long l;
    long long unsigned int i64; // For a 64-bit machine
    float f; double d;
    char c; char *s = (char*)malloc(10); // Space needs to be allocated to store the string to be read
    int *p;

    // Input shown in magenta and Output shown in gray

    scanf("%d\n", &i);     printf("%d\n", i);       // dec       // 17 17
    scanf("%x\n", &i);     printf("%x\n", i);       // hex       // 11 11
    scanf("%o\n", &i);     printf("%o\n", i);       // oct       // 21 21
    scanf("%ld\n", &l);    printf("%ld\n", l);      // long      // 19560651 19560651
    scanf("%llu\n", &i64); printf("%llu\n", i64);   // int 64    // 84012356964166717 84012356964166717
    scanf("%f\n", &f);     printf("%f\n", f);       // float     // 2.142857 2.142857
    scanf("%lf\n", &d);    printf("%lf\n", d);      // double    // 2.142857 2.142857
    scanf("%c\n", &c);     printf("%c\n", c);       // char      // x x

    // Used just 's', not &s, as it is a pointer
    scanf("%s\n", s);      printf("%s\n", s);       // string    // ppd ppd

    scanf("%p\n", &p);     printf("%p\n", p);       // pointer   // 008FFC0C 008FFC0C
}
```

Sizes and Conversion Code for scanf family

Argument Type	Size Specifier	Code
integral	hh (char), h (short), none (int), l (long), ll (long long) h (short), none (int), l (long). ll (long long)	i
integer	h (short), none (int), l (long), ll (long long)	d
unsigned int	hh (char), h (short), none (int), l (long), ll (long long)	u
character octal	hh (unsigned char)	o
integer hexadecimal	h (short), none (int), l (long), ll (long long)	x
real	none (double), l (double), L (double)	f
real (scientific)	none (double), l (double), L (double)	e
real (scientific)	none (double), l (double), L (double)	g
real (hexadecimal)	none (double), l (double), L (double)	a
character	none (char), l (wchar_t)	c
string	none (char string), l (wchar_t string)	s
pointer		p
integer (for count)	none (int), hh (char), h (short), l (long), ll (long long)	n
set	none (char), l (wchar_t)	[

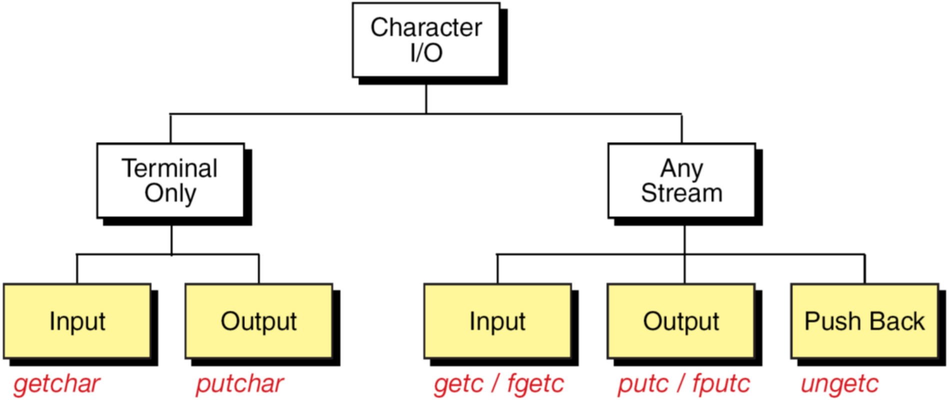
Checking scanf Results

```
#include <stdio.h>

#define FLUSH while (getchar() != ’\n’)
#define ERR1 "\aPrice incorrect. Re-enter both fields\n"
#define ERR2 "\aAmount incorrect. Re-enter both fields\n"

int main() {
    int amount;
    double price;
    int ioResult;

    // Read price and amount
    do {
        printf("\nEnter amount and price: ");
        ioResult = scanf("%d%f", &amount, &price);
        if (ioResult != 2) {
            FLUSH;
            if (ioResult == 1)
                printf(ERR1);
            else
                printf(ERR2);
        } // if
    } while (ioResult != 2);
}
```

Unformatted I/O

Unformatted I/O

Unformatted I/O: Character I/O



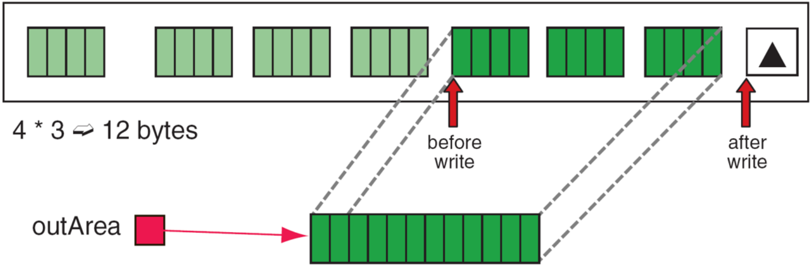
Create Text File

```
/* This program creates a text file */
#include <stdio.h>
int main() {
    FILE* spText;          // Stream
    int c, closeStatus;

    printf("This program copies input to a file.\n");
    printf("When you are through, enter <EOF>.\n\n");

    if (!(spText = fopen("My_New_Text_File.txt", "w"))) {
        printf("Error opening My_New_Text_File.txt for writing");
        return (1);
    } // if open

    while ((c = getchar()) != EOF) // Read characters from stdin. Use ^Z for EOF
        fputc(c, spText);          // Write characters to file

    closeStatus = fclose(spText);
    if (closeStatus == EOF) {
        printf("Error closing file\a\n");
        return 100;
    } // if

    printf("\n\nYour file is complete\n");
}
```

Direct Input/Output

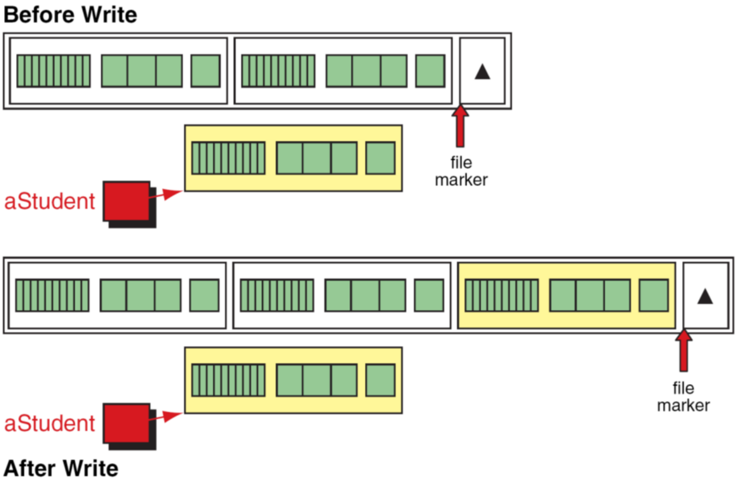
Direct Input/Output

File Write Operation

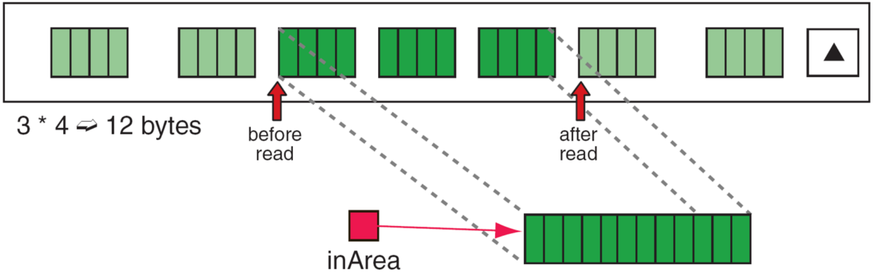


```
fwrite (outArea, sizeof (int), 3, spOut);
```

Writing a Structure

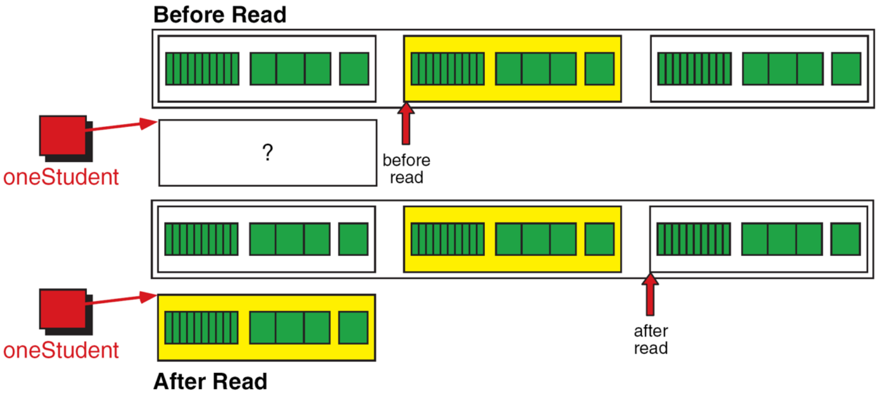


File Read Operation



```
fread (inArea, sizeof (int), 3, spData);
```

Reading a Structure



File Write / Read

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);

// buffer: pointer to the array where the read objects are stored
// size: size of each object in bytes
// count: the number of the objects to be read
```

- Writes count of objects from the given array buffer to the output stream stream. The objects are written as if by reinterpreting each object as an array of `unsigned char` and calling `fputc` `size` times for each object to write those unsigned chars into stream, in order. The file position indicator for the stream is advanced by the number of characters written
- Returns number of objects written successfully, which may be less than count if an error occurs

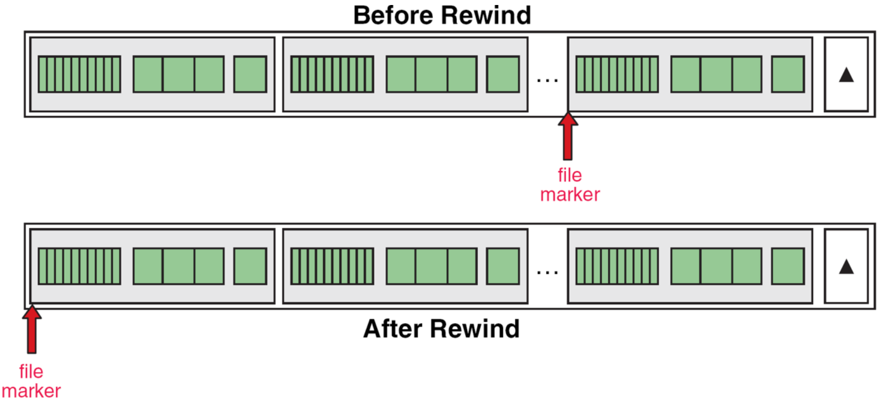
```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

- Reads up to count objects into the array buffer from the given input stream stream as if by calling `fgetc` `size` times for each object, and storing the results, in the order obtained, into the successive positions of buffer, which is reinterpreted as an array of `unsigned char`. The file position indicator for the stream is advanced by the number of characters read
- Returns number of objects read successfully, which may be less than count if an error or end-of-file condition occurs

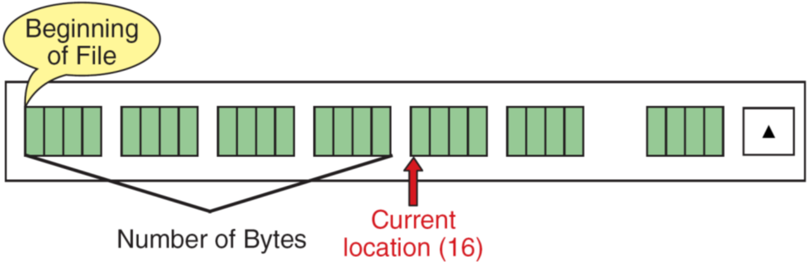
File Positioning

File Positioning

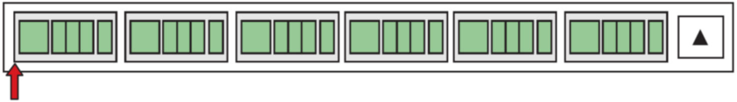
Rewind File

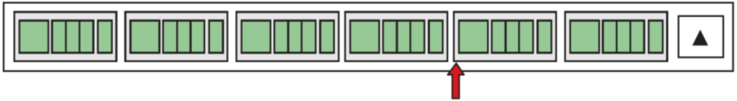


Current Location (ftell) Operation

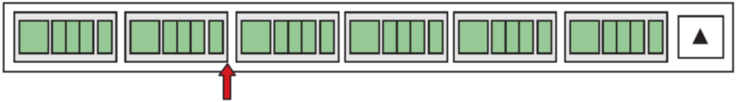


File Seek Operation

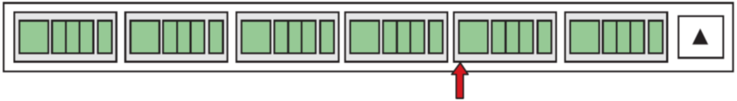




```
fseek (sp, 4 * sizeof(STRUCTURE_TYPE), SEEK_SET);
```



```
fseek (sp, - 4 * sizeof(STRUCTURE_TYPE), SEEK_END);
```



```
fseek (sp, 2 * sizeof(STRUCTURE_TYPE), SEEK_CUR);
```

File Positioning Function

```
long ftell(FILE *stream);
```

- Returns the file position indicator for the file stream `stream` on success or -1L if failure occurs
- If the stream is open in binary mode, the value obtained by this function is the number of bytes from the beginning of the file
- If the stream is open in text mode, the value returned by this function is unspecified and is only meaningful as the input to `fseek`()

```
int fseek(FILE *stream, long offset, int origin); // origin=SEEK_SET, SEEK_CUR, or SEEK_END
// offset: number of characters to shift the position relative to origin
// origin: position to which offset is added
```

- Sets the file position indicator for the file stream stream to the value pointed to by offset. Returns 0 upon success, nonzero value otherwise

```
void rewind(FILE *stream);
```

- Moves the file position indicator to the beginning of the given file stream.
- The function is equivalent to `fseek(stream, 0, SEEK_SET)`, except that `EOF` is cleared

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

Module Summary

- Discussed formatted and unformatted I/O using C Standard Library
- Discussed I/O with file and string