3.6 Binary Input / Output

Objectives

- To understand operations on text and binary files
- To understand the differences between text and binary files
- To write programs that read, write, and/or append text and binary files

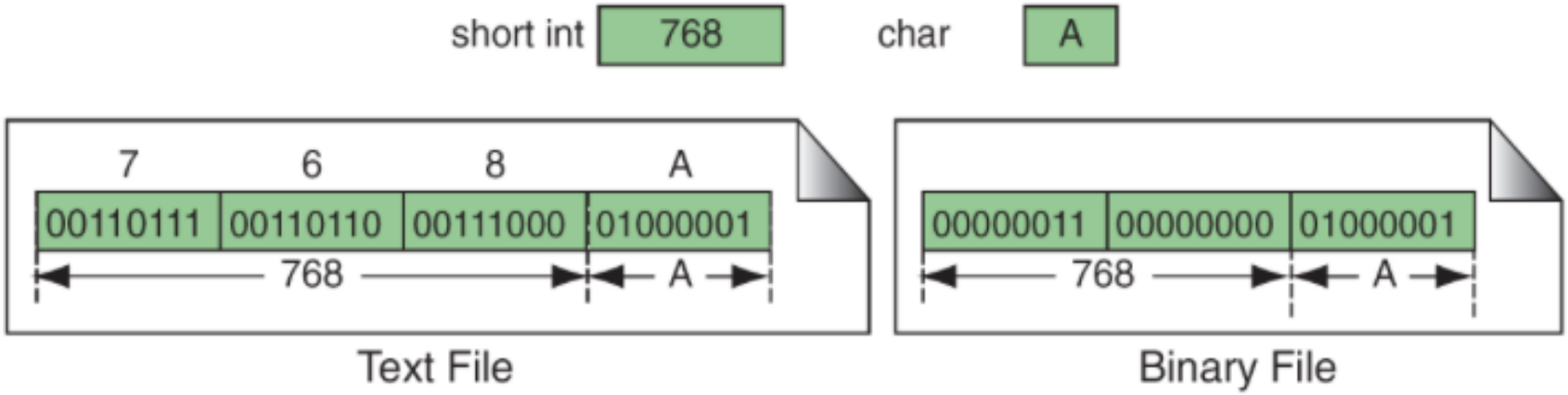
Text File vs Binary File

- Text files are divided into lines. Each byte in a text file represents a character, therefore readable by humans
- In binary files, each byte may not be a character.

NOTE :
Formatted input/output(printf,scanf) , character input/output(getchar, putchar), and string input/output(fgets, fputs) functions can be used only with text files.

An Example

- Text File stores “768” as three numeric characters and “A” as single character.
- Binary File stores “768” as a single short int character of 2 bytes and character “A” as a single character of size one byte.



- Note: Text files store data as a sequence of characters; binary files store data as they are stored in primary memory.

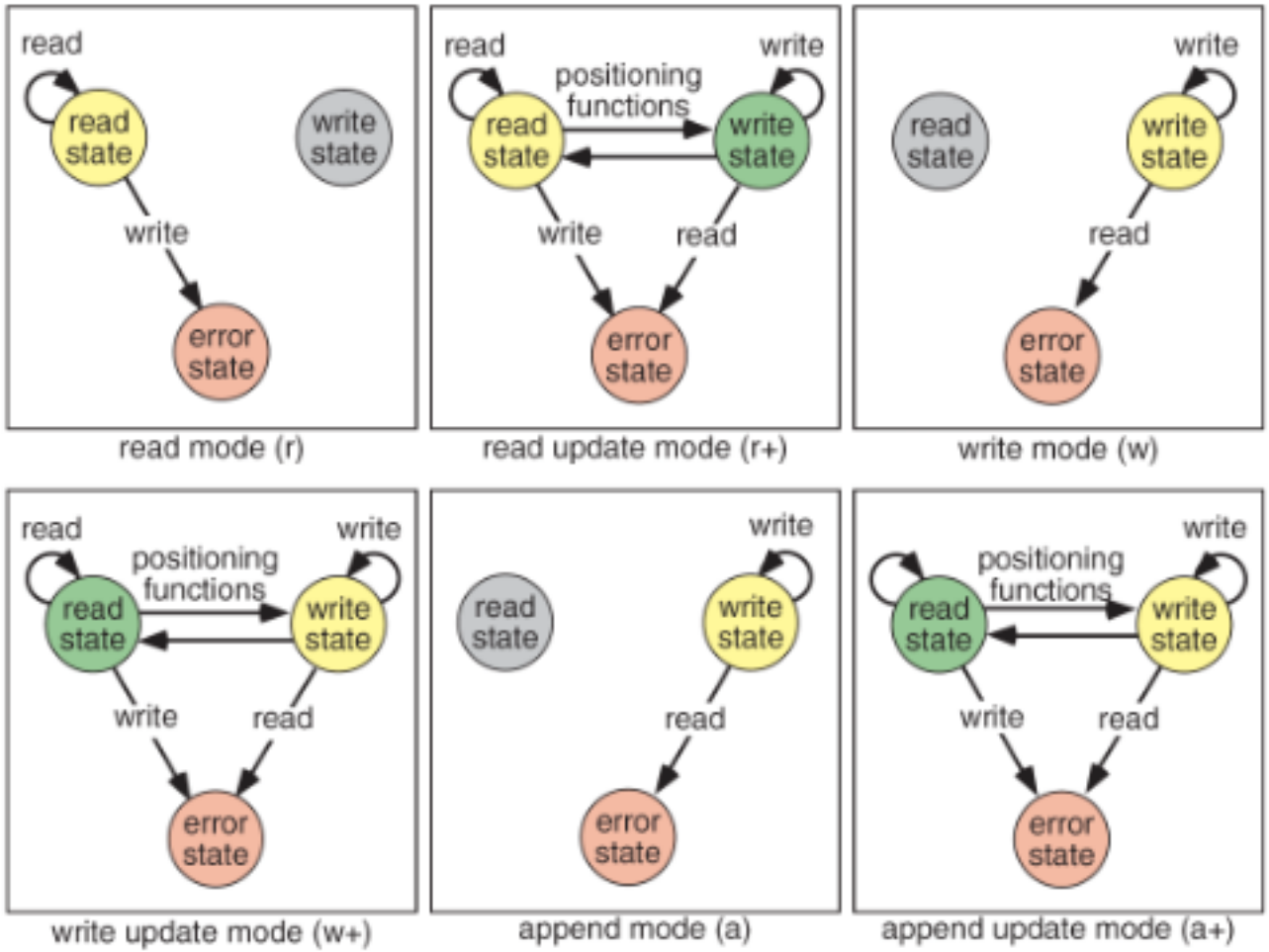
State of File

- Files can be opened in **Read**, **Write** or **Error** State
- To read from a file - open in Read state
- To write into a file - open in Write state
- Error state - Occurs when a read operation is attempted on a file opened in Write state or vise versa.

Note: A file opened in "read" state must already exist. Otherwise the open fails.

- Files can be opened in various states based on the file mode statements

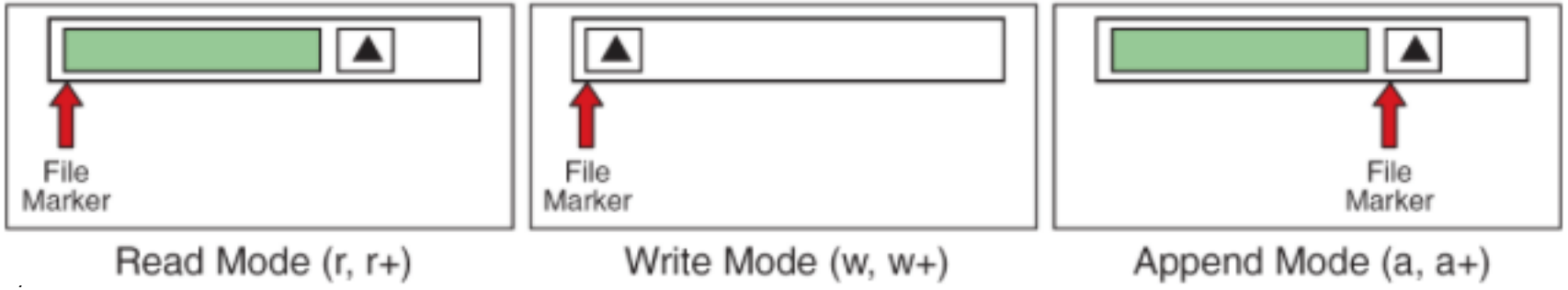
File Modes


read
read state
write state
write
error state
read mode (r)
read
write
positioning functions
read state
write state
write
read
error state
read update mode (r+)
write
read state
write state
read
error state
write mode (w)
read
write
positioning functions
read state
write state
write
read
error state
write update mode (w+)
write
read state
write state
read
error state
append mode (a)
write
positioning functions
read state
write state
write
read
error state
append update mode (a+)

File Modes Continued.

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

- Position of the EOF marker for each mode



File operations

- Open a file: `fopen()`
- Close a file: `fclose()`
- Read a binary file: `fgetc(), getc(), fgets(), puts(), fscanf(), fread()`
- Write a binary file: `fputc(), putc(), fputs(), puts(), fprintf(), fwrite()`
- File positioning: `fseek(), ftell(), fgetpos(), fsetpos()`

fopen()

```
FILE *fopen(char *name, char *mode);
void fclose(FILE* stream)
```

- fopen returns a pointer to a `FILE`.
- mode can be
  - “r” - read
  - “w” - write
  - “a” - append
  - “b” can be appended to the mode string to work with binary files. For example, “rb” means reading binary file.

Text Files – fprintf()

```
int fprintf(FILE *fp, char *format, ...);
```

Similar to printf.

On success, the total number of characters written is returned.

If a writing error occurs, a negative number is returned.

Example – create and write to a file

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char name[10];
  double balance;
  int account;
  if ((fp = fopen("clients.dat", "w")) == NULL) {
       printf("File could not be opened\n");
  }
  else {
       printf("Enter one account, name, and balance.\n");
       scanf("%d%s%lf", &account, name, &balance);
       fprintf(fp, "%d %s %.2f\n", account, name, balance);
       fclose(fp);
  }
  return 0;
}
```

Text Files – fscanf()

```
int fscanf(FILE *fp, char *format, ...);
```

•Similar to scanf.

•On success, return the number of items successfully filled. This count can match the expected number of items or be less (even zero) due to a matching failure, a reading error, or the reach of the *end-of-file*. And, if *end-of-file* happens before any data could be successfully read, EOF is returned.

```
int feof(FILE *fp);
```

•return 1/0 end-of-file is reached

Text Files – read from a file

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char name[10];
  double balance;
  int account;
  if ((fp = fopen("clients.dat", "r")) == NULL) {
       printf("File could not be opened\n");
  }
  else {
       fscanf(fp, "%d%s%lf", &account, name, &balance);
       printf("%d %s %.2f\n", account, name, balance);
       fclose(fp);
  }
  return 0;
}
```

Text Files – character I/O

```
int fputc(int c, FILE *fp);
int putc(int c, FILE *fp);
```

Write a character c to a file. putc() is often implemented as a MACRO (hence faster). On success, the character written is returned. If a writing error occurs, EOF is returned

```
int fgetc(FILE *fp);
int getc(FILE *fp);
```

Read a character c to a file. getc() is often implemented as a MACRO (hence faster). On success, the character read is returned. If a read error occurs, EOF is returned.

Text Files – character I/O

```
#include <stdio.h>
int main()
{
  FILE *source_fp, *dest_fp;
  int ch;

  if ((source_fp = fopen("source.txt", "r")) == NULL)
        printf("cannot open source file\n");
  if ((dest_fp = fopen("dest.txt", "w")) == NULL)
        printf("cannot open dest file\n");
  while ((ch = getc(source_fp)) != EOF)
        putc(ch, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Text Files – character I/O

```
int ugetc(int c, FILE *fp);
```

Push back a character read from a file pointer.

```
int feof(FILE *fp);
```

return 1/0 *end-of-file* is reached

Text Files – character I/O

```
#include <stdio.h>
int main ()
{
  FILE * fp;
  int c;
  fp = fopen ("source.txt","r");
  if (fp == NULL) {
    printf("cannot open the file\n");
    return 0;
  }
  while (!feof (fp)) {
    c = getc(fp);
    if (c == '#')
        ungetc('@', fp);
    else
        putc(c, stdout); // stdout is the screen
  }
  return 0;
}
```

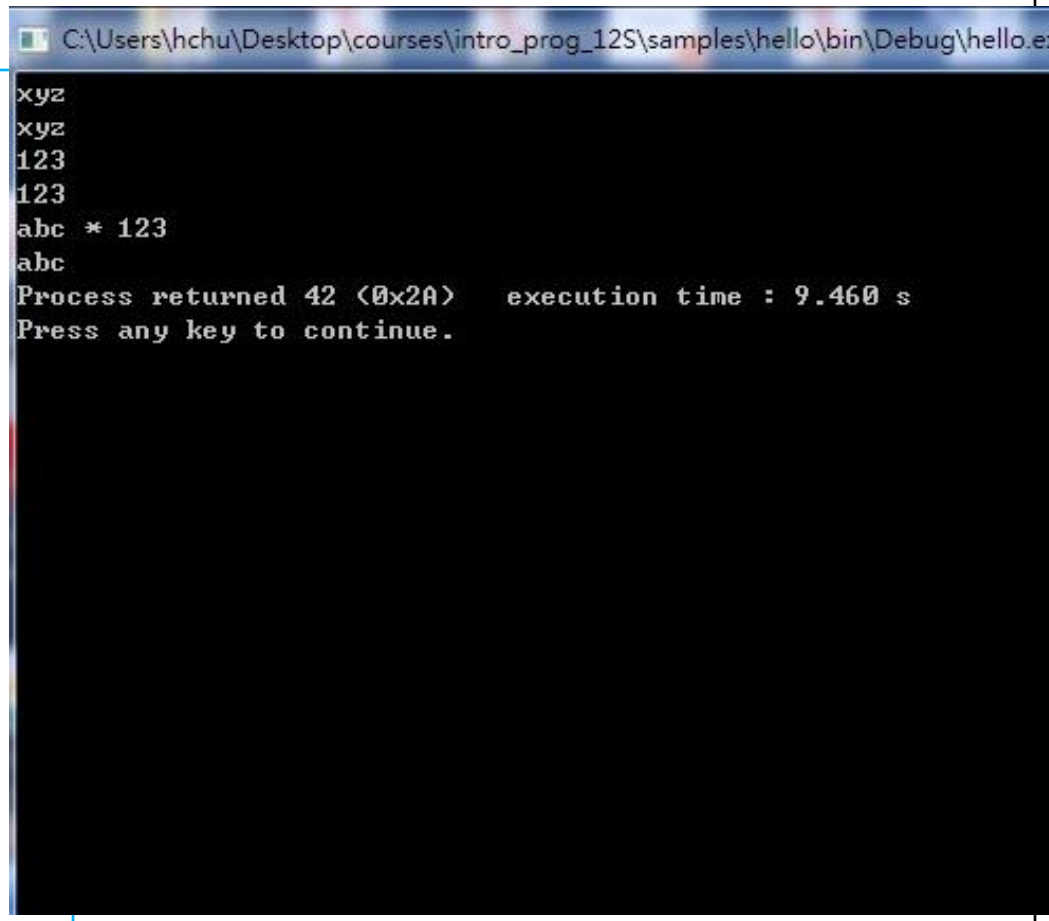
Text Files – standard input & output

- `FILE *stdin  // screen input as a file`
- `FILE *stdout // screen output as a file`

Text Files – stdin, stdout

```
#include <stdio.h>

int main ()
{
    int c;
    while ((c =
  fgetc(stdin)) != '*')
    {
        fputc(c, stdout);
    }
}
```



Text Files – Line I/O

```
int fputs(const char *s, FILE *fp);
```

Write a line of characters to a file. On success, a non-negative value is returned. On error, the function returns EOF.

```
char* fgets(char *s, int n, File *fp);
```

Read characters from a file until it reaches the first new-line or (n-1) characters, in which it places the NULL character ('\0') at the end of the string.

On success, the function returns s. If the end-of-file is encountered before any characters could be read, the pointer returned is a NULL pointer (and the contents of s remain unchanged).

Text Files – Line I/O

```
#include <stdio.h>
int main()
{
  FILE *source_fp, *dest_fp;
  char s[100];

  if ((source_fp = fopen("source.txt", "r")) == NULL)
        printf("cannot open source file\n");
  if ((dest_fp = fopen("dest.txt", "w")) == NULL)
        printf("cannot open dest file\n");

  while (fgets(s, 100, source_fp) != NULL)
        fputs(s, dest_fp);

  fclose(source_fp);
  fclose(dest_fp);
  return 0;
}
```

Opening Binary Files

- Syntax:

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

- The six binary file modes are

Examples

- The following are examples of open statements for binary files
  - // Write into a binary file
    *FILE *fpWriteBin;*
    fpWriteBin=fopen("c:\\myFile.bin", "wb");
  - // Read from a binary file
    fpReadBin = fopen ("myFile.bin", "rb");
  - // Write with update
    fpWriteUpdateBin = fopen ("myFile.bin", "w+b");
  - // Append to a binary file
    fpApndBin = fopen ("myFile.bin", "ab");

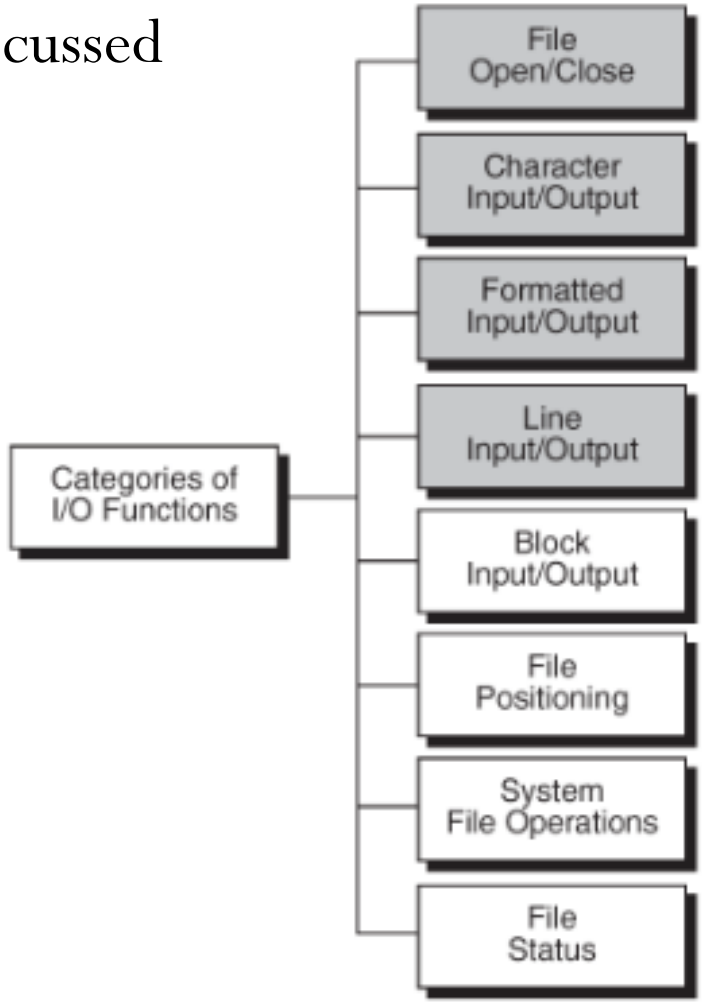
Closing a Binary File

- This is the same as closing a text file

```
int fclose (fpReadBin);
```

Standard Library Functions for Files

- C has eight categories of standard file library functions.
- The first 4 are already discussed



File operations

- Open a file: `fopen()`
- Close a file: `fclose()`
- Read a binary file: `fgetc(), getc(), fgets(), puts(), fscanf(), fread()`
- Write a binary file: `fputc(), putc(), fputs(), puts(), fprintf(), fwrite()`
- File positioning: `fseek(), ftell(), fgetpos(), fsetpos()`

1. Block I/O Functions

- Used to read and write data into binary files as discussed previously.
- With the exception of character data, we cannot "see" data in binary files. If opened in an editor it looks like hieroglyphics.
- This is because there are no format conversions .
- ***fread( )*** and ***fwrite( )*** are the block read and write functions.

1. Block I/O Functions Continued.

- File Read : *fread( )*
  - reads a specified number of bytes from a binary file and places them into memory at the specified location.
  - Syntax:

```
int fread( void* pInArea,
           int elementSize,
           int count,
           FILE* sp);
```

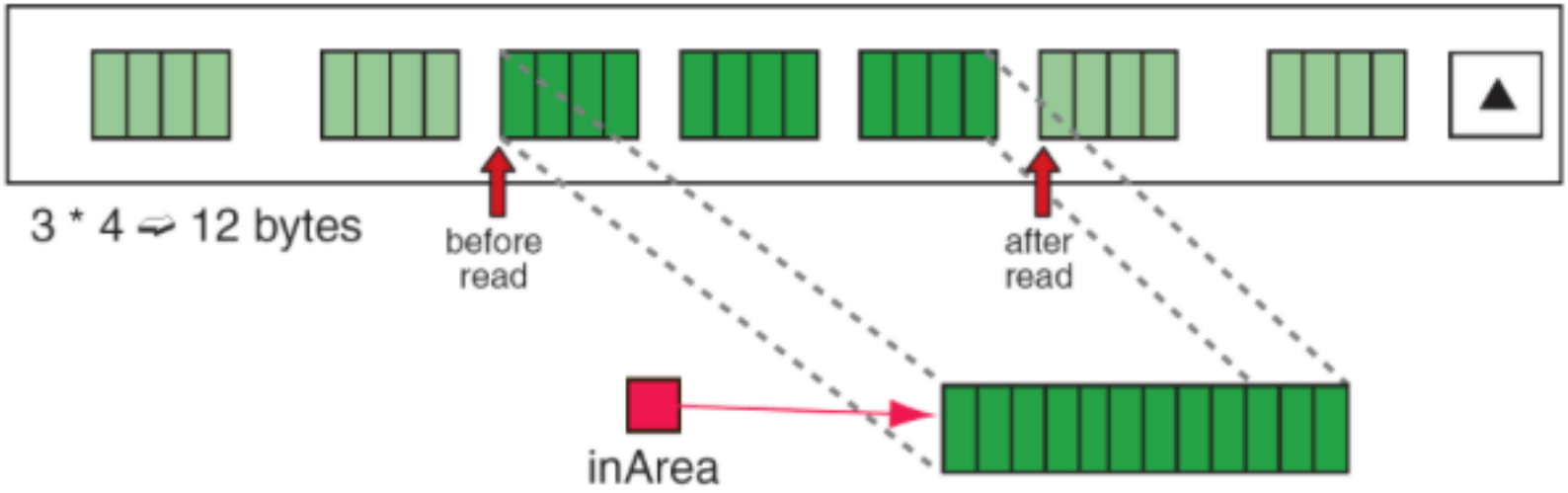
pInArea is a pointer to the input area in memory

elementSize & count are multiplied to determine the amount of data to be transferred.

the last parameter (sp)is the associated stream

1. Block I/O Functions – fread( ).

- An example of a file that reads data into an array of integers




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

- fread() transfers the next 3 integers from the file(spData) to the array(inArea)
- If only 2 integers are left to read fread() will return 2

1. Block I/O Functions – fread( ).

- Program: read a file of integers

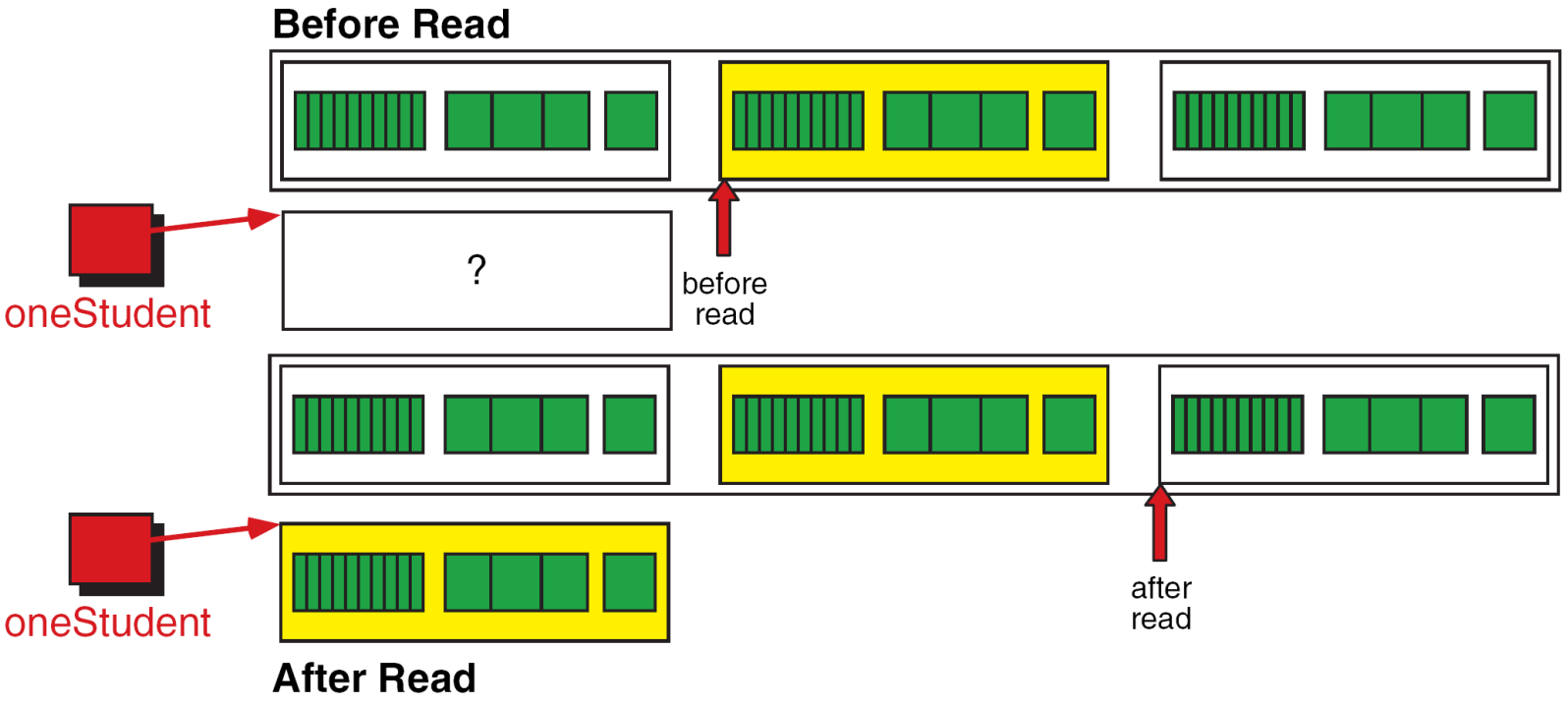
```
 1  // Read a file of integers, three integers at a time.
 2  {
 3     …
 4  // Local Declarations
 5     FILE* spIntFile;
 6     int   itemsRead;
 7     int   intAry[3];
 8
 9  // Statements
10     spIntFile = fopen("int_file.dat", "rb");
11      …
12     while ((itemsRead = fread(intAry,
13              sizeof(int), 3, spIntFile)) !=  0)
14        {
15         // process array
16              …
17        } // while
18     …
19  }  // block
```

1. Block I/O Functions – fread( ).

- fread( ) returns the number of items read
- In the previous figure it will range from 0-3
- A more common use: - Reading Structures (records)
- Advantage: Structures contain a variety of data (string, int, float,etc.)
  - Block I/O functions can transfer data one structure(record) at a time
  - No need to format data

1. Block I/O Functions – fread( ).

- Reading a Structure



1. Block I/O Functions continued.

- Program: Read Student File

```
 1  /* Reads one student's data from a file
 2        Pre   spStuFile is opened for reading
 3        Post  stu data structure filled
 4              ioResults returned
 5  */
 6  int readStudent (STU* oneStudent, FILE* spStuFile)
 7  {
 8  // Local Declarations
 9     int ioResults;
10
11  // Statements
12     ioResults = fread(oneStudent,
13                       sizeof(STU), 1, spStuFile);
14     return ioResults;
15  } // readStudent
```

1. Block I/O Functions – fwrite( ).

- Writes a specified number of items to a binary file
- Syntax

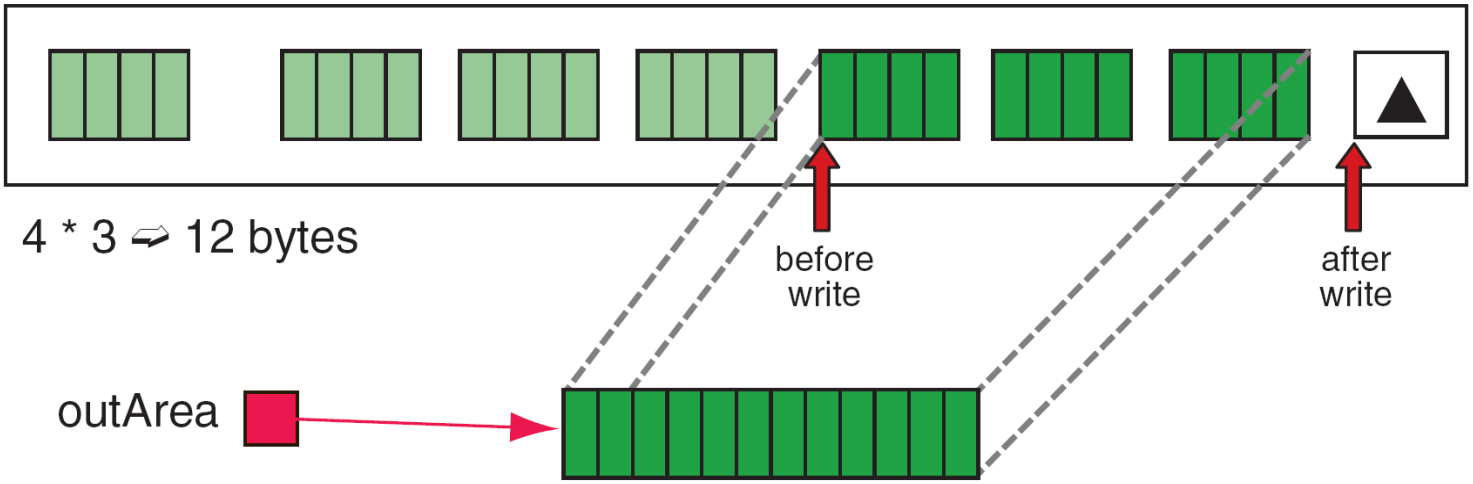
```
int fwrite (void*  pOutArea,
            int    elementSize,
            int    count,
            FILE*  sp);
```

Fwrite copies elementSize x count bytes from the address specified by pOutArea to the file

- The parameters for file write correspond exactly with the parameters for fread( ).
- It returns the number of items written.
- Eg: if it writes 3 integers, it returns 3
- If the number of items written is fewer than count, then error

1. Block I/O Functions – fwrite( ).

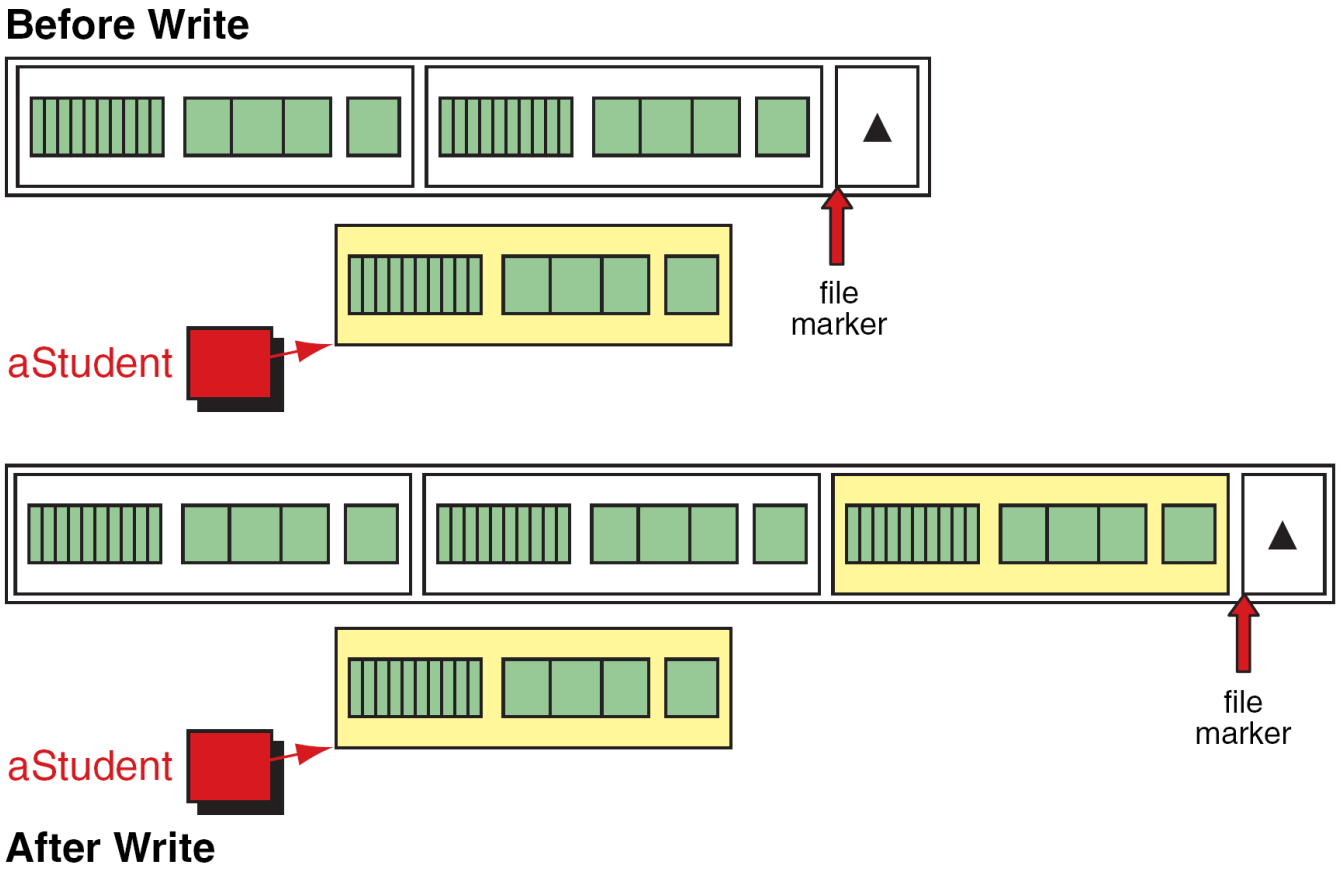
- File Write Operation




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

1. Block I/O Functions – fwrite( ).

- Writing a Structure



1. Block I/O Functions – fwrite( ).

- Program: Writing Structured Data

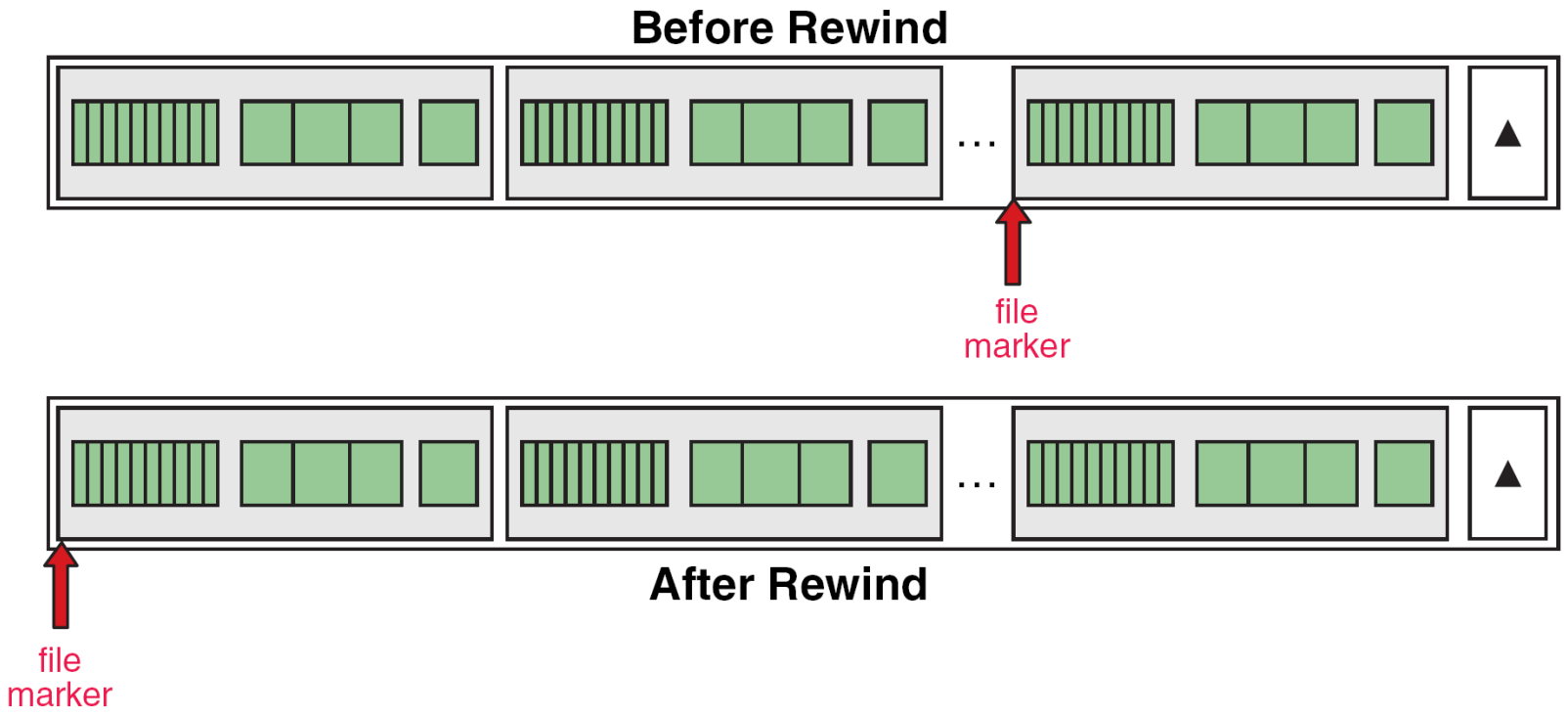
```
 1  /* Writes one student's record to a binary file.
 2        Pre  aStudent has been filled
 3             spOut is open for writing
 4        Post aStudent written to spOut
 5  */
 6  void writeStudent (STU* aStudent, FILE* spOut)
 7
 8  {
 9  // Local Declarations
10     int ioResult;
11
12  // Statements
13     ioResult = fwrite(aStudent,
14                       sizeof(STU), 1, spOut);
15     if (ioResult != 1)
16        {
17         printf("\a Error writing student file \a\n");
18         exit (100);
19        } // if
20     return;
21  }  // writeStudent
```

2. File Positioning

- These have 2 uses:
  - For randomly processing data in disk files – position the file to read the desired data.
  - Change a file state (from read to write, etc.)
- There are 3 functions
  - Rewind – *rewind( )*
  - Tell location – *ftell( )*
  - File seek – *fseek( )*

2. File Positioning – rewind( )

- Sets the file position indicator to beginning of file



2. File Positioning – rewind( )

- Syntax:

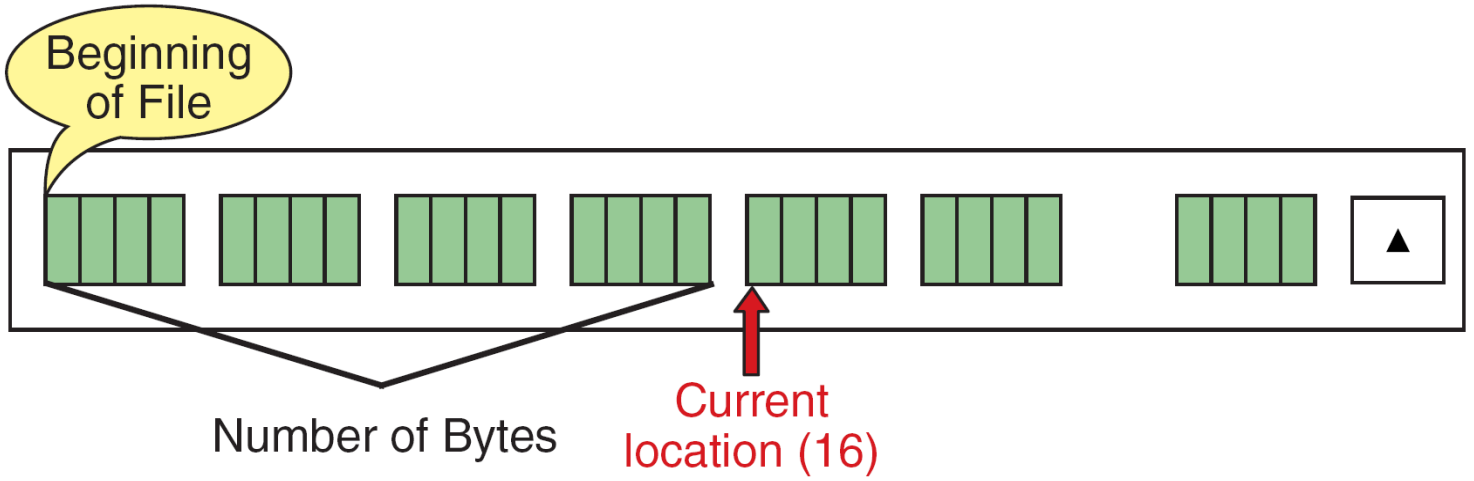
```
void rewind( FILE* stream);
```

- A common use: change a work file from write state to read state
- NOTE: recollect that to read and write a file with one open, we must open it in update mode(w+ or w+b).
- Though the same can be achieved by closing and reopening a file, rewinding is a faster operation.

2. File Positioning – ftell( )

- Reports current position of file marker in file relative to the beginning of the file
- Measures position in bytes starting from zero
- Returns a long integer
- Syntax:

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



2. File Positioning – ftell( )

- To find number of structures relative to first structure, it must be calculated.
- Example

  numChar = ftell(sp);

  numStruct = numChar / sizeof(STRUCTURE_TYPE);
- Here each structure is 4 bytes. If ftell() returns 16, it implies that there are 4 structures before this one.
- If ftell() encounters an error it returns “-1”.
- There are 2 conditions for error
  - Using ftell() with a device that cannot store data
  - When position is larger than long int

3. File Positioning – fseek( )

- Positions the file location indicator to the specified byte position in the file
- Syntax:

```
int fseek( FILE* stream, long offset, int wherefrom);
```

- The first parameter is a pointer to the open file(either rea/write)
- Second parameter is a signed integer that specifies the number of bytes : absolutely or relatively.

3. File Positioning – fseek( )

- C provides 3 named constants that specify the start position (wherefrom) of the seek operation
  - #define SEEK_SET 0
  - #define SEEK_CUR 1
  - #define SEEK_END 2
- When wherefrom is SEEK_SET or 0, the offset is measured absolutely from the beginning of the file
- Example: to set file indicator to byte 100 on a file the syntax is

```
fseek(sp, 99L, SEEK_SET);
```

3. File Positioning – fseek( )

- When wherefrom is set to SEEK_CUR or 1, the displacement is calculated from the current file position.
- If the displacement is negative, the file position moves to the beginning of the file and if displacement is positive, it moves towards the end of the file.
- It is an error to move beyond the beginning of a file
- The file is extended if the marker moves beyond the end of the file but the contents of the extended bytes are unknown.
- Example: To position the file marker to the next record in a structured file

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

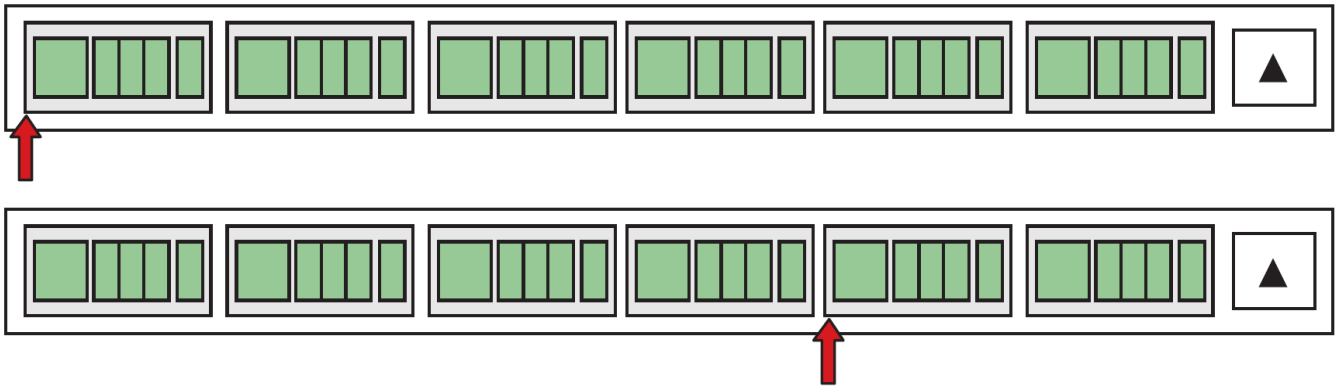
3. File Positioning – fseek( )

- If wherefrom is SEEK_END or 2, the file locator indicator is positioned relative to the end of the file.
- This is used to write a record to the end of the file as shown below

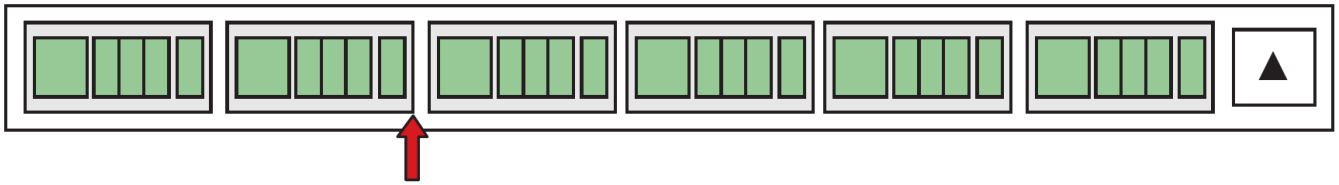
```
fseek(stuFile,0L,SEEK_END);
```

- This returns 0 if the positioning is successful and returns a non zero value if unsuccessful.

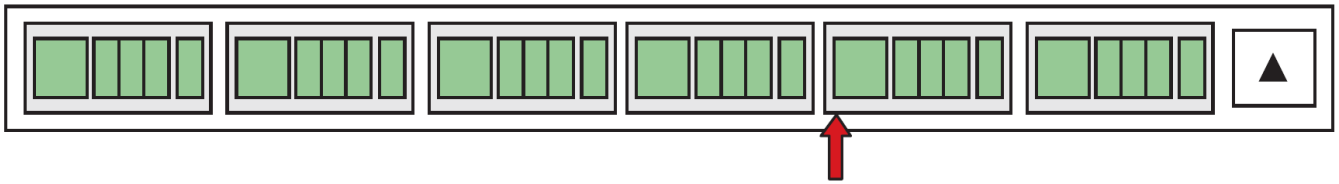
3. File Positioning – fseek( )



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



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



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

Example: Program to Append 2 Binary Files

```
17  // Statements
18     printf("This program appends two files.\n");
19     printf("Please enter file ID of the primary file: ");
20     scanf("%12s", fileID);
21     if (!(sp1 = fopen (fileID, "ab")))
22         printf("\aCan't open %s\n", fileID), exit (100);
23
24     if (!(dataCount = (ftell (sp1))))
25        printf("\a%s has no data\n", fileID), exit (101);
26     dataCount /= sizeof(int);
27
28     printf("Please enter file ID of the second file: ");
29     scanf("%12s", fileID);
30     if (!(sp2 = fopen (fileID, "rb")))
31        printf("\aCan't open %s\n", fileID), exit (110);
32
33     while (fread (&data, sizeof(int), 1, sp2) == 1)
34        {
35         fwrite (&data, sizeof(int), 1, sp1);
36         dataCount++;
37        } // while
```

Example: Program to Append 2 Binary Files Cont.

```
38
39      if (! feof(sp2))
40         printf("\aRead Error. No output.\n"), exit (120);
41
42      fclose (sp1);
43      fclose (sp2);
44
45      printf("Append complete: %ld records in file\n",
46              dataCount);
47      return 0;
48  }   // main
```