



Chapter 9: File Processing

Introduction to Computer Programming
(C language)

Nguyễn Tiến Thịnh, Ph.D.

Email: ntthinh@hcmut.edu.vn

Course Content

- ❑ C.1. Introduction to Computers and Programming
- ❑ C.2. C Program Structure and its Components
- ❑ C.3. Variables and Basic Data Types
- ❑ C.4. Selection Statements
- ❑ C.5. Repetition Statements
- ❑ C.6. Functions
- ❑ C.7. Arrays
- ❑ C.8. Pointers
- ❑ **C.9. File Processing**

References

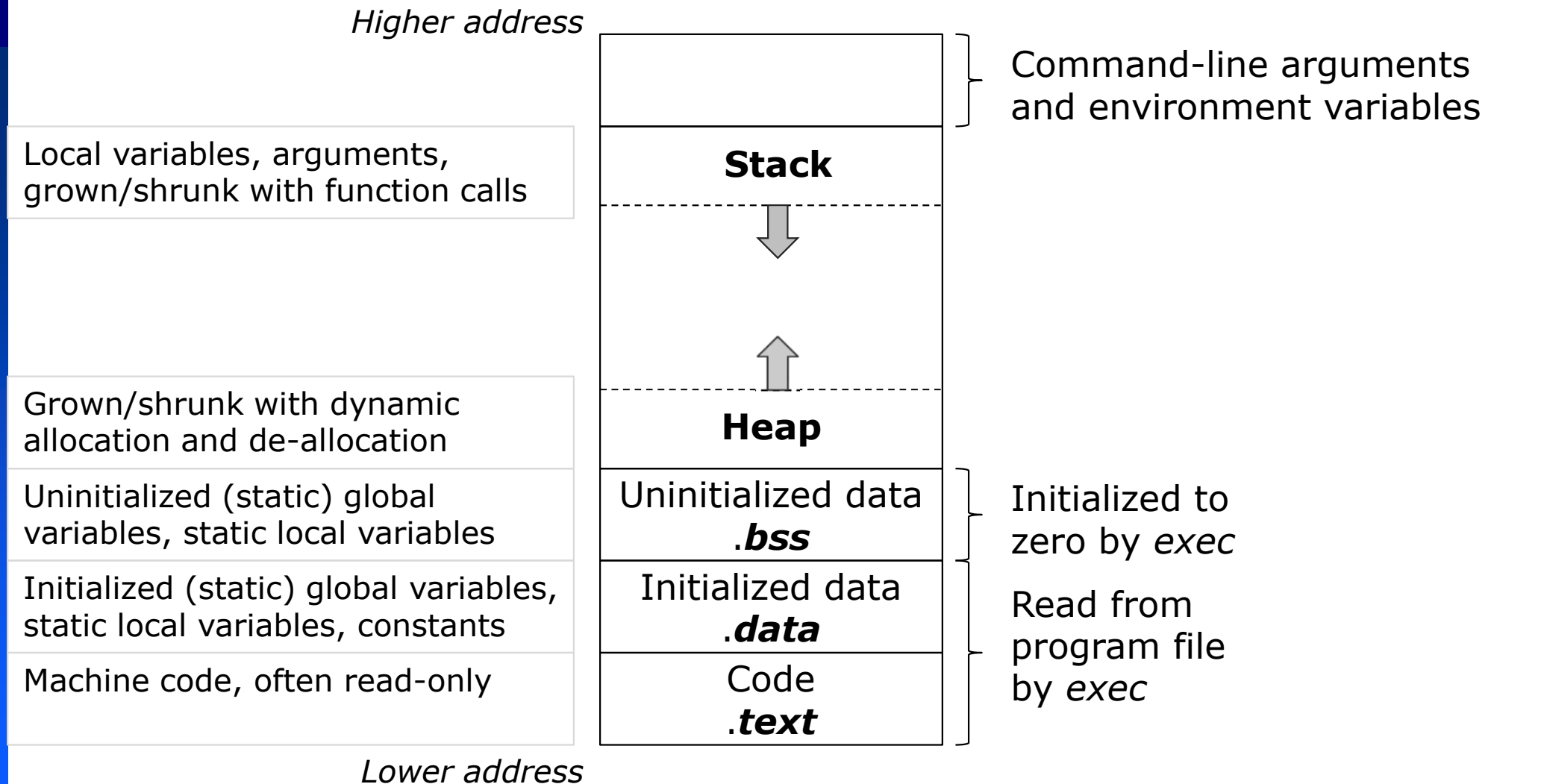
- ▣ [1] "*C: How to Program*", 7th Ed. – Paul Deitel and Harvey Deitel, Prentice Hall, 2012.
- ▣ [2] "*The C Programming Language*", 2nd Ed. – Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1988
- ▣ and others, especially those on the Internet

Content

- ▣ Introduction
- ▣ Declare files
- ▣ Open and close files
- ▣ Store and retrieve data from files
- ▣ Use macros
- ▣ Summary

Recall – Chapter 3 and Chapter 8

□ Memory layout of a C program

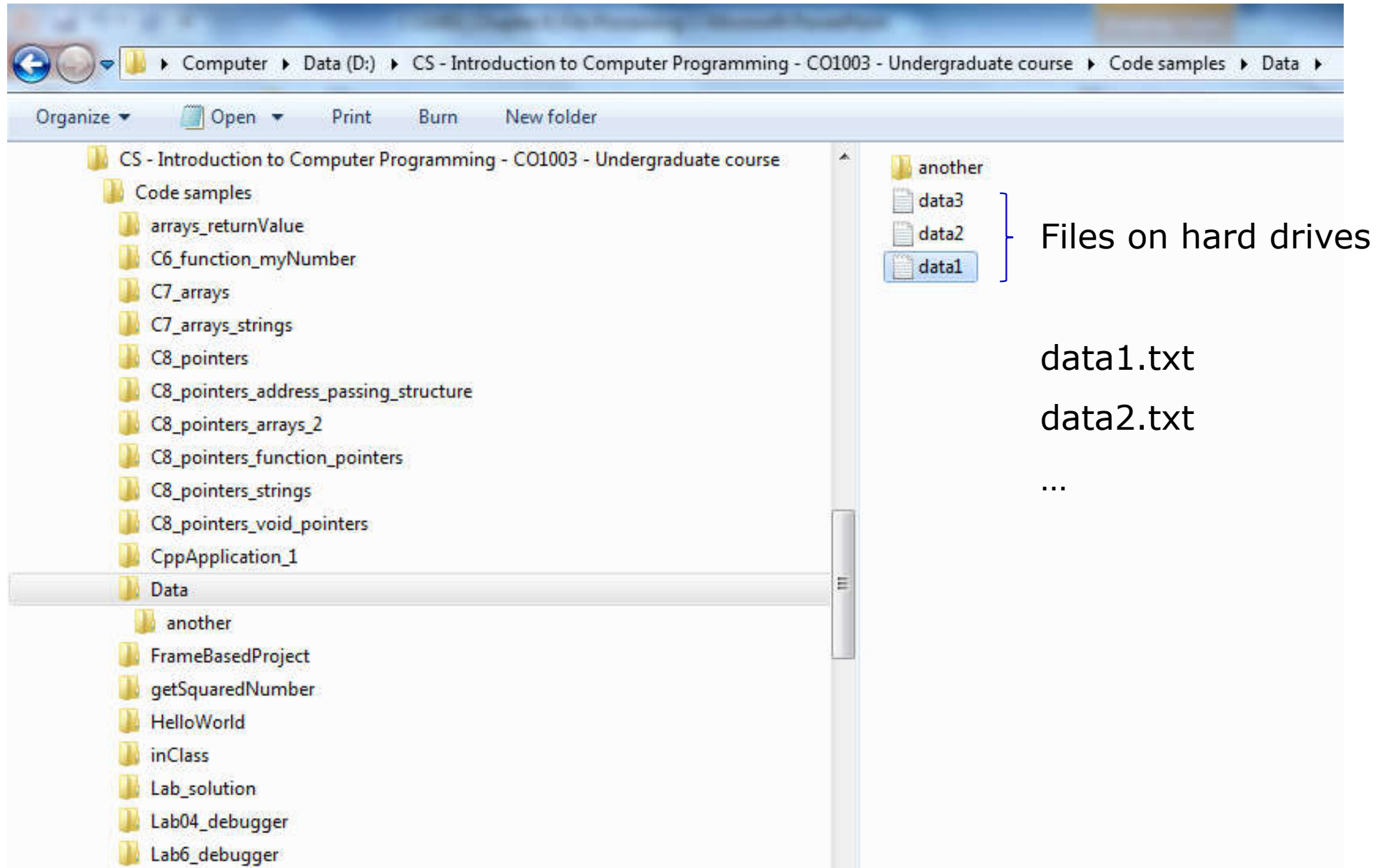


bss = block started by symbol, better save space

Introduction

- Data (input, output, supporting) in memory
 - Temporary as lost when a program terminates !!!
- Can we have permanent data before and after program execution?
- Files on secondary storage devices (hard drives, CDs, DVDs, flash drives, ...)

Introduction



Introduction

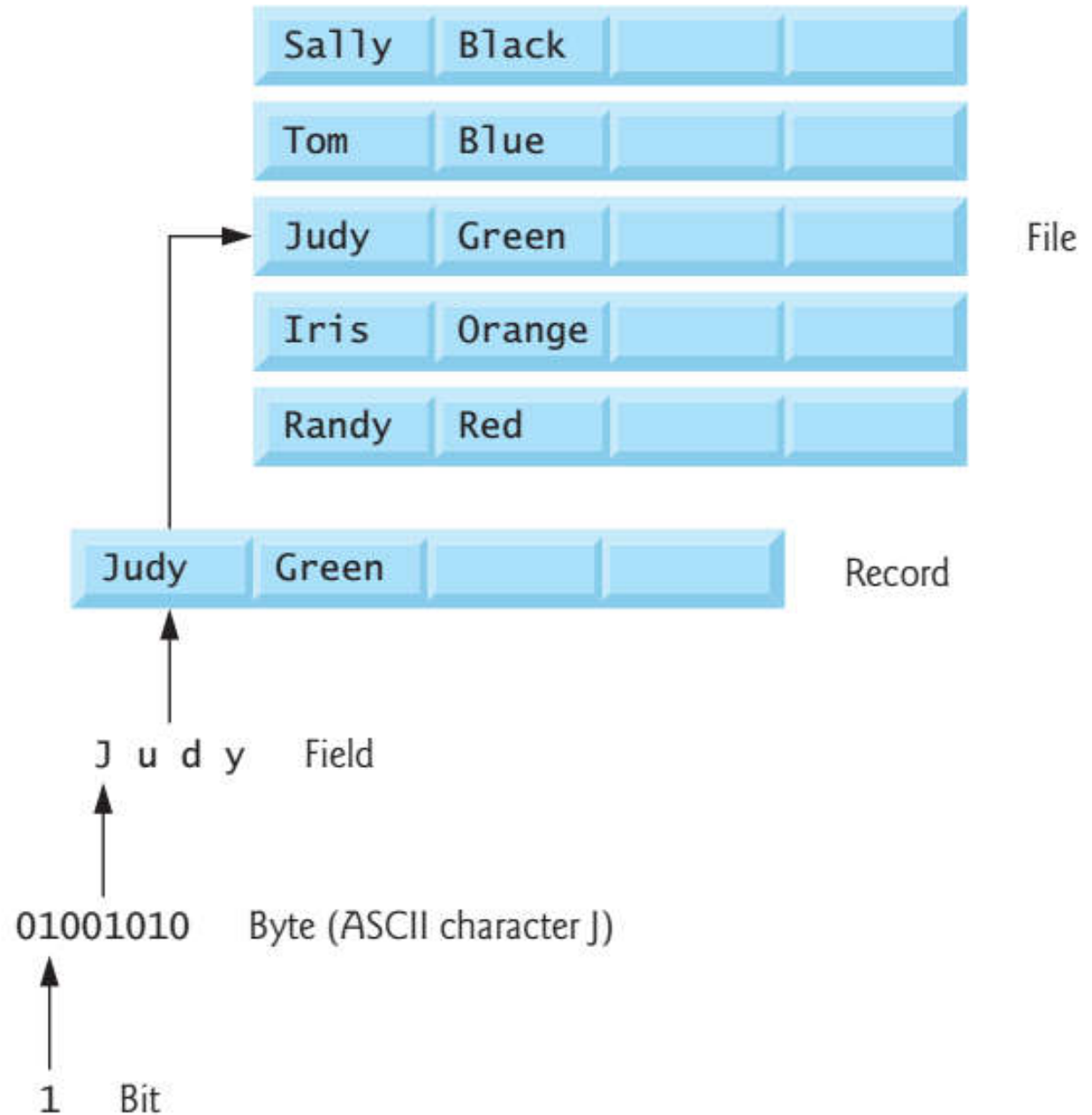


Figure 11.1. Data hierarchy
[1], pp. 419

Introduction

- ❑ C views each file as a sequential stream of bytes.
- ❑ Each file ends either with an *end-of-file* marker or at a specific byte number recorded in a system-maintained, administrative data structure.
 - *end-of-file*: ctrl-z (Windows), ctrl-d (Linux/Mac OS X/Unix)
- ❑ When a file is opened, a stream is associated with the file.
- ❑ Streams provide communication channels between files and programs.

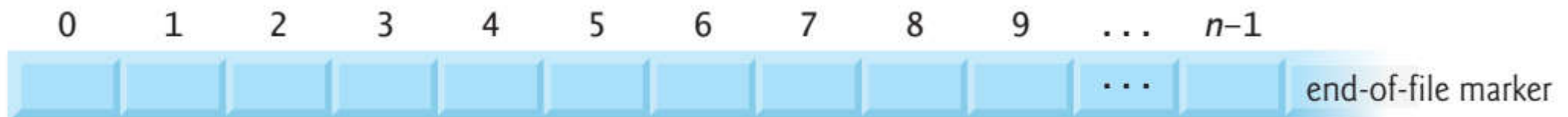


Figure 11.2. C's view of a file of n bytes
[1], pp. 420

Introduction

- C views each file as a sequential stream of bytes.
 - Fixed-length records stored in a random-access file
 - The exact location of a record relative to the beginning of the file

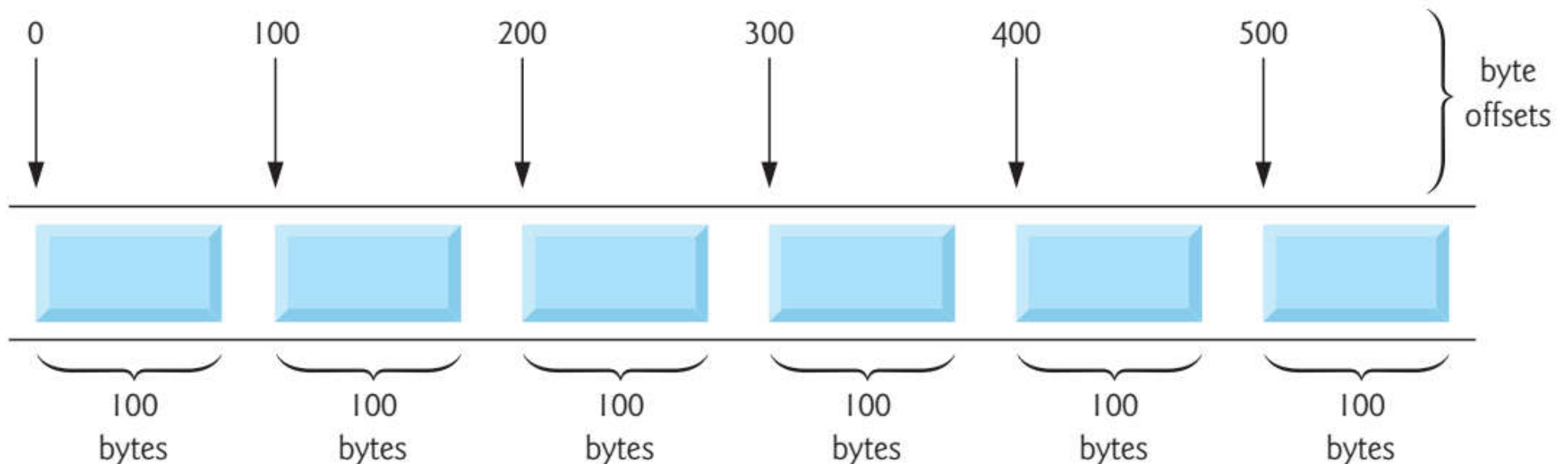


Figure 11.10: C's view of a random-access file
[1], pp. 431

Introduction

□ Operations on files

- Create
- Open
- Read
- Write (write a new file, append an existing file)
- Close

□ Access to files in C for processing

- Sequential access to a sequential stream of *bytes*
- Random access to a stream of fixed-size *records*

Introduction

- The standard library: `<stdio.h>`

- ▣ `FILE *fopen(const char *filename, const char *mode)`
- ▣ `FILE *freopen(const char *filename, const char *mode, FILE *stream)`
- ▣ `int fclose(FILE *stream)`
- ▣ `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`
- ▣ `int fgetc(FILE *stream)`
- ▣ `char *fgets(char *str, int n, FILE *stream)`
- ▣ `int fscanf(FILE *stream, const char *format, ...)`
- ▣ `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
- ▣ `int fputc(int char, FILE *stream)`
- ▣ `int fputs(const char *str, FILE *stream)`
- ▣ `int fprintf(FILE *stream, const char *format, ...)`

Declare files

- Each file communicated with C program via a stream controlled by a pointer of FILE

```
FILE* identifier;
```

```
FILE* pFile1;
```

```
FILE* pFile2 = 0;
```

```
FILE* pFile3 = NULL;
```

identifier: a valid identifier for a pointer pointing to a file stream

FILE: an object type suitable for storing information for a file stream, defined in the <stdio.h> standard library

FILE*: a pointer type of FILE type

- A file pointer will be associated with a file stream once a file is opened.

Open and close files

□ Open files

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

- Opens the file pointed to by filename using the given opening mode.

```
pFile1 = fopen("Data.txt", "r");
```

□ Close files

- `int fclose(FILE *stream)`

- Closes the stream. All buffers are flushed.

```
fclose(pFile1);
```

Open and close files

□ Open files

- Filename = "Data.txt"

- Located in the current directory

- Filename = ".\\Data\\Data.txt"

- Located in the sub-directory of the current directory

- Filename = "..\\Data.txt"

- Located in the super-directory of the current directory

- Filename = "D:\\CS - Introduction to Computer Programming - CO1003 - Undergraduate course\\Code samples\\Data\\Data.txt"

- Located in the specified directory with the absolute path

- ...

Open and close files

File opening modes

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Store and retrieve data from files

□ Store data into files (write)

- A sequence of bytes from memory to the file
 - The file plays a role of *stdout*.

□ Retrieve data from files (read)

- A sequence of bytes from the files to memory
 - The file plays a role of *stdin*.

What bytes?	Read (file -> memory)	Write (memory -> file)
A char	fgetc	fputc
A line of bytes	fgets	fputs
Formatted bytes	fscanf	fprintf
Bytes in the binary mode	fread	fwrite

Store and retrieve data from files

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)

Description

The C library function **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** reads data from the given **stream** into the array pointed to, by **ptr**.

Declaration

Following is the declaration for fread() function.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** -- This is the pointer to a block of memory with a minimum size of *size*nmemb* bytes.
- **size** -- This is the size in bytes of each element to be read.
- **nmemb** -- This is the number of elements, each one with a size of **size** bytes.
- **stream** -- This is the pointer to a FILE object that specifies an input stream.

Store and retrieve data from files

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

Description

The C library function **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** writes data from the array pointed to, by **ptr** to the given **stream**.

Declaration

Following is the declaration for fwrite() function.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Parameters

- **ptr** -- This is the pointer to the array of elements to be written.
- **size** -- This is the size in bytes of each element to be written.
- **nmemb** -- This is the number of elements, each one with a size of **size** bytes.
- **stream** -- This is the pointer to a FILE object that specifies an output stream.

Store and retrieve data from files

What bytes?	Read (file -> memory)	Write (memory -> file)
<i>A char</i>	<i>fgetc</i>	<i>fputc</i>
char aChar;	aChar = fgetc (pFile1);	fputc ('a', pFile1);
<i>A line of bytes</i>	<i>fgets</i>	<i>fputs</i>
char aStr[50];	fgets (aStr, 50, pFile1);	fputs ("Today?", pFile1);
<i>Formatted bytes</i>	<i>fscanf</i>	<i>fprintf</i>
int anInt; char aChar; float aFloat;	fscanf (pFile1, "%d %c %f", &anInt, &aChar, &aFloat);	fprintf (pFile1, "%d %d %c %f", 10, anInt, aChar, aFloat);
<i>Bytes in the binary mode</i>	<i>fread</i>	<i>fwrite</i>
char buffer[50];	fread (buffer, sizeof(int)+sizeof(char)+sizeof(float)+1, 5, pFile1);	fwrite (buffer, strlen(buffer)+1, 1, pFile1);

Process the grades of each student
in a file for their averaged grades.

```
#include <stdio.h>
#define SIZE 50

void main() {
    //Declaration for a file pointer
    FILE* pFile = NULL;

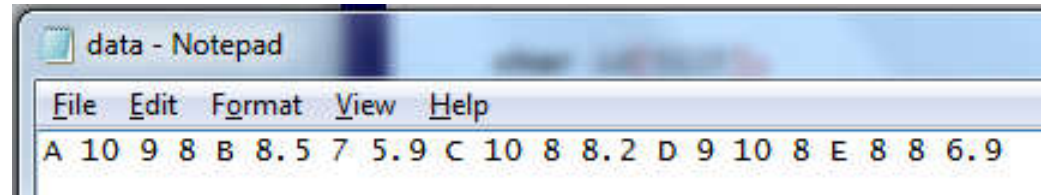
    char id[SIZE];
    float g1[SIZE], g2[SIZE], g3[SIZE];
    int n = 0;

    //Open a file for data reads
    pFile = fopen("./Data\\Data.txt", "r");
    if (pFile == NULL) {
        printf("\nUnable to open the file!\n");
        return;
    }

    //Read data from a file
    while (fscanf(pFile, "%c %f %f %f", &id[n], &g1[n], &g2[n], &g3[n])>0) {
        if (id[n] == ' ') continue;
        n++;
    }

    //Process the data read from a file
    int i;
    printf("\nID \tAVERAGE\n");
    for (i=0; i<n; i++) printf("\n%c \t%.2f\n", id[i], (g1[i]+g2[i]+g3[i])/3);

    //Close a file
    fclose(pFile);
}
```



ID	AVERAGE
A	9.00
B	7.13
C	8.73
D	9.00
E	7.63

Process the grades of each student in a file for their averaged grades. Write the resulting averaged grades into another file.

```
#include <stdio.h>
#define SIZE 50

struct avgGrade {
    char id;
    float avg;
};

int readGrades(char* filename, char* id, float* g1, float* g2, float* g3, int* n);
int writeAVG(char* filename, struct avgGrade* grd, int n);
int writeAVG_2(char* filename, struct avgGrade* grd, int n);

void main() {
    char* inFile = ".\\Data\\Data.txt";
    char* outFile = ".\\Data\\avgData.txt";
    char* outFile_2 = ".\\Data\\avgData_2.txt";

    char id[SIZE];
    float g1[SIZE], g2[SIZE], g3[SIZE];
    int n = 0;

    struct avgGrade* avgG;

    if (!readGrades(inFile, id, g1, g2, g3, &n)) {
        printf("\nUnable to read grades from the file!\n");
        return;
    }

    avgG = (struct avgGrade*)malloc(n*sizeof(struct avgGrade));
    if (avgG == NULL) {
        printf("\nUnable to have memory for averaged grade processing!\n");
        return;
    }
}
```

ID	AVERAGE
A	9.00
B	7.13
C	8.73
D	9.00
E	7.63

```
#include <stdio.h>
#define SIZE 50
```

```
struct avgGrade {
    char id;
    float avg;
};
```

```
int readGrades(char* filename, char* id, float* g1, float* g2, float* g3, int* n);
int writeAVG(char* filename, struct avgGrade* grd, int n);
int writeAVG_2(char* filename, struct avgGrade* grd, int n);
```

```
void main() {
    char* inFile = ".\\Data\\Data.txt";
    char* outFile = ".\\Data\\avgData.txt";
    char* outFile_2 = ".\\Data\\avgData_2.txt";
    .....
```

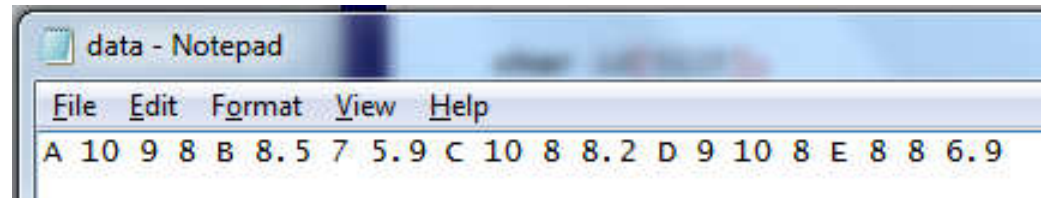
```
int i;
for (i=0; i<n; i++) {
    avgG[i].id = id[i];
    avgG[i].avg = (g1[i] + g2[i] + g3[i])/3;
}
```

```
if (!writeAVG(outFile, avgG, n))
    printf("\nUnable to write averages to the file!\n");
```

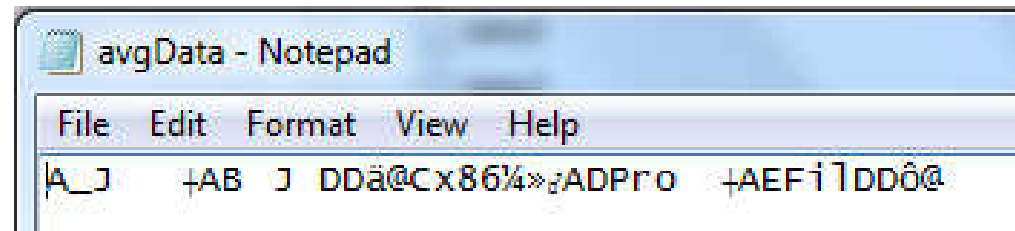
```
if (!writeAVG_2(outFile_2, avgG, n))
    printf("\nUnable to write averages to the file!\n");
```

```
}
```

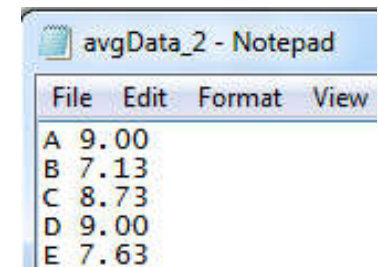
Input data file



Output data file in the binary mode



Output data file in the formatted mode




```
//return: 1: successfully written; 0: unsuccessfully written
```

```
int writeAVG(char* filename, struct avgGrade* grd, int n) {
```

```
FILE* opFile = fopen(filename, "wb");
```

Output data file in the binary mode

```
if (opFile==NULL) {  
    printf("\nUnable to open the file!\n");  
    return 0;  
}
```

```
fwrite(grd, sizeof(struct avgGrade), n, opFile);
```

```
fclose(opFile);  
return 1;
```

```
}
```

```
//return: 1: successfully written; 0: unsuccessfully written
```

```
int writeAVG_2(char* filename, struct avgGrade* grd, int n) {
```

```
FILE* opFile = fopen(filename, "w");
```

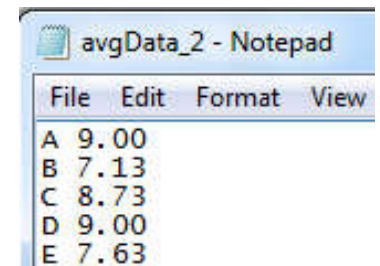
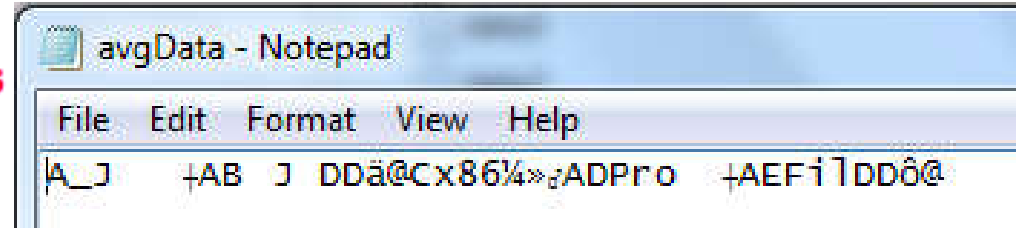
Output data file in the formatted mode

```
if (opFile==NULL) {  
    printf("\nUnable to open the file!\n");  
    return 0;  
}
```

```
int i;  
for(i=0; i<n; i++)  
    fprintf(opFile, "%c %.2f\n", grd[i].id, grd[i].avg);
```

```
fclose(opFile);  
return 1;
```

```
}
```



Use macros

□ Macro

- An identifier in `#define` preprocessor directive

`#define identifier replacement-text`

- *identifier*: a valid identifier scoped from its definition to the end of the file or to the `#undef` directive
- *replacement-text*: text on the line or longer text on many lines with a backslash (\)
- Considered as operations defined as symbols
 - With no argument ? processed like a symbolic constant
 - With arguments ? processed like an inline function
 - Arguments are substituted in the replacement text
 - The replacement text then replaces the identifier and argument list in the program

Use macros

`#define` *identifier replacement-text*

□ User-defined macros

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

```
#define PI 3.14159  
#define AREA(radius) (PI*(radius)*(radius))  
#define CIRCUMFERENCE(radius) (2*PI*(radius))
```

```
void main() {  
    float r = 1.5;  
  
    printf("\nThe given circle has a radius of %.2f.\n", r);  
  
    printf("\nArea of the given circle = %.2f\n", AREA(r));  
    printf("\nCircumference of the given circle = %.2f\n", CIRCUMFERENCE(r));  
  
    printf("\nArea of the larger circle = %.2f\n", AREA(r+1));  
    printf("\nCircumference of the given circle = %.2f\n", CIRCUMFERENCE(r+1));  
}
```

```
The given circle has a radius of 1.50.  
Area of the given circle = 7.07  
Circumference of the given circle = 9.42  
Area of the larger circle = 19.63  
Circumference of the given circle = 15.71
```

→ Parentheses need using for correct value determination.

Use macros

```
#define identifier replacement-text
```

```
#define CIRCUMFERENCE(radius) (2*PI*(radius))
```

r=1.5

$CIRCUMFERENCE(r+1) = (2*PI*(r+1)) = 15.71$



VS.

```
#define CIRCUMFERENCE(radius) (2*PI*radius)
```

r=1.5


$CIRCUMFERENCE(r+1) = (2*PI*r+1) = 10.42$



Use macros

`#define identifier replacement-text`

`#define DIAMETER(radius) ((radius) + (radius))`

$CIRCUMFERENCE = PI * DIAMETER(r+1)$ r=1.5
 $= PI * ((r+1) + (r+1)) = 15.71$ 

VS.

`#define DIAMETER(radius) (radius) + (radius)`

$CIRCUMFERENCE = PI * DIAMETER(r+1)$ r=1.5
 $= PI * (r+1) + (r+1) = 10.35$ X

Use macros

- Conditional compilation up to macro definition
 - control the execution of preprocessor directives and the compilation of program code

```
#ifdef identifier    //if identifier is defined, consider the following code  
#endif
```

```
#ifdef identifier    //if identifier is defined, consider the following code  
#else               //else consider the following code  
#endif
```

```
#ifdef identifier    //if identifier is defined, consider the following code  
#elif constant-expression  
#endif
```

```
#ifndef identifier   //if identifier is not defined, consider the following code  
#endif
```

```
#undef identifier    //remove the definition of identifier
```

Use macros – Conditional compilation

```
1  #ifndef MYNUMBER_H_INCLUDED
2  //to prevent header files
3  //from being included multiple times in the same source files
4
5  #define MYNUMBER_H_INCLUDED
6
7  #define E 2.718281
8
9  #define PI 3.141592
10
11 int isPrimeNumber(const int aNumber);
12
13 int isSquaredNumber(const int aNumber);
14
15 int genMask(const int aNumber, int *aMask);
16
17 void countDigits(const int aNumber);
18
19 double factorial(int n);
20
21 double power(double x, int n);
22
23 double rFactorial(int n);
24
25 double rPower(double x, int n);
26
27 #endif // MYNUMBER_H_INCLUDED
```

Recall – Chapter 6 – Functions
Header file: *mynumber.h*

Use macros

- Many macros predefined in the standard library files
 - `<float.h>`
 - `FLT_MIN`: the minimum finite floating-point value
 - `FLT_MAX`: the maximum finite floating-point value
 - ...
 - `<limits.h>`
 - `INT_MIN`: the minimum value for an int with 2 bytes
 - `INT_MAX`: the maximum value for an int with 2 bytes
 - ...
 - `<locale.h>`
 - `LC_ALL`: sets everything with the location specific settings
 - `LC_CTYPE`: affects all the character functions with settings
 - ...

Use macros

- Many macros predefined in the standard library files
 - `<stdarg.h>`
 - The **stdarg.h** header defines a variable type **va_list** and *three following macros* which can be used to get the arguments in a function when the number of arguments are not known i.e. variable number of arguments.
 - `void va_start(va_list ap, last_arg)`
 - `type va_arg(va_list ap, type)`
 - `void va_end(va_list ap)`
 - A function of variable arguments is defined with the ellipsis (`,...`) at the end of the parameter list.

Use macros - <stdarg.h>

- `va_list`
 - ▣ A type whose variable *ap* is used for holding information about the arguments
- `void va_start(va_list ap, last_arg)`
 - ▣ The macro initializes **ap** variable to be used with the **va_arg** and **va_end** macros. The **last_arg** is the last known fixed argument being passed to the function i.e. the argument before the ellipsis.
- `type va_arg(va_list ap, type)`
 - ▣ The macro retrieves the next argument in the parameter list of the function with type **type**.
- `void va_end(va_list ap)`
 - ▣ The macro allows a function with variable arguments which used the **va_start** macro to return. If **va_end** is not called before returning from the function, the result is undefined.

Use macros - <stdarg.h>

Define the function *isSubstring* to check if a substring is contained by a list of given strings.

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

```
String: "Today is Monday, isn't?" contains a substring "Monday".
```

```
int isSubstring(char* aSub, int nStr, ...);

void main() {
    char* str1 = "Today is Monday, isn't?";
    char* str2 = "Today is not Sunday.";
    char* str3 = "Yesterday is not Today.";

    char* query = "Monday";

    if (isSubstring(query, 1, str1))
        printf("\nString: \"%s\" contains a substring \"%s\".\n", str1, query);

    if (isSubstring(query, 2, str1, str2))
        printf("\nStrings: \"%s\" and \"%s\" contain a substring \"%s\".\n", str1, str2, query);

    if (isSubstring(query, 3, str1, str2, str3))
        printf("\nStrings: \"%s\", \"%s\", and \"%s\" contain a substring \"%s\".\n", str1, str2, str3, query);
}
```

Use macros - <stdarg.h>

```
//Return: 1: true for all strings;
//        0: false for at least one string
int isSubstring(char* aSub, int nStr, ...) {

    va_list ap;
    int i;

    va_start(ap, nStr);

    for (i=0; i<nStr; i++) {

        char* aStr = va_arg(ap, char*);

        int j, k = 0;
        while (k<strlen(aStr)) {
            if (aSub[0] == aStr[k]) {
                for (j=1; j<strlen(aSub); j++)
                    if (aSub[j] != aStr[k+j]) break;
                if (j==strlen(aSub)) break;
            }
            k++;
        }
        if (k==strlen(aStr)) return 0;
    }

    va_end(ap);

    return 1;
}
```

Variable **ap** is required to have an access to each argument in the argument list after this point.

This macro is required to start the access to the argument list.

This macro is required to have an access to an actual argument of type **char*** in the argument list.

This macro is required to end the access to the argument list before the return of the function.

Summary

- File processing in C
 - Allows data permanency
 - Performs with the standard functions in the `<stdio.h>` library
 - Open/Close
 - Read/Write/Append
 - Supports two access schemes
 - Sequential access
 - Random access

Summary

- Macros in the `#define` preprocessor directive
 - User-defined macros vs. Macros in the libraries
 - Conditional compilation based on macro definitions
 - Examples with macros in the `<stdarg.h>` for functions with the variable numbers of arguments

Chapter 9: File Processing

