Files and Errors

Fundamentals of Programming

Lecture 6

Contents

- File access
- File access functions and modes
- File write and read (text and binary)
- Memory access
- Error handling

File access

File access

- In Unix-like operating systems (Linux, BSD, macOS) the program output can be sent to external files
- Symbol (>) represents output from program to file

```
program > file.txt
```

- Symbol (<) represents input from file to program

```
program < file.txt
```

File access operations

- FPRINTF() function is used to write to a text file
- FSCANF() function is used read from a text file
- File access functions are available in stdio.h library file
- External files
  - Text files
  - Binary data files

File types

Text file

- ASCII characters
- Data stored one byte at a time
- File is slower to read and write
- Compatible with all computer systems

Binary file

- Encoded as binary data
- Data is compact
- File is faster to read and write
- Incompatible with some computer systems

Data streams

- `stdin`
  - Used for input buffering or keyboard
- `stdout`
  - Used for output buffering or screen
- `stderr`
  - Used for output buffering or screen

File mode of operation

Text Mode	Binary Mode	Operation	Description
`r`	`rb`	Read	Open file for reading
`w`	`wb`	Write	Create new file for writing, previous contents are discarded
`a`	`ab`	Append	Open or create file for writing (at end of file)
`r+`	`rb+` `r+b`	Read and append	Open file for update (reading and writing)
`w+`	`wb+` `w+b`	Write and append	Create file for update, previous contents are discarded
`a+`	`ab+` `a+b`	Append	Open or create file for update, and writing (at end of file)

File access functions

File pointer

- The external file must be access via file pointer <file_ptr>
- File pointer uses data type `FILE`

```
FILE *<identifier>;
```

Open and close files

- FOPEN() is used to initialise and open an external file
  - Mode of operations include read, write, or append
  - Only one mode can be used on a file at any given time
- FCLOSE() is used to close a file
  - So other programs will be allowed to access the file

File open

- FOPEN() is used to open an external file

```
<file_ptr> = fopen("<file>","<mode>");
```

- Example:

```
FILE *fp;
fp = fopen("info.txt", "w");
```

File close

- FCLOSE() is used to close an external file

```
fclose(<file_ptr>);
```

- Example:

```
fclose(fp);
```

End-of-file

- FEOF() checks the end-of-file (EOF) indicator in a file
- Return value from FEOF() is either 0 (FALSE) or 1 (TRUE)
- It is useful when program has to read the entire file

```
<var> = feof(<file_ptr>);
```

- Example:

```
while (!feof(fp)) {...}
```

Rewind()

- REWIND() moves the file pointer to top of file
- Preserves the previous mode of operation
- Equivalent to `fseek(fp, 0, SEEK_SET)`

```
rewind(<file_ptr>);
```

- Example:

```
rewind(fp);
```

Error in opening file

- Before an external file can be used for reading, it has to exist in the directory
- A non-existent file or an empty file can be detected by using NULL
- If the file does not exist the error message is printed to terminal, and value 1 is returned to indicate an error

```
FILE *fp;
fp = fopen("data.txt", "r");
if (fp == NULL) {
  printf("Error in opening file!\n");
  return 1;
}
```

Error in opening file example

- If the file does not exist, a user-defined error message will be printed
- Without the IF statement to check on the file, the program will fail

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char a[20];
  fp = fopen("info.txt", "r");
  if (fp == NULL) {
    printf("Error in opening file!\n");
    return 1;
  }
  fscanf(fp, "%s", a);
  printf("%s\n", a);
  fclose(fp);
  return 0;
}
```

File write

Writing text data to file

- Writing text data to file can be in the form of
  - String of characters
  - Single character
- Loops (repetition structures) can be used to help write multiple characters at a time
- Functions
  - FPRINTF()
  - FPUTC()
  - FPUTS()

Fprintf()

- FPRINTF() writes a formatted string of characters to a file

```
fprintf(<file_ptr>,"<control_string>",<arg1>,…,<argN>);
```

- Example:

```
char x[20] = "Goodbye";
fprintf(fp, "Hello");
fprintf(fp, "%s", x);
```

Fprintf() example 1

- File pointer fp is FILE data type
- FOPEN() creates a text file info.txt for writing by “w” mode
- FPRINTF() writes “Hello” to the text file via stream fp
- FCLOSE() exits file operations, so file cannot be read or written

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("info.txt", "w");
  fprintf(fp, "Hello");
  fclose(fp);
  return 0;
}
```

Fprintf() example 2

- Two words “Hello” and “Goodbye” are written into the file
- The write order depends on the sequence of the FPRINTF() functions used

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("info.txt", "w");
  fprintf(fp, "Hello");
  fprintf(fp, "Goodbye");
  fclose(fp);
  return 0;
}
```

Fputc()

- FPUTC() writes a single character to a stream
- Stream can be a file pointer, char array, char pointer

```
fputc('<char>',*<stream>);
```

- Example:

```
char y = 'K';
fputc('x', fp);
fputc(y, fp);
```

Fputc() example 1

- FPUTC() has two input parameters:
  - Character to be written
  - Stream
- FPUTC() writes one ASCII character at a time to the stream fp
- Contents of info.txt is ‘x’

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("info.txt", "w");
  fputc('x', fp);
  fclose(fp);
  return 0;
}
```

Fputc() example 2

- Several FPUTC() functions are used to write the characters of a word into the file info.txt
- Contents of info.txt is “OK\n”

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("info.txt", "w");
  fputc('O', fp);
  fputc('K', fp);
  fputc('\n', fp);
  fclose(fp);
  return 0;
}
```

Fputs()

- FPUTS() writes a string of characters to a stream
- Stream can be a file pointer, char array, char pointer

```
fputs(*<string>,*<stream>);
```

- Example:

```
char x[20] = "Wonderful";
fputs(x, fp);
fputs("Day", fp);
```

File write example 1

- FPUTS() has two input parameters:
  - Character array or string to be written
  - Stream
- FPUTS() writes the word “Hello” into the file

```
#include <stdio.h>
int main()
{
  FILE *fp;
  fp = fopen("info.txt", "w");
  fputs("Hello", fp);
  fclose(fp);
  return 0;
}
```

File write example 2

- Two character arrays contain long text
- FPUTS() is used in each statement for writing data in the arrays a[] and b[] to the file

```
#include <stdio.h>
int main()
{
  char a[20] = "Hello\n";
  char b[20] = "Goodbye\n";
  FILE *fp;
  fp = fopen("info.txt", "w");
  fputs(a, fp);
  fputs(b, fp);
  fclose(fp);
  return 0;
}
```

File read

Reading text data from file

- Reading text data to file can be in the form of
  - String of characters
  - Single character
- Loops (repetition structures) can be used to help read multiple characters at a time
- Functions
  - FSCANF()
  - FGETC()
  - FGETS()

Fscanf()

- FSCANF() reads formatted data from a file into an character array
- It will only read a string without blank spaces

```
fscanf(<file_ptr>,"<control_string>",&<arg1>,…,&<argN>);
```

- Example:

```
int x;
char y[20];
fscanf(fp, "%d", &x);
fscanf(fp, "%s", y);
```

Fscanf() example 1

- FOPEN() opens text file info.txt for reading by “r” mode
- FSCANF() has three input parameters:
  - Stream
  - Data as string/character/integer
  - Destination of character array
- One integer value is read, even though the data is ASCII characters
- FCLOSE() exits file operations

```
#include <stdio.h>
int main()
{
  int x;
  FILE *fp;
  fp = fopen("info.txt", "r");
  fscanf(fp, "%d", &x);
  printf("x = %d\n", x);
  fclose(fp);
  return 0;
}
```

Fscanf() example 2

- Three integer values are read from the file
- Data is stored in array a[]
- Array a[] data is printed to the terminal screen

```
#include <stdio.h>
int main()
{
  int a[3];
  FILE *fp;
  fp = fopen("info.txt", "r");
  fscanf(fp, "%d", &a[0]);
  fscanf(fp, "%d", &a[1]);
  fscanf(fp, "%d", &a[2]);
  printf("%d %d %d\n",a[0],a[1],a[2]);
  fclose(fp);
  return 0;
}
```

Fscanf() example 3

- Info.txt contains a string
- FSCANF() is used to read the entire string
- Once three characters are read, it is output to the screen

```
#include <stdio.h>
int main()
{
  char a[3];
  FILE *fp;
  fp = fopen("info.txt", "r");
  fscanf(fp, "%s", a);
  printf("%s\n",a);
  fclose(fp);
  return 0;
}
```

Fgetc()

- FGETC() reads a single character from a stream
- Stream can be a file pointer, char array, char pointer
- Return value must be saved in a variable

```
<var> = fgetc(*<stream>);
```

- Example:

```
char x;
x = fgetc(fp);
```

Fgetc() example 1

- FGETC() has one input parameter: stream
  - Returns a char value
- FGETC() reads a character from fp
- Data is stored in char variable x
- PRINTF() is used to output data to the screen

```c
#include <stdio.h>
int main()
{
  char x;
  FILE *fp;
  fp = fopen("info.txt", "r");
  x = fgetc(fp);
  printf("%c\n", x);
  fclose(fp);
  return 0;
}
```

Fgetc() example 2

- FGETC() is used to read three characters, one by one, from the file
- Each statement containing FGETC() reads from fp, and stores the character in an element in array x[]
- PRINTF() is used to output data to the screen

```
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("info.txt", "r");
  x[0] = fgetc(fp);
  x[1] = fgetc(fp);
  x[2] = fgetc(fp);
  printf("%c %c %c\n", x[0], x[1], x[2]);
  fclose(fp);
  return 0;
}
```

Fgets()

- FGETS() reads a string of characters including blank spaces from a stream
- Size refers to buffer size N, actual number of characters read is N-1
- Stream can be a file pointer, char array, char pointer

```
fgets(*<string>,<size>,*<stream>);
```

- Example:

```
char x[20]; char y[20];
fgets(x, 20, fp); // read 19 chars
fgets(y, 20*sizeof(char), fp); // read 19 chars
```

Sizeof()

- SIZEOF() is used to get the size of a data type in bytes

```
<var> = sizeof(<data_type>);
```

- Example:

```
int x, y, z;
x = sizeof(int); // x = 4
y = sizeof(float); // y = 4
z = sizeof(char); // z = 1
```

Fgets() example 1

- FGETS() has three input parameters:
  - Character array
  - Expected size in bytes of input data
  - Stream
- FGETS() reads from fp, expects 19 characters, and stores data in x[]
- String data is printed to the screen using PRINTF()

```c
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("info.txt", "r");
  fgets(x, 20, fp);
  printf("%s\n", x);
  fclose(fp);
  return 0;
}
```

Fgets() example 2

- FGETS() can also be used to read ASCII characters from the user’s keyboard
- User’s keyboard is represented by **stdin** data stream
- Data read from the user is stored in the file using FPRINTF()

```
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("info.txt", "w");
  printf("Enter word: ");
  fgets(x, 20, stdin);
  fprintf(fp, "%s\n", x);
  fclose(fp);
  return 0;
}
```

Sequential file access

Sequential write (1)

- File is open for writing
- File is checked if it exists, otherwise create a new file
- FOR loop runs for five rounds
- Five integers are read from user
- Five integers are written to file

```
#include <stdio.h>
int main() {
  int n;
  int x[5];
  FILE *fp;
  fp = fopen("data.txt","w");

  for (n=0; n<5; n++){
    printf("Enter integer %d: ", n+1);
    scanf("%d", &x[n]);
    fprintf(fp, "%d\n", x[n]);
  }
  fclose(fp);
  return 0;
}
```

Sequential write (2)

- At the command prompt:

```
Enter integer 1: 56
Enter integer 2: 32
Enter integer 3: 11
Enter integer 4: 3
Enter integer 5: 8
```

- In data.txt:

```
56
32
11
3
8
```

Sequential read (1)

- File opened for reading
- WHILE loop runs until EOF indicator is detected
- Data in x[] is output to the screen

```
#include <stdio.h>
int main() {
  int n = 0;
  int x[5];
  FILE *fp;
  fp = fopen("data.txt","r");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  while (!feof(fp)) {
    x[n] = 0;
    fscanf(fp, "%d", &x[n]);
    if (x[n] == '\0')
      break;
    else
      printf("%d\n", x[n]);
    n++;
  }
  fclose(fp);
  return 0;
}
```

Sequential read (2)

- At the command prompt:

56

32

11

3

8

Append data (1)

- File opened for append
- File pointer fp is reset to top of file
- One integer is read from user and stored in x
- Data in x appended to file

```c
#include <stdio.h>
int main() {
  int x;
  FILE *fp;
  fp = fopen("data.txt","a+");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  printf("Enter new integer: ");
  scanf("%d", &x);
  fprintf(fp, "%d\n", x);
  fclose(fp);
  return 0;
}
```

Append data (2)

- At the command prompt:

```
Enter new integer: 47
```

- In data.txt:

```
56
32
11
3
8
47
```

Rewind()

- If the same file has to be operated on again, REWIND() is used to reopen the same file with the previous mode of operation
- The file pointer will be located at the top of the file by default
- Use FSEEK() to move the file pointer, if required
- The only input parameter is the original file pointer

```
rewind(<file_ptr>);
```

Rewind/append example

- File data.txt is opened for append
- FSEEK() is used to move the file pointer to the last position before EOF indicator
- User is prompted for a new integer, which is then written into the file
- REWIND() reopens the file with the same mode of operation (a+)
- User is prompted again for another integer, then it is written into the file

```
#include <stdio.h>
int main()
{
  int x;
  FILE *fp;
  fp = fopen("data.txt","a+");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  printf("Enter new integer: ");
  scanf("%d", &x);
  fprintf(fp, "%d\n", x);

  rewind(fp);
  printf("Enter new integer: ");
  scanf("%d", &x);
  fprintf(fp, "%d\n", x);
  fclose(fp);
  return 0;
}
```

Rewind/write example 1

- The program will prompt the user to enter three (3) integers
- Each new integer is written into the file
- REWIND() reopens the file for write again
- The file pointer starts at the top of the file
- Any new write operation will overwrite the first integer in the file

```
#include <stdio.h>
int main()
{
  int x, n;
  FILE *fp;
  fp = fopen("data.txt","w");

  for (n=0; n<3; n++) {
    printf("Enter integer %d: ", n+1);
    scanf("%d", &x);
    fprintf(fp, "%d\n", x);
  }
  rewind(fp);
  printf("Enter new integer: ");
  scanf("%d", &x);
  fprintf(fp, "%d\n", x);
  fclose(fp);
  return 0;
}
```

Rewind/write example 2

- FSEEK() with SEEK_END option will move the file pointer to the end of the file
- Now any new write operations will continue from the last entry
- Mimics the append mode of operation

```
#include <stdio.h>
int main()
{
  int x, n;
  FILE *fp;
  fp = fopen("data.txt","w");

  for (n=0; n<3; n++) {
    printf("Enter integer %d: ", n+1);
    scanf("%d", &x);
    fprintf(fp, "%d\n", x);
  }
  rewind(fp);
  fseek(fp, 0, SEEK_END);
  printf("Enter new integer: ");
  scanf("%d", &x);
  fprintf(fp, "%d\n", x);
  fclose(fp);
  return 0;
}
```

Random access file

Random access file

- In text files, the data written or read are not constant in size
  - Read operation is difficult if the data size is not known
- In random access files, data written to the file is constant in size
- Information in a random access file is binary data

Offset	0	N	2N	3N	4N	5N	6N	7N
Data	N bytes	N bytes	N bytes	N bytes	N bytes	N bytes	N bytes	N bytes

Data orientation in binary files

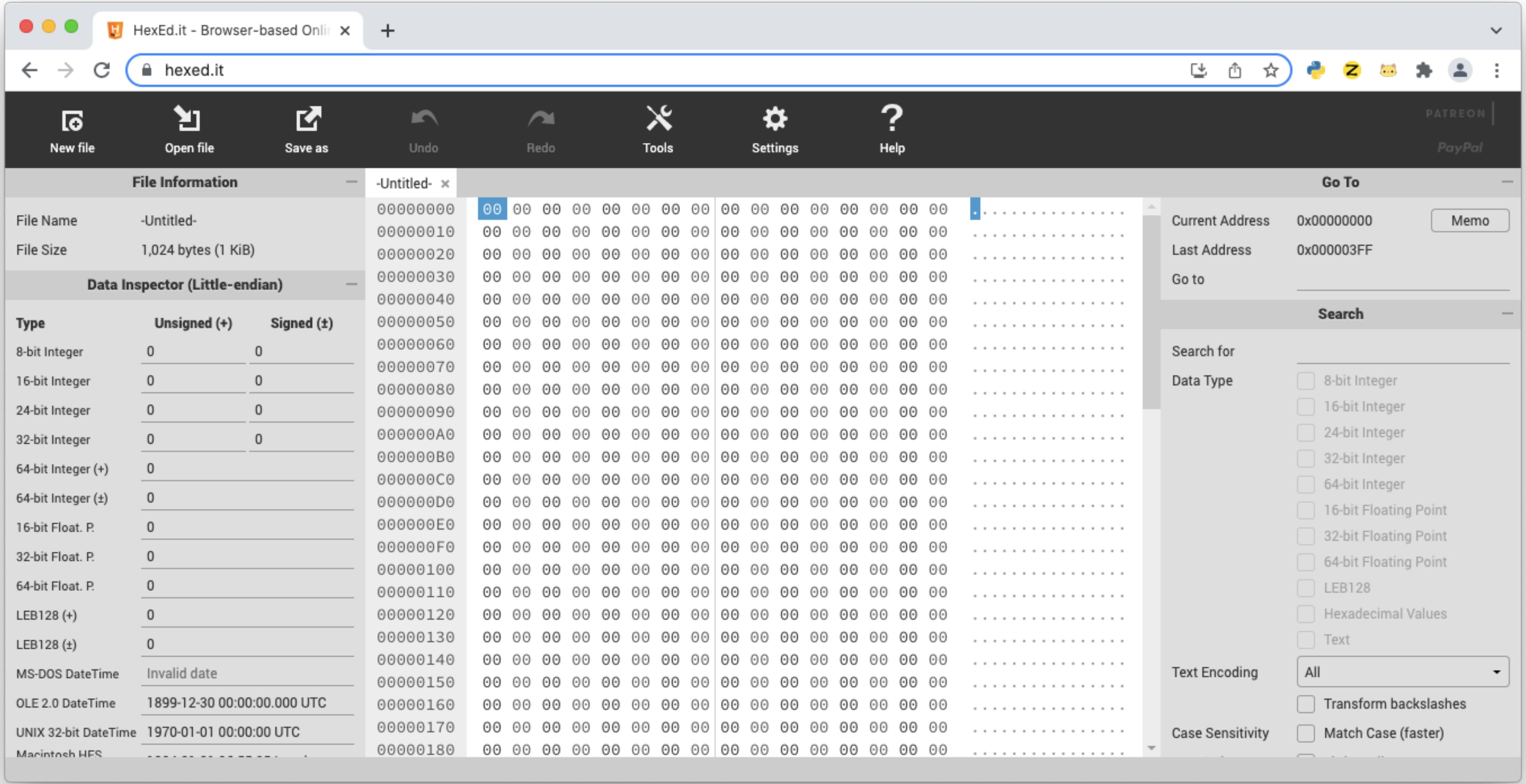
- Data is written into a binary file as hexadecimal
- The data will have the Least Significant Byte (LSB) first, and the Most Significant Byte (MSB) last
- Data can read using a hex editor or hex-to-decimal converter
- For example, the decimal value 12345 is 3039 in hex

	LSB			MSB
Hex Address	00	01	02	03
Hex Data	39	30	00	00

Hex Editor

- A hex editor is used to view and modify contents of a binary file
- It is available as a web application or a desktop application
- Data is presented in a grid form
  - Row represents the left digits of the hex address values
  - Column represents the rightmost digit of the hex address value

Hex Address		n															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0n	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	1n	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	2n	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
	3n	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F


HexEd.it - Browser-based Onli
hexed.it
New file
Open file
Save as
Undo
Redo
Tools
Settings
Help
PATREON
PayPal
File Information
-Untitled-
File Name -Untitled-
File Size 1,024 bytes (1 KiB)
Data Inspector (Little-endian)
Type Unsigned (+) Signed (±)
8-bit Integer 0 0
16-bit Integer 0 0
24-bit Integer 0 0
32-bit Integer 0 0
64-bit Integer (+) 0
64-bit Integer (±) 0
16-bit Float. P. 0
32-bit Float. P. 0
64-bit Float. P. 0
LEB128 (+) 0
LEB128 (±) 0
MS-DOS DateTime Invalid date
OLE 2.0 DateTime 1899-12-30 00:00:00.000 UTC
UNIX 32-bit DateTime 1970-01-01 00:00:00 UTC
Go To
Current Address 0x00000000
Memo
Last Address 0x000003FF
Go to
Search
Search for
Data Type
8-bit Integer
16-bit Integer
24-bit Integer
32-bit Integer
64-bit Integer
16-bit Floating Point
32-bit Floating Point
64-bit Floating Point
LEB128
Hexadecimal Values
Text
Text Encoding All
Transform backslashes
Case Sensitivity Match Case (faster)

Fwrite()

- FWRITE() writes binary data to a file

```
fwrite(*<buffer>,<size>,<count>,<file_ptr>);
```

- Example:

```
int x = 2;
int *y;
fwrite(&x, sizeof(int), 1, fp);
fwrite(y, sizeof(int), 1, fp);
```

Fwrite() example

- Block of data size = 4 bytes (int)
- Decimal 12345 = Hex 3039
- Data in info.dat is 39 30 00 00 in hexadecimal with LSB first and MSB last.

From the info.dat file

From the hex editor


```c
#include <stdio.h>
int main()
{
  int x = 12345;
  FILE *fp;
  fp = fopen("info.dat", "wb");
  fwrite(&x, sizeof(int), 1, fp);
  fclose(fp);
  return 0;
}
```

Fread()

- FREAD() reads binary data from a file

```
fread(*<buffer>,<size>,<count>,<file_ptr>);
```

- Example:

```
int x = 2;
int *y;
fread(&x, sizeof(int), 1, fp);
fread(y, sizeof(int), 1, fp);
```

Fread() example

- Block of data size = 4 bytes (int)
- Data read from info.data is printed to the terminal screen in decimal
  - Because of %d integer format specifier
- Alternatively, %x hex format specifier can also be used

```
printf("%x\n", x);
```

```
#include <stdio.h>
int main()
{
  int x = 0;
  FILE *fp;
  fp = fopen("info.dat","rb");
  fread(&x, sizeof(int), 1, fp);
  printf("%d\n", x);
  fclose(fp);
  return 0;
}
```

Fseek()

- FSEEK() is used to set the file pointer position with an offset in a random access (binary) file
- It has three input parameters:
  - File (file pointer)
  - Offset (N * size of data type)
  - Origin (SEEK_SET, SEEK_CUR, SEEK_END)
- Return value <var> is zero if successful, or a non-zero if failed

```
<var> = fseek(<file_ptr>,<offset>,<origin>);
```

Fseek() file pointer position

- Text file
  - Offset is zero
- Binary file
  - Offset is set from Origin
- Origin
  - SEEK_SET (beginning)
  - SEEK_CUR (current position)
  - SEEK_END (end of file)

Fseek() usage

- Return file pointer to beginning of the file

```
fseek(fp, 0, SEEK_SET);
```

- Move file pointer to third line in the file to read an integer

```
fseek(fp, 2*(sizeof(int)), SEEK_SET);
```

File pointer positions

File pointer location	File contents
	Beginning of file
	:
SEEK_SET Positive offset	Current
	:
	End of file

File pointer location	File contents
	Beginning of file
	:
SEEK_END Negative offset	Current
	:
	End of file

File pointer location	File contents
	Beginning of file
Neg Offset	:
SEEK_CUR	Current
Pos Offset	:
	End of file

Random access file example

Fwrite()
Binary file example (1)

- FOR loop used to read five integers from user
- FWRITE() used to write data to file
- Suppose the input values are 1111, 2222, 3333, 4444, 5555

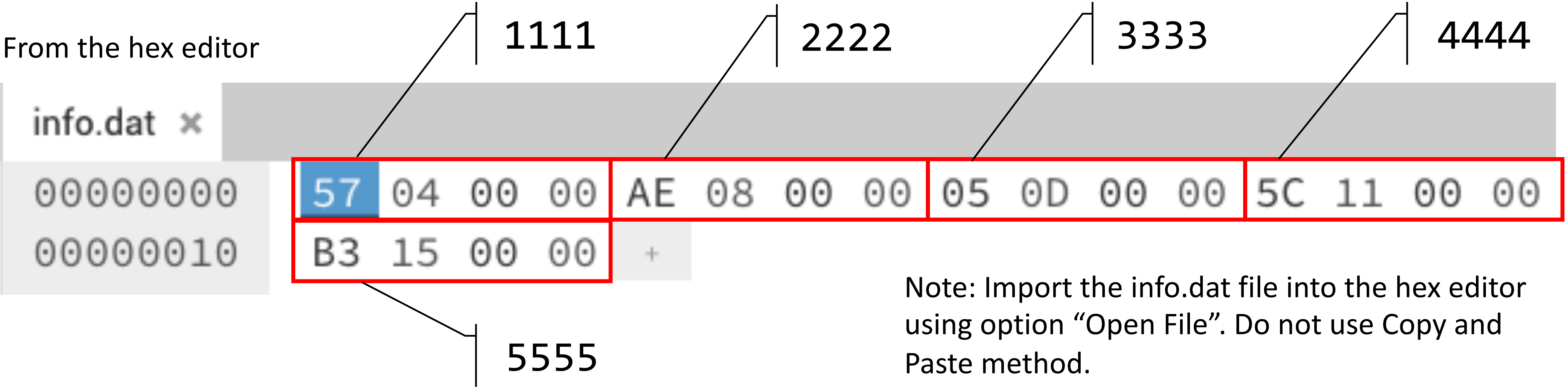
```
#include <stdio.h>
int main() {
  int n;
  int x[5];
  FILE *fp;
  fp = fopen("info.dat","wb");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  for (n=0; n<5; n++){
    printf("Enter integer %d: ", n+1);
    scanf("%d", &x[n]);
    fwrite(&x[n], sizeof(int), 1, fp);
  }
  fclose(fp);
  return 0;
}
```

Fwrite()
Binary file example (2)

From the info.dat file

Decimal	Hex
1111	0457
2222	08AE
3333	0D05
4444	115C
5555	15B3

From the hex editor



Note: Import the info.dat file into the hex editor using option “Open File”. Do not use Copy and Paste method.

Fread()
Binary file example (3)

- WHILE loop uses FREAD() to read all integers from fp, and store the data in array x[]
- When FEOF() has not detected the end-of-file (EOF) indicator, it returns a zero
- All contents in the file info.dat will be printed to the terminal screen

```c
#include <stdio.h>
int main() {
  int n;
  int x[5];
  FILE *fp;
  fp = fopen("info.dat","rb");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  n = 0;
  while (!feof(fp)) {
    fread(&x[n], sizeof(int), 1, fp);
    printf("%d\n", x[n]);
    n++;
  }
  fclose(fp);
  return 0;
}
```

Fseek()
Binary file example (4)

- The user is asked for the offset value, in order to read the data in info.dat
- Data is output to screen

Offset	Decimal	Hex
0	1111	0457
1	2222	08AE
2	3333	0D05
3	4444	115C
4	5555	15B3

```
#include <stdio.h>
int main()
{
  int x = 0;
  int offset;
  FILE *fp;
  fp = fopen("info.dat","rb");
  if (fp == NULL) {
    printf("File open error\n");
    return 1;
  }
  printf("Enter offset value: ");
  scanf("%d", &offset);
  fseek(fp, offset * sizeof(int), SEEK_SET);
  fread(&x, sizeof(int), 1, fp);
  printf("%d\n", x);
  fclose(fp);
  return 0;
}
```

Memory access

Dynamic memory allocation

- Array size is fixed beforehand in programs
- If array size cannot be predicted, then dynamic memory allocation is used
- Also known as **dynamic array**
- Memory allocation functions are found in stdlib.h
- Once memory block is allocated, it is treated like an array

Array comparison

Fixed array

- Determined during development
  - Data type of array
  - Size of array
- During program execution
  - Memory space is persistent
  - Size of array is fixed

Dynamic array

- Determined during development
  - Data type of array
  - Size of array is optional
- During program execution
  - Memory space can be persistent or freed
  - Size of array can be changed

Sizeof() used in dynamic arrays

- SIZEOF() is used to get the size of a data type in bytes
- A multiplier may be used with the return value to signify N number of data types for the dynamic array

```
<var> = sizeof(<data_type>);
```

- Example:

```
int x, y, z;
x = 20 * sizeof(int); // x = 80
y = 20 * sizeof(float); // y = 80
z = 50 * sizeof(char); // z = 50
```

Free() function

- FREE() is used to release the memory space when no longer needed

```
free(<ptr>);
```

- Example:

```
int *sp, *tp;
free(sp);
free(tp);
```

Malloc() function

- MALLOC() is used to allocate uninitialised memory space
- Pointer used must be of the same data type as the dynamic array

```
<ptr> = malloc(<size>);
```

- Example:

```
int *sp, *tp;
sp = malloc(50*sizeof(int)); // 50*int
tp = malloc(200*sizeof(int)); // 200*int
```

Malloc() example 1

- The sp pointer is assigned to memory space for 3 integers
- MALLOC() will allocate 12 bytes using 3 * sizeof(int)
- Integer values are manually entered into the memory space via sp[]
- Data in the memory space is printed to the terminal screen

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *sp;
  sp = malloc(3 * sizeof(int));
  sp[0] = 1;
  sp[1] = 2;
  sp[2] = 3;
  printf("sp[0] = %d\n", sp[0]);
  printf("sp[1] = %d\n", sp[1]);
  printf("sp[2] = %d\n", sp[2]);
  free(sp);
  return 0;
}
```

Calloc() function

- CALLOC() is used to allocate memory space initialised with zeros
- Pointer used must be of the same data type as the dynamic array

```
<ptr> = calloc(<count>,<size>);
```

- Example:

```
int *sp, *tp;
sp = calloc(50,sizeof(int)); // 50*int
tp = calloc(200,sizeof(int)); // 200*int
```

Calloc() example 1

- The sp pointer is assigned to memory space for 3 integers
- CALLOC() will allocate 12 bytes for sp
- Integer values are manually entered into the memory space via sp[]
- Data in the memory space is printed to the terminal screen

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *sp;
  sp = calloc(3, sizeof(int));
  sp[0] = 1;
  sp[1] = 2;
  sp[2] = 3;
  printf("sp[0] = %d\n", sp[0]);
  printf("sp[1] = %d\n", sp[1]);
  printf("sp[2] = %d\n", sp[2]);
  free(sp);
  return 0;
}
```

Comparing malloc() and calloc()

- To compare between malloc() and calloc(), create the dynamic arrays and assign values to the arrays
- Print the contents in the arrays
- Free the memory space of the dynamic arrays
- Re-create the dynamic arrays and print the contents in the arrays
- Use the test program in the next two slides to study the difference between malloc() and calloc()

Comparing malloc() and calloc() Test program - Part 1

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n;
  int *s, *t;
  // Create dynamic arrays
  s = malloc(4*sizeof(int));
  t = calloc(4, sizeof(int));

  // Assign values
  s[0] = 1; s[1] = 2; s[2] = 3; s[3] = 4;
  t[0] = 1; t[1] = 2; t[2] = 3; t[3] = 4;

  // Print contents in arrays
  for (n=0; n<4; n++) {
    printf("s[%d] = %d, t[%d] = %d\n", n, s[n], n, t[n]);
  }
  free(s); free(t); // Release arrays
```

Comparing malloc() and calloc() Test program - Part 2

```
  // Re-create dynamic arrays
  s = malloc(4*sizeof(int));
  t = calloc(4, sizeof(int));

  // Print contents in arrays
  for (n=0; n<4; n++) {
    printf("s[%d] = %d, t[%d] = %d\n", n, s[n], n, t[n]);
  }
  free(s); free(t); // Release arrays
  return 0;
}
```

Malloc() example 2

- Ask user for size of memory space for the dynamic array
- MALLOC() is used to create memory space
- FOR loop used to read data from user
- Data output to screen using another FOR loop

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, x;
  int *sp;
  printf("Enter memory size (int): ");
  scanf("%d", &x);
  sp = malloc(x * sizeof(int));
  for (n=0; n<x; n++) {
    printf("Enter integer %d: ", n+1);
    scanf("%d", &sp[n]);
  }
  for (n=0; n<x; n++) {
    printf("Integer %d: %d\n", n+1, sp[n]);
  }
  free(sp);
  return 0;
}
```

Calloc() example 2

- Ask user for size of memory space for the dynamic array
- CALLOC() is used to create memory space
- FOR loop used to read data from user
- Data output to screen using another FOR loop

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int n, x;
  int *sp;
  printf("Enter memory size (int): ");
  scanf("%d", &x);
  sp = calloc(x, sizeof(int));
  for (n=0; n<x; n++) {
    printf("Enter integer %d: ", n+1);
    scanf("%d", &sp[n]);
  }
  for (n=0; n<x; n++) {
    printf("Integer %d: %d\n", n+1, sp[n]);
  }
  free(sp);
  return 0;
}
```

Error handling

Errors

- The program has to be designed to ensure no syntax errors occur during compilation
- Errors may come from the user or values entered by the user
- Need to anticipate errors that could happen during program use
- C language provides specialised functions for reporting on errors
- The library header file errno.h contains useful error codes

Ferror()

- FERROR() checks for the error indicator in a stream

```
<var> = ferror(<file_ptr>);
```

- Example,

```
if (ferror(fp)) {
  printf("Error\n");
}
```

Ferror() example 1

- The program is designed to read from data.txt file
- But the error is the “w” WRITE mode of operation
- This will cause the error message to be printed
- To fix the error, change to the READ mode of operation to “r”

```c
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "w");

  fgets(x, 20, fp);
  if (ferror(fp)) {
    printf("Error reading file\n");
  } else {
    printf("%s\n", x);
  }
  fclose(fp);
  return 0;
}
```

Ferror() example 2

- The program is designed to write to data.txt file
  - Ensure data.txt exists before running the program
- But the error is the “r” READ mode of operation
- This will cause the error message to be printed
- To fix the error, change to the WRITE mode of operation to “w”

```
#include <stdio.h>
int main()
{
  char x[20] = "Hello";
  FILE *fp;
  fp = fopen("data.txt", "r");
  if (fp == NULL) {
    printf("Error in opening file!\n");
    return 1;
  }

  fputs(x, fp);
  if (ferror(fp)) {
    printf("Error writing file\n");
  }
  fclose(fp);
  return 0;
}
```

Ferror() example 3

Note:
Change the mode of operation between “w” or “r” and observe the results on the terminal screen.

```
#include <stdio.h>
int main()
{
  char x[20] = "Hello";
  FILE *fp;
  fp = fopen("data.txt", "w");
  if (fp == NULL) {
    printf("Error in opening file\n");
    return 1;
  }
  // Write
  fputs(x, fp);
  if (ferror(fp)) {
    printf("Error in writing file\n");
    return 2;
  } else {
    printf("\"%s\" was written successfully\n", x);
  }

  // Read
  fgets(x, 20, fp);
  if (ferror(fp)) {
    printf("Error in reading file\n");
    return 3;
  } else {
    printf("\"%s\" was read successfully\n", x);
  }
  fclose(fp);
  return 0;
}
```

Clearerr()

- CLEARERR() clears end-of-file and error indicators for a stream

```
clearerr(<file_ptr>);
```

- Example,

```
clearerr(fp);
if (feof(fp)) {
  printf("EOF set\n");
} else {
  printf("EOF clear\n");
}
```

Clearerr() example 1

- The program will indicate if the EOF indicator is set or clear when it reads every line in data.txt file
- However, the statement `clearerr(fp);` will cause the WHILE loop to become infinite

```
// Contents of data.txt
apple
ball
cat
```

```
// Output
EOF clear
EOF clear
EOF clear
EOF set
```

```c
#include <stdio.h>
#include <string.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "r");

  while (!feof(fp)) {
    strcpy(x, "");
    fgets(x, 20, fp);
    clearerr(fp);
    if (feof(fp)) {
      printf("EOF set\n");
    } else {
      printf("EOF clear\n");
    }
  }
  fclose(fp);
  return 0;
}
```

Clearerr() example 2

- Program has two parts when operating on data.txt file
  - Part 1: Write to the file, which causes an error message to be printed
  - Part 2: Read from the file, which will be successful
  - Change the mode of operation to “w” and see the opposite effect
- The error indicator is cleared using CLEARERR(), otherwise both write and read operations will result in error
  - To test this issue, comment out the CLEARERR() statements

```c
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "r");
  if (fp == NULL) {
    printf("Error in opening file!\n”);
    return 1;
  }

  // Write
  fputs("Hello", fp);
  if (ferror(fp)) {
    printf("Error writing file\n");
    clearerr(fp);
    return 2;
  } else {
    printf("Write successful\n”);
  }
```

```c
  // Read
  fgets(x, 20, fp);
  if (ferror(fp)) {
    printf("Error reading file\n");
    clearerr(fp);
    return 3;
  } else {
    printf("Read successful : %s\n", x);
  }
  fclose(fp);
  return 0;
}
```

Before reading a file

- After a file pointer is assigned to a file, we need to check if the file exists and readable
- Alternatively, the functions PERROR() or STRERROR() can be used to provide more information

```
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "r");
  if (fp == NULL) {
    printf("Error in opening file!\n");
    return 1;
  }
  fgets(x, 20, fp);
  printf("%s", x);
  fclose(fp);
  return 0;
}
```

Perror()

- PERROR() is used to print specific error message from the operating system

```
FILE *fp;
fp = fopen("data.txt","r");
if (fp == NULL) {
  perror("Error ");
  return 1;
}
```

Perror() example

- PERROR() will use the information provided by the operating system in the error message
- Assuming the data.txt file does not exist, then the error message is:

```
Error : No such file or directory
```

```c
#include <stdio.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "r");
  if (fp == NULL) {
    perror("Error ");
    return 1;
  }
  fgets(x, 20, fp);
  printf("%s", x);
  fclose(fp);
  return 0;
}
```

Strerror()

- STRERROR() provides textual description of the system error code (errnum)
- The errnum is usually acquired from the errno variable
- Library header files string.h and errno.h must be included

```
FILE *fp;
fp = fopen("data.txt","r");
if (fp == NULL) {
  printf("Error : %s\n", strerror(errno));
  return 1;
}
```

Strerror() example

- STRERROR() uses the error number (errno) code to provide more information in the error message
- Assuming the data.txt file does not exist, then the error message is:

```
Error : No such file or directory
```

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
int main()
{
  char x[20];
  FILE *fp;
  fp = fopen("data.txt", "r");
  if (fp == NULL) {
    printf("Error : %s\n",
      strerror(errno));
    return 1;
  }
  fgets(x, 20, fp);
  printf("%s", x);
  fclose(fp);
  return 0;
}
```

Math error

- Using errno.h library, the error number macro (errno) can report on specific errors when math operations are involved
- At start of C program, errno is zero
- For MATH_ERRNO the possible errors:
    - EDOM – domain error
    - ERANGE – pole or range error

Math error message

- Ensure math.h and errno.h has been included in the program
- Any mathematical operation that goes beyond the domain or range of the result will trigger an error
- Type of error is stored in errno variable
- Custom error message has to be prepared for the specific math error

Math error example

- If the variable result is initialised with log(-10), it will cause EDOM error
- If the variable result is initialised with log(0), it will cause ERANGE error

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
int main()
{
    float result = log(-10); // EDOM
    //float result = log(0); // ERANGE
    if (errno == ERANGE) {
        printf("ERANGE errno = %d\n", errno);
    } else if (errno == EDOM) {
        printf("EDOM errno = %d\n", errno);
    } else {
        printf("Result = %f\n", result);
    }
    return 0;
}
```

Testing for math errors

- The math.h library contains several useful functions to check on mathematical results
  - isinf() – Infinity, divide by zero
  - isnan() – Not a Number (NaN)
  - isnormal() - Normalised
  - isfinite() – Finite number

```
#include <stdio.h>
#include <math.h>
int main()
{
  float result = 4.0/0.0;
  if (isinf(result)) {
    printf("Error: divide-by-zero\n");
  } else {
    printf("Result = %f\n", result);
  }
  return 0;
}
```

Summary

- Using external files give the program permanent data storage
- Files can be either text file or binary file
- Dynamic memory allocation is useful when array size is not planned
- Error handling depends on applying suitable error handling functions at the correct location in the program

Further readings

- C: How to Program, 8th Edition, Paul Deitel & Harvey Deitel
  - Files: pp. 474–491
  - Errors: pp. 396, 491
- The C Programming Language, 2nd Edition, Kernighan & Ritchie
  - Files: pp. 142–145
  - Errors: pp. 145–146