C PROGRAMMING
Lecture 9

1st semester 2023 - 2024

Input/Output

- Most programs require interaction (input or output.
- I/O is not directly supported by C.
- I/O is handled using standard library functions defined in <stdio.h>.
  - stdio.h header file contains function declarations for I/O.

Standard I/O Library

- Streams and FILE objects
- Three types of buffering
- Open a stream
- read/write a stream
- Positioning a stream
- Formatted I/O

Streams and FILE Objects

- Standard I/O
  - File stream driven - FILE*
  - Associate a file with a stream and operate on the stream
  - 3 pre-defined streams
    - stdin
    - stdout
    - stderr

Streams and FILE Objects

```
#include <stdio.h>

int main() {
    char buffer[100];

    // Reading from keyboard (stdin)
    printf("Enter a line of text: ");
    fgets(buffer, sizeof(buffer), stdin);

    // Displaying on the screen (stdout)
    printf("You entered: %s", buffer);

    return 0;
}
```

Streams and FILE Objects

```
#include <stdio.h>

int main() {
    int num;

    // Reading from the keyboard (stdin)
    printf("Enter a number: ");
    scanf("%d", &num);

    // Simple operation (e.g., doubling the number)
    int result = 2 * num;

    // Displaying on the screen (stdout)
    printf("Result of doubling: %d\n", result);

    // Displaying an error message on the screen (stderr)
    fprintf(stderr, "This is an error message (for example purposes only).\n");

    return 0;
}
```

Buffering

- Goal of buffering is to minimize the number of read and write calls.
- Three types of buffering
  - Fully buffered – Block Buffered
  - Line buffered
  - Unbuffered

Buffering

- **Fully Buffered (Block Buffered):**

Data is accumulated in a buffer until the buffer is full or until an explicit flush operation is triggered. This is often used for disk files. For example, when writing to a file, the data is collected in the buffer until it reaches a certain size or until the buffer is explicitly flushed, at which point the entire content of the buffer is written to the file.

Buffering

- **Line Buffered:**

Data is accumulated in the buffer until a newline character is encountered. This is commonly used for streams related to terminals.
For example, when reading from or writing to a terminal, the buffer is flushed when a newline character is encountered.

Buffering

Unbuffered:

Data is not stored in a buffer, and each character is read or written directly. This can be less efficient, as it may result in a higher number of system calls. For example, error messages are often unbuffered to ensure that they are immediately displayed.

Setting Buffer Type

setbuf

```
void setbuf(FILE *stream, char *buf);
```

stream is a pointer to the FILE structure representing the stream for which buffering is to be set.

buf is a pointer to the buffer that will be used for buffering.

If buf is NULL buffering is disabled
Buffer must be of size BUFSIZ as defined in <stdio.h>

Setting Buffer Type

```
#include <stdio.h>

int main() {
    FILE *filePointer;
    char buffer[BUFSIZ];

    // Opening a file in write mode
    filePointer = fopen("example.txt", "w");

    // Check if the file was opened successfully
    if (filePointer == NULL) {
        perror("Error opening the file");
        return 1;
    }

    // Enabling buffering with a user-provided buffer
    setbuf(filePointer, buffer);

    // Writing to the file
    fprintf(filePointer, "This is a line.\n");

    // Closing the file
    fclose(filePointer);

    return 0;
}
```

Setting Buffer Type

- setvbuf

```
int setvbuf(FILE *stream, char *buf, int mode , size_t size);
```

  - Can set all 3 types of buffering depending on value of *mode*
    - _IOFBF - Fully buffered
    - _IOLBF - Line buffered
    - _IONBF - Non buffered
  - If buf is NULL, system will create its own buffer automatically
  - buf is ignored if mode is _IONBF
  - `setbuf(fp, NULL)` is the same as `setvbuf(fp, buf, _IONBF, size)`

Setting Buffer Type

```
#include <stdio.h>

int main() {
    FILE *filePointer;
    char buffer[BUFSIZ];

    // Opening a file in write mode
    filePointer = fopen("example.txt", "w");

    // Check if the file was opened successfully
    if (filePointer == NULL) {
        perror("Error opening the file");
        return 1;
    }

    // Set buffering to fully buffered with a user-provided buffer
    // You can replace _IOFBF with _IOLBF or _IONBF for different buffering modes
    if (setvbuf(filePointer, buffer, _IOFBF, BUFSIZ) != 0) {
        perror("Error setting buffer type");
        return 1;
    }

    // Writing to the file
    fprintf(filePointer, "This is a line.\n");

    // Closing the file
    fclose(filePointer);

    return 0;
}
```

Opening a Stream

```
FILE *fopen(const char *path,const char *mode);
FILE *freopen(const char *path,const char *mode,FILE *stream);
FILE *fdopen(int fildes, const char *mode);
```

- fopen opens the file given by *path*.
- freopen opens a file on a specified stream, closing the stream first if it is already open
- fdopen opens a stream from a file descriptor. Useful for streams that don’t have a regular file such a pipes

Opening a Stream

- mode
  - r or rb
  - w or wb
  - a or ab
  - r+ or r+b or rb+
  - w+ or w+b or wb+
  - a+ or a+b or ab+

Opening a Stream

Text Mode (no 'b' in the mode):

'r': Open for reading. The file must exist.
'w': Open for writing. If the file exists, it is truncated. If the file does not exist, it is created.
'a': Open for appending. If the file does not exist, it is created. Data is written at the end of the file.

Opening a Stream

Binary Mode (with 'b' in the mode):

'rb': Open for reading in binary mode.
'wb': Open for writing in binary mode. If the file exists, it is truncated. If the file does not exist, it is created.
'ab': Open for appending in binary mode. If the file does not exist, it is created. Data is written at the end of the file.

Opening a Stream

Text and Binary Mode with Read and Write (with '+' in the mode):

'r+' or 'r+b' or 'rb+': Open for both reading and writing. The file must exist.
'w+' or 'w+b' or 'wb+': Open for both reading and writing. If the file exists, it is truncated. If the file does not exist, it is created.
'a+' or 'a+b' or 'ab+': Open for both reading and appending. If the file does not exist, it is created. Data is written at the end of the file.

Opening a Stream

```
#include <stdio.h>

int main() {
    FILE *filePointer;
    char buffer[100];

    // Open file for reading
    filePointer = fopen("example.txt", "r");

    if (filePointer == NULL) {
        perror("Error opening the file");
        return 1;
    }

    // Read from the file
    fgets(buffer, sizeof(buffer), filePointer);
    printf("Content: %s", buffer);

    // Close the file
    fclose(filePointer);

    return 0;
}
```

Reading and Writing a Stream

- Three ways to read and write
  - One character at a time
  - One line at a time
  - Direct (binary) I/O
- Character at a time

```
int getc(FILE *stream);
int fgetc(FILE *stream);
int getchar(void);
```

Reading and Writing a Stream

- Handling errors

```
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
```

- “Unreading”
  - We can put a single character back into the stream

    ```
    int ungetc(int c, FILE *stream);
    ```

Reading and Writing a Stream

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

feof end-of-file (EOF) indicator for a given file stream has been set. This function is typically used in conjunction with file input/output operations to determine if the end of a file has been reached.

Reading and Writing a Stream

```
#include <stdio.h>
int main() {
    FILE *file = fopen("example.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    int character;
    while ((character = fgetc(file)) != EOF) {
        // Process the character
        putchar(character);
    }
    if (feof(file)) {
        printf("\nEnd of file reached.\n");
    } else {
        printf("\nEnd of file not reached.\n");
    }
    fclose(file);
    return 0;
}
```

Reading and Writing a Stream

The **ferror** function checks the error indicator associated with the specified file stream.
The error indicator is set by certain file operations (e.g., unsuccessful read or write operations).
If an error has occurred, ferror returns a non-zero value; otherwise, it returns 0.

Reading and Writing a Stream

```
#include <stdio.h>

int main() {
    FILE *filePointer;

    filePointer = fopen("example.txt", "r");

    if (filePointer == NULL) {
        perror("Error opening the file");
        return 1;
    }

    // Attempting to read from a file that doesn't exist
    int ch = fgetc(filePointer);

    // Check for error
    if (ferror(filePointer)) {
        perror("Error reading from the file");
    } else {
        printf("Read character: %c\n", ch);
    }

    fclose(filePointer);

    return 0;
}
```

Reading and Writing a Stream

The **clearerr** function is a standard library function in C and C++ programming languages. It is used to clear the end-of-file (EOF) and error indicators for a given file stream, allowing further input/output operations on the file to be performed.

The purpose of clearerr is to reset the end-of-file and error indicators for a file stream. After calling clearerr, you can use the file stream for additional I/O operations without being affected by previous EOF or error conditions.

Reading and Writing a Stream

```
#include <stdio.h>

int main() {
    FILE *file = fopen("example.txt", "r");

    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    int character;
    while ((character = fgetc(file)) != EOF) {
        // Process the character
        putchar(character);
    }

    if (feof(file)) {
        printf("\nEnd of file reached.\n");
    } else {
        printf("\nEnd of file not reached.\n");
    }

    // Clear the end-of-file and error indicators
    clearerr(file);

    // Now you can perform additional I/O operations on the file

    fclose(file);
    return 0;
}
```

Reading and Writing a Stream

```
int putc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int putchar(int c);
```

- Returns c if ok, EOF on error

- getc and putc may be more efficient than fgetc and fputc because macros do not have the overhead of calling a function

Reading and Writing a Stream

- Line at a time I/O
  - Read

```
char *gets(char *s);
char *fgets(char *s,int size,FILE *stream);
```

  - Write

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

Example – v1

```
#include <stdio.h>
int main ()
{
   FILE *fp;
   int i, c, cnt=0;
   char buff[1000];
   fp = fopen("file1.txt","r");
   if(fp == NULL)
   {
      perror("Error in opening file\n");
      return(-1);
   }
```

Example – v1

```
 do{
        for (cnt=0;(c=fgetc(fp)) !='\n'&& c!=EOF;cnt++){
                buff[cnt] = c;
        }
        buff[cnt] = '\0';
        for(i=cnt-1;i>=0;i--)
                printf("%c", buff[i]);
        printf("\n");
        cnt=0;
        if( feof(fp) )
                break ;
   }while(1);
   fclose(fp);
   return(0);
}
```

Example – v2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main ()
{
   FILE *fp;
   int c;
   char buff[1000];
   fp = fopen("file1.txt", "r");
   while (1) {
        if (fgets(buff,1000, fp) == NULL) break;
        for(c=(int)strlen(buff);c>=0;c--)
                printf("%c",buff[c]);
  }
return 0;
}
```