CSC215
Lecture

Input and Output

Outline

- Introduction
- Standard files
- General files I/O
- Command-line parameters
- Error handling
- String I/O

Introduction

- C has no built-in statements for input or output
- Input and output functions are provided by the standard library `<stdio.h>`
- All input and output is performed with streams:
  - Stream: a sequence of bytes
    - text stream: consists of series of characters organized into lines ending with `'\n'`
      The standard library takes care of conversion from `"\r\n" to '\n'`
    - binary stream: consists of a series of raw bytes
  - The streams provided by standard library are buffered
- Streams are represented by the data type `FILE*`
  - FILE is a struct contains the internal state information about the connection to the file

Standard Files

- Standard input stream:
  - called `stdin`
  - normally connected to the keyboard
  - OS knows it by number 0
- Standard output stream:
  - Called `stdout`
  - normally connected to the display screen
  - OS knows it by number 1
- Standard error stream:
  - called `stderr`
  - also normally connected to the screen
  - OS knows it by number 2

Standard Files

- `int putchar(int char)`
  - Writes the character (an unsigned char) `char` to `stdout`
  - returns the character printed or `EOF` on error
- `int puts(const char *str)`
  - Writes the string `str` to `stdout` up to, but not including, the null character
  - A newline character is appended to the output
  - returns non-negative value, or `EOF` on error
- `int getchar(void)`
  - reads a character (an unsigned char) from `stdin`
  - returns `EOF` on error
- `char *gets(char *str)`
  - Reads a line from `stdin` and stores it into the string pointed to by `str`
  - It stops when either: the newline character is read or when the end-of-file is reached, whichever comes first
  - Prone to overflow problem

Standard Files

- `int scanf(const char *format, ...)`
  - Reads formatted input from `stdin`
  - Prone to overflow problem when used with strings
- `int printf(const char *format, ...)`
  - Sends formatted output to `stdout`
- `void perror(const char *str)`
  - prints a descriptive error message to `stderr`
  - string `str` is printed, followed by a colon then a space.
- What does the following code do?

```
int main ( ){
  char c ;
  while ((c=getchar())!= EOF){
    if ( c >= 'A' && c <= 'Z')
      c = c - 'A' + 'a';
      putchar(c) ;
  }
  return 0;
}
```

Standard Files

- Redirecting standard streams:
  - Provided by the operating system
  - Redirecting `stdout`: `prog > output.txt`
    and to append: `prog >> output.txt`
  - Redirecting `stderr`: `prog 2> error.txt`
    and to append: `prog 2>> error.txt`
  - Redirecting to `stdin:` `prog < input.txt`
  - Redirect the output of prog1 to the input of prog2: `prog1 | prog2`

General Stream I/O

- So far, we have read from the standard input and written to the standard output
- C allows us to read data from any text/binary files
- `FILE* fopen(char *filename,char *mode)`
  - opens file `filename` using the given `mode`
  - returns a pointer to the file stream
  - or NULL otherwise.
- `int fclose(FILE* fp)`
  - closes the stream (releases OS resources).
  - all buffers are flushed.
  - returns 0 if successful, and EOF otherwise.
  - automatically called on all open files when program terminates

`r`	For reading. File must exist
`w`	Creates empty file for writing. If file exists, it content is erased.
`a`	Appends to an existent file. Creates one if not exist.
`r+`	For reading & writing. File must exist
`w+`	Creates a file for reading & writing.
`a+`	For reading and appending

General Stream I/O

- `int getc(FILE* stream)`
  - reads a single character from the stream.
  - returns the character read or EOF on error/end of file.
  - We can implement it as follows: `#define getchar() getc(stdin)`
- `char* fgets(char *line, int maxlen, FILE* fp)`
  - reads a single line (upto maxlen characters) from the input stream (including linebreak)
  - stops when reading n-1 characters, reading \n or reaching end of file
  - returns a pointer to the character array that stores the line
  - returns NULL if end of stream.
- `int fscanf(FILE* fp, char *format, ...)`
  - similar to scanf,sscanf
  - reads items from input stream fp.
  - returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure

General Stream I/O

- `int ungetc(int ch, FILE *stream)`
  - pushes `ch` (unsigned char) onto the specified `stream` to be read again.
  - returns character that was pushed back if successful, otherwise EOF
- `int putc(int ch, FILE* fp)`
  - writes a single character `ch` to the output stream.
  - returns the character written or EOF on error.
  - we can implement it as follows: `#define putchar(c) putc(c,stdout)`
- `int fputs(char *line, FILE* stream)`
  - writes a single line to the output stream.
  - returns 0 on success, EOF otherwise.
- `int fprintf(FILE *stream, const char *format, ...)`
  - sends formatted output to a stream
  - returns total number of characters written, otherwise, a negative number is returned.

General Stream I/O

- `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`.
  - size: size in bytes of each element to be read
  - nmemb: number of elements, each one with a size of size bytes.
  - returns total number of elements successfully read.
    - if differs from `nmemb`, either an error has occurred or EOF was reached.
- `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`
  - returns total number of elements successfully written
    - if differs from `nmemb`, it will show an error
- `void rewind(FILE *stream)`
  - sets file position to beginning of `stream`.
- `int fseek(FILE *stream, long int offset, int whence)`
  - sets file position of `stream` to `offset`
  - `offset` signifies number of bytes to seek from given `whence` position

SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End of file

Example: std.h

```c
typedef struct{
  int id;
  char name[25];
  float gpa;
} Student;

int save_students_data(char*, Student*, int);

Student* get_students_data(char*, int*);

Student enter_student_data();

void print_student_data(Student*);
```

Example: std.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "std.h"

int save_students_data(char* fn, Student* slist, int num){
  FILE* fp;
  int i;

  if ((fp = fopen(fn, "w"))){
    fwrite(&num, sizeof(int), 1, fp);
    for (i=0; i<num; i++)
      if (!fwrite(slist+i, sizeof(Student), 1, fp)) {
        perror("Problem writing to file");
        return -2;
      }
    fclose(fp);
    return 0;
  }
  perror("File could not be opened.");
  return -1;
}
```

```c
if ((fp = fopen(fn, "w"))){
  fwrite(&num, sizeof(int), 1, fp);
  if (!fwrite(slist,
              sizeof(Student),
              Num,
              fp)) {
      perror("Problem writing to file");
      return -2;
  }
  fclose(fp);
  return 0;
}
```

Example: std.c (cont.)

```
Student* get_students_data(char* fn, int* num){
  FILE* fp;
  Student* result;
  int i;

  if ((fp = fopen(fn, "r"))){
    fread(num, sizeof(int), 1, fp);
    result = (Student*)calloc(*num, sizeof(Student));
    for (i=0; i<*num; i++)
      if (!fread(result+i, sizeof(Student), 1, fp)){
        perror("Problem reading from file");
        return NULL;
      }
    fclose(fp);
    return result;
  }
  perror("File could not be opened.");
  return NULL;
}
```

```
if ((fp = fopen(fn, "r"))){
  fread(&num, sizeof(int), 1, fp);
  result=(Student*)calloc(num,
                          sizeof(Student));
  if (!fread(result,
             sizeof(Student),
             num,
             fp)){
    perror("Problem reading from file");
    return NULL;
  }
  fclose(fp);
  return result;
}
```

Example: std.c (cont.)

```
Student enter_student_data(){
  Student s;
  printf("Enter student's id:");
  scanf("%d", &(s.id));
  printf("Enter student's name:");
  fgets(s.name, 24, stdin);
  printf("Enter student's GPA:");
  scanf("%f", &(s.gpa));
  return s;
}

void print_student_data(Student* s){
  printf("\n-----------------\n");
  printf("Student's id: %d\n", s->id);
  printf("Student's name: %s", s->name);
  printf("Student's GPA: %.2f\n", s->gpa);
  printf("-----------------\n");
}
```

Example: test-std.c

```
#include "std.h"

int main(){
  Student slist[3], *sff;
  int i, count;
  for (i=0; i<3; i++)
    slist[i] = enter_student_data();

  save_students_data("std.dat", slist, 3);

  sff = get_students_data("std.dat", &count);

  for (i=0; i<count; i++)
    print_student_data(sff+i);

  return 0;
}
```

Handling Files

- `int remove(const char *filename)`
  - deletes the given filename so that it is no longer accessible.
  - returns 0 on success and -1on failure and `errno` is set appropriately
- `int rename(const char *old_filename, const char *new_filename)`
  - causes filename referred to, by `old_filename` to be changed to `new_filename`.
  - returns 0 on success and -1on failure and `errno` is set appropriately
- How to get a file's size?
  - Use fseek with `long int ftell(FILE *stream)`
    - returns current file position of the given stream

  -
    ```
    FILE* f; long int size=0;
    if ((f = fopen("readme.txt"))){
      fseek(f, 0, SEEK_END);
      size = ftell(f);
      fclose(f);
    }
    ```

Command line Input

- In addition to taking input from standard input and files, you can also pass input while invoking the program.
  - so far, we have used int main() as to invoke the main function.
  - however, main function can take arguments that are populated when the program is invoked.
- `int main(int argc,char* argv[])`
  - `argc`: count of arguments.
  - `argv`: an array of pointers to each of the arguments
  - note: the arguments include the name of the program as well
  - Examples:

```
./cat a.txt b.txt
( argc = 3 , argv[0] = "cat" , argv[1] = "a.txt" and argv[2] = "b.txt" )
./cat
( argc = 1 , argv[0] = "cat" )
```

Error Handling

- No direct support for error handling
- `errno.h`
  - defines the global variable `errno`, set to zero at program startup
  - defines macros that indicate some error codes
- `char* strerror(int errnum)`
  - returns a string describing error errnum, must include `string.h`
- `stderr`
  - output stream for errors
  - assigned to a program just like `stdin` and `stdout`
  - appears on screen even if stdout is redirected
- `exit` function
  - terminates the program from any function, must include `stdlib.h`
  - argument is passed to the system
    - `EXIT_FAILURE , EXIT_SUCCESS`: defined in stdlib.h

Error Handling: Example

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno ;

int main () {
   FILE* pf;
   pf = fopen ("unexist.txt", "rb");
   if (pf == NULL) {

     int e = errno;
     fprintf(stderr, "Value of errno: %d\n", e);
     perror("Error printed by perror");
     fprintf(stderr, "Error opening file: %s\n", strerror(e));
   }
   else
      fclose (pf);
   return 0;
}
```

String I/O

- Instead of writing to the standard output, the formatted data can be written to or read from character arrays.
- `int sprintf(char *str, const char *format, ...)`
  - `format` specification is the same as `printf`.
  - output is written to `str` (does not check size).
  - returns number of character written or negative value on error.
- `int sscanf(const char *str, const char *format, ...)`
  - `format` specification is the same as `scanf`;
  - input is read from `str` variable.
  - returns number of items read or negative value on error.