Mathematical Software Programming (02635)

Lecture 6 — October 11, 2018

Instructor: Martin S. Andersen

Fall 2018

This week

Topics

- Strings and files

Learning objectives

- Design, implement, and document a program that solves a mathematical problem.
- Debug and test mathematical software.

Strings

A string is a *null-terminated* array of characters

```
char s[] = "Hello World!";
printf("%s\n",s);
printf("s is a char array of length %zu\n",sizeof(s));
printf("s is a string of length %zu\n",strlen(s));
```

What is the length of the `char` array `s`?

What is the length of the string?

- null-termination character is `\0`
- `s[0]` is the character `H`, `s[11]` is the character `!`, and `s[12]` is `\0`
- the character array may be (much) longer than the string
- include `<string.h>` to use functions such as `strlen()` or `strcmp()`

Programs with arguments

```c
/* main_demo.c */
#include <stdio.h>
int main(int argc, char const *argv[]) {
    printf("The user entered %d strings:\n",argc);
    for (int i=0;i<argc;i++)
        printf("%s\n",argv[i]);
    return EXIT_SUCCESS;
}
```

Running the program with three arguments yields the following output:

```
$ ./main_demo string1 string2 string3
The user entered 4 strings:
./main_demo
string1
string2
string3
```

Unsafe functions

Example: reading a string with `gets()`

```
char name_buffer[8];
printf("What is your name? ");
gets(name_buffer);                  // C11: 'gets' removed, 'gets_s' optional
```

- What happens if the user enters a name with more than 8 characters?
- Operating system may issue a warning when starting the program: `warning: this program uses gets(), which is unsafe.`

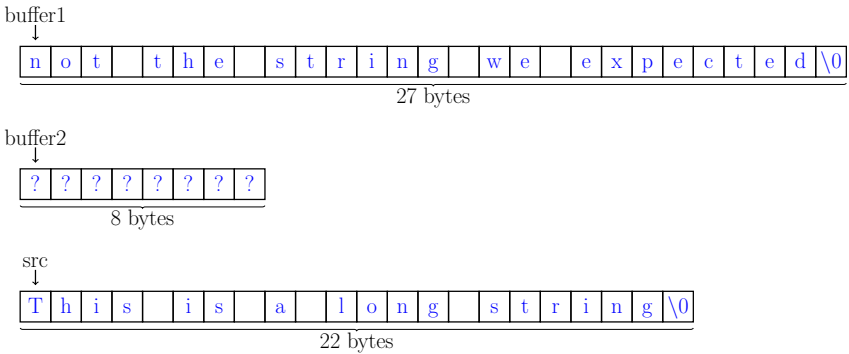
Safer alternative

```
char name_buffer[8];
printf("What is your name? ");
fgets(name_buffer, 8, stdin);
```

What happens now if the user enters a name with more than 8 characters?

String copy example

```
char buffer1[] = "not the string we expected";
char buffer2[8];
strcpy(buffer2, "This is a long string");
```



Working with text files

Opening and closing a file

```
FILE *fp = fopen("data.txt", "r"); // open file for reading

/* ... do something with the file ... */

fclose(fp);  // close file
fp = NULL;   // not necessary, but good practice
```

`fopen()` prototype

```
FILE *fopen(const char * name, const char * mode);
```

- several modes: reading (`"r"`), writing (`"w"`), appending (`"a"`)
- `FILE` is a `struct` defined in `stdio.h` (*opaque* data structure)
- returns a `FILE*` (a pointer to a `FILE`)

Reading and writing text to a file

Input prototypes

```
/* Read single character from file */
int fgetc(FILE *pfile);
/* Read string from file */
char * fgets(char *str, int nchars, FILE *pfile);
/* Read formattted input from file */
int fscanf(FILE *pfile, const char *format, ...);
```

Output prototypes

```
/* Write single character to file */
int fputc(int ch, FILE *pfile);
/* Write string to file */
int fputs(const char *str, FILE *pfile);
/* Write formatted output */
int fprintf(FILE *pfile, const char *format, ...);
```

Standard streams

Three standard streams of type `FILE*`

- standard input: `stdin` (`scanf` reads from `stdin`)
- standard output: `stdout` (`printf` writes to `stdout`)
- standard error: `stderr`

Example: write error message to `stderr`

```
double * arr = malloc(N*sizeof(*arr));
if (arr==NULL)
    fprintf(stderr,"Memory allocation failed.\n");
```

Reading strings with scanf/fscanf

```c
FILE *fp;
char buf[32];
// Open file
if ((fp = fopen("data.txt", "r")) == NULL) {
    fprintf(stderr,"Error opening file.\n");
    exit(EXIT_FAILURE);
};
// Read from file
if (fscanf(fp,"%31s",buf)!=1) {
    fprintf(stderr,"Error reading from file.\n");
    exit(EXIT_FAILURE);
}
```

- `fscanf` and `scanf` return the number of input items *assigned*
- `fscanf` and `scanf` need space for the null character `\0`
- using `%s` instead of `%[width]s` may lead to buffer overflow

Working with binary files

Opening a binary file

Mode strings: reading (`"rb"`), writing (`"wb"`), appending (`"ab"`)

Input/output prototypes

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *pfile);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *pfile);
```

Maintaining floating-point precision: binary vs. text format

- text file: format specifier `%.17g` for `double` and `%.9g` for `float`
- binary file: 8 bytes for `double` and 4 bytes for `single`

Example: write array to binary file

```c
/* Declare double array */
double data[] = {0.1,0.2,-0.1,-2.0,5.0,3.0};

/* Length of array */
size_t n = sizeof(data)/sizeof(double);

/* Open file and write data */
FILE *fp = fopen("data.dat","wb");
if ( fp == NULL ) exit(EXIT_FAILURE);
size_t ret = fwrite(data, sizeof(*data), n, fp);

/* Check return value and close file */
if ( ret != n ) fprintf(stderr,"Ups! Write error...\n");
fclose(fp);
fp = NULL;
```

Example: read array from binary file

```c
/* Declare double array */
double data[100];

/* Open file and read (at most) 100 doubles */
FILE *fp = fopen("data.dat","rb");
if ( fp == NULL ) exit(EXIT_FAILURE);
size_t ret = fread(data, sizeof(*data), 100, fp);
printf("Read %zu doubles.\n", ret);

/* Close file */
fclose(fp);
fp = NULL;
```

Error handling

- `errno.h` defines integer `errno` (initially zero)
- `string.h` defines function `char *strerror(int errnum)`
- `stdio.h` defines function `void perror(const char *s);`

```c
int main(void) {
    int errnum;
    FILE *fp;
    if ((fp = fopen("/path/to/file", "r")==NULL) {
        errnum = errno;
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror(errnum));
        return EXIT_FAILURE;
    }
    /* do something with file */
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Big-endian vs little-endian

Recall that many data types consist of multiple bytes

- a `double` consists of 8 bytes
- a `long` (typically) consists of 4 bytes or 8 bytes

What is the order of the bytes in memory?

Big-endian

Most significant byte has smallest memory address

Little-endian

Least significant byte has smallest memory address

Endianness

Checking for endianness

```
int i = 1;
char *p = (char *) &i;
if (*p == 1)
   printf("Your system is little-endian.\n");
else if (*(p+sizeof(int)-1) == 1)
   printf("Your system is big-endian.\n");
```

Common predefined macros

```
#define __BYTE_ORDER__ __ORDER_LITTLE_ENDIAN__
#define __ORDER_BIG_ENDIAN__ 4321
#define __ORDER_LITTLE_ENDIAN__ 1234
#define __ORDER_PDP_ENDIAN__ 3412
```

64-bit programming models

Model	short	int	long	long long	pointer	OS/compiler(s)
LP64	16	32	64	64	64	Most Unix-like systems
LLP64	16	32	32	64	64	Windows/Visual C++,MinGW
ILP64	16	64	64	64	64	

Remark: Most of today's 32-bit systems are ILP32 (int, long, and pointers are 32-bit)

Macros

List built-in macros with C preprocessor

```
$ cpp -dM /dev/null
#define __DBL_MIN_EXP__ (-1021)
#define __FLT_MIN__ 1.17549435e-38F
#define __CHAR_BIT__ 8
#define __WCHAR_MAX__ 2147483647
...
```

Windows: use `NUL` instead of `/dev/null`

Some macros depend on compiler options

```
$ gcc -dM -E - [options] < /dev/null
```

replace `[options]` with compiler flags

Midterm evaluation

- Please complete midterm evaluation by October 24, 2018
- It only takes a few minutes!

Quiz 2

1. Go to socrative.com on your laptop or mobile device
2. Enter "room number" **02635**
3. Answer ten quick question (the quiz is anonymous)