Chapter 18 I/O in C + Misc

Based on slides © McGraw-Hill
Modified by Diana Palsetia

I/O

From chapter 8 we know that I/O access is privileged as it involves accessing device registers.

"Normal" programs asks another privileged program such as OS to perform I/O on its behalf

Standard Library

I/O commands are not included as part of the C language

Instead, they are part of the Standard Library provided by the system

- Implementation depends on processor, operating system, etc., but interface is standard.

Since I/O commands are privileged routines written by system programmer, these are provided in library routine which then needs to included in your program.
e.g. #include <stdio.h>

Standard header files are contain declarations of functions, variables, or any structures

Basic I/O Functions

The standard I/O functions declared in the <stdio.h> header file are:

Function	*Description*
`putchar`	Displays an ASCII character to the screen.
`getchar`	Reads an ASCII character from the keyboard.
`printf`	Displays a formatted string,
`scanf`	Reads a formatted string.
`fopen`	Open/create a file for I/O.
`fprintf`	Writes a formatted string to a file.
`fscanf`	Reads a formatted string from a file.

Text Streams

All character-based I/O in C is performed on **text streams**

A stream is a **sequence of ASCII characters**, such as:

- the sequence of ASCII characters printed to the monitor by a single program
- the sequence of ASCII characters entered by the user during a single program
- the sequence of ASCII characters in a single file

Text Streams (contd..)

Characters are processed in the order in which they were added to the stream.

- E.g. a program sees input characters in the same order as the user typed them

Standard input stream (keyboard) is called **stdin**

Standard output stream (monitor) is called **stdout**

Character I/O

`int getchar(void)` - Reads one ASCII character from stdin.

- The value returned is the ascii value of the character. Similar to GETC trap routine in LC3

`int putchar (int character)` - Adds one ASCII character to stdout.

- The value returned is the written character. If an error occurs, EOF is returned
- Similar to “IN” trap routine in LC3

These functions deal with “ASCII characters” or “ASCII value” and hence no type conversion is performed.

```
char c = 'h';
...
putchar(c);
putchar('h');
putchar(104);
```

Each of these calls prints 'h' to the screen.

Buffered I/O or Delayed Effect

In many systems, characters are **buffered** during an I/O operation

Example: Keyboard (input stream)

- Characters are added to the buffer (temporary storage) and given to input stream (keyboard) only when the "Enter" key is pressed
- This allows user to correct input before confirming with “Enter”

Printing Special Character Literals in C

Certain characters cannot be easily represented by a single keystroke, because they
- correspond to whitespace (newline, tab, backspace, ...)
- are used as delimiters for other literals (quote, double quote, ...)

These are represented by the following sequences:

`\\n`	newline
`\t`	tab
`\\`	backslash
`\'`	single quote
`\"`	double quote

e.g. printf("The number is \"5\"");
Will print: The number is "5"

Missing Data Arguments printf

What happens when you don't provide a data argument for every formatting character?

```
printf("The value of nothing is %d\n");
```

`%d` will convert and print whatever is on the stack in the position where it expects the first argument.

Note: Something will be printed, but it will be a garbage value as far as our program is concerned.

scanf Conversion

For each data conversion, scanf will skip whitespace characters and then read ASCII characters until it encounters the first character that should NOT be included in the converted value.

`%d`	Reads until first non-digit.
`%x`	Reads until first non-digit (in hex).
`%s`	Reads until first whitespace character.

scanf Return Value

The scanf function returns an integer, which indicates the number of successful conversions performed.
- This lets the program check whether the input stream was in the proper format.
- This also includes any literals in format string i.e. must match literals in the conversion process

Example:

```
scanf("%d/ %d/ %d", &bMonth, &bDay, &bYear);
```

Input Stream	*Return Value*
`02/16/69`	`3`
`02/16 69`	`2`
`02 16 69`	`1`

Scanf Bad Input

Remember that characters are added to a buffer (temporary storage) and given to input stream (keyboard) only when the "Enter" key is pressed (buffered streaming)

```
#include <stdio.h>

int main(){
  int check = 0;
  int i = 0;
  while(i != 1){
    printf("Enter number\n");
    if((check = scanf("%d",&i)) != 1){
      printf("Error in input, must be a number");
    }
  }
}
```

What happens when you enter letter or float response to the prompt for an integer?

- Stuck in a while loop, why??

Scanf Bad Input (contd..)

Why?

1	2	.	4	5	'\n'

The picture shows the stream of input characters after the first call to scanf was complete. Here is what the first call did:

- Read the '1', saw that it was a digit and can be used as part of an int
- Read the '2', saw that it was a digit and can be used as part of an int
- Read the '.', saw that it was not a digit and could be not be used as part of an int. The '.' was put back on the input stream (in our e.g. stdin) so that it could be read by the next input operation
- So how do we take care of this ?

fflush(FILE * stream)

function in stdio.h

On an output stream (e.g. a file in write mode or stdout) fflush causes any buffered but unwritten data to be written

On an input stream (e.g. a file in a read or stdin), the behavior is undefined

Some platforms use fflush to clear

- E.g. fflush(stdin) has the same effect as setbuff(stdin, NULL) on eniac-s but not on eniac-l
- See demo

Scanf Bad Input (contd..)

How do we take care of it?

Need to clear/disable buffering with the input stream (in our example stdin).

void setbuf (FILE * *stream* , char * *buffer*);

- Causes the character array pointed to by the *buffer* parameter to be used instead of an automatically allocated buffer.
- If the specified *buffer* is NULL it disables buffering with the *stream*
  - E.g. setbuf(stdin, NULL)
- Use *setbuf()* function after a stream has been opened but before it is read or written.

Bad scanf Arguments

Two problems with scanf data arguments

1. Argument is not an address

```
int n = 0;
scanf("%d", n);
```

Will use the value of the argument as an address

2. Missing data argument

```
scanf("%d");
```

Will get address from stack, where it expects to find first data argument.

1. If you're lucky, program will crash because of trying to modify a restricted memory location (e.g., location 0). Runtime error: Segmentation Fault

2. Otherwise, your program will just modify an arbitrary memory location, which can cause very unpredictable behavior.

Carriage Return (CR) control character

In mechanical typewriters with levers

- CR allows the printing head to return to the left side of the paper after a line of text had been typed

Adopted in electronic typewriters and keyboards. Keys generating carriage return are keys with the following symbol/words:

- Enter
- Return

ASCII code for CR is 13 in decimal or 0D in hex

Line Feed (LF) control character

In a mechanical typewriter, LF advances the paper one line without moving the print head

In electronic typerwriter or keyboards, it is a control character that moves the position of the cursor to the next line

ASCII value for LF is 10 (in dec) or 0A(in hex)

CR+LF = used as line termination sequence

Window OS uses CR+LF to indicate line termination in the text files

Newline (“\n”)

In Unix/Linux OS, line feed is used as line termination sequence

- Implicit CR before LF is a unix invention, probably as an economy, since it saves one byte per line.

C language uses LF to terminate lines and called it the newline “\n” for convenience or readability. Hence ascii value for \n is same as LF i.e. x0A or 10

Line Termination in Text Files

UNIX text files

end of line marked by LF

Windows text files

end of line marked by CR followed by LF

Macintosh text files

end of line marked by CR

Note: Representing line termination vary between hardware platforms and operating systems, which can be problematic when exchanging data between systems of different types.

About Text Files

Text files can be displayed, edited by regular file editors

- contain only ASCII characters
- divided into lines; end of line marked by a control character e.g. CR, LF, or CR+LF
- a C source file is an example of a text file

Binary & executable files e.g. a.out

- contain arbitrary bit patterns; do not print or display with regular text editor

Word size vs. Memory Addressability

Word size

- a fixed-sized group of bits that are handled together by the machine
- Number of bits normally processed by ALU in 1 instr
- E.g. LC-3 word size is 16 bits = 2 bytes

Addressability

- How much information (in bits) can each memory location hold?
- E.g. LC3 – 16bits addressable

Byte Ordering

Memory is usually byte (s) addressable

- E.g. Linux Machines, is 1 byte addressable

An object/type is can be made of multiple words, but it is stored continuously in memory

If addressibilty < word size then

the ordering of the bytes in a word can be in two ways

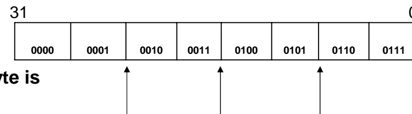
1. **Little Endian**
2. **Big Endian**

Little vs. Big Endian

Example:
# of locations in memory = 2^{16}
Addressability of memory = 1 byte = 8 bits
Word size of machine = 4 bytes = 32 bits
Sizeof(int) = 4 bytes

Addr	Data Little Endian	Data Big Endian
0x1000	x67	x01
0x1001	x45	x23
0x1002	x23	x45
0x1003	x01	x67

int i = 0x01234567 (= 19088743 in decimal)
&i = 0x1000



Little Endian: least significant byte is stored first. E.g. Intel ISA

Big Endian: Most significant byte is stored first. E.g. IBM, SUN ISA

Malloc(size)

If the size of the space requested is 0, the behavior is implementation-dependent

- The value returned will be either a null pointer or a non-null pointer
- Non-null pointer can occur if:
  - Some versions of malloc do not do error checking - assumes programmer knows best
  - Or in some versions of malloc If size is 0 then pretends that size is 1.

• So if you want to right portable code, you must do some extra checking

```
int * p = (int *)malloc(size * sizeof(int));
if(p != NULL && size != 0){
   //do something with dynamically allocated array
}
else{
   printf("Error during memory allocation")
   exit(1)
}
```


exit (int status)

- exit() is system call just like any I/O function in stdio.h
- It is in stdlib.h
- Allows termination of the program in case of an error
- Usually used when termination needs to be done in function other than main()

EXIT_SUCCESS	0	Normal termination
EXIT_FAILURE	1	Abnormal termination. Error.

Core Dump

A core dump is the name often given to the recorded state of the working memory of a computer program at a specific time, generally when the program that has terminated abnormally

Is a simply a binary file with the sequence of bytes or words containing the memory image of a particular process.

Name of file is "core" without any extensions and the file appears in the current working directory

Core Dump (contd..)

Abnormal Program Termination

- often occurs with buffer overflows, where a programmer allocates too little memory for incoming or computed data

- access to null pointers, a common coding error when an unassigned memory reference variable is accessed

Uses of core dump

- A runtime error such as "Segmentation Fault" does not describe as to where does the problem lie.

- Core dump can help reveal where the fault is occurring
  - Programmer can try to interpret the file but needs enough knowledge of structure of the programs memory use
  - Special Programs such as GDB debugger or dump analyzers can relieve the programmer from reading core file which is in hex bytes

Core Dump (contd..)

Issue with core file

- core file is often very large and takes up a lot of disk space
- One can avoid exceeding the disk quota with a large core file by limiting the core dump size

csh/tcsh shell

- limit coredumpsize 0 #zero size of core file
- limit coredumpsize 1024 # Limit core dumps to 1K
- unlimit coredumpsize # allow unlimited-sized coredumps

bash shell

- ulimit –c 0
- ulimit –c unlimited

Next Class

Review