

Chapter 18

I/O – Part I

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

CIT593

2/17

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 as library routine which then need to be included in your program.

e.g. #include <stdio.h>

header files contain function signature/prototypes, or any structures

CIT593

3/17

Basic I/O Functions

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

<i>Function</i>	<i>Description</i>
<code>putchar</code>	Displays an ASCII character to the screen.
<code>getchar</code>	Reads an ASCII character from the keyboard
<code>printf</code>	Displays a formatted string
<code>scanf</code>	Reads a formatted string

...

Some more we will look at later

CIT593

4/17

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

CIT593

5/17

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**

CIT593

6/17

Character I/O

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

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

These functions deal with "ASCII characters" or "ASCII value"

- Very limited in use

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

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

CIT593

7/17

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"

CIT593

8/17

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"

CIT593

9/17

printf

`printf` Displays a formatted string

`%d` – integer
`%u` – unsigned int
`%x` – hexadecimal integer
`%o` – octal integer
`%c` – character
`%s` – character string (input has to be char *)
`%f` – floating point
`%e` – scientific notation

CIT593

10/17

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");
```

- **Note:** Something will be printed, but it will be a garbage value as far as our program is concerned
- **Reason:** `%d` will convert and print whatever is on the stack in the position where it expects the first argument
 - More on this next time....

CIT593

11/17

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.

CIT593

12/17

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: (in scanf1.c)

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

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

CIT593

13/17

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)

//Example in scanf2.c
#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??**

CIT593

14/17

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 ?**

CIT593

15/17

Scanf Bad Input (contd..)

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)`
- **Important: Use `setbuf()` function after a stream has been opened but before it is read or written.**

CIT593

16/17

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
- 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. Missing data argument

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

- Your program will just modify an arbitrary memory location, which can cause very unpredictable behavior
 - Because it will get address from stack, where it expects to find first data argument (this will be clear when we do chp14)