

Standard I/O



# Advanced C

## Standard I/O - Why should I know this? - DIY

### Screen Shot

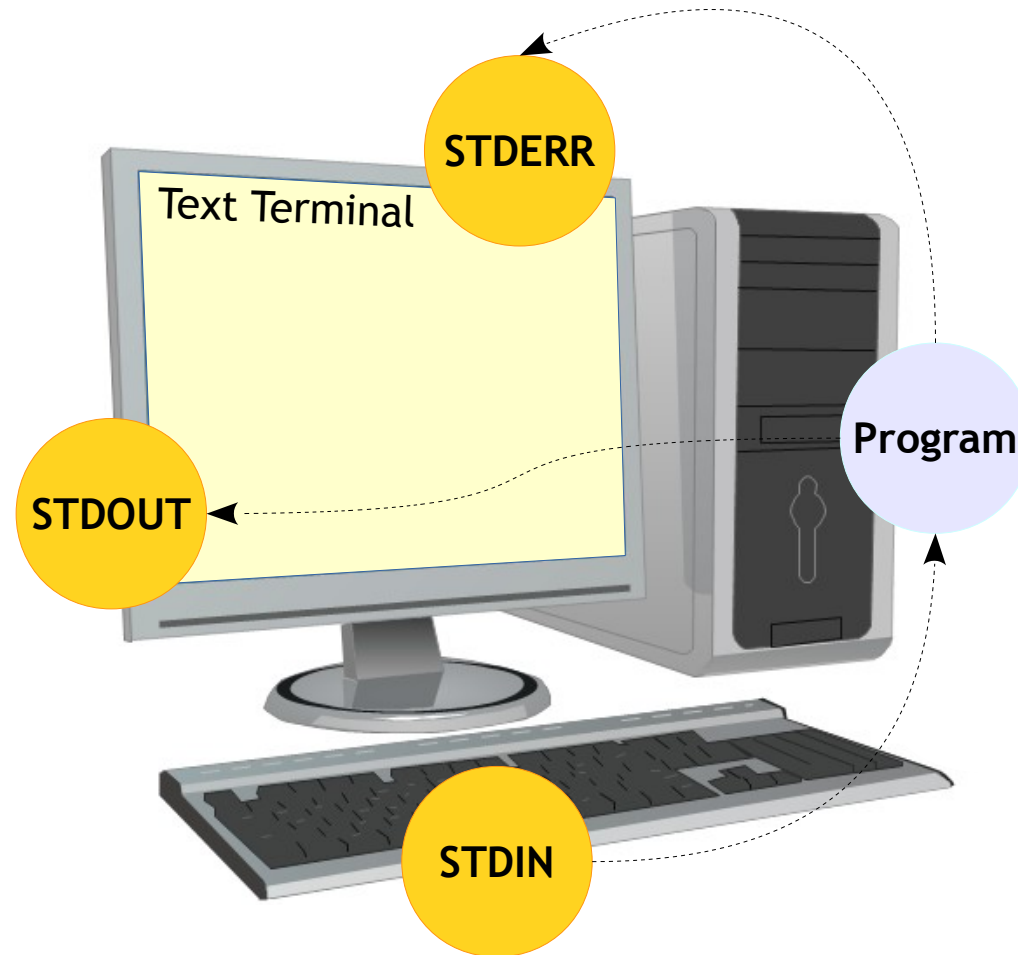
```
user@user:~]  
user@user:~] ./print_bill.out  
Enter the item 1: Kurkure  
Enter no of pcs: 2  
Enter the cost : 5  
Enter the item 2: Everest Paneer Masala  
Enter no of pcs: 1  
Enter the cost : 25.50  
Enter the item 3: India Gate Basmati  
Enter no of pcs: 1  
Enter the cost : 1050.00
```

```
-----  
S.No  Name           Quantity      Cost      Amount  
-----  
1.    Kurkure         2            5.00      10.00  
2.    Everest Paneer  1            25.50     25.50  
2.    India Gate Bas  1           1050.00   1050.00  
-----  
Total                4            1085.50  
-----
```

```
user@user:~]
```

# Advanced C

## Standard I/O



# Advanced C

## Standard I/O - The File Descriptors



- OS uses 3 file descriptors to provide standard input output functionalities
  - 0 → stdin
  - 1 → stdout
  - 2 → stderr
- These are sometimes referred as “The Standard Streams”
- The IO access example could be
  - stdin → Keyboard, pipes
  - stdout → Console, Files, Pipes
  - stderr → Console, Files, Pipes

# Advanced C

## Standard I/O - The File Descriptors



- Wait!!, did we see something wrong in previous slide?  
Both stdout and stderr are similar ?
- If yes why should we have 2 different streams?
- The answer is convenience and urgency.
  - Convenience : Diagnostic information can be printed on stderr. Example - we can separate error messages from low priority informative messages
  - Urgency : serious error messages shall be displayed on the screen immediately
- So how the C language help us in the standard IO?

# Advanced C

## Standard I/O - The header file



- You need to refer input/output library function

`#include <stdio.h>`

- When the reference is made with “<*name*>” the search for the files happen in standard path
- Header file Vs Library

# Advanced C

## Standard I/O - Unformatted (Basic)



- Internal binary representation of the data directly between memory and the file
- Basic form of I/O, simple, efficient and compact
- Unformatted I/O is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor
- `getchar()` and `putchar()` are two functions part of standard C library
- Some functions like `getch()`, `getche()`, `putch()` are defined in `conio.h`, which is not a standard C library header and is not supported by the compilers targeting Linux / Unix systems

# Advanced C

## Standard I/O - Unformatted (Basic)

001\_example.c

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    int ch;

    for ( ; (ch = getchar()) != EOF; )
    {
        putchar(toupper(ch));
    }

    puts("EOF Received");

    return 0;
}
```



# Advanced C

## Standard I/O - Unformatted (Basic)

002\_example.c

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    int ch;

    for ( ; (ch = getc(stdin)) != EOF; )
    {
        putc(toupper(ch), stdout);
    }

    puts("EOF Received");

    return 0;
}
```

# Advanced C

## Standard I/O - Unformatted (Basic)

003\_example.c

```
#include <stdio.h>

int main()
{
    char str[10];

    puts("Enter the string");
    gets(str);
    puts(str);

    return 0;
}
```

# Advanced C

## Standard I/O - Unformatted (Basic)

004\_example.c

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    char str[10];

    puts("Enter the string");
    fgets(str, 10, stdin);
    puts(str);

    return 0;
}
```

# Advanced C

## Standard I/O - Formatted



- Data is formatted or transformed
- Converts the internal binary representation of the data to ASCII before being stored, manipulated or output
- Portable and human readable, but expensive because of the conversions to be done in between input and output
- The `printf()` and `scanf()` functions are examples of formatted output and input

# Advanced C

## Standard I/O - printf()

005\_example.c

```
#include <stdio.h>

int main()
{
    char a[8] = "Emertxe";

    printf(a);

    return 0;
}
```

- What will be the output of the code on left side?
- Is that syntactically OK?
- Lets understand the printf() prototype
- Please type  
man printf  
on your terminal

# Advanced C

## Standard I/O - printf()

### Prototype

```
int printf(const char *format, ...);  
or  
int printf("format string", [variables]);
```

where format string arguments can be

`%[flags][width][.precision][length]type_specifier`

`%type_specifier` is mandatory and others are optional

- Converts, formats, and prints its arguments on the standard output under control of the format
- Returns the number of characters printed

# Advanced C

## Standard I/O - printf()

### Prototype

```
int printf(const char *format, ...);
```

What is this!?

# Advanced C

## Standard I/O - printf()

### Prototype

```
int printf(const char *format, ...);
```

What is this!?

- Is called as ellipses
- Means, you can pass any number (i.e 0 or more) of “optional” arguments of any type
- So how to complete the below example?

### Example

```
int printf("%c %d %f", );
```

What should be written here and how many?



# Advanced C

## Standard I/O - printf()

### Example

```
int printf("%c %d %f", arg1, arg2, arg3);
```

Now, how to you decide this!?

Based on the number of format specifiers

- So the **number of arguments** passed to the printf function should exactly **match** the **number of format specifiers**
- So lets go back the code again

# Advanced C

## Standard I/O - printf()

### Example

```
#include <stdio.h>

int main()
{
    char a[8] = "Emertxe";

    printf(a);

    return 0;
}
```

Isn't this a string?

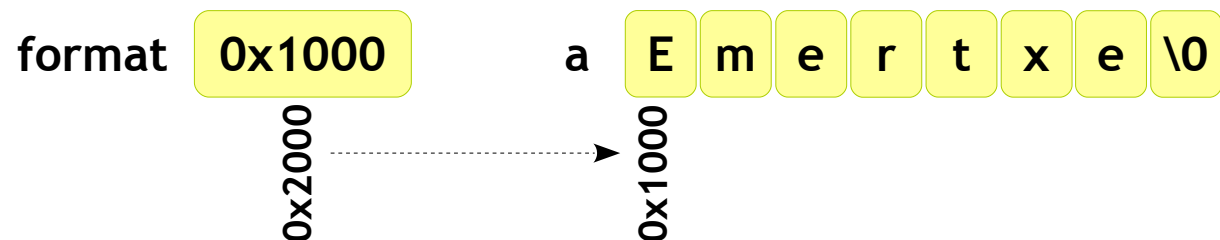
And strings are nothing but array of characters terminated by null

So what get passed, while passing a array to function?

```
int printf(const char *format, ...);
```

Isn't this a pointer?

So a pointer hold a address, can be drawn as



So the base address of the array gets passed to the pointer, Hence the output

Note: You will get a warning while compiling the above code.  
So this method of passing is not recommended

# Advanced C

## Standard I/O - printf() - Type Specifiers



006\_example.c

Specifiers	Example	Expected Output
%c	printf("%c", 'A')	A
%d %i	printf("%d %i", 10, 10)	10 10
%o	printf("%o", 8)	10
%x %X	printf("%x %X %x", 0xA, 0xA, 10)	a A a
%u	printf("%u", 255)	255
%f %F	printf("%f %F", 2.0, 2.0)	2.000000 2.000000
%e %E	printf("%e %E", 1.2, 1.2)	1.200000e+00 1.200000E+00
%a %A	printf("%a", 123.4) printf("%A", 123.4)	0x1.ed9999999999ap+6 0X1.ED9999999999AP+6
%g %G	printf("%g %G", 1.21, 1.0)	1.21 1
%s	printf("%s", "Hello")	Hello

# Advanced C

## Standard I/O - printf() - Type Length Specifiers



007\_example.c

Length specifier	Example	Example
%[h]X	printf(“%hX”, 0xFFFFFFFF)	FFFF
%[l]X	printf(“%lX”, 0xFFFFFFFFl)	FFFFFFFF
%[ll]X	printf(“%llX”, 0xFFFFFFFFFFFFFFFF)	FFFFFFFFFFFFFFFF
%[L]f	printf(“%Lf”, 1.23456789L)	1.234568

# Advanced C

## Standard I/O - printf() - Width



008\_example.c

Width	Example	Expected Output
%[x]d	printf(“%3d %3d”, 1, 1)	1 1
	printf(“%3d %3d”, 10, 10)	10 10
	printf(“%3d %3d”, 100, 100)	100 100
%[x]s	printf(“%10s”, “Hello”)	Hello
	printf(“%20s”, “Hello”)	Hello
%*[specifier]	printf(“%*d”, 1, 1)	1
	printf(“%*d”, 2, 1)	1
	printf(“%*d”, 3, 1)	1

# Advanced C

## Standard I/O - printf() - Precision



009\_example.c

Precision	Example	Expected Output
%[x].[x]d	printf(“%3.1d”, 1)	1
	printf(“%3.2d”, 1)	01
	printf(“%3.3d”, 1)	001
%0.[x]f	printf(“%0.3f”, 1.0)	1.000
	printf(“%0.10f”, 1.0)	1.0000000000
%[x].[x]s	printf(“%12.8s”, “Hello World”)	Hello Wo

# Advanced C

## Standard I/O - printf() - Flags



010\_example.c

Flag	Example	Expected Output
%[#]x	printf(“%#x %#X %#x”, 0xA, 0xA, 10) printf(“%#o”, 8)	0xa 0XA 0xa 010
%[-x]d	printf(“%-3d %-3d”, 1, 1) printf(“%-3d %-3d”, 10, 10) printf(“%-3d %-3d”, 100, 100)	1 1 10 10 100 100
%[ ]3d	printf(“% 3d”, 100) printf(“% 3d”, -100)	100 -100

# Advanced C

## Standard I/O - printf() - Escape Sequence



011\_example.c

Escape Sequence	Meaning	Example	Expected Output
<code>\n</code>	New Line	<code>printf("Hello World\n")</code>	Hello World (With a new line)
<code>\r</code>	Carriage Return	<code>printf("Hello\rWorld")</code>	World
<code>\t</code>	Tab	<code>printf("Hello\tWorld")</code>	Hello    World
<code>\b</code>	Backspace	<code>printf("Hello\bWorld")</code>	HelWorld
<code>\v</code>	Vertical Tab	<code>printf("Hello\vWorld")</code>	Hello World
<code>\f</code>	Form Feed	<code>printf("Hello World\f")</code>	Might get few extra new line(s)
<code>\e</code>	Escape	<code>printf("Hello\eWorld")</code>	Helloorld
<code>\\</code>		<code>printf("A\\B\\C")</code>	A\\B\\C
<code>\"</code>		<code>printf("\\"Hello World\"")</code>	"Hello World"



# Advanced C

## Standard I/O - printf()

- So in the previous slides we saw some 80% of printf's format string usage.

What?? Ooh man!!.. Now how to print **80%??**

# Advanced C

## Standard I/O - printf() - Example



012\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string[] = "Hello World";

    printf("%d %c %f %s\n", num1 , ch, num2, string);
    printf("%+05d\n", num1);
    printf("%.2f %.5s\n", num2, string);

    return 0;
}
```

# Advanced C

## Standard I/O - printf() - Return

013\_example.c

```
#include <stdio.h>

int main()
{
    int ret;
    char string[] = "Hello World";

    ret = printf("%s\n", string);

    printf("The previous printf() printed %d chars\n", ret);

    return 0;
}
```

# Advanced C

## Standard I/O - sprintf() - Printing to string

### Prototype

```
int sprintf(char *str, const char *format, ...);
```

- Similar to printf() but prints to the buffer instead of stdout
- Formats the arguments in arg1, arg2, etc., according to format specifier
- buffer must be big enough to receive the result

# Advanced C

## Standard I/O - sprintf() - Example

014\_example.c

```
#include <stdio.h>

int main()
{
    int num1 = 123;
    char ch = 'A';
    float num2 = 12.345;
    char string1[] = "sprintf() Test";
    char string2[100];

    sprintf(string2, "%d %c %f %s\n", num1 , ch, num2, string1);
    printf("%s", string2);

    return 0;
}
```

# Advanced C

## Standard I/O - Formatted Input - scanf()

### Prototype

```
int scanf(char *format, ...);  
or  
int scanf("string", [variables]);
```

- Reads characters from the standard input, interprets them according to the format specifier, and stores the results through the remaining arguments.
- Almost all the format specifiers are similar to printf() except changes in few
- Each “optional” argument must be a **pointer**

# Advanced C

## Standard I/O - Formatted Input - scanf()

- It returns as its value the number of successfully matched and assigned input items.
- On the end of file, EOF is returned. Note that this is different from 0, which means that the next input character does not match the first specification in the format string.
- The next call to scanf() resumes searching immediately after the last character already converted.

# Advanced C

## Standard I/O - scanf() - Example

015\_example.c

```
#include <stdio.h>

int main()
{
    int num1;
    char ch;
    float num2;
    char string[10];

    scanf("%d %c %f %s", &num1 , &ch, &num2, string);
    printf("%d %c %f %s\n", num1 , ch, num2, string);

    return 0;
}
```



# Advanced C

## Standard I/O - scanf() - Format Specifier



016\_example.c

Flag	Examples	Expected Output
%*[specifier]	scanf("%d%*c%d%*c%d", &h, &m, &s)	User Input → HH:MM:SS Scanned Input → HHMMSS  User Input → 5+4+3 Scanned Input → 543

# Advanced C

## Standard I/O - scanf() - Format Specifier



017\_example.c

Flag	Examples	Expected Output
%[]	scanf("%[a-z A-Z]", name)	User Input → Emertxe Scanned Input → Emertxe
		User Input → Emx123 Scanned Input → Emx
	scanf("%[0-9]", id)	User Input → 123 Scanned Input → 123
		User Input → 123XYZ Scanned Input → 123

# Advanced C

## Standard I/O - scanf() - Return



018\_example.c

```
#include <stdio.h>

int main()
{
    int num = 100, ret;

    printf("The enter a number [is 100 now]: ");
    ret = scanf("%d", &num);

    if (ret != 1)
    {
        printf("Invalid input. The number is still %d\n", num);
        return 1;
    }
    else
    {
        printf("Number is modified with %d\n", num);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - sscanf() - Reading from string

### Prototype

```
int sscanf(const char *string, const char *format, ...);
```

- Similar to scanf() but read from string instead of stdin
- Formats the arguments in arg1, arg2, etc., according to format

# Advanced C

## Standard I/O - sscanf() - Example



019\_example.c

```
#include <stdio.h>

int main()
{
    int age;
    char array_1[10];
    char array_2[10];

    sscanf("I am 30 years old", "%s %s %d", array_1, array_2, &age);
    sscanf("I am 30 years old", "%*s %*s %d", &age);
    printf("OK you are %d years old\n", age);

    return 0;
}
```

# Advanced C

## Standard I/O - DIY

### Screen Shot

```
user@user:~]  
user@user:~] ./print_bill.out  
Enter the item 1: Kurkure  
Enter no of pcs: 2  
Enter the cost : 5  
Enter the item 2: Everest Paneer Masala  
Enter no of pcs: 1  
Enter the cost : 25.50  
Enter the item 3: India Gate Basmati  
Enter no of pcs: 1  
Enter the cost : 1050.00
```

```
-----  
S.No  Name           Quantity      Cost      Amount  
-----  
1.    Kurkure         2            5.00      10.00  
2.    Everest Paneer  1            25.50     25.50  
3.    India Gate Bas  1           1050.00   1050.00  
-----  
Total                4            1085.50  
-----
```

```
user@user:~]
```

# Advanced C

## Standard I/O - DIY



### Screen Shot

```
user@user:~]  
user@user:~] ./progress_bar.out  
Loading [-----] 50%  
user@user:~]
```

# Advanced C

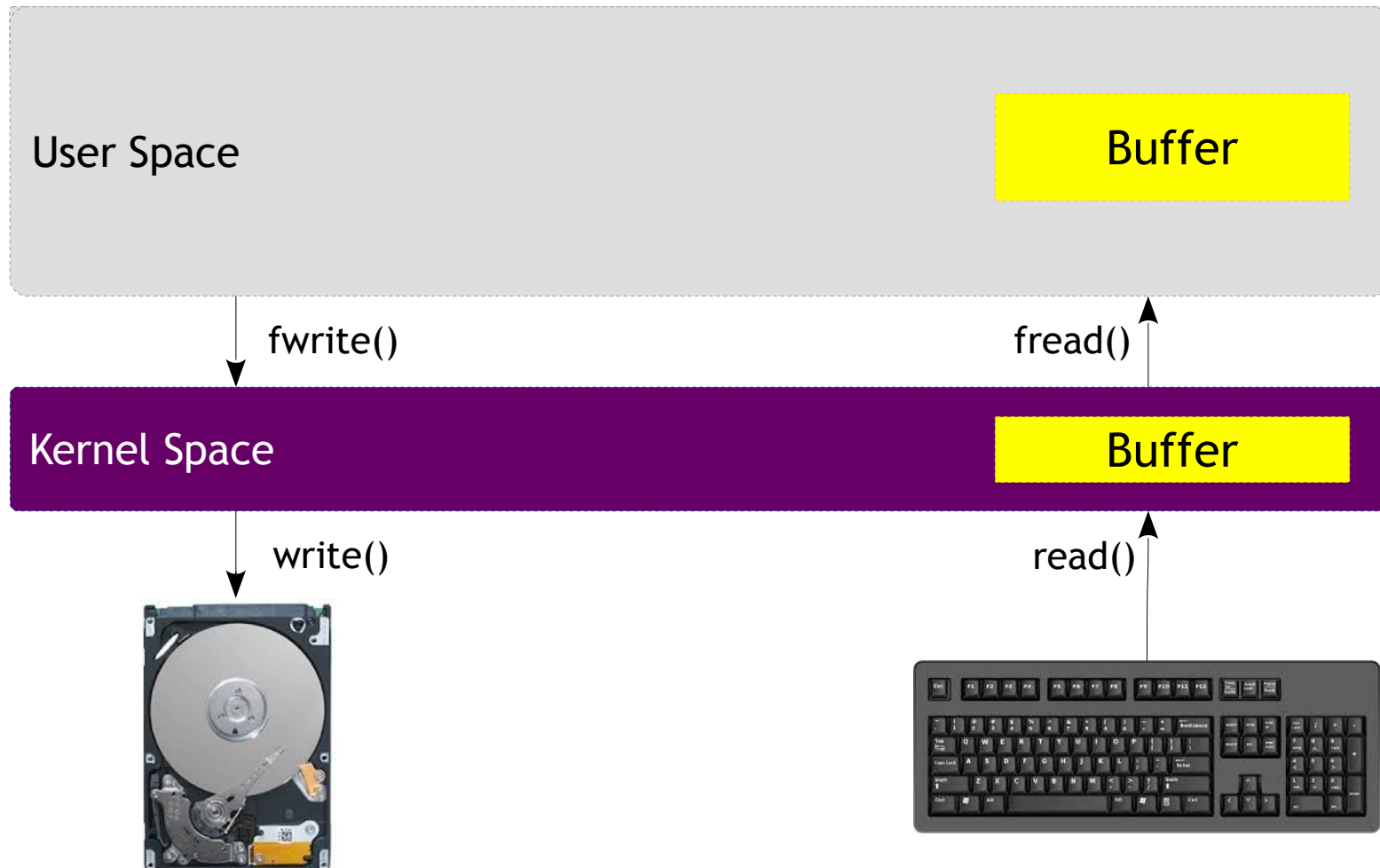
## Standard I/O - Buffering





# Advanced C

## Standard I/O - Buffering



# Advanced C

## Standard I/O - User Space Buffering



- Refers to the technique of temporarily storing the results of an I/O operation in user-space before transmitting it to the kernel (in the case of writes) or before providing it to your process (in the case of reads)
- This technique minimizes the number of system calls (between user and kernel space) which may improve the performance of your application

# Advanced C

## Standard I/O - User Space Buffering



- For example, consider a process that writes one character at a time to a file. This is obviously inefficient: Each write operation corresponds to a `write()` system call
- Similarly, imagine a process that reads one character at a time from a file into memory!! This leads to `read()` system call
- I/O buffers are temporary memory area(s) to moderate the number of transfers in/out of memory by assembling data into batches

# Advanced C

## Standard I/O - Buffering - stdout



- The output buffer get flushed out due to the following reasons
  - Normal Program Termination
  - '\n' in a printf
  - Read
  - fflush call
  - Buffer Full

# Advanced C

## Standard I/O - Buffering - stdout

020\_example.c

```
#include <stdio.h>

int main()
{
    printf("Hello");

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdout



### 021\_example.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    while (1)
    {
        printf("Hello");
        sleep(1);
    }

    return 0;
}
```

### Solution

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    while (1)
    {
        printf("Hello\n");
        sleep(1);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdout

022\_example.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int num;

    while (1)
    {
        printf("Enter a number: ");
        scanf("%d", &num);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdout

023\_example.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    while (1)
    {
        printf("Hello");
        fflush(stdout);
        sleep(1);
    }

    return 0;
}
```



# Advanced C

## Standard I/O - Buffering - stdout



### 024\_example.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char str[BUFSIZ] = "1";

    while (1)
    {
        printf("%s", str);
        sleep(1);
    }

    return 0;
}
```

### Solution

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char str[BUFSIZ] = "1";

    setbuf(stdout, NULL);

    while (1)
    {
        printf("%s", str);
        sleep(1);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdin



- The input buffer generally gets filled till the user presses and enter key or end of file.
- The complete buffer would be read till a '\n' or EOF is received.

# Advanced C

## Standard I/O - Buffering - stdin

### 025\_example.c

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != 'n')
    {
        scanf("%c", &ch);

        printf("%c", ch);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stdin



### Solution 1

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != 'n')
    {
        scanf("%c", &ch);
        __fpurge(stdin);
        printf("%c", ch);
    }

    return 0;
}
```

### Solution 2

```
#include <stdio.h>

int main()
{
    char ch = 'y';

    printf("Enter a string: ");

    while (ch != 'n')
    {
        scanf("%c", &ch);
        while (getchar() != '\n');
        printf("%c", ch);
    }

    return 0;
}
```

# Advanced C

## Standard I/O - Buffering - stderr

- The stderr file stream is unbuffered.

### 026\_example.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    while (1)
    {
        fprintf(stdout, "Hello");
        fprintf(stderr, "World");

        sleep(1);
    }

    return 0;
}
```