C STANDARD I/O

-used for reading, writing from/to standard output & formatting-

STANDARD I/O

In this session we will learn:

1. Escape sequence
2. Character and string
3. Standard input/output (examples):
   a) Standard input – `printf()`
   b) Standard output – `scanf()`

STANDARD I/O

- standard library functions for file input and output.
- C abstracts all file operations into operations on streams of bytes, which may be "input streams" or "output streams".
- C has no direct support for random-access data files.
- So, to read from a record in the middle of a file, the programmer must create a stream, seek to the middle of the file, and then read bytes in sequence from the stream.

STANDARD I/O

- In this topic, indirectly, we also will learn the C standard built-in function.
- And also the variadic functions (functions which having variable number of parameters) such as `printf()` and `scanf()`.
- Let start with escape sequence…

STANDARD I/O

Escape Sequence

- Character combinations consisting of a backslash (\) followed by a letter or by a combination of digits.
- Regarded as a single character and is therefore valid as a character constant.
- Must be used to represent a newline character, single quotation mark, or certain other characters in a character constant.
- Used to specify actions such as carriage returns and tab movements on terminals and printers.
- Used to provide literal representations of nonprinting characters and characters that usually have special meanings, such as the double quotation mark (").

STANDARD I/O

- Allow you to send non-graphic control characters to a display device. e.g, the `ESC` character (`\033`) is often used as the first character of a control command for a terminal or printer.
- Some escape sequences are device-specific. e.g, the vertical-tab and formfeed escape sequences (`\v` and `\f`) do not affect screen output, but they do perform appropriate printer operations.
- You can also use the backslash (`\`) as a continuation character.
- When a newline character (equivalent to pressing the `RETURN` key) immediately follows the backslash, the compiler ignores the backslash and the newline character and treats the next line as part of the previous line.
- This is useful primarily for preprocessor definitions longer than a single line. e.g.

```
#define assert(xpr)   \
 ((xpr) ? (void)0 : _assert(#xpr, __FILE__, __LINE__))
```

STANDARD I/O

List of the ANSI escape sequences and what they represent

Escape sequence	Description	Representation
`\'`	single quote	byte `0x27`
`\"`	double quote	byte `0x22`
`\?`	question mark	byte `0x3f`
`\\`	backslash	byte `0x5c`
`\0`	null character	byte `0x00`
`\a`	audible bell	byte `0x07`
`\b`	backspace	byte `0x08`
`\f`	form feed - new page	byte `0x0c`
`\n`	line feed - new line	byte `0x0a`
`\r`	carriage return	byte `0x0d`
`\t`	horizontal tab	byte `0x09`
`\v`	vertical tab	byte `0x0b`
`\nnn`	arbitrary ASCII character in octal value. At least one digit and maximum 3 digits. e.g. `\10` or `\010` for backspace	byte *nnn*
`\xnn`	arbitrary ASCII character in hexadecimal value. Number of hexadecimal digits is unlimited. e.g. `\x31`, `\x5A`	byte *nn*
`\xnnnn`	Unicode character in hexadecimal notation if this escape sequence is used in a wide-character constant or a Unicode string literal. e.g. `L'\x4e00'`	-
`\unnnn`	arbitrary Unicode value. May result in several characters.	code point *U+nnnn*
`\Unnnnnnnn`	arbitrary Unicode value. May result in several characters.	code point *U+nnnnnnnn*

STANDARD I/O

Character and string

- A character constant is formed by enclosing a single character from the representable character set within single quotation marks (''').

  e.g: `'3','\b','T',L'p',L'\x4e00'`

- A string is an array of characters. String literals are words surrounded by double quotation marks ("").

  e.g:

```
"This is a string, lateral string"
L"This is a wide string"
"123abc"
"a4"
L"1234*abc@"
```

- In C, to store strings we could use array or pointers type constructs.

STANDARD I/O

C number representation

- Computers store information in binary (base-2).
- Anything you write in a program and gets executed eventually gets converted to binary.
- However for human reading we have base-10 (decimal), base-8 (octal) and base-16 (hexadecimal).
- Summary:

Base	Number Representation	Calculation	Example	
Binary	0,1	$1x2^4+0x2^3+1x2^2+1x2^1+1x2^0$	0010111	0010111
Decimal	0,1,2,3,4,5,6,7,8,9	$2x10^1+3x10^0$	23	-
Octal	0,1,2,3,4,5,6,7	$2x8^1+7x8^0$	27	000 **010 111**
Hexadecimal	0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f **or** 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	$1x16^1+7x16^0$	17	**0001 0111**

STANDARD I/O

C number representation

- Integer constants are constant data elements that have no fractional parts or exponents.
- Always begin with a digit.
- Can specify integer constants in decimal, octal, or hexadecimal form.
- Can specify signed or unsigned types and long or short types.
- To specify a decimal constant, begin the specification with a nonzero digit.
- e.g.

```
int iNum = 187; // decimal constant
```

STANDARD I/O

- To specify an octal constant, begin the specification with `0`, followed by a sequence of digits in the range `0` through 7. e.g.

```
int iNum = 0774; // octal constant
int jNum = 0597; // error: 9 is not an octal digit
```

- To specify a hexadecimal constant, begin the specification with `0x` or `0X` (the case of the "`x`" does not matter), followed by a sequence of digits in the range `0` through `9` and `a` (or `A`) through `f` (or `F`).
- Hexadecimal digits `a` (or `A`) through `f` (or `F`) represent values in the range 10 through 15. e.g.

```
int iNum = 0x4f2a; // hexadecimal constant
int jNum = 0X4F2A; // equal to iNum
```

STANDARD I/O

- To specify an unsigned type, use either the `u` or `U` suffix.
- To specify a long type, use either the `l` or `L` suffix. For example:

```
unsigned unVal = 238u; // unsigned value
long lgVal = 0x6FFFFFL; // long value specified
                        // as hex constant
unsigned long unlgVal = 0677342ul; // unsigned long
                                   // value
```

- To specify a 64-bit integral type, use the `LL`, `ll` or `i64` suffix.

STANDARD I/O

	Byte character	Wide character	Description

Formatted input/output	`scanf()`
	`fscanf()`
	`sscanf()`
		`wscanf()`
		`fwscanf()`
		`swscanf()`
			reads formatted byte/`wchar_t` input from stdin (standard input), a file stream or a buffer

	`vscanf()`
	`vfscanf()`
	`vsscanf()`
		`vwscanf()`
		`vfwscanf()`
		`vswscanf()`
			reads formatted input byte/`wchar_t` from stdin, a file stream or a buffer using variable argument list

	`printf()`
	`fprintf()`
	`sprintf()`
	`snprintf()`
		`wprintf()`
		`fwprintf()`
		`swprintf()`
			prints formatted byte/`wchar_t` output to stdout (standard output), a file stream or a buffer

	`vprintf()`
	`vfprintf()`
	`vsprintf()`
	`vsnprintf()`
		`vwprintf()`
		`vfwprintf()`
		`vswprintf()`
			prints formatted byte/`wchar_t` output to stdout, a file stream, or a buffer using variable argument list

Unformatted input/output	`fgetc()` `getc()`	`fgetwc()` `getwc()`	reads a byte/`wchar_t` from a file stream
	`fgets()`	`fgetws()`	reads a byte/`wchar_t` line from a file stream
	`fputc()` `putc()`	`fputwc()` `putwc()`	writes a byte/`wchar_t` to a file stream
	`fputs()`	`fputws()`	writes a byte/`wchar_t` string to a file stream
	`getchar()`	`getwchar()`	reads a byte/`wchar_t` from stdin
	`gets()`	N/A	reads a byte string from stdin (deprecated in C99, obsolete in C11)
	`putchar()`	`putwchar()`	writes a byte/`wchar_t` to stdout
	`puts()`	N/A	writes a byte string to stdout
	`ungetc()`	`ungetwc()`	puts a byte/`wchar_t` back into a file stream

STANDARD I/O

File access	`fopen()`	opens a file
	`freopen()`	opens a different file with an existing stream
	`fflush()`	synchronizes an output stream with the actual file
	`fclose()`	closes a file
	`setbuf()`	sets the buffer for a file stream
	`setvbuf()`	sets the buffer and its size for a file stream
	`fwide()`	switches a file stream between wide character I/O and narrow character I/O
Direct input/output	`fread()`	reads from a file
	`fwrite()`	writes to a file
File positioning	`ftell()`	returns the current file position indicator
	`fgetpos`	gets the file position indicator
	`fseek()`	moves the file position indicator to a specific location in a file
	`fsetpos()`	moves the file position indicator to a specific location in a file
	`rewind()`	moves the file position indicator to the beginning in a file
Error handling	`clearerr()`	clears errors
	`feof()`	checks for the end-of-file
	`ferror()`	checks for a file error
	`perror()`	displays a character string corresponding of the current error to stderr
Operations on files	`remove()`	erases a file
	`rename()`	renames a file
	`tmpfile()`	returns a pointer to a temporary file
	`tmpnam()`	returns a unique filename

STANDARD I/O

The `printf()` family

- Loads the data from the given locations, converts them to character string equivalents and writes the results:
  a) to stdout (standard output) such as `printf()` and `wprintf()`.
  b) to a file stream `stream` such as `fprintf()` and `fwprintf()`.
  c) to a character string buffer such as `sprintf()` and `swprintf()`.
  d) At most (`buf_size` – 1) characters are written such as `snprintf()`. The resulting character string will be terminated with a null character, unless `buf_size` is zero.

STANDARD I/O

- Implementation dependant example (Microsoft Visual C++):

`_printf_l()`	Same as `printf()` but use the locale parameter passed in instead of the current thread locale.
`_wprintf_l()`	Same as `wprintf()` but use the locale parameter passed in instead of the current thread locale.
`printf_s()`, `_printf_s_l()`, `wprintf_s()`, `_wprintf_s_l()`	Same as `printf()`, `_printf_l()`, `wprintf()` and `_wprintf_l()` but with security enhancements

STANDARD I/O

Sample function	
Name	`printf()`
Syntax	`int printf(const char *format [,argument]...);`
Parameters	`format` - Format control. `argument` - Optional arguments. `...` (additional arguments) - Depending on the format string, the function may expect a sequence of additional arguments, each containing a value to be used to replace a format specifier in the format string (or a pointer to a storage location, for n). There should be at least as many of these arguments as the number of values specified in the format specifiers. Additional arguments are ignored by the function.
Usage	Formats and prints a series of characters and values to the standard output stream, stdout. If arguments follow the format string, the format string must contain specifications that determine the output format for the arguments
Return value	Returns the number of characters printed (excluding the null byte used to end output to strings), or a negative value if an error occurs. If format is NULL, the invalid parameter handler is invoked
Example	`printf("%ld, %#X, %-7d, %7s, %8.5f \n", 3, 120, 11, "STRING", 1.234);`
Remarks	Also called a variadic function.

STANDARD I/O

The syntax for format specifications fields, used in `printf()`, `wprintf()` and related functions.

- A format specification, which consists of optional and required fields, has the following form:

% [flags] [width] [.precision] [{h | l | ll | I | I32 | I64}]type

- More readable form:

% [*flags*] [*field_width*] [*.precision*] [*length_modifier*] *conversion_character*

- Where components in brackets **[]** are optional.
- Examples:

```
printf("%#x\n", 141);
printf("%g\n", 5.1234567);
printf("%07d\n", 123);
printf("%+d\n", 456);
printf("%-7d,%-5d,\n", 33, 44);
printf("%7s\n", "123456");
printf("%4f\n", 41.1234);
printf("%8.5f\n", 3.234);
printf("%.3f\n", 15.4321);
printf("%hd\n", 7);
printf("%ld\n", 9);
printf("%ls\n", "my name");
printf("%Lg\n", 45.23456123);
```

STANDARD I/O

- Each field of the format specification is a single character or a number signifying a particular format option.
- The simplest format specification contains only the percent sign and a type character, for example:

```
printf("A string: %s", "This is a string")
```

- If a percent sign is followed by a character that has no meaning as a format field, the character is copied to stdout. For example, to print a percent-sign character, use `%%`.
- The optional fields, which appear before the *`type`* character, control other aspects of the formatting, as follows:

STANDARD I/O

flags	Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes (refer to the "Flag Characters" table). More than one flag can appear in a format specification.
Width	Optional number that specifies the minimum number of characters output (refer to the `printf()` Width Specification.
precision	Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values (refer to the Precision Specification table).
h | l | ll | I | I32 | I64	Optional prefixes to type that specify the size of argument (refer to Size Specification table).
type	Required character that determines whether the associated argument is interpreted as a character, a string, or a number (refer to the `printf()` Type Field Characters table).

Flag Directives

STANDARD I/O

Flag Characters		
It is the first optional field of the format specification after % sign. Is a character that justifies output and prints signs, blanks, decimal points, octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.		
Flag	**Meaning**	**Default**
–	Left align the result within the given field width.	Right align.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
0	If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d) and a precision specification is also present (for example, %04.d), the 0 is ignored.	No padding.
' '	A blank. Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with the e, E, f, a or A format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Ignored when used with c, d, i, u, or s.	Decimal point appears only if digits follow it. Trailing zeros are truncated.

STANDARD I/O

Code example:

```
printf("%%#x: %#x\n", 120 );
printf("%%x: %x\n", 12 );
printf("%%#X: %#X\n", 120 );
printf("%%X: %X\n", 12 );
printf("%%#o: %#o\n", 120 );
printf("%%o: %o\n", 12 );
printf("%%#2f: %#2f\n", 120.567 );
printf( "%%g: %g\n", 3.1415926 );
printf( "%%g: %g\n", 93000000.0 );
printf( "%%G: %G\n", 93000000.0 );
printf("%%07d: %07d\n", 102 );
printf("%+d\n", 102 );
printf("%%-7d,%%-5d,: %-7d,%-5d,\n", 11, 22 );
printf("%%#010x: %#010x\n", 121 );
printf("%%#010X: %#010X\n", 121 );
```

STANDARD I/O

Output example:

```
%#x: 0x78
%x: c
%#X: 0X78
%X: C
%#o: 0170
%o: 14
%#2f: 120.567000
%g: 3.14159
%g: 9.3e+007
%G: 9.3E+007
%07d: 0000102
+102
%-7d,%-5d,: 11     ,22   ,
%#010x: 0x00000079
%#010X: 0X00000079
```

STANDARD I/O

- The second optional field.
- The width argument is a **non-negative decimal integer** controlling the **minimum number of characters printed**.
- If the number of characters in the output value is less than the specified width, **blanks** are added to the left or the right of the values, depending on whether the `-` (minus) flag (for left alignment) is specified, until the minimum width is reached.
- If width is prefixed with `0`, zeros are added until the minimum width is reached (not useful for left-aligned numbers).
- The width specification never causes a value to be truncated.

STANDARD I/O

Width Specification

- If the number of characters in the output value is greater than the specified width, or if width is not given, all characters of the value are printed (but subject to the precision specification).
- If the width specification is an asterisk (*), an `int` argument from the argument list supplies the value.
- The width argument must precede the value being formatted in the argument list.
- A non-existent or small field width does not cause the truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

Example:

```
printf("%%7s: %7s, %%7s: %7s,\n", "abc", "123456" );

%7s:     abc, %7s:  123456,
```

STANDARD I/O

Precision Specification

- The third optional field.
- It specifies a **nonnegative decimal integer**, preceded by a period (**.**), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits as summarized in the following table.
- Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value.
- If precision is specified as `0` and the value to be converted is `0`, the result is no characters output, as shown below:

```
printf("%.0d", 0);  /* no characters output */
```

STANDARD I/O

Precision Specification

- If the precision specification is an asterisk (`*`), an `int` argument from the argument list supplies the value.
- The precision argument must precede the value being formatted in the argument list.
- The type determines the interpretation of precision and the default when precision is omitted, as shown in the following table.

STANDARD I/O

How Precision Values Affect Type

Type	Meaning	Default
a, A	The precision specifies the number of digits after the point.	Default precision is 6. If precision is 0, no point is printed unless the # flag is used.
c, C	The precision has no effect.	Character is printed.
d, i, u, o, x, X	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision.	Default precision is 1.
e, E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, with any trailing zeros truncated.
s, S	The precision specifies the maximum number of characters to be printed. Characters in excess of precision are not printed.	Characters are printed until a null character is encountered.

STANDARD I/O

How Precision Values Affect Type, continue…

If the argument corresponding to a floating-point specifier is infinite, indefinite, or `NAN 9` (Not-A-Number), `printf()` gives the following output.

Value	Output
+ infinity	1.#INFrandom-digits
– infinity	–1.#INFrandom-digits
Indefinite (same as quiet NaN)	digit.#INDrandom-digits
NAN	digit.#NANrandom-digits

STANDARD I/O

Code Example:

```
    printf("%%4f: %4f\n", 12.4321 );
    printf( "%%8.5f: %8.5f\n", 1.234 );
    printf("%%.1f: %4.1f\n", 12.4321 );
    printf("%%.3f: %.3f\n", 12.4321 );
    printf( "%%.3f: %.3f\n%%.3g:
%.3g\n%%.3f: %.3f\n%%.3g: %.3g\n", 100.2,
100.2, 3.1415926, 3.1415926 );
    printf( "%%.5s: %.5s\n", "abcdefg" );
```

STANDARD I/O

Output example:

```
%4f: 12.432100
%8.5f:  1.23400
%.1f: 12.4
%.3f: 12.432
%.3f: 100.200
%.3g: 100
%.3f: 3.142
%.3g: 3.14
%.5s: abcde
```

Size Specification

STANDARD I/O

- Also known as length modifiers.
- The optional prefixes to type, `h`, `l`, `I`, `I32`, `I64`, and `ll` specify the "size" of argument (`long` or `short`, 32- or 64-bit, single-byte character or wide character, depending upon the type specifier that they modify).
- These type-specifier prefixes are used with type characters in `printf()` functions or `wprintf()` functions to specify interpretation of arguments, as shown in the following table.
- These prefixes are Microsoft (and some other implementations) extensions example and are not ANSI-compatible.

STANDARD I/O

Size Specification

Size prefixes for `printf()` and `wprintf()` format-type specifiers		
To specify	**Use prefix**	**With type specifier**
`long int`	`l` (lowercase L)	`d`, `i`, `o`, `x`, or `X`
`long unsigned int`	`l`	`o`, `u`, `x`, or `X`
`long long`	`ll`	`d`, `i`, `o`, `x`, or `X`
`short int`	`h`	`d`, `i`, `o`, `x`, or `X`
`short unsigned int`	`h`	`o`, `u`, `x`, or `X`
`__int32`	`I32`	`d`, `i`, `o`, `x`, or `X`
`unsigned __int32`	`I32`	`o`, `u`, `x`, or `X`
`__int64`	`I64`	`d`, `i`, `o`, `x`, or `X`
`unsigned __int64`	`I64`	`o`, `u`, `x`, or `X`
`ptrdiff_t` (that is, `__int32` on 32-bit platforms, `__int64` on 64-bit platforms)	`I`	`d`, `i`, `o`, `x`, or `X`

STANDARD I/O

Size Specification

To specify	Use prefix	With type specifier
`size_t` (that is, `unsigned __int32` on 32-bit platforms, `unsigned __int64` on 64-bit platforms)	`I`	`o`, `u`, `x`, or `X`
long double	`l` or `L`	`f`
Single-byte character with `printf()` functions	`h`	`c` or `C`
Single-byte character with `wprintf()` functions	`h`	`c` or `C`
Wide character with `printf()` functions	`l`	`c` or `C`
Wide character with `wprintf()` functions	`l`	`c` or `C`
Single-byte – character string with `printf()` functions	`h`	`s` or `S`
Single-byte – character string with `wprintf()` functions	`h`	`s` or `S`
Wide-character string with `printf()` functions	`l`	`s` or `S`
Wide-character string with `wprintf()` functions	`l`	`s` or `S`
Wide character	`w`	`c`
Wide-character string	`w`	`s`

STANDARD I/O

- Thus to print **single-byte** or **wide-characters** with `printf()` functions and `wprintf()` functions, use format specifiers as follows.

To print character as	Use function	With format specifier
single byte	`printf()`	`c`, `hc`, or `hC`
single byte	`wprintf()`	`C`, `hc`, or `hC`
wide	`wprintf()`	`c`, `lc`, `lC`, or `wc`
wide	`printf()`	`C`, `lc`, `lC`, or `wc`

- To print strings with `printf()` functions and `wprintf()` functions, use the prefixes `h` and `l` analogously with format type-specifiers `s` and `S`.

STANDARD I/O

Code example:

```
short int i = 3;
long int j = 3;
wchar_t* wide_str = L"This is a wide string";
long double d = 3.1415926535;

printf( "%%hd: %hd\n", i );
printf( "%%ld: %ld\n", j );
printf( "%%ls: %ls\n", wide_str );
printf( "%%Lg: %Lg\n", d );
```

STANDARD I/O

Output example:

```
%hd: 3
%ld: 3
%ls: This is a wide string
%Lg: 3.14159
```

STANDARD I/O

Type character

- The type character of the format specification indicates that the corresponding argument is to be interpreted as a **character**, **string**, or **number**.
- The type character is the only required format field, and it appears after any optional format fields.

Type character

STANDARD I/O

printf() Type Field Characters		
Character	**Type**	**Output format**
c	int	When used with printf() functions, specifies a single-byte character; when used with wprintf() functions, specifies a wide character.
d, i	int	Signed decimal integer.
o	int	Unsigned octal integer.
u	int	Unsigned decimal integer.
x	int	Unsigned hexadecimal integer, using "abcdef."
X	int	Unsigned hexadecimal integer, using "ABCDEF."
e	double	Signed value having the form [–]d.dddd e [sign]dd[d] where d is a single decimal digit, dddd is one or more decimal digits, dd[d] is two or three decimal digits depending on the output format and size of the exponent, and sign is + or –.
E	double	Identical to the e format except that E rather than e introduces the exponent.
f	double	Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.

STANDARD I/O

Type character

Character	Type	Output format
g	double	Signed values are displayed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).
p	Pointer to void	Displays the argument as an address in hexadecimal digits (as if by %#x or %#lx)
n	Pointer to integer	Number of characters successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.
a	double	Signed hexadecimal double precision floating point value having the form [-]0xh.hhhh p±dd, where h.hhhh are the hex digits (using lower case letters) of the mantissa, and dd are one or more digits for the exponent. The precision specifies the number of digits after the point.
A	double	Signed hexadecimal double precision floating point value having the form [-]0Xh.hhhh P±dd, where h.hhhh are the hex digits (using capital letters) of the mantissa, and dd are one or more digits for the exponent. The precision specifies the number of digits after the point.
s	String	When used with printf() functions, specifies a single-byte-character string; when used with wprintf() functions, specifies a wide-character string. Characters are displayed up to the first null character or until the precision value is reached.

STANDARD I/O

Source code for C program examples in txt file format:

1. Example 0
2. Example 1
3. Example 2

`scanf()` family STANDARD I/O

- Read formatted data from a variety of standard input stream which are:
  1. data from `stdin` (standard input) such as `scanf()` and `wscanf()`
  2. data from file stream stream `fscanf()` and `fwscanf()`
  3. data from null-terminated character string buffer `sscanf()` and `swscanf()`
- and interprets it according to the format then stores the results in its arguments.

STANDARD I/O

- Implementation dependant (Microsoft visual C++) example is listed in the following table.

Function	Description
`_scanf_l()`	Same as `scanf()` except they use the locale parameter passed in instead of the current thread locale.
`_wscanf_l()`	Same as `wscanf()` except they use the locale parameter passed in instead of the current thread locale.
`_scanf_s_l()`	Same as `_scanf_l()` with security enhancements.
`_wscanf_s_l()`	Same as `_wscanf_l()` with security enhancements

STANDARD I/O

Function example	
Name	`scanf()`
Syntax	`int scanf(const char *format [,argument]...);`
Parameters	`format` - pointer to a null-terminated character string specifying how to read the input or a format control string. `argument` - optional arguments. `...` - receiving arguments
Usage	Read formatted data from the standard input stream
Return value	Returns the number of fields successfully converted and assigned; the return value does not include fields that were read but not assigned. A return value of 0 indicates that no fields were assigned. If format is a `NULL` pointer, the invalid parameter handler is executed. If execution is allowed to continue, these functions return `EOF`.
Example	`scanf("%d%5f%f%s", &i, &j, &x, name);`
Remarks	Also called variadic function.

STANDARD I/O

Format Specification Fields for `scanf()` and `wscanf()` family

- Describes the symbols used to tell the `scanf()` functions how to parse the input stream, such as from `stdin`, into values that are inserted into (program) variables.
- Has the following form, components in brackets **[ ]** are optional.

```
% [*] [width] [{h | l | ll | I64 | L}]type
```

- More readable form:

```
% [*] [field_width] [length_modifier] conversion_character
```

- Where components in brackets **[ ]** are optional.
- Example:

```
scanf("%f", &x);
scanf("%4f", &x);
scanf("%ls", &x);
scanf("%*d %[0123456789]", name);
```

STANDARD I/O

1. The `format` argument specifies the interpretation of the input and can contain one or more of the following:
   a. White-space characters: blank (' '); tab ('\t'); or newline ('`\n`'). A white-space character causes `scanf()` to read, but not store, all consecutive white-space characters in the input up to the **next non-white-space character**. One white-space character in the format matches any number (including 0) and combination of white-space characters in the input.
   b. Non-white-space characters: except for the percent sign (`%`). A non-white-space character causes `scanf()` to read, but not store, a matching non-white-space character. If the next character in the input stream does not match, `scanf()` terminates.
   c. Format specifications: introduced by the percent sign (`%`). A format specification causes `scanf()` to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

STANDARD I/O

- The format is read from left to right.
- Characters outside format specifications are expected to match the sequence of characters in the input stream; the matching characters in the input stream are scanned but not stored.
- If a character in the input stream conflicts with the format specification, `scanf()` terminates, and the character is left in the input stream as if it had not been read.
- When the first format specification is encountered, the value of the first input field is converted according to this specification and stored in the location that is specified by the first argument.
- The second format specification causes the second input field to be converted and stored in the second argument, and so on through the end of the format string.

STANDARD I/O

- An **input field** is defined as all characters (a string of non-white-space character) up to the first white-space character (space, tab, or newline), or up to the first character that cannot be converted according to the format specification, or until the field width (if specified) is reached.
- If there are too many arguments for the given specifications, the extra arguments are evaluated but ignored.
- The results are unpredictable if there are not enough arguments for the format specification.
- Each field of the format specification is **a single character** or **a number** signifying a particular format option.
- The type character, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number.
- The simplest format specification contains only the percent sign and a type character. e.g. `scanf(%s)`

STANDARD I/O

- If a percent sign (%) is followed by a character that has no meaning as a format-control character, that character and the following characters (up to the next percent sign) are treated as an ordinary sequence of characters, that is, a sequence of characters that must match the input.
- For example, to specify that a percent-sign character is to be input, use %%.
- An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

STANDARD I/O

`scanf()` Width Specification

- format strings in the `scanf()` family of functions.
- These functions normally assume the input stream is divided into a sequence of tokens.
- Tokens are separated by whitespace (space, tab, or newline), or in the case of numerical types, by the natural end of a numerical data type as indicated by the first character that cannot be converted into numerical text.
- However, the width specification may be used to cause parsing of the input to stop before the natural end of a token.
- The *width* specification consists of characters between the `%` and the type field specifier, which may include a positive integer called the *width* field and one or more characters indicating the size of the field, which may also be considered as modifiers of the type of the field, such as an indication of whether the integer type is `short` or `long`.
- Such characters are referred to as the size prefix. e.g.

```
scanf("%4f", &x);
```

STANDARD I/O

The Width Field

- The *width* field is a positive decimal integer controlling the maximum number of characters to be read for that field.
- No more than *width* characters are converted and stored at the corresponding argument.
- Fewer than *width* characters may be read if a whitespace character (space, tab, or newline) or a character that cannot be converted according to the given format occurs before *width* is reached.

STANDARD I/O

The Size Prefix

1. The optional prefixes `h`, `l`, `ll`, `I64`, and `L` indicate the size of the argument (`long` or `short`, single-byte character or wide character, depending upon the type character that they modify).
2. Used with type characters in `scanf()` or `wscanf()` functions to specify interpretation of arguments as shown in the following table.
3. The type prefix `I64` is a Microsoft extension example and is not ANSI compatible.
4. The `h`, `l`, and `L` prefixes are Microsoft extensions when used with data of type `char`.
5. The type characters and their meanings are described in the "Type Characters for `scanf()` functions" table.

STANDARD I/O

Size Prefixes for `scanf()` and `wscanf()` Format-Type Specifiers		
To specify	**Use prefix**	**With type specifier**
`double`	**`l`**	**`e`**, **`E`**, **`f`**, **`g`**, or **`G`**
`long double` (same as double)	**`L`**	**`e`**, **`E`**, **`f`**, **`g`**, or **`G`**
`long int`	**`l`**	**`d`**, **`i`**, **`o`**, **`x`**, or **`X`**
`long unsigned int`	**`l`**	**`u`**
`long long`	**`ll`**	**`d`**, **`i`**, **`o`**, **`x`**, or **`X`**
`short int`	**`h`**	**`d`**, **`i`**, **`o`**, **`x`**, or **`X`**
`short unsigned int`	**`h`**	**`u`**
`__int64`	**`I64`**	**`d`**, **`i`**, **`o`**, **`u`**, **`x`**, or **`X`**
Single-byte character with `scanf()`	**`h`**	**`c`** or **`C`**
Single-byte character with `wscanf()`	**`h`**	**`c`** or **`C`**
Wide character with `scanf()`	**`l`**	**`c`** or **`C`**
Wide character with `wscanf()`	**`l`**	**`c`**, or **`C`**
Single-byte character string with `scanf()`	**`h`**	**`s`** or **`S`**
Single-byte character string with `wscanf()`	**`h`**	**`s`** or **`S`**
Wide-character string with `scanf()`	**`l`**	**`s`** or **`S`**
Wide-character string with `wscanf()`	**`l`**	**`s`** or **`S`**

STANDARD I/O

- The following examples use `h` and `l` with `scanf()` functions and `wscanf()` functions:

```
scanf( "%ls", &x );      // read a wide-character string
wscanf( "%hC", &x );     // read a single-byte character
```

STANDARD I/O

Reading Undelimited strings

- To read strings not delimited by whitespace characters, a set of characters in brackets (`[]`) can be substituted for the **s** (string) type character.
- The set of characters in brackets is referred to as a control string.
- The corresponding input field is read up to the first character that does not appear in the control string.
- If the first character in the set is a caret (`^`), the effect is reversed: The input field is read up to the first character that does appear in the rest of the character set.
- `%[a-z]` and `%[z-a]` are interpreted as equivalent to `%[abcde...z]`.
- This is a common `scanf()` function extension, but note that the ANSI standard does not require it.

STANDARD I/O

- Some implementation supports a nonstandard extension that causes the library to dynamically allocate a string of sufficient size for input strings for the `%s` and `%a[range]` conversion specifiers.
- To make use of this feature, specify a as a length modifier (thus `%as` or `%a[range]`). The caller must free the returned string, as in the following example:

```
char *p;
int q;
errno = 0;

q = scanf("%a[a-z]", &p);
if (q == 1)
{
        printf("read: %s\n", p);
        free(p);
}
else if (errno != 0)
{
        perror("scanf");
}
else
{
        printf("No matching characters\n");
}
```

- (the `a` is interpreted as a specifier for floating-point numbers).

STANDARD I/O

Reading Unterminated strings

- To store a string without storing a terminating null character (`'\0'`), use the specification `%`*n*`c` where *n* is a decimal integer.
- The `c` type character indicates that the argument is a pointer to a character array.
- The next *n* characters are read from the input stream into the specified location, and no null character (`'\0'`) is appended.
- If *n* is not specified, its default value is `1`.

STANDARD I/O

When `scanf()` stops reading a field?

- The `scanf()` function scans each input field, character by character. It may stop reading a particular input field before it reaches a space character for a variety of reasons:
  1) The specified width has been reached.
  2) The next character cannot be converted as specified.
  3) The next character conflicts with a character in the control string that it is supposed to match.
  4) The next character fails to appear in a given character set.
- For whatever reason, when the `scanf()` function stops reading an input field, the next input field is considered to begin at the first unread character.
- The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

STANDARD I/O

Type Field Characters for `scanf()` family

- The following information applies to any of the `scanf()` family of functions.
- The type character is the only required format field.
- Appears after any optional format fields.
- The type character determines whether the associated argument is interpreted as a character, string, or number.

STANDARD I/O

Character	Type of input expected	Type of argument	Size argument in secure version?
c	Character. When used with `scanf()` functions, specifies single-byte character; when used with `wscanf()` functions, specifies wide character. White-space characters that are ordinarily skipped are read when `c` is specified. To read next non-white-space single-byte character, use `%1s`; to read next non-white-space wide character, use `%1ws`.	Pointer to `char` when used with `scanf()` functions, pointer to `wchar_t` when used with `wscanf()` functions.	Required. Size does not include space for a null terminator.
C	Opposite size character. When used with `scanf()` functions, specifies wide character; when used with `wscanf()` functions, specifies single-byte character. White-space characters that are ordinarily skipped are read when `C` is specified. To read next non-white-space single-byte character, use `%1s`; to read next non-white-space wide character, use `%1ws`.	Pointer to `wchar_t` when used with `scanf()` functions, pointer to char when used with `wscanf()` functions.	Required. Size argument does not include space for a null terminator.
d	Decimal integer.	Pointer to `int`.	No.

STANDARD I/O

Continue…

d	Decimal integer.	Pointer to int.	No.
i	An integer. Hexadecimal if the input string begins with "0x" or "0X", octal if the string begins with "0", otherwise decimal.	Pointer to int.	No.
o	Octal integer.	Pointer to int.	No.
u	Unsigned decimal integer.	Pointer to unsigned int.	No.
x	Hexadecimal integer.	Pointer to int.	No.
e , E, f, g, G	Floating-point value consisting of optional sign (+ or –), series of one or more decimal digits containing decimal point, and optional exponent ("e" or "E") followed by an optionally signed integer value.	Pointer to float.	No.
n	No input read from stream or buffer.	Pointer to int, into which is stored number of characters successfully read from stream or buffer up to that point in current call to scanf() functions or wscanf() functions.	No.

Continue…

STANDARD I/O

s	String, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets ([ ]), as discussed in `scanf()` Width Specification.	When used with `scanf()` functions, signifies single-byte character array; when used with `wscanf()` functions, signifies wide-character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.
S	Opposite-size character string, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets ([ ]), as discussed in `scanf()` Width Specification.	When used with `scanf()` functions, signifies wide-character array; when used with `wscanf()` functions, signifies single-byte-character array. In either case, character array must be large enough for input field plus terminating null character, which is automatically appended.	Required. Size includes space for a null terminator.

STANDARD I/O

- The `a` and `A` specifiers are not available with `scanf()`.
- The `size` arguments, if required, should be passed in the parameter list immediately following the argument they apply to. For example, the following code:

```
char string1[11], string2[9];
scanf("%10s %8s", string1, 11, string2, 9);
```

- reads a string with a maximum length of 10 into `string1`, and a string with a maximum length of 8 into `string2`.
- The buffer sizes should be at least one more than the width specifications since space must be reserved for the null terminator.

STANDARD I/O

- The format string can handle single-byte or wide character input regardless of whether the single-byte character or wide-character version of the function is used.
- Thus, to read single-byte or wide characters with `scanf()` and `wscanf()` functions, use format specifiers as follows:

To read character as	Use this function	With these format specifiers
single byte	`scanf()` functions	`c`, `hc`, or `hC`
single byte	`wscanf()` functions	`C`, `hc`, or `hC`
wide	`wscanf()` functions	`c`, `lc`, or `lC`
wide	`scanf()` functions	`C`, `lc`, or `lC`

- To scan strings with `scanf()` functions, and `wscanf()` functions, use the above table with format type-specifiers `s` and `S` instead of `c` and `C`.

STANDARD I/O

Program examples:

1. Example 1: Try the following inputs: `58 71.3 A t` byte characters
2. Example 2:
3. Example 3: See next slide (character and string issues with `scanf()`)

STANDARD I/O

Run the following program and precede the `%c`'s with spaces.

```
#include <stdio.h>

int main(void)
{
      char a, b;
      int i, j;
      printf("Enter two char-int pairs: ");
      scanf(" %c %d", &a, &i);
      scanf(" %c %d", &b, &j);
      printf("%c:%d:\n", a, i);
      printf("%c:%d:\n", b, j);
      return 0;
}
```

White space will terminate a string

```
Enter two char-int pairs: R8 T3
R:8:
T:3:
Press any key to continue . . .
```

STANDARD I/O

1. Did the values get read into the variables as they should have been? **YES**
2. Try the same experiment again without the leading spaces in the format strings for integers e.g. `scanf(" %c%d", &a, &i);`. Did you get the results as before? **YES**
3. Try the same experiment again without the leading spaces in the format strings for the characters (e.g. `scanf("%c %d", &a, &i);`. Did you get the same result as before? **NO**
4. When reading in integers, spaces are not needed, true or false? **TRUE**
5. When reading in characters, we would add the spaces before the `%c`'s, true or false? **TRUE**

Format strings for floats (`%f`) behave like integers and those for strings (`%s`) behave like characters.

END of C STANDARD I/O