

## C LAB WORKSHEET 6

### C scanf(), scanf\_s() family 1

Items in this page:

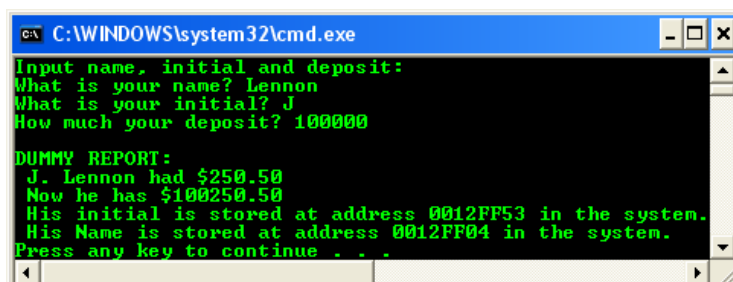
1. The scanf() and scanf\_s() function family – reading data from standard input.
2. Tutorial references are: [C/C++ intro & brief history](#), [C/C++ data type 1](#), [C/C++ data type 2](#), [C/C++ data type 3](#) and [C/C++ statement, expression & operator 1](#), [C/C++ statement, expression & operator 2](#) and [C/C++ statement, expression & operator 2](#). More scanf() and its family examples can be found in [C formatted input/output](#).

- In the previous practice you already learned how to use the printf() function to write/display/print/output character/string/text on your screen/terminal/standard output. In this lab you will learn another function, scanf() or a new secure version scanf\_s(), on how to accept input from your standard input such as keyboard or user.
- Create a new empty Win32 Console Application project named **myscanf** and add a C++ source file named **myscanfsrc** to the project. Make sure you set your project property to be compiled as C codes.
- Type the following source code.

```
// preprocessor directive
#include <stdio.h>
// another preprocessor directive, means replace all INIT_BALANCE
// occurrences in this program with 250.50
#define INIT_BALANCE 250.50

void main(void)
{
    float Deposit;
    // name is an array variable, more on this later...
    // means reserve 65 bytes of storage including terminated
    // string, '\0'
    char Initial, Name[65];
    // some prompt to user...
    printf("Input name, initial and deposit:\n");
    printf("What is your name? ");
    // read the user input and store the data at...
    scanf("%64s", Name);
    printf("What is your initial? ");
    // don't forget the space before %c!!! else the output
    // will be proceeded to the next line...bug or what?
    // read the user input and store the data at...
    // & means storage/memory address of Initial variable...
    scanf(" %c", &Initial);
    printf("How much your deposit? ");
    // read the user input and store the data at...
    scanf("%f", &Deposit);
    // C4996: scanf and wscanf are deprecated; consider
    // using a secure version, scanf_s and wscanf_s
    // now it is time to read the stored data...
    printf("\nDUMMY REPORT:\n");
    printf(" %c. %s had $%.2f\n", Initial, Name, INIT_BALANCE);
    printf(" Now he has $%.2f\n", (INIT_BALANCE + Deposit));
    printf(" His initial is stored at address %p in the system.\n", &Initial);
    // No need to put the & because an array variable without []
    // is already a pointer...
    printf(" His Name is stored at address %p in the system.\n", Name);
}
```

- Build, run the program and show the output.



```
C:\WINDOWS\system32\cmd.exe
Input name, initial and deposit:
What is your name? Lennon
What is your initial? J
How much your deposit? 100000

DUMMY REPORT:
J. Lennon had $250.50
Now he has $100250.50
His initial is stored at address 0012FF53 in the system.
His Name is stored at address 0012FF04 in the system.
Press any key to continue . . .
```

- In this example the program use `scanf()` function to accept input from user. The program reads data from user and stores it somewhere in the memory (denoted by the hex addresses), then writes it on the standard output. Opposite to the `printf()` write formatted data to the standard output, `scanf()` read formatted data from the **standard input**, the keyboard.

### The `scanf()`, `_scanf_l()`, `wscanf()` and `_wscanf_l()`

- The definition of the `scanf()` family is listed in the following Table.

Item	Description
Function	<code>scanf()</code> family.
Use	Read formatted data from the standard input stream. These functions are deprecated and it is better to use a more secure versions <code>scanf_s()</code> , <code>_scanf_s_l()</code> , <code>wscanf_s()</code> , <code>_wscanf_s_l()</code> .
Prototype	<ol style="list-style-type: none"> <li><code>int scanf(const char *format [, argument]...);</code></li> <li><code>int _scanf_l(const char *format, locale_t locale [, argument]...);</code></li> <li><code>int wscanf(const wchar_t *format [, argument]...);</code></li> <li><code>int _wscanf_l(const wchar_t *format, locale_t locale [, argument]...);</code></li> </ol>
Parameters	<b>format</b> - Format control string. <b>argument</b> - Optional arguments. <b>locale</b> - The locale to use.
Example	<code>scanf("%64s", Name);</code>
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 <code>0</code> indicates that no fields were assigned. If format is a <code>NULL</code> pointer, the invalid parameter handler is invoked. If execution is allowed to continue, these functions return <code>EOF</code> and set <code>errno</code> to <code>EINVAL</code> .
Include file	<code>&lt;stdio.h&gt;</code> or <code>&lt;wchar.h&gt;</code> for <code>wscanf()</code> , <code>_wscanf_l()</code>
Remark	<p>The <code>scanf()</code> function reads data from the standard input stream <code>stdin</code> and writes the data into the location given by argument. Each argument must be a pointer to a variable of a type that corresponds to a type specifier in format. If copying takes place between strings that overlap, the behavior is undefined.</p> <p>When reading a string with <code>scanf()</code>, always specify a width for the <code>%s</code> format (for example, <code>"%32s"</code> instead of <code>"%s"</code>); otherwise, improperly formatted input can easily cause a buffer overrun. Alternately, consider using the secure version <code>scanf_s()</code>, <code>_scanf_s_l()</code>, <code>wscanf_s()</code>, <code>_wscanf_s_l()</code> or <code>fgets()</code>.</p> <p><code>wscanf()</code> is a wide-character version of <code>scanf()</code>; the format argument to <code>wscanf()</code> is a wide-character string. <code>wscanf()</code> and <code>scanf()</code> behave identically if the stream is opened in ANSI mode. <code>scanf()</code> doesn't currently support input from a <code>UNICODE</code> stream.</p> <p>The versions of these functions with the <code>_l</code> suffix are identical except that they use the <code>locale</code> parameter passed in instead of the current thread locale.</p>

Table 1

- The information here applies to the entire `scanf()` family of functions, including the secure versions and describes the symbols used to tell the `scanf()` functions how to parse the input stream, such as the input stream `stdin` (standard input) for `scanf()`, into values that are inserted into program variables. A format specification has the following form:

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

- The format argument specifies the interpretation of the input and can contain one or more of the following:

Input	Description
<b>White-space characters</b>	Blank (' '); tab ('\t'); or newline ('\n'). A white-space character causes <code>scanf()</code> 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</b>	Except for the percent sign (%). A non-white-space character causes <code>scanf()</code> to read, but not store, a matching non-white-space character. If the next character in the input stream does not match, <code>scanf()</code> terminates.
<b>Format specifications</b>	Introduced by the percent sign (%). A format specification causes <code>scanf()</code> 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.

Table 2.

- 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.
- An input field is defined as all characters 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 (for example, %s). 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. For example scanf\_s("%d", x, 20) where 20 byte reserved for the x.
- The secure versions (those with the \_s suffix) of the scanf() family of functions require that a buffer size parameter be passed preceding each parameter of type c, C, s, S or [.
- 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

- 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. For example scanf("%10d", x) where 10 is the field width.

### 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.
- The width specification is separate and distinct from the buffer size argument required by the secure versions of these functions (i.e., scanf\_s(), wscanf\_s(), etc.). In the following example, the width specification is 20, indicating that up to 20 characters are to be read from the input stream. The buffer length is 21, which includes room for the possible 20 characters plus the null terminator:

```
char str[21];
scanf("%20s", str, 21);
```

- If the width field is not used, scanf() will attempt to read the entire token into the string. If the size specified is not large enough to hold the entire token, nothing will be written to the destination string. If the width field is specified, then the first width characters in the token will be written to the destination string along with the null terminator.

### The Size Prefix

- 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). These format-specification characters are used with type characters in scanf() or wscanf() functions to specify interpretation of arguments as shown in the following table. The type prefix I64 is a Microsoft extension and is not ANSI/ISO compatible. The type characters and their meanings are described in the "Type Characters for scanf() functions" table in scanf() Type Field Characters. The h, l, and L prefixes are Microsoft extensions when used with data of type char. You might not find it in other C/C++ compilers.

### Size Prefixes for scanf() and wscanf() Format-Type Specifiers

- The size prefixes for scanf() and wscanf() are listed in the following Table.

To specify	Use prefix	With type specifier
double	l (el letter)	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

Table 3.

- The following examples use h and l with `scanf_s()` functions and `wscanf_s()` functions:

```
scanf_s("%ls", &x, 2); // read a wide-character string
wscanf_s("%hC", &x, 2); // read a single-byte character
```

- If using an unsecured function in the `scanf()` family, omit the size parameter indicating the buffer length of the preceding argument.

### 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.
- Note that `%[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.

### Reading Unterminated strings

- To store a string without storing a terminating null character (`'\0'`), use the specification `%nc` where `n` is a decimal integer. In this case, 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.

### Type Characters for `scanf()` functions

- The type character is the only required format field; it appears after any optional format fields. The type character determines whether the associated argument is interpreted as a character, string, or number. The following Table summarizes the type character used for `scanf()/scanf_s()` family function.

Character	Type of input expected	Type of argument	Size argument in secure version?
<b>c</b>	Character. When used with <code>scanf()</code> functions, specifies single-byte character; when used with <code>wscanf()</code> functions, specifies wide character. White-space characters that are ordinarily skipped are read when <code>c</code> is specified. To read next non-white-space single-byte character, use <code>%1s</code> ; to read next non-white-space wide character, use <code>%1ws</code> .	Pointer to <code>char</code> when used with <code>scanf()</code> functions, pointer to <code>wchar_t</code> when used with <code>wscanf()</code> functions.	Required. Size does not include space for a null terminator.
<b>C</b>	Opposite size character. When used with <code>scanf()</code> functions, specifies wide character; when used with <code>wscanf</code> functions, specifies single-byte character. White-space characters that are ordinarily skipped are read when <code>C</code> is specified. To read next non-white-space single-byte character, use <code>%1s</code> ; to read next non-white-space wide character, use <code>%1ws</code> .	Pointer to <code>wchar_t</code> when used with <code>scanf</code> functions, pointer to <code>char</code> when used with <code>wscanf()</code> functions.	Required. Size argument does not include space for a null terminator.
<b>d</b>	Decimal integer.	Pointer to int.	No.
<b>i</b>	Decimal, hexadecimal, or octal integer.	Pointer to int.	No.
<b>o</b>	Octal integer.	Pointer to int.	No.
<b>u</b>	Unsigned decimal integer.	Pointer to unsigned int.	No.
<b>x</b>	Hexadecimal integer.	Pointer to int.	No.
<b>e, E, f, g, G</b>	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.
<b>n</b>	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 <code>scanf</code> functions or <code>wscanf()</code> functions.	No.
<b>s</b>	String, up to first white-space character (space, tab or newline). To read strings not delimited by space characters, use set of square brackets ( <code>[]</code> ), as discussed in <code>scanf()</code> Width Specification.	When used with <code>scanf()</code> functions, signifies single-byte character array; when used with <code>wscanf()</code> 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.
<b>S</b>	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 ( <code>[]</code> ), as discussed in <code>scanf()</code> Width Specification.	When used with <code>scanf()</code> functions, signifies wide-character array; when used with <code>wscanf</code> 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.

Table 4.

- 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**.
- 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	<code>scanf()</code> functions	<code>c</code> , <code>hc</code> , or <code>hC</code>
single byte	<code>wscanf()</code> functions	<code>C</code> , <code>hc</code> , or <code>hC</code>
wide	<code>wscanf()</code> functions	<code>c</code> , <code>lc</code> , or <code>lC</code>
wide	<code>scanf()</code> functions	<code>C</code> , <code>lc</code> , or <code>lC</code>

Table 5.

- 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`.

#### A Secure Version: `scanf_s()`, `_scanf_s_l()`, `wscanf_s()` and `_wscanf_s_l()`

- The following Table describe the secure version of `scanf_s()` family.

Item	Description
Function	<code>scanf_s()</code> , <code>_scanf_s_l()</code> , <code>wscanf_s()</code> and <code>_wscanf_s_l()</code>
Use	Read formatted data from the standard input stream. These are versions of <code>scanf()</code> , <code>_scanf_l()</code> , <code>wscanf()</code> , <code>_wscanf_l()</code> with security enhancements.
Prototype	<ol style="list-style-type: none"> <li><code>int scanf_s(const char *format [, argument]...);</code></li> <li><code>int _scanf_s_l(const char *format, locale_t locale [, argument]...);</code></li> <li><code>int wscanf_s(const wchar_t *format [, argument]...);</code></li> <li><code>int _wscanf_s_l(const wchar_t *format, locale_t locale [, argument]...);</code></li> </ol>
Parameters	<p><code>format</code> - Format control string.</p> <p><code>argument</code> - Optional arguments.</p> <p><code>locale</code> - The locale to use.</p>
Example	<code>scanf_s("%d %f %c %C %s %S", &amp;i, &amp;fp, &amp;c, 1, &amp;wc, 1, s, 60, ws, 60);</code>
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. The return value is <code>EOF</code> for an error or if the end-of-file character or the end-of-string character is encountered in the first attempt to read a character. If <code>format</code> is a <code>NULL</code> pointer. If execution is allowed to continue, <code>scanf_s()</code> and <code>wscanf_s()</code> return <code>EOF</code> and set <code>errno</code> to <code>EINVAL</code> .
Include file	<code>&lt;stdio.h&gt;</code> or <code>&lt;wchar.h&gt;</code> for <code>wscanf_s()</code> , <code>_wscanf_s_l()</code>
Remark	<p>The <code>scanf_s()</code> function reads data from the standard input stream <code>stdin</code> and writes the data into the location given by <code>argument</code>. Each argument must be a pointer to a variable of a type that corresponds to a type specifier in <code>format</code>. If copying takes place between strings that overlap, the behavior is undefined.</p> <p><code>wscanf_s()</code> is a wide-character version of <code>scanf_s()</code>; the <code>format</code> argument to <code>wscanf_s()</code> is a wide-character string. <code>wscanf_s()</code> and <code>scanf_s()</code> behave identically if the stream is opened in ANSI mode. <code>scanf_s()</code> doesn't currently support input from a <code>UNICODE</code> stream.</p> <p>The versions of these functions with the <code>_l</code> suffix are identical except that they use the <code>locale</code> parameter passed in instead of the current thread locale.</p> <p>Unlike <code>scanf()</code> and <code>wscanf()</code>, <code>scanf_s()</code> and <code>wscanf_s()</code> require the <b>buffer size to be specified for all input parameters</b> of type <code>c</code>, <code>C</code>, <code>s</code>, <code>S</code>, or <code>[</code>. The buffer size is passed as an additional parameter immediately following the pointer to the buffer or variable. For example, if reading a string, the buffer size for that string is passed as follows:</p> <pre>char s[10]; scanf("%9s", s, 10);</pre> <p>The buffer size includes the terminating null. A width specification field may be used to ensure that the token read in will fit into the buffer. If no width specification field is used, and the token read is too big to fit in the buffer, nothing will be written to that buffer. In the case of characters, one may read a single character as follows:</p> <pre>char c; scanf("%c", &amp;c, 1);</pre> <p>When reading multiple characters for non-null terminated strings, integers are used as the width specification and the buffer size.</p> <pre>char c[4]; scanf("%4c", &amp;c, 4); // not null terminated</pre>

Table 6.

- Most of the security-enhanced CRT (C Run-Time) functions and many of the preexisting functions validate their parameters. This could include checking pointers for NULL, checking that integers fall into a valid range, or checking that enumeration values are valid. When an invalid parameter is found, the invalid parameter handler is executed.

### Invalid Parameter Handler Routine

- The behavior of the C Runtime when an invalid parameter is found is to call the currently assigned invalid parameter handler. The default invalid parameter handler raises an Access Violation exception, which normally makes continued execution impossible. In **Debug mode**, an assertion is also raised. Also, if Watson crash reporting is enabled, which is the default, the application will crash and prompt the user if they want to load the crash dump to Microsoft for analysis.
- This behavior can be changed by using the function `_set_invalid_parameter_handler()` to set the invalid parameter handler to your own function. If the function you specify does not terminate the application, control is returned to the function that received the invalid parameters, and these functions will normally cease execution, return an error code, and set `errno` to an error code. In many cases, the `errno` value and the return value are both `EINVAL`, indicating an invalid parameter. In some cases, a more specific error code is returned, such as `EBADF` for a bad file pointer passed in as a parameter.

---

[| Main](#) |< [C/C++ Variable, Operator & Type 3](#) | [scanf\(\) and scanf\\_s\(\) 2](#) >| [Site Index](#) | [Download](#) |

The C `scanf()` and `scanf_s()` family: [Part 1](#) | [Part 2](#) | [Part 3](#)