Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Strings: mintf(…)

Due 9/26

Goals

This assignment has the following goals:

1. Understand number bases
2. Understand how printed representations of values relate to in-memory representations
3. Practice designing simple algorithms.
4. Practice using *test-driven development*.

Overview

You will create a function called `mintf(…)` that is similar to the standard `printf(…)` but in place of the usual format codes, your `mintf(…)` will support the following format codes:

`%d`	integer (int, short, or char), expressed in decimal notation, with no prefix
`%x`	integer (int, short, or char), expressed in hexadecimal notation with the prefix "0x"; use lowercase letters for digits beyond 9
`%b`	integer (int, short, or char), expressed in binary notation with the prefix "0b"
`%$`	double, formatted with **exactly** two digits to the right of the decimal point, and a dollar sign to the left of the first digit. Example: "$35.99"
`%s`	string (char* or string literal)
`%c`	character (int, short, or char, between 0 and 255) expressed as its corresponding ASCII character
`%%`	a single percent sign (no parameter)

For each occurrence of any of the above codes, your program shall print one of the arguments (after the format) to `mintf(…)` in the specified format. **Anything else in the format string should be expressed as is.** For example, if the format string included "%z", then "%z" would be printed. Likewise, a lone % at the end of the string would also be printed as is (e.g., example #5 below) because anything that doesn't match one of the format specifiers above should be expressed as is.

Your code will also handle `\n` and other backslash escapes, but you will not need to do anything special. The C compiler converts those for you. If this is confusing, just ignore it, and then try putting a `\n` in the format string when calling your `mintf(…)`.

You will use your `print_integer(…)` from HW02. You must finish HW02 before you begin this assignment.

Prepare

Read this assignment description carefully. (We will assume you have already read the Code Quality Standards and policies on homework and academic integrity.

Next, make sure you understand each of the topics below. As with HW02, we have suggested several concise references to help you get up to speed.

variadic functions

A variadic function is any function that takes a variable number of parameters. You may have noticed how printf, unlike most C functions you have used, can take any number of additional parameters, after the initial format string. You don't need to know this deeply, but you will need to look up the syntax, and also understand how a program determines the number of *arguments*. This page has a good overview of variadic functions, and a working example. We have another working example in the Q&A below. (Reminder: Learn from their example but do not copy it.)

printf

Make sure you understand the %d, %x, %o, %s, and %c format codes. Try writing very small programs with them, and make sure you can predict the output. See pages 418, 677, and the inside back cover of your textbook or the links on the Resources page. More detailed information can be found by typing `man printf` from bash.

Differences between `mintf(…)` and `printf(…)`

- `mintf(…)` supports `%$` but `printf(…)` does not.
- `mintf(…)` adds a prefix ("0x") to `%x` but `printf(…)` does not.
  For example, `mintf("%x", 768336)` will print `0xbb950` but `printf("%x", 768336)` will print `bb950`.
- `mintf(…)` supports `%b` but `printf(…)` does not.
  Some non-standard variants of C do support %b but we are concerned with standard ISO C99 in this class.
- `printf(…)` supports many format specifiers that `mintf(…)` does not (e.g., `%o`, `%p`, `%f`, `%e`, `%u`, etc.).

Instructions

Start by getting the files. Type `264get hw06` and then `cd hw06` from bash.

You will find only one file: mintf.h. That is the *header file*. It defines the signature of two of the functions you will implement: `mintf(…)` and `print_integer(…)`.

You will be creating three new files: mintf.c, test_mintf.c, and test_mintf.txt. There is no starter code (other than the header file).

mintf.c is where your `mintf(…)` and `print_integer(…)` implementations will go. Copy your `print_integer(…)` from HW02 into it. Copy the function signature for `mintf(…)` from mintf.h into mintf.c. Make sure all of the following headers are included at the top of your mintf.c. (Type these manually into your file.)

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

```
#include <stdarg.h>
#include "mintf.h"
```

test_mintf.c will contain a `main(…)` function that will test your `mintf(…)` and `print_integer(…)`. It must exercise all of the functionality in those two functions. For this homework, that means that in the course of running your test_mintf.c file, every line of code in your mintf.c should be executed at some point. To do this, you simply have several mintf(…) that include all of the format codes above, including a variety of valid values, including positive integers, negative integers, positive float values (for currency), negative float values (for currency), 0, strings, and an empty string. It should also have a few print_integer(…) statements to test that on its own. This will give you a very simple way to know for sure if your code works.

Use test-driven development to do this assignment incrementally.

test_mintf.txt will contain the expected output from running your tests in test_mintf.c. It will be a simple text file.

To test if your code works perfectly, you will run the following:
`gcc -o test_mintf test_mintf.c mintf.c` # compile your test together with your mintf.c to create an executable called test_mintf
`./test_mintf > test_mintf.actual` # run your executable and send the output to a file called test_mintf.actual (That filename is arbitrary.)
`diff test_mintf.actual test_mintf.txt` # compare the actual output with the expected output using the diff command
The diff command prints the *differences* so if you see any output at all, then your test failed. If you see no output, then it passed.

```
gcc -o test_mintf test_mintf.c mintf.c && ./test_mintf | diff test_mintf.txt -
```

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents	
mintf.c	functions	**`print_integer`**`(int n, int radix, char* prefix)` → *return type:* void Print the number `n` to the console (stdout) in the specified number base (radix), with the `prefix` immediately before the first digit. • `radix` may be any integer between 2 and 36 (inclusive). • For values of `radix` above 10, use lowercase letters to represent the digits following 9. For example, `print_integer(165, 16, "")` should print "`a5`". (The number *one-hundred sixty-five* would be written as "`a5`") in hexadecimal, i.e., base 16.) the number "one-hundred sixty-five". • `print_integer(…)` should *not* print a newline (`'\n'` or `'\r'`). For example, `print_integer(123, 10, ""); print_integer(-456, 10, "")` should print `"123-456"`. • More exaples are given in HW02.

file	contents	
		mintf(const char *format, ...) → *return type:* void Print `format` with any format codes replaced by the respective additional arguments, as specified above. • `%d` should work with any `int` between `INT_MIN` and `INT_MAX`. • `%$` should work with any `double` in the range of `INT_MIN-0.99` up to `INT_MAX + 0.99`. ~~Round to the nearest~~ Truncate to a multiple of 0.01 (e.g., 768.336 ⇒ 768.33, -768.336 ⇒ 768.33). • Note: Until Sat 9/23, this description contained contradictory instructions with respect to rounding. In one place it said to round to the nearest 0.01. In another place it said to truncate. The tester will accept both. If you have not implemented this part yet, please truncate.
mintf.h	declarations	declaration of `print_integer(…)`
test_mintf.c	functions	**main**(int argc, char* argv[]) → *return type:* int Test your `mintf(…)`. Your `main(…)` must return EXIT_SUCCESS (0).
test_mintf.txt	output	Expected output from running your test_mintf.c.

2. Your test_mintf.c must exercise all functionality (e.g., all format codes, negative numbers, any `radix` between 2 and 36, largest possible `n`, smallest possible `n`, etc.).
3. For format codes related to numbers (%d, %x, %b, %$), your program should handle any valid int value on the system it is being run on, including 0, positive numbers, and negative numbers.
4. Your code may not make any assumptions about the size of an `int`.
5. Your test_mintf.c should work—and give the same output—with *any* correct mintf.c, and likewise for your mintf.c.
6. For negative numbers, print the "-" before the prefix (e.g., "-$3.00" not "$-3.00", "-0x12AD" not "0x-12AD").
7. Only the following external header files, functions, and symbols are allowed in your mintf.c. That means you may use `printf(…)` in your test_mintf.c but not in your mintf.c. You may use `fputc(…)` and `stdout` in either one (or both).

header	*functions/symbols*	*allowed in…*
`limits.h`	`INT_MAX`, `INT_MIN`	`test_mintf.c`
`stdarg.h`	`va_list`, `va_start`, `va_arg`, `va_end`, `va_copy`	`mintf.c`, `test_mintf.c`
`stdbool.h`	`bool`, `true`, `false`	`mintf.c`, `test_mintf.c`
`stdio.h`	`fputc`, `stdout`, `fflush`	`mintf.c`, `test_mintf.c`
`stdio.h`	`printf`	`test_mintf.c`
`assert.h`	`assert`	`mintf.c`, `test_mintf.c`
`stdlib.h`	`EXIT_SUCCESS`, `abs`	`mintf.c`, `test_mintf.c`

All others are prohibited unless approved by the instructor. Feel free to ask if there is something you would like to use.
8. Repeat: Do not call `printf(…)`
9. Submissions must meet the code quality standards and the policies on homework and academic integrity.

How much work is this?

Parsing the format string feel like a puzzle. Think it through before you start writing a lot of code. The rest will involve a modest amount of code. Expect to do plenty of testing and debugging to get this right.

The instructor's `mintf(…)` and `print_integer(…)` are 51 and 20 sloc*, respectively. cloc reports 75 sloc for the whole file.
* *sloc* = "source lines of code" (excluding comments and blank lines)

Examples

These are to help you understand the spec. Your test cases must be your own. **Do not copy these (even with minor modifications).**

1.	Code:	`mintf("768336");`
	Output:	`768336`

2.	Code:	`mintf("My favorite number is %d!", 768336);`
	Output:	`My favorite number is 768336!`

3.	Code:	`mintf("%d written in hex is %x.", 768336, 768336);`
	Output:	`768336 written in hex is 0xbb950.`

4.	Code:	`mintf("%d written in binary is %b.", 768336, 768336);`
	Output:	`768336 written in binary is 0b10111011100101010000.`

5.	Code:	`mintf("Chance of coalescent cloudburst:  0.375%");`
	Output:	`Chance of coalescent cloudburst: 0.375%`

6.	Code:	`mintf("Chance of coalescent cloudburst:  0.375%%");`
	Output:	`Chance of coalescent cloudburst: 0.375%`

7.	Code:	`mintf("%s the cat has only %d lives left.", "Orlicasta", 8);`
	Output:	`Orlicasta the cat has only 8 lives left.`

8.	Code:	`mintf("Orlicasta's shoes cost %$.", 768.336);`
	Output:	`Orlicasta's shoes cost $768.33.`

9.	Code:	`mintf("Orlicasta's bank balance is %$.", -768.336);`
	Output:	`Orlicasta's bank balance is -$768.33.`

10.	Code:	`mintf("Excess parameters are ignored: %d", 768336, 768337, 768338);`
	Output:	`Excess parameters are ignored: 768336`
		This should take no extra effort to code.

11.	Code:	`mintf("Insufficient parameters may lead to undefined behavior: %d %d %d", 768336);`
	Output:	`Insufficient parameters may lead to undefined behavior: 768336 □□□□□□□□□□□□□□□`
		You do not need to test for this.

12.	Code:	`mintf("Mismatched parameters may lead to undefined behavior: %d", "Orlicasta");`
	Output:	`Mismatched parameters may lead to undefined behavior: □□□□□□□□□□□□□□□`
		You do not need to test for this.

These are to help you understand the spec. Your test cases must be your own. **Do not copy these (even with minor modifications).**

Submit

In general, to submit any assignment for this course, you will use the following command:

```
264submit ASSIGNMENT FILES…
```

For HW06, you will type `264submit hw06 mintf.c test_mintf.c test_mintf.txt` from inside your hw06 directory.

You can submit as often as you want, even if you are not finished with the assignment. That saves a backup copy which we can retrieve for you if you ever have a problem.

Q&A

1. **Is an int always in decimal (base 10)?**
   No. See the next question.

2. **Does my print_integer need to deal with inputs that are specified in number bases other than decimal? For example, does my print_integer need to be able to convert from hex to binary?**
   An int does not have a number base. It is just a value. The base (or "radix") of a number only refers to the way it is written, i.e., in your C code, or in the output of a program you write. For example, if n is the number of fingers on two hands, you could write that as 10, 0b1010, or 0xa, but it the same quantity no matter how you write it.

   When gcc reads your code for the first time, it interprets any integer literals (raw numbers in your code) according to how you write your code. Once your code has been compiled, the base that you used to write it becomes completely moot.

   Example: To gcc, the following are equivalent and have exactly the same effect.

   ```
   int n = 65;
   printf("n can be written as %d in decimal, as 0x%x in hexadecimal, or as
   %c if interpreted as an ASCII character.", n, n, n);

   int n = 0x41;
   printf("n can be written as %d in decimal, as 0x%x in hexadecimal, or as
   %c if interpreted as an ASCII character.", n, n, n);

   int n = 0b1000001;  // non-standard, but accepted by gcc; do not use for
   ECE 264
   printf("n can be written as %d in decimal, as 0x%x in hexadecimal, or as
   %c if interpreted as an ASCII character.", n, n, n);
   ```

   They all print the following:

   ```
   n can be written as 65 in decimal, as 0x41 in hexadecimal, or as A if
   interpreted as an ASCII character.
   ```

3. **What if there is a `'\0'` in the middle of the string?**
   By definition, a string ends at the first `'\0'`, so you would stop printing characters when you see the `'\0'`.

4. **What if mintf is called without any optional parameters?**
   That is fine and normal. For example, the following should have the same effect:

   ```
   mintf("Hello world!\n");
   ```

```
printf("Hello world!\n");
```

5. **What's the deal with variadic functions / va_arg / va_list?**
A variadic function is any function that takes an unknown number of optional parameters. The optional parameters are represented by three dots (e.g., int foo(int n, ...)). Those dots are part of the C language. The optional arguments are accessed using va_arg. You must call va_start once in that function, before the first use of va_arg. You must call va_end once in the function, after the last use of va_arg. You must include stdarg.h at the top of the file (i.e., `#include <stdarg.h>`). There is no way to know from va_arg how many optional arguments there are, so you need to use some other information (in this case the format string) to know how many there are (and hence how many times to call va_arg).

Here's an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>                                 // - This is required in
any program that uses
                                                    //   variadic functions
(additional arguments).
void print_many_numbers(int num_of_nums, ...) {     // - The ... stands for
any number of arguments.
    va_list more_args;                              // - Get a handle to the
additional arguments.
    va_start(more_args, num_of_nums);               // - This is required to
access them.  Yes, it
                                                    //   is weird.
    for(int i = 0; i < num_of_nums; i++) {          //
        int next_num = va_arg(more_args, int);      // - Get the next
additional argument.
        printf("%d\n", next_num);                   // - Print it.
    }                                               //
                                                    //
    va_end(more_args);                              // - This is required
after you're done.
}                                                   //   Yes, this is weird
(too).

int main(int argc, char* argv[]) {
    print_many_numbers(4, 768336, 768337, 768338, 768339);
    return EXIT_SUCCESS;
}
```

Here is the output:

```
768336
768337
768338
768339
```

Other examples can be found in the articles in the Resources section.

6. **Should hex digits above 10 be uppercase or lowercase?**
Lowercase.

7. **The specification states, "For each occurrence of any of the above codes, your program shall print one of the arguments (after the format) to mintf(…) in the specified**

format." What does this mean?
It works just like printf. For each format code (e.g., "%d", "%x", etc) other than "%%" in the format string, your mintf should take one of the optional arguments and print it in the specified format. For example, `printf("I knew that %s was %d%% Martian because of the %$ tab for breakfast.", "Mason", 99, 67.8)` would print "I knew that Mason was 99% martian because of the $67.80 tab for breakfast.".

8. **The "0x" prefix for hexadecimal and "0b" prefix for binary output seem somewhat arbitrary. Where do these come from?**
   These prefixes mimic the formats that gcc accepts when you specify raw integers in your code. The following statements are completely identical in the eyes of gcc.
   ```
   int n = 10;
   int n = 0xa;
   int n = 0b1010;  // non-standard, but accepted by gcc; do not use for ECE
   264
   ```
9. **Why do we need the prefix parameter to print_integer?**
   In a sense, it is a convenience. When printing a negative number that requires a prefix, the prefix should go between the minus sign and the first digit. For example, `mintf("%$", -12.3)` should print "-$12.30" not "$-12.30". Similarly, `mintf("%x", -10)` should print "-0xa" not "0x-a". Since your mintf will most likely depend on your print_integer, having this prefix parameter to print_integer will make your mintf code simpler than it would be otherwise. If this still doesn't make sense, just think through how you would structure your mintf. You will most likely find that the only reasonable way to do it is to have a separate function like print_integer.
10. **What is the relationship between print_integer and mintf?**
    Your print_integer will almost certainly be used by your mintf. For that reason, you may want to write your print_integer first.
11. **Will my print_integer be called directly by external code?**
    Yes.
12. **How do I handle `\n`?**
    See the note in the assignment description about this. If your code works for everything other than `\n`, then it should work for `\n`, too. You shouldn't need to do anything special. When gcc reads your code, it will automatically interpret the `\n` as ASCII character 10 (newline).
13. **How do I deal with the double for the `"%$"`?**
    Here's a hint: To extract the integer portion of a double (e.g., 3.5 ⇒ 3), put `(int)` before the double. For example, the following code prints 3.
    ```
    double f = 3.5;
    int n = (int) f;
    printf("%d", n);
    ```
    Also note that for the "%$" format code, you print *exactly* 2 decimal digits. For example, `mintf("%$", 1.2)` would print "$1.20". This makes things much easier for you than they could be otherwise.
14. **What if someone passes an integer instead of a double `"%$"`?**
    You may assume the argument for `%$` is a double.
15. **What radixes (number bases) must my print_integer support?**
    Your print_integer should support any value for radix between 2 and 36 (inclusive). For radixes above 10, it should represent digits beyond than 9 with lowercase letters. For example, `print_integer(11, 16, "")` would print "b" because b is the (11-9)th lowercase

letter of the alphabet. Similarly, `print_integer(20, 21, "")` would print "k" because k is the (20-9)th letter of the alphabet.

This converter may help you get a sense (and test your code).

16. **What is the difference between a "radix" and a "base"?**
They are synonyms for the same thing.

17. **What should my program do if someone passes the wrong number of parameters to mintf?**
You may assume that the caller will only pass the correct number of parameters. There will be no penalty if your mintf crashes or prints garbage in case there aren't enough parameters for the given format string.

18. **What should my program do if the type of a parameter does not match the format code (e.g., mintf("%d", "birthday"))?**
You may assume that the type of parameters will match the corresponding format code. There will be no penalty if your mintf misbehaves in case of such a mismatch.

19. **Should I write `#include "mintf.h"` (quotation marks) or `#include <mintf.h>` (angle brackets)?**
Use quotation marks for header files in your project. Use angle brackets for standard header files. Thus, you should start your mintf.c with this:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "mintf.h"
```

20. **May we modify mintf.h?**
No.

21. **May we turn in our own mintf.h?**
Yes, you may turn it in, but that file will be ignored. We will copy your files into an empty directory and then copy our test files, including mintf.h on top. Thus, any mintf.h you submit will be overwritten.

22. **How does mintf relate to printf?**
Your mintf will have the same behavior as printf for the format codes "%d", "%c", "%s", and "%%". For "%x" your mintf is different in that yours will automatically add the "0x" prefix, whereas printf does not. For example, `mintf("%x", 10)` prints "0xa" whereas `printf("%x", 10)` prints "a". In addition, your mintf will support two format codes that printf does not support at all: "%b" and "%$".

23. **What types can I use with each format code?**
For `%d`, `%b`, or `%x`, you would normally pass an int, although a char or short will also work. For `%$`, pass a double. For `%s`, pass a string. For `%c`, you would normally pass a char (e.g., 'c') but an int or short will also work, as long as they are in the range of 0 to 255 (inclusive). For `%%`, you would not pass any argument. This is implied by the table at the top of this assignment description.

The underlying rule is explained more explicitly here, in one of the two recommended articles. For most people, these details shouldn't be necessary. If you were thinking of trying unsigned types or long types, don't.

24. **For %$, if the number has >2 digits to the right of the decimal point, should we round or truncate?**
    Truncate. For example, `mintf("%$", 768.336)` should print "$768.33".

The Q&A section of HW02 answers many more questions, including how to handle `INT_MIN`.

Pre-tester ●

The pre-tester for HW06 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw06
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Updates

9/22/2017	Allow assert(…); bounds for `%$` should be `INT_MIN - 0.99` to `INT_MAX + 0.99`.
9/23/2017	Truncate floating point numbers to a multiple of 0.01 (e.g., 768.336 ⇒ 768.33,

	-768.336 ⇒ -768.33). Previously, the specification was contradictory. In one place it said to round to the nearest 0.01. In another place, it said to truncate. The tester will accept both. If you haven't implemented this part yet, please truncate.
11/26/2017	Adjust line count of `print_integer(…)` from 16 to 20.