Programming Assignment #1: Varg[1]

COP 3502, Spring 2017

Due: Sunday, January 22, *before* 11:59 PM

Table of Contents

1 "Varg" is not a real word. It's a portmanteau of **var** and **arg**, which in this case stand for "**var**iable number of **arg**uments." The correct technical term for functions that take a variable number of arguments is "variadic."

Abstract

In COP 3223 (Introduction to C Programming), you encountered many functions – such as *printf()* – that were capable of taking a different number of arguments every time you call them. However, the functions you wrote in that class probably never had that ability; instead, they were always restricted to taking the same number of arguments each time they were called.

In this programming assignment, you will write two variadic functions – that is, functions that take a variable number of arguments. By completing this assignment, you will acquire a new, advanced C programming skill. Because of one of the restrictions placed on the functions, you will also gain a small amount of experience developing an algorithm with a focus on runtime efficiency.

On the software engineering side of things, you will learn to construct programs that use multiple source files and custom header (.h) files. This assignment will also help you hone your ability to acquire new knowledge by reading technical specifications. If you end up pursing a career as a software developer, the ability to rapidly digest technical documentation and work with new software libraries will be absolutely critical to your work, and this assignment will provide you with a gentle exercise in doing just that.

Finally, this assignment is specifically designed to require relatively few lines of code so that you can make your first foray into the world of Linux without a huge, unwieldy, and intimidating program to debug. As the semester progresses, the programming assignments will become more lengthy and complex, but for now, this assignment should provide you with a gentle introduction to debugging in a foreign development environment.

Attachments

Varg.h, *varsum.c*, *testcase{01-06}.c*, *output{01-06}.txt*, and *test-all.sh*

Deliverables

Varg.c

(***Note!*** Capitalization and spelling of your filename matters!)

1. Overview

In this assignment, you'll write two variadic functions – functions that take variable numbers of arguments, just like `printf()` is capable of doing. (There are also two other functions for you to write. A complete list of required functions, including their functional prototypes, is given below in Section 8, "Function Requirements".)

You will submit a single source file, named `Varg.c`, that contains all required function definitions, as well as any auxiliary functions you deem necessary. In `Varg.c`, you will have to #include any header files necessary for your functions to work, including the custom `Varg.h` file we have distributed with this assignment (see Section 5, "Varg.h").

Note: You will *not* write a main() function in the source file you submit! Rather, we will compile your source file with our own `main()` function(s) in order to test your code. We have attached example source files that have `main()` functions, which you can use to test your code. You can write your own `main()` functions for testing purposes, but the code you submit must not have a `main()` function. We realize this is completely new territory for most of you, so don't panic. We've included instructions for compiling multiple source files into a single executable (e.g., mixing your `Varg.c` with our `Varg.h` and `testcaseXX.c` files) in Sections 11 and 12 of this PDF.

Although we have included test cases with sample `main()` functions to get you started with testing the functionality of your code, we encourage you to develop your own test cases, as well. Ours are by no means comprehensive. We will use much more elaborate test cases when grading your submission.

Start early. Work hard. Good luck!

2. Important Note: Test Case Files Look Wonky in Notepad

Included with this assignment are several test cases, along with output files showing exactly what your output should look like when you run those test cases. You will have to refer to those as the gold standard for how your output should be formatted. Please note that if you open those files in Notepad, they will appear to be one long line of text. That's because Notepad handles end-of-line characters differently from Linux and Unix-based systems. One solution is to view those files in an IDE (like CodeBlocks), or you could use a different text editor (like Atom or Sublime).

3. Adopting a Growth Mindset

A word of advice before we dive in to the details: When faced with an assignment like this, which has many different facets, some of which might appear foreign and/or challenging, it's important not to look at it as an instrument that is being used to measure how much you already know (whether that's knowledge of programming, operating systems, or even what some might call "natural intellectual capability"). Rather, it's important to view this assignment as a chance to learn something new, grow your skill set, and embrace a new challenge.

It's also important to view intellectual capability as something that can grow and change over one's lifetime. Adopting that mindset will allow you to reap greater benefits from this assignment (and from

all your college-level coursework) regardless of the grades you earn.

For more on the importance of adopting a growth mindset throughout your academic career and how that can impact your life, see the following:

> Growth Mindset vs Fixed Mindset: An Introduction (Watch time: 2 min 42 sec)
>
> The Power of Belief – Mindset and Success (Watch time: 10 min 20 sec)

4. Writing Variadic Functions

One of the trickiest things about writing a variadic function is that you must somehow tell the function how many arguments you're passing to it. When you call `printf()`, you do this implicitly, because `printf()` goes through your first argument (a string), and every time it sees a conversion code (such as "%d" or "%c"), it knows there is another input argument waiting to be processed. By reading the format string, the function figures out how many arguments to read after the initial one.

Included with this assignment is a file called `varsum.c`, which I wrote to show you how to implement a simple function to add up an arbitrary number of integer arguments passed to a function. The file has two versions of that function: `mySum()` and `myOtherSum()`. The only difference between the two implementations is how each function figures out how many integers it will be processing.

From that source file, you'll see that when writing a variadic function, all the magic happens with the `va_list` data type and the `va_start` and `va_arg` functions. In order to use `va_list`, `va_start`, and `va_arg`, you must include `stdarg.h` at the top of your source code, like so:

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

Note that `stdarg.h` is a standard system library, just like `stdio.h` and `string.h`.

5. Varg.h

Included with this assignment is a customer header file that includes functional prototypes for all the functions you will be implementing. You should #include this file from your `Varg.c` file, like so:

```
#include "Varg.h"
```

The "quotes" (as opposed to <brackets>) indicate to the compiler that this header file is found in the same directory as your source, not a system directory.

You should not modify `Varg.h` in any way, and you should not send `Varg.h` when you submit your assignment. We will use our own unmodified copy of `Varg.h` when compiling your program.

If you write auxiliary functions ("helper functions") in your `Varg.c` file (which is strongly encouraged!), you should ***not*** add those functional prototypes to `Varg.h`. Our test case programs will not call your helper functions directly, so they do not need any awareness of the fact that your helper functions even exist. (We only list functional prototypes in a header file if we want multiple source files

to be able to call those functions.) So, just put the functional prototypes for any helper functions you write at the top of your `Varg.c` file.

Think of `Varg.h` as a bridge between source files. It contains functional prototypes for functions that might be defined in one source file (such as your `Varg.c` file) and called from a different source file (such as the `testcaseXX.c` files we have provided with this assignment).

6. Test Cases and the test-all.sh Script

We've included multiple test cases with this assignment, which show some ways in which we might test your code. These test cases are not comprehensive. You should also create your own test cases if you want to test your code comprehensively. In creating your own test cases, you should always ask yourself, "How could these functions be called in ways that don't violate the program specification, but which haven't already been covered in the test cases included with the assignment?"

We've also included a script, `test-all.sh`, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with `Varg.c`, `Varg.h`, and all the test case files, and typing:

```
bash test-all.sh
```

7. Output

The functions you write for this assignment should not produce any output. If your functions cause anything to print to the screen, it will interfere with our test case evaluation.

Please be sure to disable or remove any `printf()` statements you have in your code before submitting this assignment.

8. Function Requirements

In the source file you submit, `Varg.c`, you must implement the following functions. You may implement auxiliary functions (helper functions) to make these work, as well, although that is probably unnecessary for this assignment. Please be sure the spelling, capitalization, and return types of your functions match these prototypes exactly.

```
char mostFrequentChar(int n, ...);
```

> **Description:** This function takes a single non-negative integer, *n*, followed by a list of exactly *n* lowercase, alphabetic characters (any characters on the range 'a' through 'z'), and returns the most frequently occurring character it received as input. If multiple characters are tied for the distinction of "most frequently occurring," you should return the first one that occurred that number of times when processing the argument list from left to right. For example, when processing the input to the function call `mostFrequentChar(5, 'a', 'b', 'b', 'a', 'c')`,

both ‘a’ and ‘b’ are tied (with two occurrences each), but the first character to rack up two occurrences when reading the list in order (from left to right) is ‘b’. (For further examples, see the test cases included with this assignment.)

Special Restriction: As you read the list of input arguments, you are not allowed to store that list anywhere in memory (for example, in a linked list or a string). In other words, you can read the list of arguments passed to your function exactly once. You should not try to save that list in memory so that you can re-read it multiple times within your function.

Output: This function should not print anything to the screen.

Return Value: The most frequently occurring character in the argument list. (See the description above for an explanation of how to handle ties.) If *n* is equal to zero, your function should return the ‘\0’ character, which is C’s null sentinel.

```
char fancyMostFrequentChar(char c, ...);
```

Description: This function takes as its input a list of characters. The function is guaranteed to receive at least one argument. The last argument passed to the function will always be the ‘\0’ character, which is C’s null sentinel. All other characters in the list will be lowercase, alphabetic characters (any characters on the range ‘a’ through ‘z’). Your function should return the most frequently occurring character in the list. In the event of ties, you should handle them in the same way that the `mostFrequentChar()` function handles ties (described above). Note that it’s possible for ‘\0’ to be the only argument given, in which case you should return ‘\0’.

Special Restriction: This function is subject to the same special restriction listed above for the `mostFrequentChar()` function.

Output: This function should not print anything to the screen.

Return Value: The most frequently occurring character in the argument list. (See the description above for an explanation of how to handle ties.) If ‘\0’ is the only argument passed to this function, then you should return ‘\0’.

```
double difficultyRating(void);
```

Output: This function should not print anything to the screen.

Return Value: A double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
double hoursSpent(void);
```

Output: This function should not print anything to the screen.

Return Value: An estimate (greater than zero) of the number of hours you spent on this assignment.

9. Input Specifications Are a Contract

In testing, you can rest assured that we will only call your functions in ways that jive with the function descriptions above. For example, we will never call `mostFrequentChar()` without specifying an integer as the first argument, and that integer is always guaranteed to correctly indicate the number of arguments that follow. Similarly, we will never call `fancyMostFrequentChar()` without ending the list of parameters with a null sentinel ('\0'), and neither function will receive unexpected characters in the argument list (such as a capital 'Q' or punctuation marks such as '!').

10. ASCII Character Values

Note: If you haven't thought about how to solve this problem yet, it might not be immediately obvious why the information in this section might be useful to you in this assignment.

In C, each character has an underlying integer value called its ASCII value. (ASCII is an international standard for character numbering. If you want to read more about the topic, see the ASCII article on Wikipedia. For this assignment, however, you don't need to know any more about ASCII than what I've written up in this section.)

To see a character's underlying integer value, you can always just print it as an integer, like so:

```
printf("%d", 'a');
```

The ASCII value for 'a' is 97, so the above line of code should print out 97.

Lowercase letters are numbered sequentially: 'a' is 97, 'b' is 98, 'c' is 99, and so on. If for some reason you wanted to convert the characters 'a' through 'z' to integers 0 through 25 (maybe so you could use them to access indices in an array of length 26?), you could just subtract 97 from those characters. For example:

```
printf("%d", 'a' - 97);  // 'a' - 97 = 97 - 97 = 0
printf("%d", 'b' - 97);  // 'b' - 97 = 98 - 97 = 1
printf("%d", 'c' - 97);  // 'c' - 97 = 99 - 97 = 2
```

Personally, I never hard-code the value 97 when converting characters to integers (partly because I never used to be able to remember it off the top of my head). Instead, I just use 'a' in place of 97, because the math will always work out:

```
printf("%d", 'a' - 'a');  // 'a' - 'a' = 97 - 97 = 0
printf("%d", 'b' - 'a');  // 'b' - 'a' = 98 - 97 = 1
printf("%d", 'c' - 'a');  // 'c' - 'a' = 99 - 97 = 2
```

Continued on the following page...

11. Compilation and Testing (CodeBlocks)

The key to getting multiple files to compile into a single program in CodeBlocks (or any IDE) is to create a project. Here are the step-by-step instructions for creating a project in CodeBlocks, which involves importing `Varg.h`, `testcase01.c`, and the `Varg.c` file you've created (even if it's just an empty file so far).

1. Start CodeBlocks.
2. Create a New Project (*File -> New -> Project*).
3. Choose "Empty Project" and click "Go."
4. In the Project Wizard that opens, click "Next."
5. Input a title for your project (e.g., "Varg").
6. Choose a folder (e.g., Desktop) where CodeBlocks can create a subdirectory for the project.
7. Click "Finish."

Now you need to import your files. You have two options:

1. Drag your source and header files into CodeBlocks. Then right click the tab for **<u>each</u>** file and choose "Add file to active project."

   – or –

2. Go to *Project -> Add Files...*. Browse to the directory with the source and header files you want to import. Select the files from the list (using CTRL-click to select multiple files). Click "Open." In the dialog box that pops up, click "OK."

You should now be good to go. Try to build and run the project (F9).

Note that if you import both `testcase01.c` *and* `testcase02.c`, the compiler will complain that you have multiple definitions for `main()`. You can only have one of those in there at a time. You'll have to swap them out as you test your code.

Yes, constantly swapping out the test cases in your project will be a bit annoying. You can avoid this if you're willing to migrate away from an IDE and start compiling at the command line instead. If you're interested in doing that in Windows, please look around online for instructions on how to make that happen, and see a TA in office hours if you get stuck. Alternatively, you might consider installing Linux on a separate partition of your hard drive. If you take that approach, just be sure to back up your hard drive first.

Note! Even if you develop your code with CodeBlocks on Windows, you ultimately have to transfer it to the Eustis server to compile and test it there. See the following page (Section 12, " Compilation and Testing (Linux/Mac Command Line)") for instructions on command line compilation in Linux.

12. Compilation and Testing (Linux/Mac Command Line)

To compile multiple source files (`.c` files) at the command line:

```
gcc Varg.c testcase01.c
```

By default, this will produce an executable file called `a.out`, which you can run by typing:

```
./a.out
```

If you want to name the executable file something else, use:

```
gcc Varg.c testcase01.c -o Varg.exe
```

...and then run the program using:

```
./Varg.exe
```

Running the program could potentially dump a lot of output to the screen. If you want to redirect your output to a text file in Linux, it's easy. Just run the program using the following command, which will create a file called `whatever.txt` that contains the output from your program:

```
./Varg.exe > whatever.txt
```

Linux has a helpful command called `diff` for comparing the contents of two files, which is really helpful here since we've provided several sample output files. You can see whether your output matches ours exactly by typing, e.g.:

```
diff whatever.txt output01.txt
```

If the contents of `whatever.txt` and `output01.txt` are exactly the same, `diff` won't have any output. It will just look like this:

```
seansz@eustis:~$ diff whatever.txt output01.txt
seansz@eustis:~$ _
```

If the files differ, it will spit out some information about the lines that aren't the same. For example:

```
seansz@eustis:~$ diff whatever.txt output01.txt
1c1
< fail whale :(
---
> Hooray!
seansz@eustis:~$ _
```

13. Getting Started: A Guide for the Overwhelmed

Okay, so, this might all be overwhelming, and you might be thinking, “Where do I even start with this assignment?! I’m in way over my head!”

Don’t panic! There are plenty of TA office hours where you can get help, and here’s my general advice on starting the assignment:

1. First and foremost, start working on this assignment early. Nothing will be more frustrating than running into unexpected errors or not being able to figure out what the assignment is asking you to do on the day that it is due.

2. Secondly, glance through this entire PDF to get a general overview of what the assignment is asking you to do, even if you don’t fully understand each section right away.

3. Thirdly, before you even start programming, open up the test cases and sample output files included with this assignment and trace through a few of them to be sure you have an accurate understanding of what the function calls are supposed to be doing.

4. Once you’re ready to begin coding, start by creating a skeleton `Varg.c` file. Add a header comment with your name and NID, add some standard `#include` directives, and be sure to `#include "Varg.h"` from your source file. Then copy and paste each functional prototype from `Varg.h` into `Varg.c`, transform those prototypes into functions by removing the semicolon and setting up curly braces for each function signature, and then set up all those functions to return dummy values (some arbitrary `char` or `float`, as appropriate).

5. Test that your `Varg.c` source file compiles. Do this before you’ve even taken a stab at making the functions work! If you’re at the command line on a Mac or in Linux, your source file will need to be in the same directory as `Varg.h`, and you can test compilation like so:

   ```
   gcc -c Varg.c
   ```

   Alternatively, you can try compiling it with one of the test case source files, like so:

   ```
   gcc Varg.c testcase01.c
   ```

   For more details, see Section 12, “ Compilation and Testing (Linux/Mac Command Line).”

   If you’re using an IDE (i.e., you’re coding with something other than a plain text editor and the command line), open up your IDE and start a project using the instructions above in Section 11, “ Compilation and Testing (CodeBlocks)”. Import `Varg.h`, `testcase01.c`, and your new `Varg.c` source file, and get the program compiling and running before you move forward. (Note that CodeBlocks is the only IDE we officially support in this class.)

6. Once you have your project compiling, go back to Section 8, “Function Requirements”, and read through the function requirements. Do some brainstorming on how you want to solve the problem. Don’t even try to write any code yet. Just assume you’ll be able to get your function to

read all the arguments one by one, and think about what you'll do with each argument as you receive it.

7. Load up `varsum.c` and give it a read. You might need to read it a couple times before it all starts to click. Don't be discouraged by that. Your brain is re-wiring yourself as you read, and setting itself up to comprehend documents like that much more rapidly in the future.

8. Once you kind of have a grasp of how to write variadic functions (even if it feels a bit vague at first), dive in fearlessly. Be bold! Go back to the list of required functions (Section 8, "Function Requirements"), and try to implement one function at a time, while referring to `varsum.c` regularly. Always stop to compile and test your code before moving on to another function! Throughout the semester, if you frequently stop to check whether your code is compiling, you will save yourself a lot of headaches.

9. If you get stuck while working on this assignment, draw diagrams on paper or a whiteboard. Make boxes for all the variables in your program. Trace through your code carefully, step by step, using these diagrams.

10. You're bound to encounter errors in your code at some point. Use `printf()` statements liberally to verify that your code is producing the results you think it should be producing (rather than making assumptions that certain components are working as intended). You should get in the habit of being immensely skeptical of your own code and using `printf()` to provide yourself with evidence that your code does what you think it does.

11. If you encounter a segmentation fault, you should always be able to use `printf()` and `fflush()` to track down the *exact* line you're crashing on.

12. This semester (especially in later programs), you'll need to examine a lot of debugging output. You might want to set up a function that prints debugging strings only when you #define a `DEBUG` value to be something other than zero, so you can easily flip debugging output on and off. (Just be sure to remove your debugging statements before you submit your assignment, so your code is nice and clean and easy for us to read.)

13. When you find a bug, or if your program is crashing on a huge test case, don't trace through hundreds of iterations of some for-loop to track down the error. Instead, try to cook up a new `main()` function with a very small test case (as few lines as possible) that directly calls the function that's crashing. The less code you have to trace through, the easier your debugging tasks will be.

14. Deliverables

Submit a single source file, named `Varg.c`, via Webcourses. The source file should contain definitions for all the required functions (listed above), as well as any auxiliary functions you need to make them work.

Your source file **must not** contain a `main()` function. Do not submit additional source files, and do not submit a modified `Varg.h` header file. Your source file (without a `main()` function) should compile at

the command line using the following command:

```
gcc -c Varg.c
```

It must also compile at the command line if you place it in a directory with `Varg.h` and a test case file (for example, `testcase01.c`) and compile like so:

```
gcc Varg.c testcase01.c
```

Be sure to include your name and NID as a comment at the top of your source file.

15. Grading

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

50%	correct output for test cases
25%	implementation details (manual inspection of your code)
5%	`difficultyRating()` is implemented correctly
5%	`hoursSpent()` is implemented correctly
5%	source file is named correctly (`Varg.c`); spelling and capitalization count
10%	adequate comments and whitespace; source includes student name and NID

Note! Your program must be submitted via Webcourses, and it must compile and run on Eustis to receive credit. Programs that do not compile will receive an automatic zero.

Your grade will be based largely on your program's ability to compile and produce the *exact* output expected. Even minor deviations (such as capitalization or punctuation errors) in your output will cause your program's output to be marked as incorrect, resulting in severe point deductions. The same is true of how you name your functions and their parameters. Please be sure to follow all requirements carefully and test your program throughly.

Additional points will be awarded for style (appropriate commenting and whitespace) and adherence to implementation requirements. For example, the graders might inspect your `mostFrequentChar()` and `fancyMostFrequentChar()` functions to see whether they meet the special requirement listed in the function descriptions above.

Start early. Work hard. Good luck!