# P1 Username, Email, and Password Verification

New Attempt

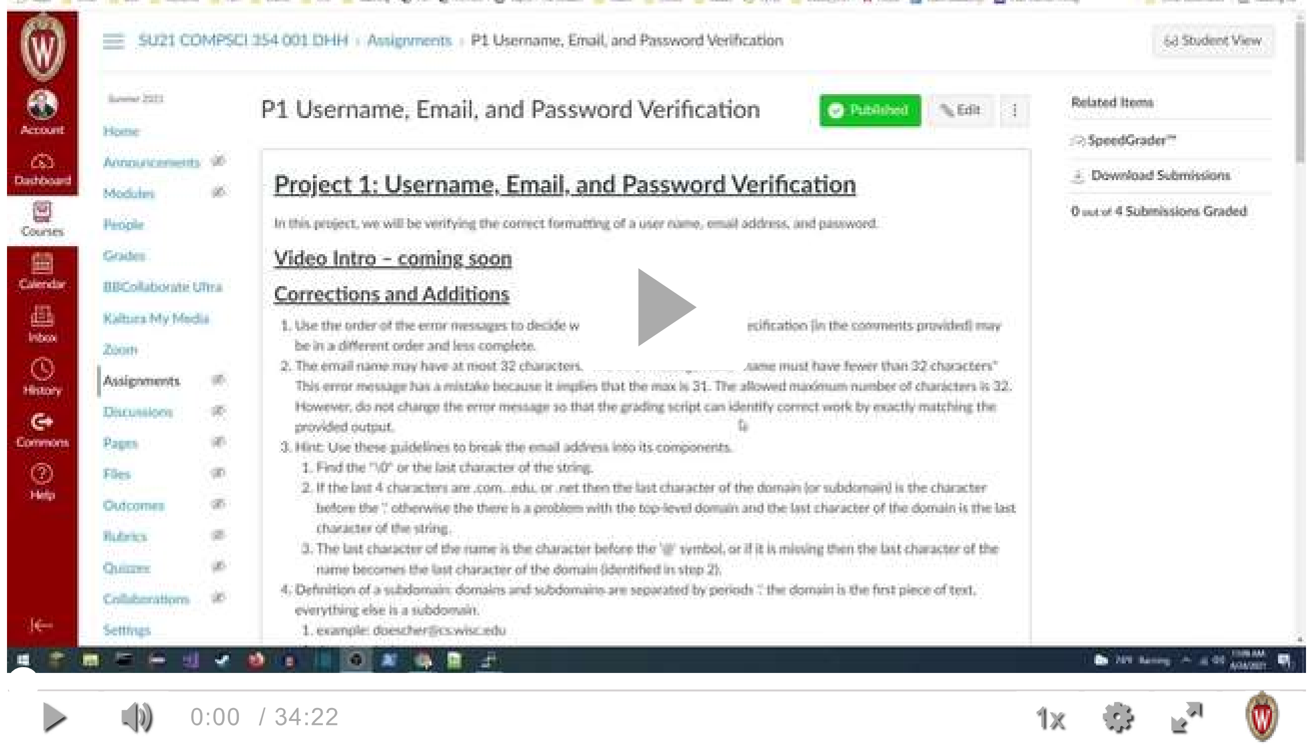
**Due** Jul 1 by 11:59pm **Points** 50 **Submitting** a file upload **File Types** c **Available** Jun 21 at 12am - Aug 14 at 11:59pm about 2 months

## Project 1: Username, Email, and Password Verification

In this project, we will be verifying the correct formatting of a user name, email address, and password.

## Video Intro

P1_Intro_And_Strategy



P1 Username, Email, and Password Verification

Project 1: Username, Email, and Password Verification

In this project, we will be verifying the correct formatting of a user name, email address, and password.

Video Intro – coming soon

Corrections and Additions

1. Use the order of the error messages to decide w … ecification (in the comments provided) may be in a different order and less complete.
2. The email name may have at most 32 characters. … name must have fewer than 32 characters" This error message has a mistake because it implies that the max is 31. The allowed maximum number of characters is 32. However, do not change the error message so that the grading script can identify correct work by exactly matching the provided output.
3. Hint: Use these guidelines to break the email address into its components.
   1. Find the "\0" or the last character of the string.
   2. If the last 4 characters are .com, .edu, or .net then the last character of the domain (or subdomain) is the character before the '.' otherwise the there is a problem with the top-level domain and the last character of the domain is the last character of the string.
   3. The last character of the name is the character before the '@' symbol, or if it is missing then the last character of the name becomes the last character of the domain (identified in step 2).
4. Definition of a subdomain: domains and subdomains are separated by periods '.' the domain is the first piece of text, everything else is a subdomain.
   1. example: doescher@cs.wisc.edu

## Corrections and Additions

1. Use the order of the error messages to decide which test to run first. The specification (in the comments provided) may be in a different order and less complete.
2. The email name may have at most 32 characters. The error message reads "name must have fewer than 32 characters" This error message has a mistake because it implies that the max is 31. The allowed maximum number of characters is 32. However, do not change the error message so that the grading script can identify correct work by exactly matching the provided output.
3. Hint: Use these guidelines to break the email address into its components.
   1. Find the "\0" or the last character of the string.
   2. If the last 4 characters are .com, .edu, or .net then the last character of the domain (or subdomain) is the character before the '.' otherwise the there is a problem with the top-level domain and the last character of the domain is the last character of the string.
   3. The last character of the name is the character before the '@' symbol, or if it is missing then the last character of the name becomes the last character of the domain (identified in step 2).
4. Definition of a subdomain: domains and subdomains are separated by periods '.' the domain is the first piece of text, everything else is a subdomain.
   1. example: doescher@cs.wisc.edu

    2. doescher is the name
    3. cs is the domain
    4. wisc is a subdomain
    5. edu is the top-level domain
5. Passwords: Check the first five conditions after the first password has been entered.  Then ask for the second password and verify they match.
6. Test Scripts: A set of tests have been released, which you can find here: **P1userTests.zip** This file can also be navigated to from the 'files' tab on canvas.
    1. In order to run these tests, you must download the .zip file, and transfer it into the folder where you have your project stored.
    2. Each test (labelled t1 - t5) contains some tricky inputs that you can run with your program.
    3. The output you should get are in files labelled o1 - o5.
    4. In order to run the tests, you can use the "<" utility (also known as redirection). Try running, ./verify < userTests/t***i*** in order to see the output of running the test. ***i** is any number between and including 1 and 5.*

## Learning Goals

This project will help you gain familiarity working with C programming and practice using arrays and pointers.

## Specifications

This project simulates making part of an online account for a website. The program asks a user to enter their username, email address, and password.  Then it verifies the correct formatting of each of these pieces of data, and it either reports success or tells the user what error they made and exits.

Please see the template for the formatting rules, the order they are applied, and the output statements generated for each condition.

For example, the username must begin with a letter [A-Z, a-z], have a maximum of 32 characters, and may only contain letters, digits, or the underscore [A-Z, a-z, 0-9, _].  These conditions are tested in this order, so if the username is “CS354isThe_Best_Most_AwesomeClass!@#$%^&*(Ever)!!!”. The first test will pass (this doesn’t generate any output), but the second test will fail that username has 50 characters. When the second test fails, the output “Max 32 characters” is printed, and the program ends (you will need to add a return 0 to your conditional statement;).

When each of the three pieces of data has been verified, the program should print a success message.

All messages printed on the screen have been written for you.  We test your code using exact match output testing.  If you change any of the messages, the tests will fail to match. (Hopefully, I haven’t made any typos, but if I did, please do not correct them, so our grading script works.)

## Comments

I have broken the username section down into individual pieces with comments.  For the other two sections, please use the specification and output messages to generate appropriate comments.

## Files

**verify_template.c** **(https://canvas.wisc.edu/courses/248855/files/20538073/download?download_frd=1)**

The first thing you should do is rename the verify_template.c file to verify.c (or anything else that will help you remember to turn in your work instead of the original template file)

## Compiling and running

```
>>> gcc -g -o verify verify.c -m32 -Wall
>>> ./verify
```

- do not the >>> this represents the prompt generate by the terminal
- gcc is the name of the compiler
- -g turns on debugging symbols

- -o verify indicates the name of the output file
- c is the name of your source code
- -m32 compiles the code for a 32 bit machine
- -Wall turns on all warnings.
- ./ indicates the current directory

## Test the Code

```
>>> ./verify < userTests/t1
```

The above line is an example of how to run the provided test named t1.

```
>>> cat userTests/o1
```

The above line is an example of how to view the actual output, from a provided output file named o1.

Test your code by running it and entering both valid and invalid usernames, email addresses, and passwords. If you come up with any tricky or clever tests, please post them to Piazza under the P1 heading to share with other class members.

## Strategy

Write your code in small pieces and test each line written by printing out a message.  This technique is called scaffolding. Remove the debugging messages after you verify the success of your code.

There is a lot of repetition in the required tasks to verify each of the three pieces.  Put the redundant code in functions, so you only have to get it written correctly once. For example, all three data items require length verification.  Write one function to test the length of the strings.

## Turn in

Upload your verify.c file to Canvas. Canvas renames all project files uploaded so you can name your work anything you like. Be sure to double-check to confirm that you haven't uploaded the original template.

## Style

1. Use meaningful variable names. Either use underscores for multi-word variables or CamelCase. Be consistent.
2. Be consistent with capitalization. For example, capitalize function names, use all caps for structs, enums, and #defines, and use lower case for variables.
3. Indent to match scope. Use four spaces instead of tabs.
4. Organize your code into sections as follows:
   1. #include <>
   2. #include ""
   3. #defines
   4. Data Types (e.g., structures and typedef)
   5. Global variables – (use global variables only when necessary)
   6. Function Prototypes
   7. Code - main() should be either the first or last function
5. Comments (It can be helpful to write the comments first)
   1. Describe what each block of code is trying to do.
   2. Good variable names may make this description easier (or potentially unnecessary).
   3. For functions, describe the purpose of the function and indicate the purpose of input parameters, output parameters, and the return value.
6. Multi-statement lines

1. Should include statements that logically go together.
2. For example: if (ptr == NULL) { printf(“error message\n”); exit(1); }